A Heavily Commented Linux Kernel Source Code

Zhao Jiong

www.oldlinux.org Jiong.Zhao@TongJi.edu.cn

A Heavily Commented Linux Kernel Source Code

Kernel Version 0.12

(Chinese Revision 5)

Zhao Jiong
Jiong.zhao@TongJi.edu.cn

WWW.OLDLINUX.ORG

2019-01-24

Abstract

This book provides detailed and comprehensive comments and explanations on all source code of the early Linux kernel (V0.12), aiming to enable readers to gain a comprehensive and profound understanding of the working mechanism of Linux in a shortest possible time and to lay a solid foundation for further study of modern Linux systems. Although the version of the analysis is very low, the kernel has been able to compile and run, and it already includes the essence of the working principle of Linux.

The book first briefly introduced the development history of the Linux kernel, explained the main differences between the various kernel versions and improvements, and gave the reasons for choosing the 0.12 kernel source code as the study object. Then it gives the basic knowledge needed to read the source code, outlines the hardware structure of the PC running the Linux system, the assembly language used by the kernel, the extends of C language, and focuses on the 80X86 processor in protected mode. Then we introduced the kernel code overview, given the kernel source directory tree structure, and according to the organizational structure of all kernel, programs and files are described in detail. In order to deepen the reader's understanding of the working principle of the kernel, the last chapter gives a number of related operational debugging tests. All relevant information in the book can be downloaded from the website www.oldlinux.org.

This book suits as the assistant and practical teaching material of university computer major student study operating system course, also suitable for self-study reference book of Linux lovers as learning kernel operating principle, also can be used as the reference book that the general technical personnel develops the embedded system.

Copyright statement

Dedicated to:

Sun Hongfang - my dear mother
Zhao Bichen - my beloved father

In teaching me to grow up
You have spent your hard life

Your son
Zhao Jiong

献给

孙洪芳 – 我亲爱的母亲
赵碧臣 – 我敬爱的父亲

在教诲我成长过程中
度过了你们艰辛一生

你们的儿子
赵炯

"RTFSC – Read The F**king Source Code :)!"

–Linus Benedict Torvalds

Table of Contents

Preface

Under the general trend of intelligent manufacturing and networking direct control of objects, the Linux operating system has become the most important basic platform for operation control in today's embedded systems. This book is a primer on the basic workings of the Linux operating system kernel.

The main goal of this book

The main goal of this book is to use a minimal amount of space or within a limited space to dissect the complete Linux kernel source code in order to obtain a full understanding of the basic functions and actual implementation of the operating system. To achieve a complete and profound understanding of the Linux kernel, a true understanding and introduction of the basic operating principles of the Linux operating system.

This book's readership is positioned to know the general use of Linux systems or has a certain programming basis, but it lacks the basic knowledge to read the current new kernel code and is eager to understand the working principle and actual code of the UNIX operating system kernel as soon as possible. Realize the lovers.

Features of this book

At the time of writing this book, there are books on the market that describe the Linux kernel that try to use the newer Linux kernel version (such as version 2.6.24 used by Fedora 8) to describe the kernel working mechanism. However, since the size of kernel source code is already very large (for example, 2.2.20 version has 2.68 million lines!), these books can only selectively explain and describe the Linux kernel source code, and many system implementation details are ignore. Therefore, it is difficult to have a clear and complete description of the Linux kernel.

The book “Linux Kernel Source Code Analysis” written by Scott Maxwell is basically oriented to the advanced level readers of Linux. It needs a more comprehensive basic knowledge to fully understand. And may be due to space limitations, the book does not comment on all the Linux kernel code, omitted a lot of kernel implementation details, such as the various header files used in the kernel (*.h), the tool to generate the kernel code image file The role of the program, each make file, and its implementation are not covered. Therefore, reading the book is difficult for readers who are at entry level.

The book "Leon's UNIX source code analysis" written by John Lions is a good book for learning UNIX source code of the operating system kernel, but because it uses the UNIX version V6, some of the code in the system call is With the assembler language of the long-deprecated PDP-11 series machine, it is difficult to conduct experiments when reading and understanding the source code related to the hardware part.

Andrew S. Tanenbaum's book "Operating Systems: Design and Implementation" is a good primer on operating system kernel implementation, but the MINIX system described in this book is a message-based kernel implementation mechanism, and Linux There are differences in the implementation of the kernel. Therefore, after learning this book, it is not very easy to start working on the newer Linux kernel source code.

When using these books for learning, there will be a feeling of "blind people feel like elephants". It is not easy to understand the overall concept of the specific implementation of the Linux kernel system, especially when the Linux system beginners use those books to learn the principle of the kernel, the overall operating structure of the kernel. It cannot be clearly formed in the mind. This has profound experience in my many years of experience in the Linux kernel learning. In October 1991, Linux founder Linus Torvalds mentioned the same problem in an article written during the development of Linux version 0.03. In this article titled "LINUX--a free unix-386 kernel", he said: "The development of Linux is for the use, learning and entertainment of those operating system enthusiasts and computer science students." Today's popular Linux systems have become larger and more complex, so they are no longer suitable as a starting point for beginners learning the operating system.

In order to fill this vacancy, this book uses a minimal amount of space or within a limited space to conduct a complete dissection of the complete Linux kernel source code in order to obtain a full understanding of the basic functions and actual implementation of the operating system. To achieve a complete and profound understanding of the Linux kernel, a true understanding and introduction of the basic operating principles of the Linux operating system.

Other benefits of reading early kernel code

At present, there have been many kernel versions developed specifically for embedded systems based on Linux's early kernels, such as DJJ's x86 operating system, Uclinux, etc. Many people in the world also realize the benefits of learning through the early Linux kernel source code. At present, people in China are already organizing human annotations to publish books similar to this article. Therefore, by reading the source code of the Linux kernel version earlier, it is indeed an effective way to learn the Linux system, and it is also very helpful for the research and application of the Linux embedded system.

In commenting on early kernel source code, the author found that early kernel source code was almost a condensed version of the newer kernels in use today. It already includes almost all the basic functional principles of the current version. As Leland L. Beck, author of "System Software: An Introduction to System Programming," introduced system programs and operating system design, he introduced an extremely simplified Simple Instruction Computer (SIC) system to illustrate the design and implementation of all system programs. The principle, which not only avoids the complexity of the actual computer system, but also a thorough description of the problem. Here, select the early kernel version of Linux as a learning object, and its guiding ideology is the same as that of Leland. This is one of the best choices for beginners of Linux kernel learning. The basic working principle of the Linux kernel can be deeply understood in the shortest possible time.

For those who are already familiar with the working principle of the kernel, it is necessary to read the kernel source code in order to allow the actual operation mechanism of the system in the actual work to produce no feeling of castle in the air.

Of course, using the early kernel as a learning object also has its disadvantages. The selected Linux early kernel version does not include support for virtual file system VFS, support for network systems, support for only a.out executable files, and description of complex subsystems in some other existing kernels. However, since this book is an introductory textbook that is used as a working mechanism for the Linux kernel, this is one of the advantages of choosing an earlier kernel version. By studying this book, you can lay a solid foundation for further studying these advanced contents.

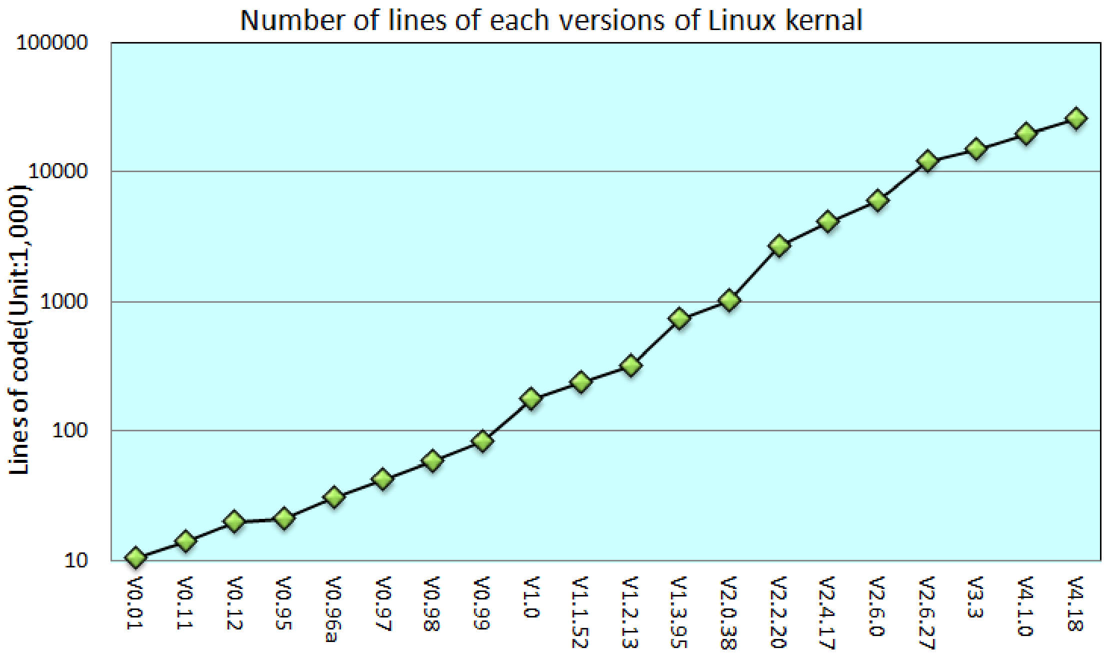
The importance and necessity of reading the complete code

Just as the founder of the Linux system stated in a newsgroup submission, to understand the true operating mechanism of a software system, be sure to read its source code (RTFSC – Read The F**king Source Code). The system itself is a complete whole, with many seemingly unimportant details. However, ignoring these details will make it difficult to understand the entire system and fail to truly understand the implementation method and means of an actual system.

Although some classic books on operating system principles (such as Mr. MJBach's "UNIX Operating System Design") can be used to theoretically guide the working principle of the UNIX-like operating system, the actual composition of the operating system is The understanding of the realization of internal relations is still not very clear. As Andrew S. Tanenbaum said, "many operating system textbooks are theoretical and light practice." "Most books and courses consume a lot of time and space for scheduling algorithms and completely ignore I/O. In fact, the former is usually less than one page of code. The latter often has to account for one-third of the total code of the entire system.” A large number of important details in the kernel are not mentioned. Therefore, it does not allow readers to understand the true beauty of a real operating system. Only after reading the complete kernel source code in detail will there be a sense of openness to the system and a deep understanding of the entire system's operational process. When you choose the newest or newer kernel source code to learn later, you will not encounter major problems and basically will be able to understand the new code content smoothly.

How to select the kernel code version to read

So, how can we choose to meet the above requirements without being confused by too much content and choose a suitable version of the Linux kernel to learn and improve the efficiency of learning? After comparing and selecting a large number of kernel versions, the author finally chose the 0.12 kernel that is similar to the current basic functions of the Linux kernel and is very short, as the best version for getting started. The following figure shows the statistics for some major Linux kernel version lines.



The current Linux kernel source code amount is in the number of millions of lines, the 2.6.0 version of the kernel code line is about 5.92 million lines, and the 4.18.X version of the kernel code is extremely large, and it has exceeded 25 million lines! So it is almost impossible to fully annotate and elaborate on these kernels. The 0.12 version of the kernel does not exceed 20,000 lines of code, so it can be explained and commented clearly in a book. Small but complete. In order to have an inductive understanding of the system under study and to use experiments to deepen the understanding of the principle, the author has also specifically rebuilt the Linux 0.12 system that is based on this kernel. Since it contains the GNU gcc compilation environment, using this system can also do some simple development work.

In addition, the use of this version can avoid the use of existing newer kernel versions that have become more and more complicated to study the various subsystems (such as virtual file system VFS, ext2, or ext3 file systems, network subsystems, new complex Memory management mechanisms, etc.)

The basic knowledge required by the book

When reading this book, readers must have some basic C language knowledge and Intel CPU assembly language knowledge. The best reference for C language is still the book "The C Programming Language" written by Mr. Brain W. Kernighan and Mr. Dennis M. Ritchie. Assembler language data can refer to any assembly language textbook that explains Intel CPU. Also need some embedded assembly language information. The authoritative information about embedded assembly is contained in the GNU gcc compiler manual. We can also search for some valuable essays about embedded assembly from the Internet. The book also contains some basic syntax descriptions for inline assembly (Section 5.5).

In addition, I hope readers have the following basic knowledge or related reference books. One is knowledge or information about 80x86 processor architecture and programming. For example, the 80x86 programming manual (INTEL 80386 Programmer's Reference Manual) can be downloaded from the Internet; the second is about 80x86 hardware architecture and interface programming knowledge or information. There is a lot of information in this regard; the third should also have the simple skills of using the Linux system at the beginning.

Since Linux kernel implementation was first developed according to the basic principles of the "UNIX operating system design" book, many variables or function names in the source code come from the book. Therefore, if you read this book properly, it will be easier to understand the kernel source code.

When Linus first developed the Linux operating system, he referred to the MINIX operating system. For example, the original Linux kernel version completely copied the MINIX 1.0 file system. Therefore, when reading this book, A.S. Tanenbaum's book "Operating System: Design and Implementation" also has great reference value.

Is reading an earlier version out of date?

On the surface, the book notes the contents of Linux's early kernel version as if the Linux operating system has just been released. Tanenbaum thinks that it is outdated (Linux is obsolete). However, by studying the content of this book, you will find that because of the small amount of early kernel source code and lean, using this book to learn the Linux kernel will have a very high learning efficiency, can do more with less, and get started quickly. And lay a solid foundation for continuing to further select the source code of the new kernel part. After completing this book, you will have a very complete and practical concept of how the system works. This complete concept makes it easy to further select and learn any part of the new kernel source code without having to Read the complete source code in the new kernel with a large amount of code.

Ext file system and MINIX file system

The Ext3 file system currently used on Linux systems was developed after kernel 1.x. Its function is detailed and its performance is also very complete and stable. It is the default standard file system on the current Linux operating system. However, as part of the introductory learning of the full working principle of the Linux operating system, in principle, the more streamlined the better. In order to achieve a complete understanding of an operating system, and without being overwhelmed by the complex and excessive details of the various subsystems, the principle of choosing the kernel version for learning is as simple as possible, as long as the system code can explain the actual working principle. The Linux kernel version 0.12 contained only the simplest MINIX 1.0 file system at the time, which is enough to understand the actual composition and working principle of a file system in an operating system. This is one of the main reasons to choose the early Linux kernel version for learning.

After reading this book in its entirety, I believe you will send this kind of sigh: "For the Linux kernel system, I'm finally getting started!" At this point, you should have enough confidence to further study the working principle and process of each part of the latest Linux kernel.

Dr. Zhao Jiong
Tongji University
2019.1

1 Overview

This chapter first reviews the process of the birth, development, and growth of the Linux operating system. This can be used to understand why the book chose an earlier version of the Linux system as a learning object. It then explains in detail the advantages and disadvantages of choosing an early version of the Linux kernel for learning and how to begin further learning. Finally, we briefly introduced the contents of each chapter.

1.1 The birth and development of Linux

Linux is a clone system of the UNIX operating system. It was born on October 5, 1991 (this is the time for the first official announcement). Since then, with the aid of the Internet, through the joint efforts of computer enthusiasts all over the world, it has now become the most widely used type of UNIX operating system in the world, and the number of users is still growing rapidly.

The birth, development, and growth of the Linux operating system depend on the following five pillars: the UNIX operating system, the MINIX operating system, the GNU project, the POSIX standard, and the Internet network. Based on these five basic clues, we follow the development history of Linux, its brewing process and its initial development. First of all, I will introduce the four basic elements, and then follow Linux founder Linus Torvalds to learn computer knowledge from her own interest in computers, start brewing her own operating system, release to the initial release of the Linux kernel version 0.01, and how difficult it will be. Step by step, with the help of hackers all over the world, the development of the more mature version 1.0 was finally introduced. It also describes the history of Linux's early development in detail.

Of course, the current Linux kernel version has been developed to version 4.18.x. The kernel used in most Linux systems is a stable 4.4.x-4.16.x kernel (where the second digit is an odd number, it means that it is being developed and cannot guarantee system stability). For the general history of Linux development, many articles and books have been introduced and will not be repeated here.

1.1.1 The birth of the UNIX operating system

The Linux operating system is a cloned version of the UNIX operating system. The UNIX operating system is a time-sharing operating system developed by Bell Labs's Ken. Thompson and Dennis Ritchie on the DEC PDP-7 minicomputer in the summer of 1969.

In order to be able to run his favorite Star Travel game on an idle PDP-7 computer, Ken Thompson developed UNIX operations within a month in the summer of 1969 while his wife returned home to California. The prototype of the system. At that time, the BCPL language (basic combination programming language) was used. After being rewritten by Dennis Ritchie in 1972 with a highly portable C language, the UNIX system was promoted in universities and colleges.

1.1.2 MINIX operating system

The MINIX system was developed by Andrew S. Tanenbaum (AST). AST is a mathematics and computer science system at Vrije University in Amsterdam, the Netherlands. He is a senior member of ACM and IEEE (only a few people in the world are senior members of the two associations). A total of more than 100 articles and 5 computer books were published.

Although AST was born in New York, it was a Dutch expatriate (his grandfather came to the United States in 1914). He studied at a high school in New York, a university at M.I.T, and a doctorate at the Berkeley campus of the University of California. Due to his postdoctoral studies, he came to his hometown of the Netherlands. Since then, it has been with the hometown. Later, I started teaching and graduate school at Vrije University. Amsterdam, the capital city of the Netherlands, is a year-round rainy city, but for the AST, this is best, because in this environment he can often stay at home to play with his computer.

MINIX was created in 1987 and is mainly used by students to learn operating system principles. By 1991 the version was 1.5. There are currently two major versions in use: Version 1.5 and Version 2.0. At that time, the operating system was free at university, but other uses were not. Of course, the current MINIX system is free and can be downloaded from many FTP sites.

For the Linux system, he later expressed his compliments to the developer Linus. However, he believes that the development of Linux is largely due to the fact that in order to keep MINIX small, he will be able to complete the learning within one semester, thus failing to accept the expansion requirements of MINIX from many people around the world. So under this premise inspired Linus to write a Linux system. Of course, Linus also just caught this good time.

As an operating system, MINIX is not an excellent one, but it also provides system source code written in C and assembly language. This is the first time that aspiring programmers or hackers have been able to read the operating system's source code. At the time, this source code was a secret that software vendors had been carefully guarding.

1.1.3 GNU Project

The GNU Project and the Free Software Foundation were founded by Richard M. Stallman in 1984 to develop a complete operating system similar to UNIX and free software: the GNU system (GNU is "GNU's Not". Recursive abbreviation for Unix, which is pronounced "guh-NEW"). Various GNU operating systems using Linux as the core are being widely used. Although these systems are often referred to as "Linux," Stallman believes that, strictly speaking, they should be referred to as GNU/Linux systems.

By the early 1990s, the GNU project had developed many high-quality free software, including the famous emacs editing system, bash shell program, gcc series compiler, gdb debugger and so on. These softwares create a suitable environment for the development of the Linux operating system. This is one of the foundations for the birth of Linux, so that many people now refer to the Linux operating system as the "GNU/Linux" operating system.

1.1.4 POSIX standard

POSIX (Portable Operating System Interface for Computing Systems) is a cluster of standards developed by IEEE and ISO/IEC. The standard is based on existing UNIX practices and experiences and describes the operating system's call service interface. The applications used to ensure the compilation can be ported and run on multiple operating systems at the source code level. It was based on the early work of a UNIX user group (usr/group) in the early 1980s. The UNIX user group originally attempted to re-integrate the distinction between AT&T's System V operating system and Berkeley CSRG's BSD operating system's call interface. In 1984, the /usr/group standard was customized.

In 1985, the IEEE Operating System Technical Committee Standards Subcommittee (TCOS-SS) began, under the aegis of ANSI, instructing the IEEE Standards Committee to establish a formal standard for program source code portability operating system service interfaces. In April 1986, the IEEE developed a trial standard. The first formal standard was approved in September 1988 (IEEE 1003.1-1988), and also the POSIX.1 standard that is often mentioned later.

By 1989, POSIX's work was transferred to the ISO/IEC community and the 15 working group continued to develop it as an ISO standard. By 1990, POSIX.1, in conjunction with the already adopted C language standard, was formally approved as the IEEE 1003.1-1990 (also ANSI) and ISO/IEC 9945-1:1990 standards.

POSIX.1 only specifies system service application programming interfaces (APIs) and only summarizes basic system service standards. Therefore, the working group expects to establish standards for other functions of the system. So the work of IEEE POSIX began. Ten approval plans were in progress at the beginning, and nearly 300 people participated in the quarterly one-week meeting. The work that started was command and tool standard (POSIX.2), test method standard (POSIX.3), and real-time API (POSIX.4). In the first half of 1990, 25 plans were already in progress and 16 working groups were involved. At the same time, some organizations are also developing similar standards such as X/Open, AT&T, and OSF.

In the early 1990s, the formulation of the POSIX standard was in the final stage of voting, which was between 1991 and 1993. At this point, when Linux was just starting out, this UNIX standard provided extremely important information for Linux, enabling Linux to be developed under the guidance of standards and compatible with most UNIX operating systems. In the original Linux kernel source code (versions 0.1, 0.11, and 0.12) the Linux system was ready for compatibility with the POSIX standard. Several symbolic constants for POSXI standard requirements have been defined in the /include/unistd.h file of the Linux version 0.01 kernel, and Linus wrote in the comment: "OK, this may be a joke, but I'm working on it. It does."

On July 3, 1991, Linus mentioned on the post posted on comp.os.minix that it is collecting POSIX data. It revealed that he is working on the development of an operating system, and at the beginning of development, he had thought of the problem of compatibility with POSIX.

1.1.5 The birth of the Linux operating system

In 1981, IBM introduced the world-renowned microcomputer IBM PC. Between 1981 and 1991, the MS-DOS operating system was always the master of the microcomputer operating system. At this time, although the price of computer hardware has decreased year by year, software prices have remained high. At that time, Apple's MACs operating system can be said to be the best performance, but its price is so that no one can easily get close.

Another computer technology camp at the time was the UNIX world. However, the UNIX operating system is not only an expensive issue. In order to seek high profit margins, UNIX dealers have pushed prices extremely high, and PC users can't get close to it. The UNIX source code that once received permission from Bell Labs to be used for teaching in the university has also been carefully guarded against disclosure. For the majority of PC users, large vendors in the software industry have never given effective solutions to this problem.

At this time, the MINIX operating system appeared, and a book describing the principles of its design and implementation was issued at the same time. Since this book written by AST was very detailed and well-articulated, almost all computer enthusiasts around the world began to read this book in order to understand how the operating system works. It also includes Linus Benedict Torvalds, the founder of the Linux system. At that time (in 1991), he was a sophomore in the Department of Computer Science at the University of Helsinki and a self-taught computer hacker. The 21-year-old Finnish young man likes to drum up his computer and test the performance and limitations of the computer. But what he lacked at the time was a professional-level operating system.

During the same year, the GNU program has developed a number of software tools. The most anticipated

GNU C compiler has emerged, but the free GNU operating system has not yet been developed. Even the MINIX operating system used in teaching has begun to have copyright, and it is necessary to purchase it to get the source code. Although the GNU operating system HURD has been under development, it did not appear to have been completed within a few years.

To better learn computer knowledge (perhaps just for interest), Linus purchased a 386-compatible computer using Christmas lucky money and loans, and mailed a MINIX system software from the United States. While waiting for the MINIX software, Linus learned the hardware knowledge of Intel 80386 seriously. In order to be able to connect to the school's mainframe through a Modem dial-up, he uses assembly language and uses the multitasking features of the 80386 CPU to create a terminal emulation program. Later, in order to copy his own software on an old computer to a new computer, he also compiled drivers for floppy disk drives, keyboards, and other hardware devices.

Through programming practice and recognizing the many limitations of the MINIX system during the learning process (MINIX is good, but it is only a simple operating system for teaching purposes, rather than a powerful and practical operating system), Linus already has something similar. The code for the operating system device driver, so he began to have the idea of a new operating system. At this point, the GNU Project has developed many tools and software, among which the most anticipated GNU C compiler has appeared. Although GNU's free operating system HURD is under development. But Linus has waited for no hurry.

From April 1991, he began to develop his own operating system by modifying the terminal emulation program and hardware drivers. At the beginning, his purpose was simple, just to learn the programming techniques of the Intel 386 architecture protection mode operation. However, the development of Linux has completely changed its original intention. According to Linus's news release on the comp.os.minix newsgroup, we can see that he has gradually evolved from learning the MINIX system stage to developing his own Linux system.

Linus delivered the message to comp.os.minix for the first time on March 29, 1991. The title of the posted post is "gcc on minix-386 doesn't optimize". It is about the gcc compiler running optimized on the MINIX-386 system (MINIX-386 is an improvement from Bruce Evans using Intel 386 features 32 Bit MINIX system). From this it can be seen that Linus had already begun to study the MINIX system in depth in early 1991, and during this time there has been an improvement of the MINIX operating system. After further learning about the MINIX system, this idea gradually evolved into the idea of redesigning a new operating system based on the Intel 80386 architecture.

When he answered someone's question on MINIX, the first sentence said was "Read the F**ing Source Code :-)"). He thinks the answer lies in the source program. This also shows that for the learning system software, we not only need to understand the basic working principles of the system, but also need to combine the actual system to learn how to implement the actual system. After all, theory is a theory, in which many branches are omitted. Although these branch problems do not have much theoretical content, they are a necessary part of the system, just like a feather in a sparrow.

From April 1991, Linus spent almost all of his time researching the MINIX-386 system (Hacking the kernel) and trying to port GNU software to the system (GNU gcc, bash, gdb, etc.). And announced on comp.os.minix on April 13 that he had successfully ported bash to MINIX, and he could not afford to leave the shell software.

The first Linux-related news was released on comp.os.minix on July 3, 1991. (Of course, there was no such name as Linux at that time. Linus thought that the name might be FREAK , FREAX. The English meaning is grotesque, monsters, whimsical, etc.). It revealed that he is developing Linux system and he has already thought of the problem of compatibility with POSIX.

In another announcement by Linus (comp.os.minix, August 25, 1991), he asked all MINIX users "What do you most want to see in the MINIX system?" ("What would you like to see?" In minix?”), in which he revealed for the first time that a (free) 386(486) operating system is being developed, and that he is only interested in it. The code will not be large and will not be as professional as GNU. I hope you will give us some feedback on what features the MINIX system likes and dislikes, and explain that due to practical and other reasons, the newly developed system is just like MINIX (and uses MINIX's file system). And it has successfully ported bash (version 1.08) and gcc (version 1.40) to the new system and it will be practical in months.

Finally, Linus stated that the operating system he developed does not use a single line of MINIX source code; because of the task switching feature of the 386, the operating system is not portable (no portability) and only AT hard disks are used. Linus did not consider the issue of Linux portability. But at present, Linux can run on almost any kind of hardware architecture.

On October 5th, 1991, Linus published a message on the comp.os.minix newsgroup, officially announcing the birth of the Linux kernel system (Free minix-like kernel sources for 386-AT). This news can be called the birth declaration of Linux, and has been widely circulated. Therefore, October 5 was a special day for the Linux community, and many later Linux versions had chosen this date. So RedHat chose this day to release its new system is not accidental.

1.1.6 Linux operating system version changes

Since the birth of the Linux operating system to the 1.0 release, a number of major releases have been released as shown in Table 1–1. Linus looked at all of the previous versions of 1.0 when he started learning to use the version management tool BitKeeper in September 2003. In fact, the Linux system does not have this version of 0.00, but since Linus' experiment on his own 80386 compatible machine succeeded in switching between two tasks under the control of clock interruption, he further enhanced his idea of developing his operating system to some extent. . Therefore we are also listed as a version. The Linux version of the kernel version was completed on September 17, 1991. However, Linus has no copyright awareness at all, so only one copy of copyright information appears in this version of the include/string.h file. The keyboard driver for this version of the kernel is hard-coded only into Finnish code, so only the Finnish keyboard is supported. Only 8MB physical memory is supported. Due to a mistake by Linus, the subsequent 0.02, 0.03 version of the kernel source code was destroyed and lost.

Table 1-1 Earlier major versions of the kernel

Version No.	Release date	Description
0.00	1991.2-4	The two processes display 'AAA...' and 'BBB...' on the screen, respectively. (Note: No release)
0.01	1991.9.17	The first official release of the Linux kernel version. Multi-threaded file system, segmentation, and paging memory management. Does not include floppy disk drivers yet.
0.02	1991.10.5	This version and version 0.03 is an internal version that is currently unavailable. Features the same as above.
0.10	1991.10	The Linux kernel version released by Ted Ts’o. Added memory allocation library functions. The boot directory contains a script that converts as86 assembler syntax to gas assembler syntax.
0.11	1991.12.8	Basically functioning kernel. Supports hard disk and floppy drive devices as well as

		serial communications.
0.12	1992.1.15	The more stable version mainly increases the software simulation program of the math coprocessor. Added job control, virtual console, file symbolic links, and virtual memory swapping capabilities.
0.95.x (ie 0.13)	1992.3.8	Virtual file system support was added in this version, but it still contains only one MINIX file system. Added login functionality. Improves the performance of floppy disk drivers and file systems. Changed hard disk naming and numbering. The original naming method is the same as that of the MINIX system. At this time, it is the same as the current Linux system. Support CDROM.
0.96.x	1992.5.12	Began to add UNIX Socket support. Added ext file system alpha tester. SCSI drivers are officially added to the kernel. Floppy disk type is automatically recognized. Improved serial driver, cache, memory management performance, support for dynamic link libraries, and the ability to run X-Windows programs. The keyboard driver written in the original assembly language has been rewritten with C. Compared with the 0.95 kernel code, there are great changes.
0.97.x	1992.8.1	Added support for new SCSI drivers; dynamic caching; msdos and ext file system support; bus mouse drivers. The kernel is mapped to the beginning of the linear address 3GB.
0.98.x	1992.9.28	Improve support for TCP/IP (0.8.1) networks and correct extfs errors. Rewritten memory management section (mm), each process has 4GB of logical address space (the kernel occupies 1GB). Starting from 0.98.4, each process can open 256 files at the same time (originally 32), and the process's kernel stack uses a single memory page independently.
0.99.x	1992.12.13	Re-design the process of the use of memory allocation, each process has 4G linear space. Constantly improving the network code. NFS support.
1.0	1994.3.14	The first official version.

The existing 0.10 version of the kernel code is a version of Ted Ts'o that was preserved at the time, Linus's own has also been lost. This version is a great improvement over the previous versions. On this version of the kernel system, GNU gcc has been used to compile the kernel, and has begun to support the operation of mounting/unmounting file systems. From this kernel version, Linus added copyright information for each file: "(C) 1991 Linus Torvalds". Some other changes in this version include: the original boot program boot/boot.s split into two programs boot/bootsect.s and boot/setup.s; 1 supports up to 16MB of physical memory; 2 drivers and memory management procedures Created their own subdirectories separately; 3 Added floppy driver; 4 Supported file read-ahead operations; 5 Supported dev/port and dev/null devices; 6 Rewritten kernel/signal.c code, added sigaction() Support etc.

Relative to the 0.10 version of the kernel, Linux 0.11 version of the changes are relatively small. However, this version is also the first stable version, and other people are beginning to participate in kernel development. The main additions in this version are: 1 load requirements for the execution program; 2 execute the /etc/rc initial file at startup; 3 build the math coprocessor simulation program frame program structure; 4Ted Ts'o adds a script program The processing code; 5 Galen Hunt added support for multiple display cards; 6John T Kohl modified the kernel/console.c program to enable the console to support tweet and KILL characters; 7 provides support for multilingual keyboards.

Linux 0.12 is a more satisfactory kernel version of Linus and a more stable kernel. During the Christmas season in 1991, he compiled the virtual memory management code so that "large" software like gcc could be used on machines with only 2MB of memory. This version makes Linus feel that releasing the 1.0 kernel version is not something that is out of sight, so he immediately upgraded the next version (0.13 version) to version 0.95. Another implication of Linus's ability to do this is to make everyone not feel that they are still far from version 1.0. However, due to the hasty release of the 0.95 version, which also contains more errors, so when the 0.95 version was just released, there were more Linux enthusiasts encountered problems in use. At that time, Linus felt like he had encountered a catastrophe. However, he has accepted this lesson since then. Every time a new kernel version is released later, he will undergo more careful testing and let a few good friends try it out before officially publishing it. The main changes in the 0.12 version of the kernel are: 1Ted Ts'o adds support for terminal signal processing; 2 can change the screen ranks used when starting up; 3 corrects a race condition caused by a file IO; 4 adds support for shared libraries Support, saving memory usage; 5 symbolic link handling; 6 deletion of directory system calls; 7 Peter MacDonald implements virtual terminal support, making Linux even superior to certain commercial versions of UNIX at the time; Function support, which was modified by Peter MacDonald based on patches provided by some people for MINIX, but MINIX did not adopt these patches; 9 re-executable system calls; 10Linus compiled math coprocessor simulation code.

Version 0.95 was the first Linux kernel version to use the GNU GPL copyright. There are actually three sub-versions of this version. Due to some problems encountered when the first 0.95 release was released on March 8, 1992, another 0.95a version was immediately released in less than 10 days (March 17). And in a month later (April 9th), 0.95c+ was released again. The biggest improvement in this version is the introduction of the virtual file system VFS structure. Although only the MINIX file system was supported at the time, the program structure has been extensively adjusted to support multiple file systems. The code for the MINIX file system is put into a separate MINIX subdirectory. Some of the other changes in the 0.95 kernel include: 1 Added login interface; 2 Ross Biro added debugging code (ptrace); 3 Floppy disk drive track buffering; 4 Non-blocking pipeline file operations; 5 System restart (Ctrl-Alt-Del); Swapon() system call to select swap devices in real time; 6 support for recursive symbolic links; 7 support for 4 serial ports; 8 support for hard disk partitions; 9 support for more types of keyboards; 10James Wiegand compiles initial parallel port drivers, etc. .

In addition, starting with the 0.95 release, many of the kernel's improvements (providing patches) were dominated by others, and Linus's main task began to become the maintenance of the kernel and decide whether to adopt a patch. Until now, the latest kernel version is version 4.16.16 released in June 2018. Its use of gz compressed source code package also has about 152MB! The latest version of each major stable release is shown in Table 1–2. Table 1-2

Table 1-2 New kernel source code size

Version number	Release date	Size (after gz compression)
2.0.40	2004.2.8	7.2 MB
2.2.26	2004.2.25	19 MB
2.6.25	2008.4.17	58 MB
3.0.10	2011.11.21	92MB
4.4.10	2016.5.11	127MB
4.16.16	2018.6.16	152MB

1.1.7 The reason for the Linux name

At the beginning of the Linux operating system, it was not called Linux. Linus named his operating system

FREAK. Its English meaning is grotesque, monster, whimsical. When he uploaded the new operating system to the ftp.funet.fi server, administrator Ari Lemke disliked the name very much. He believes that since it is Linus's operating system, take its homonym Linux as the operating system's directory, so the name of Linux began to pass down.

In Linus's autobiography "Just for Fun," Linus explains:

“Honest: I didn't want to ever release it under the name Linux because it was too egotistical . What was the name I reserved for any eventual release ? Freax. (Get it? Freaks with the requisite X.) In fact, some of the early make files-the files that describe how to compile the sources-included the word "Freax" for about half a year. But it really didn't matter. At that point I didn't need a name for it because I wasn't releasing it to anybody.”

“And Ari Lemke, who insured that it made its way to the ftp site, hated the name Freax. He preferred the other working name I was using-Linux-and named my posting: pub/OS/Linux. I admit that I didn't put up much of a fight. But it was his doing . So I can honestly say I wasn't egotistical, or half-honestly say I wasn't egotistical . But I thought, okay, that's a good name, and I can always blame somebody else for it, which I'm doing now.”

1.1.8 The main contributor to the development of early Linux systems

As can be seen from the early Linux source code, one of the most famous developers of the Linux system in addition to Linus himself is Theodore Ts'o (Ted Ts'o). He graduated from MIT Computer Science in 1990. In college time he actively participated in various student activities held in the school. He likes cooking, cycling, and of course Hacking on Linux. Later he began to like the amateur radio telegram campaign. He currently works at IBM on system programming and other important issues. He is also an IETF member of the International Network Design, Operations, Sales and Research Open Group.

The popularity of Linux in the world also has his great credit. As early as when the Linux operating system came out, he provided Maillist with great enthusiasm for the development of Linux. Almost since Linux was first released, he has been contributing to Linux. He was also the first person to add programs to the Linux kernel (the ramdisk.c virtual disk driver and the kernel memory allocation program kmalloc.c in the Linux kernel version 0.10). Until now he is still engaged in Linux-related work. In North America, he first established the Linux ftp site (tsx-11.mit.edu), and this site still provides services for the majority of Linux users. One of his biggest contributions to Linux was to propose and implement the ext2 file system. The file system has now become the de facto file system standard in the Linux world. Recently he introduced the ext3 file system. The system greatly improves the stability and access efficiency of the file system. As his admiration, the Linux Journal issue of the 97th issue (May 2002) used him as a cover character and interviewed him. He currently works for the IBM Linux Technology Center and is working on the Linux Standard Base (LSB).

Another famous person in the Linux community is Alan Cox. He originally worked at Swansea University College in Wales. At first, he particularly likes to play computer games, especially MUD (Multi-User Dungeon or Dimension). In the posts of games.mud news group in the early 90s you can find a lot of posts he posted. He even wrote a history of MUD development (rec.games.mud news group, March 9, 1992, A history of MUD). As MUD games are closely related to the internet, he slowly became fascinated with computer networks. In order to play the game and improve the speed of the computer running the game and the network transmission speed, he needs to choose a most satisfactory operating platform. So he began to contact various types of operating systems. Because of the lack of money, he could not afford even the MINIX system. When Linux 0.1x and 386BSD were released, he took a long time to purchase a 386SX computer. Since the 386BSD requires math coprocessor support and the computer with the Intel 386SX CPU does not have a math coprocessor, he installed the Linux system. So he started to learn Linux with free source code and started to have interest in Linux systems, especially with regard to networking. In the discussion of Linux single-user mode of operation, he even praised Linux for being beautifully implemented.

After the release of Linux 0.95, he began writing patches (modification programs) for the Linux system (remembering that his two earliest patches were not adopted by Linus) and became the earliest users of TCP/IP network code on Linux systems. one. Later, he gradually joined the Linux development team and became one of the main responsible for maintaining the Linux kernel source code. It can also be said to be the most important figure in the Linux community after relaying Linus. Later Microsoft invited him to join, but he simply refused. Since 2001, he is responsible for maintaining the Linux kernel 2.4.x code. Linus is mainly responsible for the development of the latest development version of the kernel (odd version, such as 2.5.x version).

Michael K. Johnson, author of The Linux Kernel Hackers' Guide, was also one of the first people to contact the Linux operating system (from version 0.97). He is also one of the initiators of the well-known Linux Document Project (LDP). He once worked for Linux Journal and now works for RedHat.

The Linux system is not the only backbone that can develop into what it is today. There are many computer experts who have made great contributions to Linux. We will not list them here. The specific list of major contributors can be found in the CREDITS file in the Linux kernel, which lists in alphabetical order the list of more than 400 people who contributed significantly to Linux, including their email address and mailing address, home page, and major contributions. Deeds and other information.

Through the above explanation, we can sum up the above five pillars of Linux as follows:

- UNIX Operating System -- UNIX was born in Bell Labs in 1969. Linux is a UNIX clone system. The importance of UNIX goes without saying.
- The MINIX operating system -- The MINIX operating system is also a UNIX clone system. It was developed in 1987 by the famous computer professor Andrew S. Tanenbaum. Due to the emergence of the MINIX system and the availability of source code (which can only be used free of charge in universities), the whirlwind of learning the UNIX system was spurred by universities around the world. Linux first started development in 1991 with reference to the MINIX system.
- The GNU Project -- The development of the Linux operating system, and most of the software used on Linux is basically from the GNU program. Linux is only a kernel of the operating system. Without the GNU software environment (such as the bash shell), Linux will be difficult to move.
- POSIX Standard -- This standard has played an important role in the development of the Linux operating system after the formal development. It is the beacon of Linux's progress.
- Internet - If you don't have an Internet network and don't have the unselfish dedication of countless computer hackers all over the world, then Linux can only grow to a level of 0.13 (0.95).

1.2 Content review

This book will mainly describe and comment on the early Linux kernel version 0.12. The Linux-0.12 version was released on January 15, 1992. Include the following files when publishing:

```
bootimage-0.12.Z - a compressed boot image file with a U.S. keyboard code;
rootimage-0.12.Z - 1200kB compressed root file system image file;
linux-0.12.tar.Z - Kernel source code file. The size is 130KB, and only 463KB after expansion;
```

as86.tar.Z	- Bruce Evans' binary execution file. 16-bit assembler and loader;
INSTALL-0.11	- Updated installation information file.

bootimage-0.12.Z and rootimage-0.12.Z are compressed floppy image files. Bootimage is the boot image file, which mainly includes disk boot sector code, operating system loader, and kernel execution code. When the PC starts, the program in the ROM BIOS reads the boot sector code and data from the default boot drive into memory, and the boot sector code reads the operating system loader and kernel execution code into memory and then controls It is up to the operating system loader to further prepare the kernel for initialization, and the final loader will give control to the kernel code. Kernel code needs file system support to function properly. Rootimage is the root file system used to provide the most basic support to the kernel, including the operating system at least some configuration files and command execution procedures. For UNIX-based file system used in Linux system, it mainly includes some specified directories, configuration files, device drivers, development programs, and all other user data or text files. The combination of these two disks is equivalent to a bootable DOS operating system disk.

as86.tar.Z is a 16-bit assembler linker package. linux-0.12.tar.Z is the compressed Linux 0.12 kernel source code. INSTALL-0.11 is a simple installation documentation for the Linux 0.11 system. It also applies to Linux systems that use the 0.12 kernel.

At present, in addition to the original rootimage-0.12.Z file, the other four files can be found. However, the author has used the resources on the Internet to re-create a fully usable rootimage-0.12 root file system for Linux 0.12. The gcc 1.40 compiler that can be used in the 0.12 environment is recompiled and the available experimental development environment is configured. Currently, these files can be downloaded from the oldlinux.org website. The specific download directory location is:

- http://oldlinux.org/Linux.old/images/ This directory contains the kernel image file bootimage and the root file system image file rootimage that have been created.
- http://oldlinux.org/Linux.old/kernels/ This directory contains the kernel source code programs, including the Linux 0.12 kernel source code program described in this book.
- http://oldlinux.org/Linux.old/bochs/ This directory contains Linux systems that have been set up to run under the computer simulation system bochs.
- http://oldlinux.org/Linux.old/Linux-0.12/ This directory contains some of the other tools that can be used in the Linux 0.12 system and some of the original installation instructions.

This book mainly analyzes all the source code programs in linux-0.12 kernel in detail, and makes detailed comments on each source program file, including comments on Makefile files. The analysis process is mainly carried out according to the computer startup process. Therefore, the consistency of the analysis until the end of the initialization kernel starts calling the shell program. The rest of the programs are for their own analysis, there is no coherence, so you can read according to their own needs. However, some application examples are provided during the analysis.

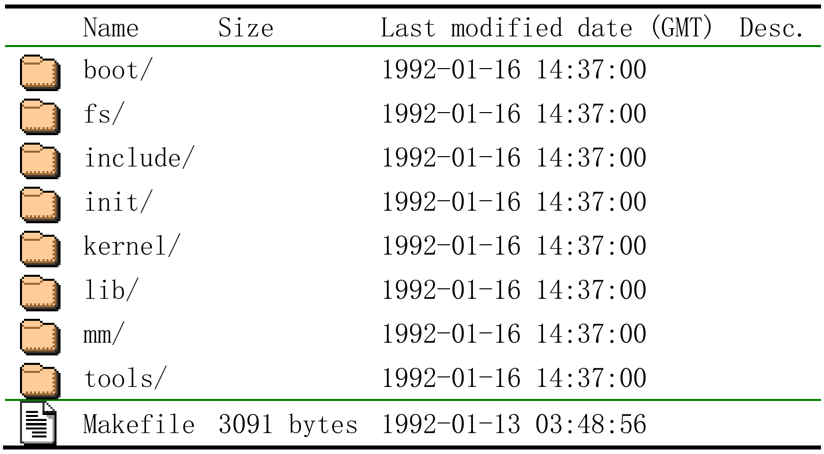
In the process of analyzing all programs, if the author thinks it is difficult to understand the statement, it will give a detailed description of the relevant knowledge. For example, when an input/output operation to the interrupt controller is encountered, a detailed description of the Intel Interrupt Controller (8259A) chip will be given and the used commands and methods will be listed. This will help deepen the understanding of the code, but also better understand the use of the hardware used, the author believes that this method of interpretation than a separate chapter to the overall introduction of hardware or other knowledge is much more efficient.

Taking the Linux 0.12 kernel to "dissect" is to increase the efficiency of our understanding of Linux's operating mechanism. Linux-0.12 version of the entire kernel source code is only about 463K bytes, including the content is basically the essence of Linux. The latest kernel version 2.6.XX is very large, 200 megabytes. Even if you spend a lifetime learning to read it may not be able to read all. Maybe you have to ask, "Since you want to start with Jane, why not analyze the smaller version of the Linux kernel source code for version 0.01? It's only about 240K bytes." The main reason is because the 0.01 version of the kernel code has too many shortcomings, not even Including drivers for floppy disks also does not relate well to the use of math coprocessors and instructions for login procedures. And the structure of the bootstrapping boot program is not the same as the current version, and the 0.12 boot boot program structure is basically the same as now. Another reason is that you can find the earlier version 1.22 of the already compiled and compiled kernel image file (bootimage-0.12), which can be used for boot demonstrations. If you add a simple root file system image (rootimage-0.12), it will be able to run normally.

There are also deficiencies in learning with Linux 0.12. For example, the kernel version does not include some very important code related to special process waiting queues, TCP/IP networks, etc. The allocation and use of memory is also different from the current kernel. Fortunately, the network code in Linux is basically self-contained, and the relationship with the kernel mechanism is not very large, so you can analyze the code after you understand the basic principles of Linux work.

This book describes all the code in the Linux kernel. In order to maintain the integrity of the structure, the description of the code is based on the structure of the source code in the kernel. Basically, the contents of each source code is a chapter. The order of the source files introduced can be found in the previous file list index. The directory structure of the entire Linux kernel source code is shown in Listing 1-1. All directory structures are based on Linux as the current directory.

List 1-1 Linux/ directory



	Name	Size	Last modified date (GMT)	Desc.
📁	boot/		1992-01-16 14:37:00	
📁	fs/		1992-01-16 14:37:00	
📁	include/		1992-01-16 14:37:00	
📁	init/		1992-01-16 14:37:00	
📁	kernel/		1992-01-16 14:37:00	
📁	lib/		1992-01-16 14:37:00	
📁	mm/		1992-01-16 14:37:00	
📁	tools/		1992-01-16 14:37:00	
📄	Makefile	3091 bytes	1992-01-13 03:48:56	

The content of this book can be divided into five parts. Chapters 1 to 4 are basics. The operating system is closely related to the hardware environment being run. If you want to thoroughly understand the entire operation of the operating system, then you need to understand its hardware operating environment, especially the processor multi-task operating mechanism. This part introduces in more detail the hardware composition of the microcomputer, the programming language used to compile the Linux kernel program, and the programming principle under Intel 80X86 protection mode; the second part includes chapters 5 through 7, describing the kernel boot boot and 32-bit operation. The preparation phase of the method should be fully read as a beginner to learn the kernel; the third part from Chapter 8 to Chapter 13 is the main part of the kernel code. The contents of Chapter 8 can be used as the main clue to read the subsequent chapters of this section. Chapters 14 to 16 are the contents of the fourth section and can serve as reference for reading the third part of the source code. The last part includes only Chapter 17, which describes how to use the PC simulation software system Bochs to conduct various experimental activities on the Linux 0.12 kernel.

The second chapter is based on the hardware block diagram of the traditional microcomputer system. It mainly introduces the components of the IBM PC/AT386 microcomputer running on the Linux kernel. Describe the functions and relationships of each major section. At the same time, it is also compared with the block diagram of the latest microcomputer. This will provide enough relevant information for readers who have not learned the principles of computer composition.

Chapter 3 introduces the programming language, object file format, and compilation environment used in the Linux 0.12 kernel. The main goal is to provide the assembly language and GNU C language extension knowledge needed to read the Linux 0.12 kernel source code. This chapter first introduced the syntax and usage of as86 and GNU as assembler in more detail, and then explained the common C language extensions such as inline assembly, statement expressions, register variables, and inline functions in the GNU C language. The mutual calling mechanism between C and assembly functions is described in detail. Finally, the use of the Makefile is briefly described.

Chapter 4 describes the architecture of the 80X86 CPU and some basic knowledge of protected mode programming. It lays a solid foundation for preparing to read the Linux kernel source code based on the 80X86 CPU. These include: 80X86 basics, protected mode memory management, interrupt and exception handling, task management, and a simple multitasking kernel example.

Chapter 5 outlines the Linux operating system architecture, the organization of the kernel source code files, and the general functionality of each file. It also introduces the use of Linux for physical memory allocation, several stacks of the kernel, and how they are used, and the use of virtual linear addresses. Finally, it begins to comment the first file seen in the Linux/ directory in the kernel package, which is the contents of the overall Makefile of the kernel code. This file is the compilation management configuration file for all kernel source programs and is used by the build management tool software make.

Chapter 6 will explain in detail the three assembly language programs in the boot/ directory, including the bootdisk.ss of the disk boot program, the setup.s assembler that takes the parameters in the BIOS, and the 32-bit run start code program head.s. The three assembler programs complete the bootloading of the kernel from the block device into memory and detect system configuration parameters, completing all the work before entering the 32-bit protected mode. Prepare for the kernel system to perform further initialization work.

Chapter 7 mainly introduces the initialization program main.c of the kernel system in the init/ directory. It is a key point for the kernel to complete all initialization work and enter normal operation. After completing all the initialization of the system, a process for the shell is created. In the introduction of the program will need to see the other programs it calls, so the reading of the subsequent chapters can be performed in the order called here. Since memory management program functions are widely used in the kernel, this chapter should be read first. When you can really understand all the programs up to the main.c program, you should already have a certain understanding of the Linux kernel. It can be said that half of them are already started, but you also need to file systems, system calls, each Drivers, etc. for a deeper reading.

Chapter 8 mainly introduces all programs in the kernel/ directory. The most important part of the process is the process scheduler(), sleep_on(), and program related system calls. At this point you should already know some of the important programs. From the beginning of this chapter, we will encounter many assembly language statements embedded in C language programs. The basic syntax for embedded assembly statements is described in Chapter 3.

Chapter 9 explains the block device program in the kernel/blk_drv/ directory. This chapter mainly contains drivers for block devices such as hard disks and floppy disks. It is mainly used to deal with file systems and high-speed buffers, and contains more hardware-related content. Therefore, you need to refer to some hardware information when reading this chapter. It's best to first look at the sections of the file system.

Chapter 10 Annotates the character device drivers in the kernel/chr_drv/ directory. This chapter mainly deals with serial line drivers, keyboard drivers, and monitor drivers. These drivers constitute the serial terminal and console terminal devices supported by the 0.12 kernel. Therefore, this chapter also contains more hardware-related content. Need to refer to related hardware books when reading.

Chapter 11 introduces the math coprocessor simulation program in the kernel/math/ directory. Due to the version of the kernel annotated in this book, coprocessors have not really started to be supported yet, so the content of this chapter is relatively small and relatively simple. Just have a general understanding.

The 12th chapter introduces the file system program in the fs/ directory of the kernel source code. When reading this chapter, we recommend that you pause for a while to read about the MINIX file system in Andrew S. Tanenbaum's book “Operating System Design and Implementation”. Chapters, because the original Linux system only supports MINIX file system, Linux 0.12 version is no exception.

Chapter 13 explains the memory management program in the mm/ directory. To thoroughly understand this aspect, we need to have a sufficient understanding of the protection mode operation mode of the Intel 80X86 microprocessor. Therefore, when reading this chapter of the program, you can refer to the overview of the operation mode of the 80X86 protection mode included in the appropriate place in this chapter. In addition to the description, you should also refer to Chapter 4 at the same time. Since this chapter explains the use of examples in the source code as objects, you can better understand how memory management works.

Existing Linux kernel analysis books generally lack the description of the kernel header file, so for a beginner, there are many obstacles to reading the kernel program. Chapter 14 of this book details all the header files in the include/ directory. Basically, each definition, each constant, or data structure is commented in detail. In order to facilitate reference during reading, this book also summarizes some important data structures and variables that are frequently used in the appendix, but these contents can actually be found in the header files of this chapter. Although the contents of this chapter are mainly used for reading the procedures in other chapters, if you want to thoroughly understand the kernel's operating mechanism, you still need to understand many of the details in these header files.

Chapter 15 describes all the files in the Linux 0.12 kernel source code lib/ directory. These library function files mainly provide interface functions to the system programs such as the compilation system, which will help the future understanding of the system software. Because of this lower version, there is not much here, so we can read it quickly. This is one of the reasons why we chose the 0.12 version.

Chapter 16 introduces the build.c program in the tools/ directory. This program is not included in the compiled and generated kernel image file. It is only used to connect the disk boot block in the kernel with other major kernel modules into a complete kernel image file.

Chapter 17 introduces the experimental environment for learning the kernel source code and the methods for hands-on experimentation. It mainly introduces the method of using and compiling Linux kernel under Bochs simulation system and the method of making disk image files. It also explains how to modify the syntax of the Linux 0.12 source code so that it can successfully compile the correct kernel under the RedHat 9 system.

The last is the appendix and index. The appendix gives some constant definitions and basic data structure definitions in the Linux kernel, as well as a concise description of the protection mode operating mechanism.

For ease of reference, the information on PC hardware used in the kernel is also listed separately in the appendix of this book. In the reference literature, we only provided books, articles, and other information that we can refer to when reading the source code. We did not provide all kinds of complicated and messy literature lists. For example, when referring to a file in the LDP (Linux Document Project) of the Linux Documentation Project, we will explicitly list which HOWTO article we need to refer to, and not just the LDP's website address.

When Linus first developed the Linux operating system kernel, he mainly referred to three books. One is "UNIX Operating System Design" by M. J. Bach, which describes the working principle and data structure of the UNIX System V kernel. Linus uses the algorithms for many of the functions in the book. The names of many important functions in the Linux kernel source code are taken from the book. Therefore, when reading this book, this is an essential reference book on the working principle of the kernel. The other is “Programming the 80386” edited by John H. Crawford et al. and is a good book explaining the 80x86 protected mode programming method. There is also a first edition of the book "MINIX Operating System Design and Implementation" by Andrew S. Tanenbaum. Linus mainly uses the MINIX file system version 1.0 described in this book, and also supports only this file system in the early Linux kernel, so when reading this chapter about the file system, the working principle of the file system It is fully available from Tanenbaum's book.

In the explanation of each program, we first briefly explain the main purpose and purpose of the program, input and output parameters and the relationship with other programs, and then list the complete code of the program and make detailed comments on the code, the original The program code or text is not altered or deleted in any way, because C language is a kind of English language. The original small amount of English comments in the program also provides a lot of useful information for constant symbols, variable names, and so on. Behind the code is a more in-depth anatomy of the program and a description of some of the language or hardware related knowledge that appears in the code. If you look back through the program after reading this information, you will have a deeper understanding.

The introduction of some basic concept knowledge needed to read this book is scattered in the corresponding parts of the various chapters. This is mainly for the convenience of finding, and when you combine the source code reading, you can have a deeper understanding of some basic concepts.

The last thing to note is that when you have fully understood everything explained in this book, it does not mean that you have become a Linux expert. You just embarked on the journey of Linux, with some initial knowledge of becoming a Linux kernel master. At this point you should read more source code, preferably incrementally from version 1.0 up to the latest odd-numbered version under development. The latest Linux kernel at the time of revision of this book is version 4.16.16. When you can quickly understand the latest versions of these developments and even come up with their own suggestions and patches, I'm willing to take a plunge.

1.3 Summary

This chapter first elaborated on the indispensable pillars of the birth and development of Linux: UNIX's initial open source version provided the basic principles and algorithms for Linux implementation, and Richard Stallman's GNU program provided a variety of free and practical utilities for Linux systems. The emergence of tools and POSIX standards provides Linux with reference guides for implementing standards-compliant systems. AST's MINIX operating system has served as an indispensable reference for the birth of Linux, and the Internet is a necessary environment for Linux to grow and grow. Finally, the chapter outlines the basic content of the book.

2 Microcomputer structure

Any system can be seen as a model consisting of four basic parts, as shown in Figure 2-1. The input part is used to receive information or data entering the system; after being processed by the processing center, the output part is sent out. The energy section provides the energy supply for the operation of the entire system, including the input and output part of the energy required for operation.

The composition of the computer system is no exception, it is also mainly composed of these four parts. Internally, however, the channels or interfaces between the processing center and the input/output portion of the computer system can be used in common, and therefore (b) in Figure 2-1 should more appropriately abstractly represent a computer system. Of course, for computers or many complex systems, each of them can be regarded as a complete subsystem independently and can also be described using this model, and a complete computer system is composed of these subsystems.

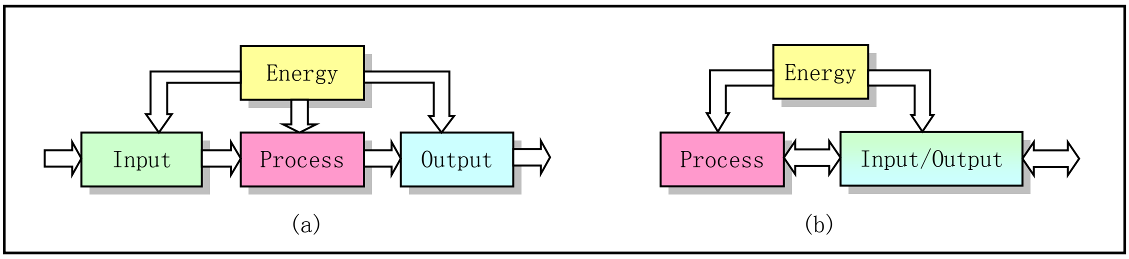


Figure 2-1 The basic composition of a system

Computer systems can be divided into hardware and software, but they are interdependent. The hardware part is the visible part of the computer system and is the platform for software operation and storage. Software is a stream of instructions that control hardware operations and actions. Just as information and thoughts stored in the human brain control the thoughts and actions of the human body, software can be seen as information and thinking in the "brain" of the computer. The theme described in this book is the operating mechanism of a computer system. It mainly explains the hardware composition principle of the processing center and the input/output part of the system and the realization of software control. On the hardware side, we outline the hardware system of an IBM PC microcomputer based on the Intel 80X86 CPU (Central Processing Unit) and its compatibles. The CPU chip of the computer can be regarded directly as the processing center of the system. The bus interface is connected with other parts; for the software running on it, we specifically describe the implementation of the Linux operating system kernel.

It can be seen that the operating system is closely related to the hardware environment being run. If you want to thoroughly understand the entire operating system, you need to understand its operating hardware environment. This chapter is based on the hardware block diagram of the traditional microcomputer system and introduces the functions of each major part of the microcomputer. These contents have basically established the hardware basis for reading the Linux 0.12 kernel. For ease of illustration, the term PC/AT will be used to refer to IBM PCs with 80386 or greater CPUs and their compatible microcomputers, while PCs are used generically to refer to all microcomputers, including IBM PC/XTs and their compatible microcomputers.

2.1 The Microcomputer Composition

From the perspective of overlooking, we illustrate the structure of a PC system with an 80386 or higher CPU. The structure of a conventional microcomputer hardware is shown in Figure 2-2. Among them, the CPU communicates with other parts of the system via a local bus (or internal bus) consisting of address lines, data lines, and control signal lines. The address line is used to provide the address of a memory or I/O device, which indicates the specific location where data needs to be read/written. Data lines are used to provide data transfer channels between the CPU and memory or I/O devices, while control lines are responsible for directing specific read/write operations. For PCs using the 80386 CPU, there are 32 internal address lines and data lines, respectively, which are all 32-bit. Therefore, the address space has 2^32 bytes, ranging from 0 to 4GB.

In the figure, the upper controller and memory interface are usually integrated on a computer motherboard. These controllers are each a functional circuit composed mainly of a large-scale integrated circuit chip. For example, the interrupt controller is composed of Intel 8259A or its compatible chips; the DMA controller is usually composed of Intel 8237A chips; the timing counter is at the core of the Intel 8253/8254 timing chip; and the keyboard controller is using Intel 8042 chip with the keyboard. The scanning circuit communicates.

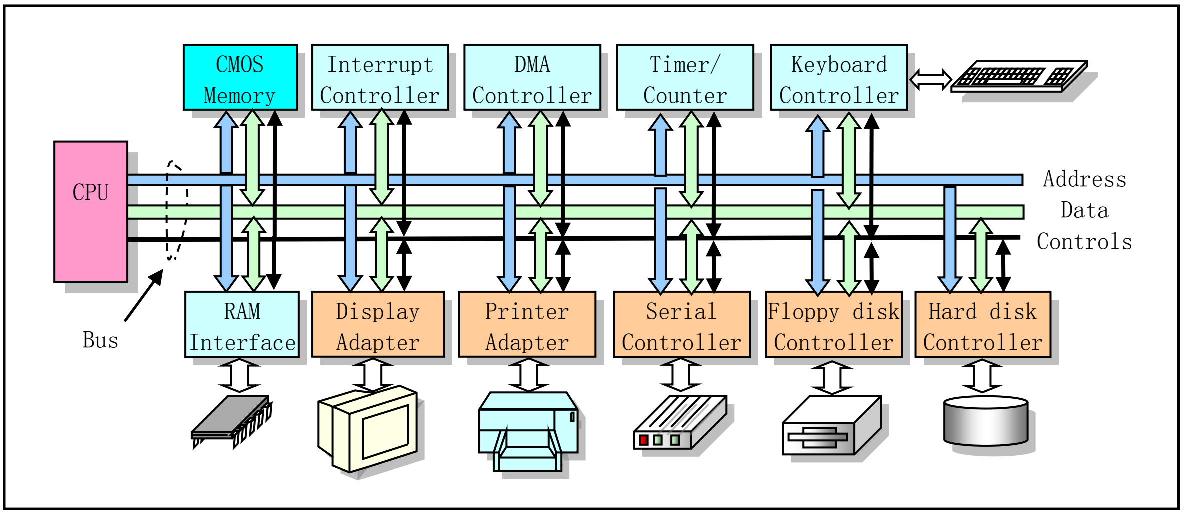


Figure 2-2 A block diagram of a traditional IBM PC and its compatibles

The control card (or adapter) in the lower part of the figure is connected to the system bus on the motherboard through an expansion slot. The bus slot is the standard connection interface to the extended device controller of the system address bus, data bus, and control line. These bus interface standards generally include an industry standard architecture ISA (Industry Standard Architecture) bus, an extended industrial standard architecture bus (EISA), a Peripheral Component Interconnect (PCI) bus, and an Accelerated Graphics Port (AGP) video. Bus and so on. The main difference between these bus interfaces is the data transfer rate and control flexibility. With the development of computer hardware, bus interfaces with higher transmission rates and more flexible control are still being introduced, such as the high-speed PCIE (PCI Express) bus using serial communication point-to-point technology. The original 80386 machine had only the ISA bus, so the system and external I/O devices can only use 16-bit data lines for data transfer.

With the development of computer technology, many functions (such as hard disk controller functions) that were originally implemented using control cards have been integrated in a few VLSI chips on a computer main board. Several even one such chip is The main features and functions of the main board are determined, and the bus structure has undergone great changes in order to allow the different parts of the system to reach their highest transmission rates. The composition of modern PCs can often be described using Figure 2-3. In addition to the CPU, modern PC motherboards mainly use two chipsets or chipsets composed of ultra-large-scale chips: Northbridge chips and Southbridge chips. The Northbridge chip is used to interface with the CPU, memory, and AGP video. These interfaces have very high transmission rates. The North Bridge chip also plays a role in memory control. Therefore, Intel labels the chip as an MCH (Memory Controller Hub) chip. The South Bridge chip is used to manage low- and medium-speed components such as PCI bus, IDE hard disk interface, USB port, etc. Therefore, the name of the South Bridge chip is ICH (I/O Controller Hub). The reason for using the “South and North” bridges to collectively refer to these two chips is that they are located on the typical PC motherboards published by Intel Corporation. They are located at the lower and upper ends of the main version (that is, on the south and north of the map), and plays the role of channel bridging with the CPU.

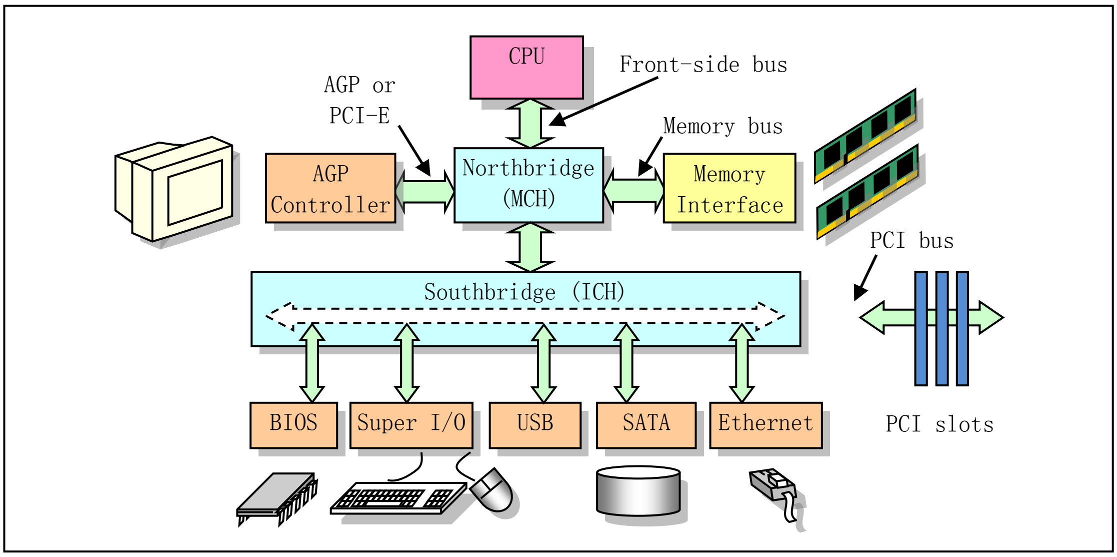


Figure 2-3 Modern PC chipset block diagram

Although the bus interface has undergone great changes, even the Northbridge and Southbridge will be combined in the future, but for our programmers, these changes are still compatible with the traditional PC architecture. Therefore, the program for the traditional PC hardware structure can still run on the current PC. This can be confirmed by Intel's development manual. Therefore, in order to facilitate the entry learning, we still discuss and study the composition and programming methods of the PC in the framework of the traditional PC architecture. Of course, these methods are still suitable for the modern PC architecture. Below we outline the working principle of each of the main controllers and control cards in Figure 2-2, and their actual programming methods are postponed until the corresponding kernel source code is read.

2.2 I/O Port addressing & access control

Before starting to transfer data between the CPU and the I/O adapter, it is necessary to first determine the I/O position of the communication adapter, that is, the port address. In the transmission of data between the CPU and the I/O interface, there may be a variety of transmission control modes. Generally, program loop query, interrupt processing, and DMA transfer may be used.

2.2.1 I/O ports and addressing

To access the data and status information on the I/O interface controller or control card, the CPU needs to specify their addresses first. This type of address is called an I/O port address or simply port. Usually an I/O controller includes a data port for accessing data, a command port for outputting commands, and a status port for accessing controller execution status. There are two ways to set the port address: unified addressing and independent addressing.

The principle of unified addressing of ports is to put the port address in the I/O controller into the memory addressing address space. Therefore, this addressing method also becomes memory image addressing. The operation of the CPU to access a port is the same as the operation of accessing the memory, and the instruction for accessing the memory is also used. The method of port independent address is to treat the addressing space of I/O controller and control card as a separate address space, which is called I/O address space. Each port has an I/O address corresponding to it and uses special I/O instructions to access the port.

The IBM PC and its compatible microcomputers mainly use an independent addressing mode and use a separate I/O address space to address and access the registers in the control device. Traditional PCs using the ISA bus architecture have I/O address space ranging from 0x000 to 0x3FF with 1024 I/O port addresses available. The default port address range used by each controller and control card is shown in Table 2-1. The use and programming methods of these ports will be described in detail later when the relevant hardware is specifically involved.

In addition, the IBM PC also partially uses the unified addressing mode. For example, the address of the display memory on the CGA display card directly occupies the memory address space 0xB800 -- 0xBC00 range. Therefore, if you want to display a character on the screen, you can directly use a memory operation instruction to perform a write operation to this memory area.

Table 2-1 I/O port address assignment

Address range	Allocation description
0x000 -- 0x01F	8237A DMA controller 1
0x020 -- 0x03F	8259A Programmable Interrupt Controller 1
0x040 -- 0x05F	8253/8254A Timer Counter
0x060 -- 0x06F	8042 Keyboard Controller
0x070 -- 0x07F	Access CMOS RAM/Real-Time Clock RTC Port
0x080 -- 0x09F	DMA page register access port
0x0A0 -- 0x0BF	8259A Programmable Interrupt Controller 2
0x0C0 -- 0x0DF	8237A DMA Controller 2
0x0F0 -- 0x0FF	Coprocessor access port
0x170 -- 0x177	IDE hard disk controller 1
0x1F0 -- 0x1F7	IDE hard disk controller 0
0x278 -- 0x27F	Parallel printer port 2
0x2F8 -- 0x2FF	Serial Controller 2
0x378 -- 0x37F	Parallel printer port 1
0x3B0 -- 0x3BF	Monochrome MDA display controller
0x3C0 -- 0x3CF	Color CGA display controller
0x3D0 -- 0x3DF	Color EGA/VGA display controller

0x3F0 -- 0x3F7	Floppy drive controller
0x3F8 -- 0x3FF	Serial Controller 1

For modern PCs using bus architectures such as EISA or PCI, 64 KB of I/O address space is available. The range of I/O addresses used by related controllers or settings can be obtained by looking at the /proc/ioports file under normal Linux systems. See the following:

```
[root@plinux root]# cat /proc/ioports
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0070-007f : rtc
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
02f8-02ff : serial(auto)
0376-0376 : ide1
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial(auto)
0500-051f : PCI device 8086:24d3 (Intel Corp.)
0cf8-0cff : PCI conf1
da00-daff : VIA Technologies, Inc. VT6102 [Rhine-II]
  da00-daff : via-rhine
e000-e01f : PCI device 8086:24d4 (Intel Corp.)
  e000-e01f : usb-uhci
e100-e11f : PCI device 8086:24d7 (Intel Corp.)
  e100-e11f : usb-uhci
e200-e21f : PCI device 8086:24de (Intel Corp.)
  e200-e21f : usb-uhci
e300-e31f : PCI device 8086:24d2 (Intel Corp.)
  e300-e31f : usb-uhci
f000-f00f : PCI device 8086:24db (Intel Corp.)
  f000-f007 : ide0
  f008-f00f : ide1
[root@plinux root]#
```

2.2.2 Interface access control

The PC I/O interface data transmission control mode can generally adopt program loop inquiry mode, interrupt processing mode and DMA transmission mode. As the name suggests, the cycle inquiry mode means that the CPU judges whether it can exchange data with the device by looping through the program to query the status in the specified device controller. This approach does not require excessive hardware support, and is relatively simple to use and program, but it consumes valuable CPU time. Therefore, this method should not be used in multitasking operating systems unless the waiting time is extremely short or necessary. In the Linux operating system, this method is only used in a few places when the device or controller can immediately return information.

The interrupt handling control method needs the support of the interrupt controller. In this control mode, only when the I/O device requests a processing request from the CPU by interrupting, the CPU temporarily interrupts the currently executed program and executes the corresponding I/O interrupt processing service process. After executing the interrupt handling service process, the CPU will continue to execute the program that was just interrupted. When an I/O controller or device issues an interrupt request, the CPU addresses the entry address of the corresponding interrupt handling service process by using an interrupt vector table (or an interrupt descriptor table). Therefore, when using the interrupt control mode, it is necessary to first set the interrupt vector table and compile the corresponding interrupt processing service process. Most device I/O controls in the Linux operating system use interrupt handling.

The direct memory access (DMA) method is used for batch data transfer between the I/O device and the system memory. The entire operation process requires the use of a dedicated DMA controller without CPU intervention. Since there is no need for software intervention during the transmission, the operation is very efficient. In the Linux operating system, floppy disk drivers use interrupts and DMA methods to achieve data transfer.

2.3 Main memory, BIOS and CMOS memory

A typical PC often contain three types of memory, one is the main memory RAM (Random Access Memory) used to run programs and temporarily save data; the other is ROM (Read Only Memory) memory, stores the system boot diagnostics and Initialize the hardware program; the third is a small amount of CMOS memory used to store the computer's real-time clock information and system hardware configuration information.

2.3.1 Main memory

When the IBM PC was first introduced in 1981, the system only had 640 KB of RAM main memory (referred to as memory). Since the 8088/8086 CPU used has only 20 address lines, the memory address range is up to 1024KB (1MB). At the time of the popular DOS operating system, the 640K or 1MB memory capacity was basically sufficient for ordinary applications. With the rapid development of computer software and hardware technology, current computers are usually configured with 512 MB or more of physical memory capacity, and all use Intel 32-bit CPUs, that is, PC/AT computers. Therefore, the CPU's physical memory addressing range has been up to 4GB (by using the new features of the CPU, the system can even address 64GB of physical memory capacity). However, in order to be compatible with the original PC in software, the allocation of physical memory below the system 1MB still remains basically the same as the original PC, but the BIOS of the original system ROM has always been the highest in the memory that the CPU can address. At the end location, the original location of the BIOS will be used as the shadow area of the BIOS when the computer is initially initialized, ie the BIOS code will still be copied to this area. See Figure 2-4.

When the computer is powered on, physical memory is set to a contiguous area starting at address 0. All memory except the range of addresses 0xA0000 to 0xFFFFF (384K to 1M total 384K) and 0xFFFE0000 to 0xFFFFFFFF (the last 64K at 4G) can be used as system memory. These two specific ranges are used for I/O devices and BIOS programs. If we have 16MB of physical memory in our computer, 0-640K will be used to hold kernel code and data on Linux 0.1x systems. The Linux kernel does not use BIOS functions nor does it use the interrupt vector table set by the BIOS. The 384K between 640K and 1M is still reserved for the use indicated in the figure. Among them, the 128K starting from address 0xA0000 is used as the display memory buffer, and then the part is used for the ROM BIOS of other control cards or its mapping area, and 0xF0000 to 1M range is used for the mapping area of the high-end system ROM BIOS. 1M-16M will be used by the kernel as an assignable main memory area. In addition, high-speed buffers and memory virtual disks also occupy a part of the memory area behind the kernel code and data. This area usually spans 640K to 1M.

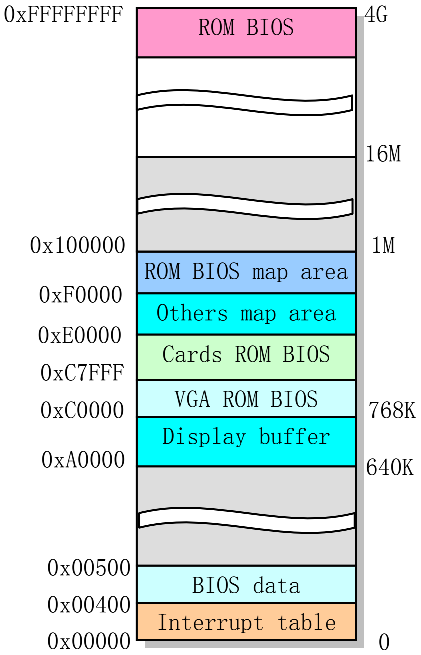


Figure 2-4 PC/AT machine memory area usage map

2.3.2 Basic input/output program BIOS

The system BIOS program stored in the ROM is mainly used to execute the self-check of various parts of the system when the computer is turned on, and various configuration tables that the operating system needs to use, such as an interrupt vector table and a hard disk parameter table, are established. It also initializes the processor and the rest of the system to a known state, and also provides hardware device interface services for operating systems such as DOS. However, since these services provided by the BIOS are not reentrancy (ie, the programs cannot be run concurrently), and considering the access efficiency, the Linux operating system runs at the same time, except that it uses the BIOS to provide some system parameters during initialization. Do not use the features in the BIOS.

When the computer system is powered on or a reset button on the chassis is pressed, the CPU automatically sets the code segment register CS to 0xF000, its segment base address is set to 0xFFFF0000, and the segment length is set to 64 KB. The IP is set to 0xFFF0, so the CPU code pointer now points to 0xFFFFFFF0, which is the last 16 bytes of the last 64K in 4G space. From the above figure, this is where the system ROM BIOS is stored. And the BIOS will store here a jump instruction JMP to jump to an instruction in the 64KB range in the BIOS code to start execution. Since the BIOS capacity of PC/AT microcomputers is mostly 1MB to 2MB, and is stored in the Flash Memory ROM, it is far from the 0--1M address space in order to be able to execute or access the BIOS in more than 64 KB range. Other BIOS code or data, the BIOS program will first use 32-bit access to set the data segment register access range to 4G (rather than the original 64K), so that the CPU can execute and manipulate data in the range of 0 to 4G. After that, after the BIOS performs some column hardware detection and initialization operations, it will copy the 64 KB BIOS code and data compatible with the original PC to the 64K at the low end of the 1M memory, and then jump to this place and let the CPU be real. Run in real address mode, as shown in Figure 2-5. Finally, the BIOS will load the operating system boot program from the hard disk or other block device into memory at 0x7c00 and jump to this location to continue the boot process.

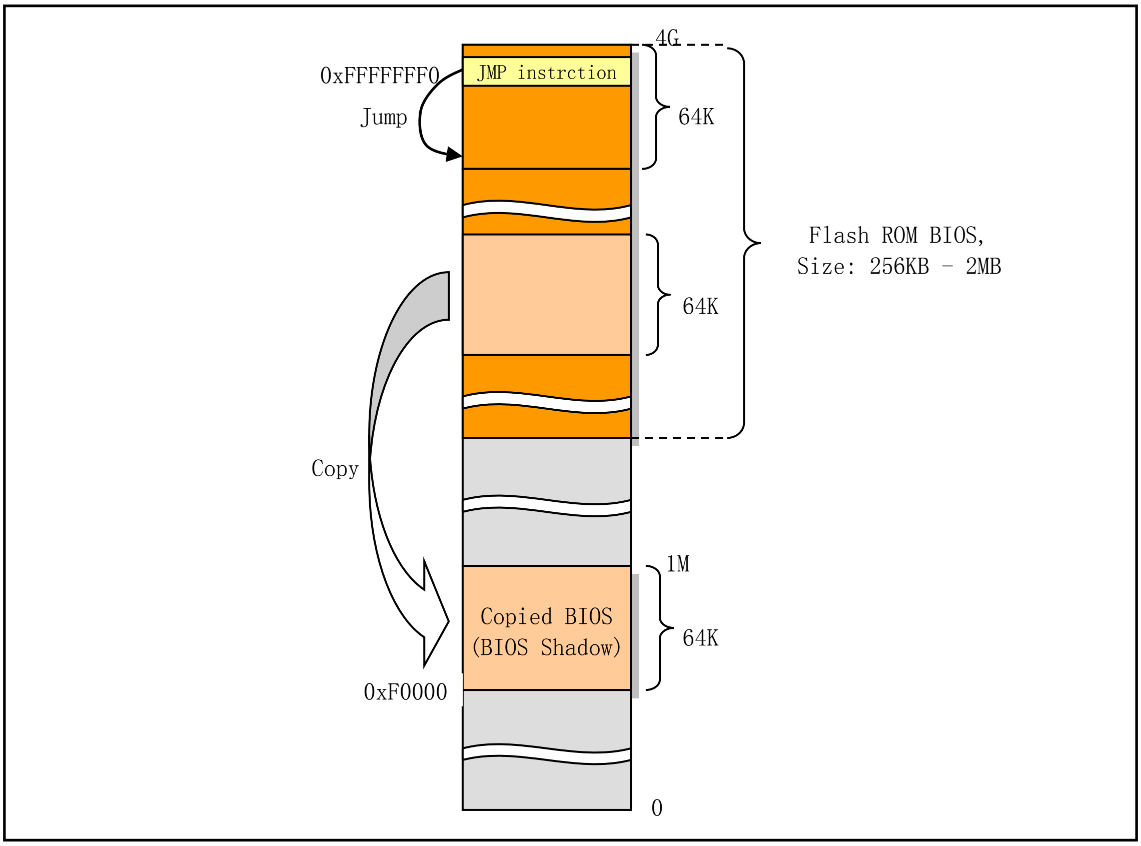


Figure 2-5 Flash ROM BIOS location and copy mapping area

2.3.3 CMOS memory

In PC/AT machines, in addition to the need to use memory and ROM BIOS, CMOS memory with little storage capacity (64 or 128 bytes) is used to store the real-time clock information and system hardware configuration information of the computer. This part of the memory is usually in an integrated block with the Real Time Chip. The address space of the CMOS memory is outside the basic memory address space and needs to be accessed using I/O instructions.

2.4 Controllers and Control Cards

As can be seen from Figure 2-2, a PC contains a variety of control cards and controllers used to transfer data and control computer operations. These controllers and control cards mainly include interrupt controllers, DMA controllers, keyboard controllers, floppy/hard disk control cards, serial communication control cards, and display control cards. Here the term "controller" refers to a control component that is integrated on a computer motherboard, and "control card" refers to a control card component that is inserted into the computer through an expansion slot. Because the control device can exist in the form of an independent control card, or it can be integrated into the main board with the increase in the degree of computer integration, there is no substantial difference between the controller and the control card. Below we describe these control devices in detail.

2.4.1 Interrupt controller

The IBM PC/AT 80X86-compatible microcomputer uses two cascaded 8259A programmable interrupt control chips to form an interrupt controller for I/O device interrupt control data access, and can provide independent interrupts for 15 devices. The control function is shown in Figure 2-6. During the initial boot-up of the computer, the ROM BIOS initializes two 8259A chips and assigns 15 levels of interrupt priority to the clock timer, keyboard, serial port, print port, floppy disk controller, coprocessor, and hard disk. Use equipment or controllers. At the same time, an interrupt vector table is created in the 0x000-0xFFF area at the beginning of the memory, and these interrupt requests are mapped to the interrupt vector number starting from 0x08, as shown in Table 2-2.

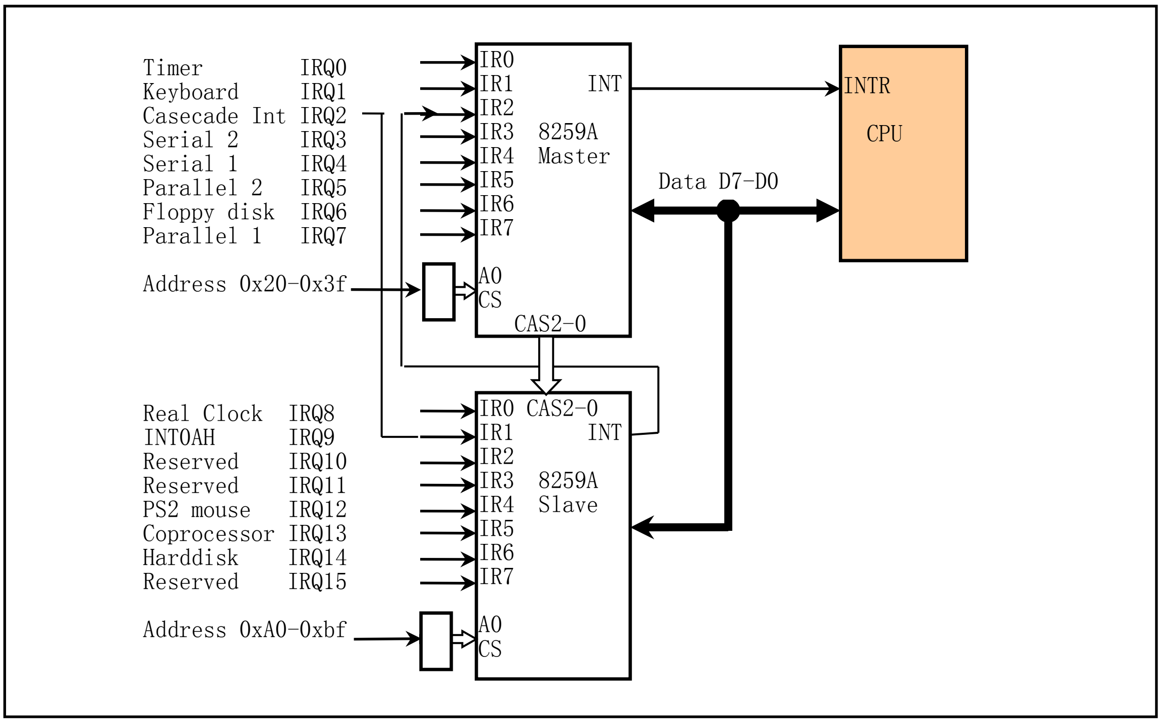


Figure 2-6 PC/AT microcomputer connected 8259 control system

However, since the interrupt number 0x00--0x1F belongs to Intel specifically reserved for the CPU, these BIOS settings conflict with Intel's requirements. To solve this problem, the Linux operating system does not directly use these interrupt numbers set by the BIOS. At power-on startup, the Linux operating system will re-set the 8259A during kernel initialization, mapping all system hardware interrupt request numbers to 0X20 and above interrupt numbers. See the subsequent sections for a detailed description of how the interrupt controller works and how to program it.

Table 2-2 Hardware request interrupt number set by ROM BIOS at power on

IRQ number	Interrupt number Set by the BIOS	Usage description
IRQ0	0x08 （8）	8250 issued 100HZ clock interrupt
IRQ1	0x09 （9）	Keyboard interrupt
IRQ2	0x0A （10）	The slave chip's interrupt

IRQ3	0x0B （11）	Serial port 2
IRQ4	0x0C （12）	Serial port 1
IRQ5	0x0D （13）	Parallel port 2
IRQ6	0x0E （14）	Floppy disk drive
IRQ7	0x0F （15）	Parallel port 1
IRQ8	0x70 （112）	Real-time clock interrupt
IRQ9	0x71 （113）	Change to INT 0x0A
IRQ10	0x72 （114）	Reserved
IRQ11	0x73 （115）	Reserved (network interface)
IRQ12	0x74 （116）	PS/2 mouse port interrupt
IRQ13	0x75 （117）	Math coprocessor interrupt
IRQ14	0x76 （118）	Hard disk controller interrupt
IRQ15	0x77 （119）	Reserved

2.4.2 DMA controller

As mentioned earlier, the main function of the DMA controller is to enhance the performance of the system by letting external devices transfer data directly to memory. Usually it is implemented by the Intel 8237 chip or its compatible chip on the machine. By programming the DMA controller, data transfer between peripherals and memory can be performed without CPU control. So during data transfer, the CPU can do other things.

In the PC/AT machine, two 8237 chips are used, so the DMA controller has 8 independent channels available. The last four of these are 16-bit channels. The floppy disk controller is specifically designated to use DMA channel 2. You must first set it before using a channel. This involves operations on three ports, the page register port, the (offset) address register port, and the data count register port. Since the DMA register is 8-bit, and the address and count value are 16-bit values, each needs to be sent twice.

2.4.3 Timer/counter

The Intel 8253/8254 is a Programmable Interval Timer (PIT) chip designed to handle precise time delays in computers. The chip provides three independent 16-bit counter channels. Each channel can work in different modes of operation, and these modes of operation can all be set using software. One way to perform a delay in software is to execute a loop operation statement, but doing so consumes CPU time. If the 8253/8254 chip is used in the machine, the programmer can configure the 8253 to meet its own requirements and use one of the counter channels to achieve the desired delay. After the time delay expires, 8253/8254 will send an interrupt signal to the CPU.

For the PC/AT and its compatible microcomputer system, the 8254 chip is used. The three timer/counter channels are used for time-of-day clocked interrupts, dynamic memory DRAM refresh timing circuits, and host speaker tone synthesis. The Linux 0.12 operating system only resets channel 0 so that the counter operates in mode 3 and sends a signal every 10 milliseconds to generate an interrupt request signal (IRQ0). The interrupt request generated at this interval is the pulse of the Linux 0.12 kernel. It is used to periodically switch the currently executed task and count the amount of system resources (time) used by each task.

2.4.4 Keyboard controller

The keyboard we are using now is a PC/AT compatible keyboard from IBM in 1984. It is usually called an AT-PS/2 compatible keyboard and has 101 to 104 buttons. There is a processor (Intel 8048 or compatible chip) on the keyboard, which is called a keyboard encoder. It is used to scan and collect the status information (ie scan code) of all key presses and release, and sends it to the keyboard controller on the main board of the host computer. . When a key is pressed, the scan code sent by the keyboard is called Make code, or simply called the connect code; the scan code sent when a pressed key is released is called disconnected. Break code, or simply break code.

The host keyboard controller is specifically designed to decode the received keyboard scan code and send the decoded data to the operating system's keyboard data queue. Because the on and off codes of each key are different, the keyboard controller can determine which key the user is operating based on the scan code. The on and off codes of all the keys on the entire keyboard form a scan code set of the keyboard. According to the development of computers, there are currently three sets of scan codes available. They are:

- The first set of scan codes -- The original XT keyboard scan code set. The current keyboard has rarely sent such scan codes;
- The second set of scan codes -- The default scan code set used by modern keyboards, commonly referred to as the AT keyboard scan code set;
- The third set of scan codes -- PS/2 keyboard scan code set. The scan code set used by the original IBM launch of the PS/2 microcomputer has rarely been used.

The AT keyboard sends the second set of scan codes by default. In spite of this, the host keyboard controller will still convert all received second keyboard scan codes into the first scan code for compatibility with PC/XT software, as shown in Figure 2-7. Therefore, we usually only need to know the first set of scan codes when programming keyboard controllers. This is also the reason why only the XT keyboard scan code set is given when it comes to keyboard programming.

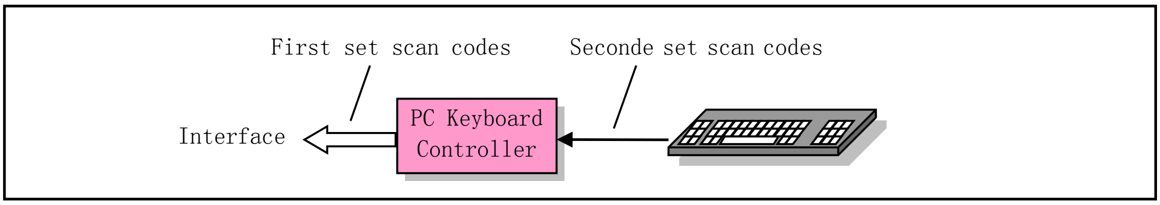


Figure 2-7 Keyboard controller conversion of scan code set

Keyboard controllers typically use Intel 8042 single-chip microprocessor chips or their compatible circuits. Today's PCs have integrated the keyboard controller in the motherboard chipset, but the functionality is still compatible with controllers that use the 8042 chip. The keyboard controller receives the 11-bit serial format data sent from the keyboard. The first bit is the start bit, the second 2-9 bits are the 8-bit keyboard scan code, the 10th bit is the parity check bit, and the 11th bit is the stop bit. See the description of the serial control card in the next section. After receiving the 11-bit serial data, the keyboard controller converts the keyboard scan code into a PC/XT standard keyboard-compatible system scan code, and then sends an interrupt request to the CPU through the IRQ1 pin of the interrupt controller. When the CPU responds to the interrupt request, the keyboard interrupt handler is invoked to read the XT keyboard scan code in the controller.

When a key is pressed, we can receive an XT keypad access code from the keyboard controller port. This scan code only indicates that the key at a location on the keyboard was pressed, but it has not yet been mapped to a character code. The connection code is usually one byte wide. For example, the key-on code for the key "A" is 30 (0x1E). When a pressed key is released, a break code is received from the keyboard controller port. For the XT keyboard (the scan code received by the keyboard controller programming port), the disconnection code is the connection code when its connection code plus 0x80, that is, the most significant bit (bit 7) is set. For example, the break code for the above "A" key is 0x80 + 0x1E = 0x9E.

But for those newly added ("extended") AT keyboard keys (such as the right Ctrl key and the right Alt key, etc.) for PC/XT standard 83-key keyboards, their on and off scan codes usually have 2 to 4 bytes, and the first byte must be 0xE0. For example, pressing the non-extended Ctrl key on the left produces a 1-byte passcode 0x1D, and pressing the Ctrl key on the right produces an extended 2-byte pass code 0xE0, 0x1D. Corresponding break codes are 0xE0, 0x9D. Table 2-3 shows several examples of turning on and off scan codes. In addition, the complete first set of scan codes is also given in the appendix.

Table 2-3 Example of the first scan code set received on the keyboard controller port

Pressed key	Connect scan code	Break scan code	Description
A	0x1E	0x9E	Non-expanding ordinary keys
9	0x0A	0x8A	Non-expanding ordinary keys
Function key F9	0x43	0xC3	Non-expanding ordinary keys
Arrow key right	0xe0, 0x4D	0xe0, 0xCD	Extended keys
Right Ctrl key	0xe0, 0x1D	0xe0, 0x9D	Extended keys
Left Shift + G	0x2A, 0x22	0xAA, 0xA2	Press and release Shift first

In addition, the output port P2 of the keyboard controller 8042 is used for other purposes. The P20 pin is used to implement the reset operation of the CPU, and the P21 pin is used to control the opening of the A20 signal line. When the output port bit 1 (P21) is 1, it turns on (gates) the A20 signal line, and 0 disables the A20 signal line. Today's motherboards no longer include a separate 8042 chip, but other integrated circuits on the motherboard will emulate the functionality of the 8042 chip for compatibility purposes. So now the keyboard programming is still using the 8042 programming method.

2.4.5 Serial control card

1. Asynchronous serial communication principle

Two computers/equipment exchange data, ie communication, must use the same language as people talk. In computer communication terminology, we refer to the "language" between a computer/device and a computer/device as a communication protocol. The communication protocol specifies the format for transmitting a unit of valid data length. Usually we use the term "frame" to describe this format. In order to allow the communication parties to determine the order of sending/sending and to perform some error detection operations, in addition to the necessary data, other information used for synchronization and error detection is also included in the transmitted one frame of information, for example, before the start of transmission of the data information. Send the start/synchronization or communication control information first, and then transmit some verification information after sending the required data information, as shown in Figure 2-8.

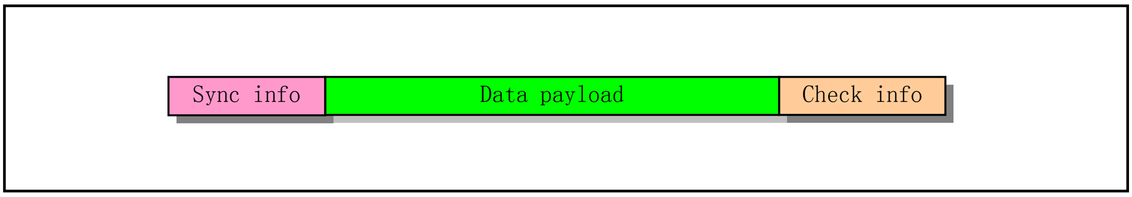


Figure 2-8 The general structure of communication frames

Serial communication refers to a communication method in which bit-bit data streams are transmitted one bit at a time on a line. Serial communication can be classified into asynchronous and synchronous serial communication. The main difference between them is the difference in the length of the communication units or frames that are synchronized during transmission. Asynchronous serial communication transmits a character as a communication unit or a frame, and synchronous serial communication transmits a sequence of a plurality of characters or bytes as one frame of data. If we use the dialogue between people as an analogy, then asynchronous communication is like the slow speed of conversation between the two parties. When speaking, a word is “worded out” and it can be paused for any length of time after each word is spoken. Synchronous communications, on the other hand, are like conversations between two parties in a consistent sentence. It can be seen that if we actually reduce the transmission unit to one bit (with letters!), then one-character asynchronous serial communication can also be regarded as a synchronous transmission of simultaneous transmission clock signals. way of communication.

2. Asynchronous serial transmission format

The frame format of asynchronous serial communication transmission is shown in Figure 2-9. Transmission of a character consists of a start bit, a data bit, a parity bit, and a stop bit. The start bit plays a role of synchronization and the value is always 0. The data bits are the actual data transmitted, ie a one-character code. Its length can be 5-8 bits. Parity bit is optional, set by the program. The stop bit is always 1, which can be set by the program to 1, 1.5, or 2 bits. Both parties must be set to the same format before communications begin sending information. If it has the same number of data bits and stop bits. In the asynchronous communication specification, the transmission 1 is called a MARK and the transmission 0 is called a SPACE. So we use these two terms in the following description.

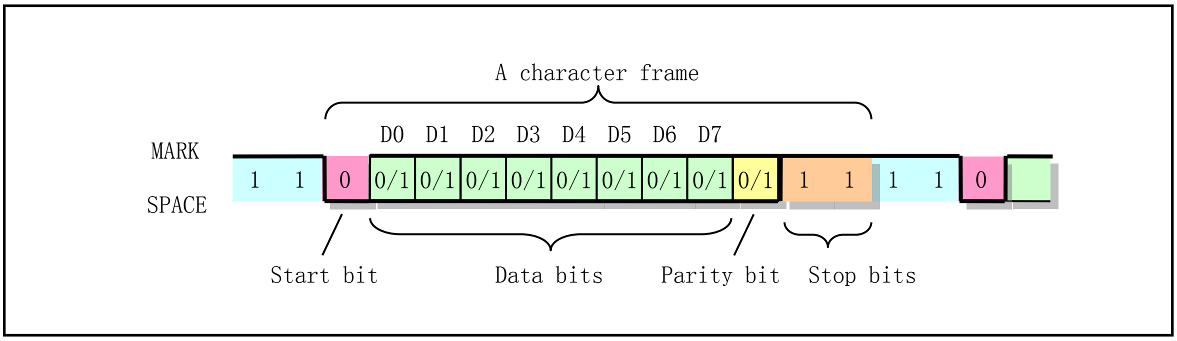


Figure 2-9 Asynchronous serial communication character transmission format

When there is no data transmission, the sender is in the MARK state, and sends 1 continuously. If it is necessary to send data, the sender needs to first send a space start bit of the bit interval. After receiving the space number, the receiver starts to synchronize with the sender and then receives the subsequent data. If the parity bit is set in the program, the parity bit needs to be received after the data transmission. Last is the stop bit. After a character frame is sent, the next character frame may be sent immediately, or the password may be temporarily sent, and the character frame may be sent after a moment.

When receiving a character frame, the receiver may detect one of three errors: (1) Parity error. At this point, the program should ask the other party to resend the character; (2) An overspeed error. This error occurs because the program fetches characters at a slower rate than the receiving speed. At this point you should modify the program to speed up the fetching of the character frequency; (3) The frame format is incorrect. This error occurs when the format information requested to be received is incorrect. For example, the empty number was received when the stop bit should be received. Usually this kind of error is caused by the difference in frame format between the two parties except the line interference.

3. Serial controller

To achieve serial communication, PCs are usually equipped with two RS-232C-compliant serial interfaces and are processed using a serial controller consisting of a Universal Asynchronous Receiver/Transmitter (UART). Send and receive serial data. The serial interface on the PC usually uses a 25-pin or 9-pin DB-25 or DB-9 connector, which is mainly used to connect MODEM devices to work. Therefore, the RS-232C standard specifies many MODEM dedicated interface pins.

Previous PCs all use National Semiconductor's NS8250 or NS16450 UART chips. Today's PCs use the 16650A and its compatible chips, but are compatible with the NS8250/16450 chips. The main difference between the NS8250/16450 and the 16650A is that the 16650A chip also supports FIFO transfer. In this mode, the UART can cause an interrupt only after it has received or transmitted a maximum of 16 characters, which can relieve the system and CPU. When the PC is powered on, the RESET signal passes through the MR pin of the NS8250 to reset the UART internal registers and control logic. If you want to use the UART afterwards, you need to perform initial programming operations to set the UART's operating baud rate, data bits, and operating mode.

2.4.6 Display control

IBM PC/AT and its compatible computers can use color and monochrome video cards. IBM's earliest PC video system standards include monochrome MDA and color CGA standards, as well as EGA and VGA standards. All of the later advanced graphics cards (including today's AGP graphics cards) have extremely high graphics processing speeds and smart acceleration processing capabilities, but they all support these initial standards. The Linux 0.1x operating system uses only the text display methods supported by these standards.

1. MDA display standard

The monochrome display adapter MDA (Monochrome Display Adapter) only supports black and white display. And only supports the unique text character display mode (BIOS display mode 7). Its screen display specifications are 80 columns x 25 lines (column number x = 0..79; line number y = 0..24), and a total of 2000 characters can be displayed. Each character also has 1 attribute byte, so it takes 4 KB to display one screen (one frame). The even address byte stores the character code, and the odd address byte stores the display attribute. The MDA card is configured with 8KB of display memory. 8 KB space (0xb0000 -- 0xb2000) starting from 0xb0000 is occupied in the memory address range of the PC. If the display screen number is video_num_lines = 25; the number of columns is video_num_colums = 80, then the position of the characters and attributes located at the screen column row values x, y in memory is:

```
    Character byte position = 0xb0000 + video_num_colums * 2 * y + x * 2;
    Attribute byte position = Character byte position + 1;
```

In the MDA monochrome text display mode, the attribute byte format of each character is shown in Table 2-4. Among them, D7 is set to 1 will cause the character to flash; D3 is set to 1 to highlight the character. It is basically the same as the attribute byte of the color text character in Figure 2-10, but with only two colors: white (0x111) and black (0x000). Their combined effect is shown in the table.

Table 2-4 Monochrome display character attribute byte settings

Background color D6D5D4	Foreground color D2D1D0	Attribute value No flash low	display effect	example
0 0 0	0 0 0	0x00	Characters are not visible.	
0 0 0	1 1 1	0x07	White characters displayed on a black background (normal display).	Normal
0 0 0	0 0 1	0x01	White underlined characters displayed on a black background.	Underline
1 1 1	0 0 0	0x70	Black characters displayed on a white background (inverse).	Reverse
1 1 1	1 1 1	0x77	Show white squares.	■

2. CGA display standard

The color graphics adapter CGA (Color Graphics Adapter) supports seven kinds of color and graphics display (BIOS display 0--6). In the 80 column X 25 column text character display mode, there are two monochrome and 16 color display modes (BIOS display mode 2--3). The CGA card comes standard with 16KB of display memory (occupying the memory address range 0xb8000 -- 0xbc000), so a total of 4 frames of display information can be stored therein. Similarly, in the 4KB display memory per frame, the even address byte stores the character code, and the odd address byte stores the character display attribute. However, only 8KB of display memory (0xb8000 -- 0xba000) is used in the console.c program. In the CGA color text display mode, the definition of the attribute byte format for each display character is shown in Figure 2-10.

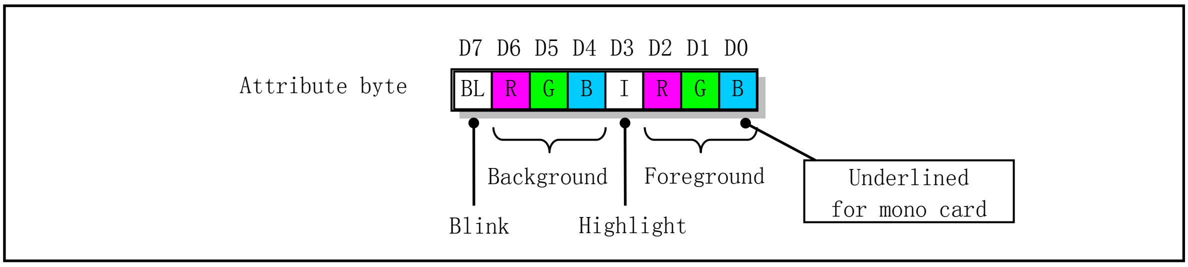


Figure 2-10 Character attribute format definition

As with the monochrome display, D7 is set to 1 to flash the display character; D3 is set to 1 to highlight the character; bits D6, D5, D4 and D2, D1, and D0 can be combined to create 8 colors. The combination of foreground and high-brightness bits can display the other 8 character colors. The color of these combinations is shown in Table 2-5.

Table 2-5 Foreground color and background color (left half)

I R G B	Value	Color name	I R G B	Value	Color name
0 0 0 0	0x00	Black	1 0 0 0	0x08	Dark grey
0 0 0 1	0x01	Blue	1 0 0 1	0x09	Light blue
0 0 1 0	0x02	Green	1 0 1 0	0x0a	Light green

0 0 1 1	0x03	Cyan	1 0 1 1	0x0b	Light cyan
0 1 0 0	0x04	Red	1 1 0 0	0x0c	Light red
0 1 0 1	0x05	Magenta	1 1 0 1	0x0d	Light magenta
0 1 1 0	0x05	Brown	1 1 1 0	0x0e	Yellow
0 1 1 1	0x07	Light grey	1 1 1 1	0x0f	White

3. EGA/VGA display standard

Enhanced Graphics Adapters (EGAs) and Video Graphics Adapters (VGAs) also support other display enhancements in graphics, in addition to or in addition to MDA and CGA support. In the MDA and CGA compatible display mode, the starting position and range of the occupied memory address are the same. However, EGA/VGA comes standard with at least 32KB of display memory. The physical memory address space starting from 0xa0000 is occupied graphically.

2.4.7 Floppy disk and hard disk controller

The floppy disk control subsystem of the PC consists of a floppy disk and a floppy disk drive. Because floppy disks can store programs and data and are easy to carry, floppy disk drives have long been one of the standard configuration devices on PCs. The hard disk also consists of a disk and a drive, but usually the hard disk's metal disk is fixed in the drive and cannot be removed. Because the hard disk has a large storage capacity, and read and write speed is very fast, it is the largest external storage device in the PC, usually also called external storage. Both floppy disks and hard disks use magnetic media to store information and have a similar storage operation. So here we use the hard disk as an example to briefly explain how they work.

The basic way to store data on a disc is to use a layer of magnetic media on the surface of the disc after magnetization. Floppy disks usually use polyester film as the substrate, while hard disk disks usually use metal aluminum alloy as the substrate. A floppy disk contains a polyester film disc. The upper and lower heads are used to read and write data on both sides of the disc. The disc rotation speed is about 300 rpm. For a floppy disk with a capacity of 1.44MB, the two sides of the disk are divided into 80 tracks, each track can store 18 sectors of data, so there are 2 X 80 X 18 = 2880 sectors. Table 2-6 shows the basic parameters of several common types of floppy disks.

Table 2-6 Common floppy disk basic parameters

Disk type and capacity	tracks/face	Sectors/tracks	Total sectors	Rotate speed (r/min)	Data transmission rate (Kbps)
5¼ inch 360KB	40	9	720	300	250
3½ inch 720KB	80	9	1440	360	250
5¼ inch 1.2MB	80	15	2400	360	500
3½ inch 1.44MB	80	18	2880	360	500
3½ inch 2.88MB	80	36	5760	360	1000

The hard disk usually includes at least two or more metal disks, and thus has more than two read/write heads. For example, there are four physical heads for a hard disk that contains two disks, and eight read and write heads for a disk that contains four disks. See Figure 2-11. The hard disk rotation speed is usually fast at 4500 rpm to 10,000 rpm, so the hard disk data transfer speed is usually up to several megabits/second.

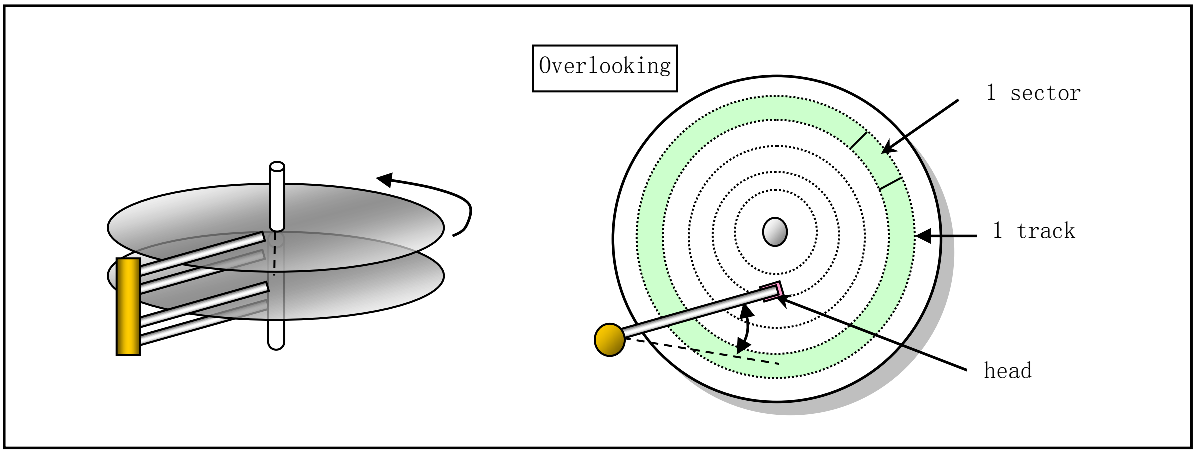


Figure 2-11 Typical hard disk internal structure with 2 disks

The magnetic head on the disk surface has a read coil and a write coil, respectively. During the read data operation, the head is first moved to a position on the rotating disk. Since the magnetic disk rotates, the magnetic medium moves at a uniform speed with respect to the magnetic head, so the magnetic head actually cuts magnetic lines of force on the magnetic medium. As a result, current is generated in the reading coil due to induction. Depending on the direction of the remanent state of the disk surface, the direction of the current induced in the coil is also different, so that 0 and 1 data recorded on the disk are read out, so that the bit stream can be sequentially read out from the disk. Since each track read by the head has a specific format for storing information, the disk circuit can discriminate and read the data in each sector on the track by recognizing the format in the read bit stream. See Figure 2- 12 shows. Among them, GAP is an interval field used for isolation. Usually GAP is 12 bytes of 0. The address field of each sector address field stores the cylinder number, head number (face number), and sector number of the relevant sector, so a sector can be uniquely determined by reading the address information in the address field.

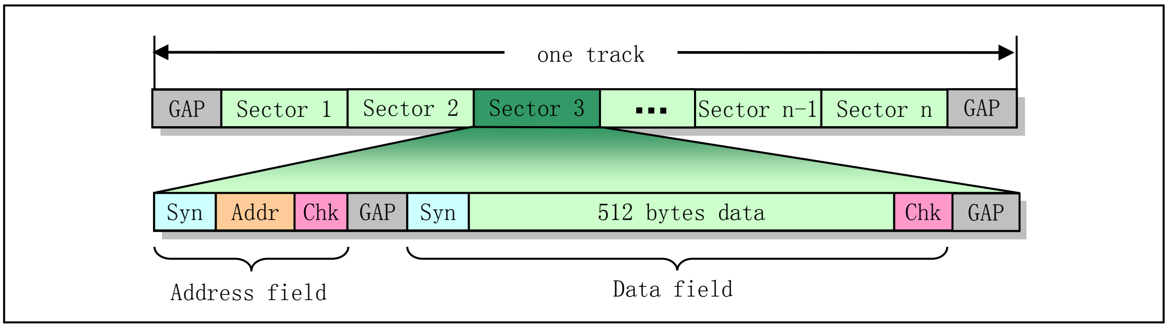


Figure 2-12 Disc track format

To read and write data on disks, you must use a disk controller. The disk controller is a logical interface circuit between the CPU and the driver. It receives request commands from the CPU, sends seeks, read/write, and control signals to the driver and controls and converts the data flow patterns. The data transferred between controller and driver includes the sector address information and timing and clock information in Figure 2-12. The controller must separate these address information and some encoding, decoding and other control information from the actual read/write data. In addition, the data transfer with the driver is a serial bit stream, so the controller needs to convert between parallel byte data and serial bit stream data.

The FDC (Floppy Disk Controller) in the PC/AT machine uses the NEC μPD765 or its compatible chip. It is mainly used to receive commands issued by the CPU and output various hardware control signals to the driver according to the command requirements, as shown in Figure 2-13. When performing a read/write operation, it needs to perform data conversion (string-parallel), encoding and verify operations, and constantly monitor the operating state of the drive.

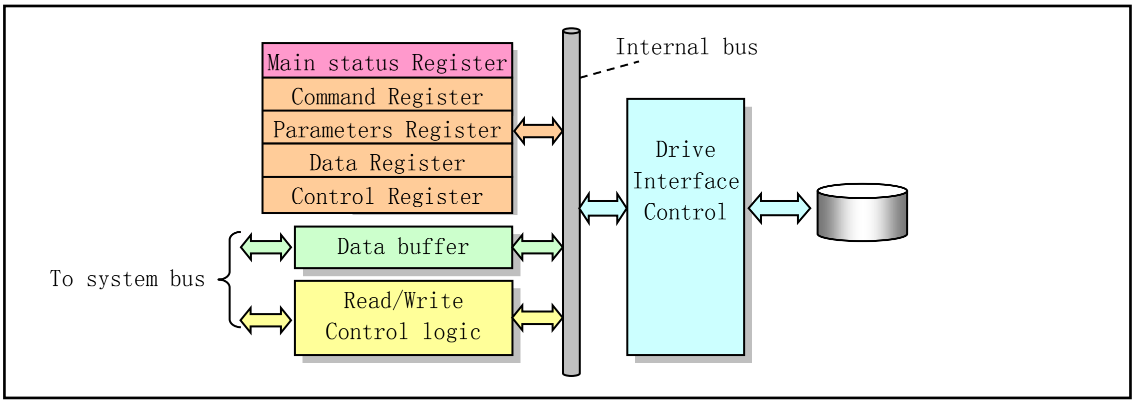


Figure 2-13 Disk controller internals

The programming process for the disk controller is to set the contents of the relevant registers in the controller through the I/O port and obtain the result information of the operation through the register. As for the transmission of sector data, the floppy disk controller is different from the PC/AT hard disk controller. The floppy disk controller circuit uses DMA signals and therefore requires the use of a DMA controller for data transfer. The AT hard disk controller uses high-speed data blocks for transmission without the intervention of a DMA controller. Because floppy disks are relatively vulnerable to damage (mould or scratches), floppy disk drives have not been deployed in computers at present. Instead, they use larger-capacity and more portable USB flash drives.

2.5 Summary

Hardware is the basic platform for operating systems. Understanding the hardware environment in which the operating system runs is a necessary condition for an in-depth understanding of the operating system on which it is running. This chapter briefly introduces each major part of the microcomputer based on the hardware composition of traditional microcomputers. In the next chapter, we describe the two assembly language syntaxes and related compilers used by the Linux kernel from a software perspective, and also introduce the contents of the GNU gcc syntax extensions used by the kernel.

3 Kernel programming language and environment

The language compilation process is the process of converting high-level languages that humans can understand into binary machine instructions that computer hardware can understand and execute. This conversion process usually generates some code that is not very efficient, so some of the code that requires high operating efficiency or has a large impact on performance is usually written directly in low-level assembly language, or the assembler generated by high-level language compilation. Then manually modify the optimization process. This chapter describes the programming language, object file format, and compilation environment used in the Linux 0.12 kernel. The main goal is to provide the assembly language and GNU C language extension knowledge needed to read the Linux 0.12 kernel source code. First of all, it introduces the syntax and usage of as86 and GNU as assembler in more detail. Then it uses C language extensions in kernel source code such as inline assembly, statement expressions, register variables, and inline functions in GNU C language. Introduced, and detailed description of the mutual calling mechanism between C and assembly functions. Because understanding the object file format is one of the most important prerequisites for understanding how the assembler works, when introducing two assembly languages, the basic format of the target file will be briefly described first, and Linux 0.12 will be given in more detail later in this chapter. The a.out object file format used in the system. Finally, the use of the Makefile is briefly described.

The content of this chapter is reference information when reading the Linux kernel source code. So you can take a brief look at the contents of this chapter, and then read the next section, and then refer back to this chapter when you encounter problems.

3.1 as86 Assembler

Two assemblers are used in Linux 0.1x systems. One is an as86 assembler that produces 16-bit code, using the companion ld86 linker; the other is the GNU assembler gas(as), which uses the GNU ld linker to link the resulting object files. Here we first describe how to use the as86 assembler, and the use of the assembler is described in the next section.

as86 and ld86 are Intel 8086, 80386 assembler compilers and linkers written by Bruce Evans, one of the main developers of MINIX-386. Linus already ported it to a Linux system when he first started developing the Linux kernel. Although it can compile 32-bit code for the 80386 processor, the Linux system uses it only to create the 16-bit boot boot sector program boot/bootsect.s and the binary setup code for the initial setup program boot/setup.s in real mode. The compiler is fast and compact, and has some features that GNU gas does not have, such as macros and more error detection methods. However, the compiler's syntax is incompatible with the syntax of the GNU as assembly compiler and more closely resembles the syntax of assemblers such as Microsoft's MASM, Borland's Turbo ASM, and NASM. These assemblers all use Intel's assembly language syntax (eg, the order of the operands is the opposite of GNU as, etc.).

The syntax of as86 is based on the assembly language syntax of the MINIX system, while the assembly syntax of the MINIX system is based on the assembler syntax of the PC/IX system. PC/IX was a UN*X operating system that was running on an Intel 8086 CPU long ago. Andrew S. Tanenbaum developed the MINIX system on a PC/IX system.

Bruce Evans is one of the major revision programmers for the 32-bit version of the MINIX operating system. He is a close friend of Linux founder Linus Torvalds. In the early days of Linux kernel development, Linus learned a lot from Bruce Evans about UNIX-like operating systems. The inadequacy of the MINIX operating system is also the result of two good friends discussing each other. These disadvantages of MINIX are just one of the main drivers that inspired Linus to develop a new concept operating system on the Intel 80386 architecture. Linus once said: "Bruce is my hero", so we can say that the birth of the Linux operating system and Bruce Evans also have a close relationship.

The source code for this compiler and linker can be downloaded from the FTP server ftp.funet.fi or from the website www.oldlinux.org. On modern Linux systems, RPM packages containing as86/ld86 can be installed directly, for example dev86-0.16.3-8.i386.rpm. Since the Linux system only uses as86 and ld86 to compile and link the two 16-bit assemblers bootsect.s and setup.s mentioned above, only the assembler syntax and assembly commands (assembler) used in these two programs are described here. The role and use of indicator).

3.1.1 as86 assembly language syntax

The assembler is designed to compile low-level assembly language programs into binary programs or object files containing machine code. The assembler will compile an input assembly language program (such as srcfile) into an object file (objfile). The basic format of the command line for assembly is:

```
    as [options] -o objfile srcfile
```

The options are used to control the compilation process to produce the target file with the specified format and settings. The input assembly language program srcfile is a text file. The contents of the file must consist of a series of lines of text that end with a newline character. Although GNU as can use semicolons to include multiple statements on a single line, it is common to include only one statement per line when programming assembly language programs.

Statements can be empty lines that contain only spaces, tabs, and line breaks, as well as assignment statements (or definition statements), pseudo-operator statements, and machine instruction statements. Assignment statements are used to assign a value to a symbol or identifier. It consists of an identifier followed by an equal sign followed by an expression, for example: "BOOTSEG = 0x07C0". Pseudo-operator statements are indicators used by the assembler and usually do not generate any code. It consists of pseudo-opcodes and zero or more operands. Each opcode begins with a dot character '.'. The dot character '.' itself is a special symbol that represents the position counter during compilation. The value is the address of the first byte of the machine instruction where the dot symbol appears.

The machine instruction statement is a mnemonic of an executable machine instruction and consists of an operation code and 0 or more operands. In addition, any statement can be preceded by a label. The label consists of an identifier followed by a colon ':'. During compilation, when the assembler encounters a label, the current position counter value is assigned to this label. Therefore, an assembly statement usually consists of three fields: label (optional), instruction mnemonic (instruction name), and operand. The label is located in the first field of an instruction. It represents the address of its location and usually indicates the destination of a jump instruction. Finally, you can also follow the comment section that starts with the comment.

The object file objfile generated by the assembler compilation usually contains at least three segments or sections, namely, a text segment (.text), a data segment (.data), and an uninitialized data segment (.bss). A text segment (or a code segment) is an initialized segment that usually contains program execution code and read-only data. The data segment is also an initialized segment that contains read/write data. The uninitialized data segment is an uninitialized segment. Usually the output object file generated by the assembler will not reserve space for the segment, but the operating system will initialize the contents of the segment to 0 when the object file is linked into the execution program. During compilation, statements that generate code or data in assembly language programs generate code or data in one of these three segments. Compiled bytes are stored starting from the '.text' section. We can use segment control pseudo operators to change the written segment. The target file format will be described in detail in the section "Linux 0.12 Object File Format" below.

3.1.2 as86 assembly programs

Below we use a simple framework sample program boot.s to illustrate the structure of the as86 assembler and the syntax of the statements in the program, and then give a compilation link and run method. Finally, use the as86 and ld86 usage methods and compilation options. The sample program is shown below. This example is a framework program for bootsect.s that compiles and generates the boot sector code. In order to demonstrate the use of certain statements, deliberately added a meaningless line 20 statements.

```
 1 !
 2 ! boot.s --  bootsect.s framework program. Replace 1 character in the string msg1
 3 !            with code 0x07 and display it on the first line of the screen.
 4 .globl begtext, begdata, begbss, endtext, enddata, endbss   ! Global id used for ld86 links
 5 .text                                ! Text segment
 6 begtext:
 7 .data                                ! Data segment
 8 begdata:
 9 .bss                                 ! Uninitialized data segment
10 begbss:
11 .text                                ! Text segment
12 BOOTSEG = 0x07c0                     ! Original segment address for the loaded bootsect code.
13
14 entry start                          ! Inform the linker the program starts executing from here.
15 start:
16         jmpi    go,BOOTSEG           ! Jump between segments. INITSEG indicates the jump
                                        ! section address, the label go is the offset address.
17 go:     mov     ax,cs                ! The value of the segment register cs -->ax is used
18         mov     ds,ax                ! to initialize the data segment registers ds and es.
19         mov     es,ax
20         mov     [msg1+17],ah         ! 0x07-> Replaces 1 dot in the string and beep once.
21         mov     cx,#20               ! 20 chars displayed, including cr & lf.
22         mov     dx,#0x1004           ! String displayed on screen at line 17, column 5.
23         mov     bx,#0x000c           ! Character display attribute (red).
24         mov     bp,#msg1             ! Point to a string (required by interrupt call).
25         mov     ax,#0x1301           ! Write string and move cursor to the end of the string.
26         int     0x10                 ! The BIOS interrupt call 0x10, function 0x13, subfunc 01.
27 loop1:  jmp     loop1                ! Dead cycle.
28 msg1:   .ascii "Loading system ..."! Message to be displayed, total of 20 ASCII characters.
29         .byte 13,10
30 .org 510                             ! Indicates statement is stored from address 510 (0x1FE).
31         .word 0xAA55                 ! Active boot sector flag, used by the BIOS.
32 .text
33 endtext:
34 .data
35 enddata:
```

```
36 .bss
37 endbss:
```

We first introduce the function of the program, and then explain in detail the role of each statement. This program is a simple boot sector program. Compile and link the generated executable program can be placed in the first sector of the floppy disk directly used to boot the computer. After starting, the red string "Loading system .." is displayed at line 17 and column 5 of the screen, and the cursor moves down one line. Then the program loops endlessly on code line 27.

The first three lines of the program are comment statements. In an as86 assembly language program, statements beginning with an exclamation mark '!' or a semicolon ';' are followed by comment text. The comment statement can be placed after any statement, or it can start with a new line.

".globl" on line 4 is an assembly directive (or assembly directive, pseudo-operator). Assembler indicators start with a single character '.' and do not generate any code at compile time. Assembler directives consist of a pseudo opcode followed by zero or more operands. For example, 'globl' on the 4th line is a pseudo-opcode, and the labels following it 'begtext, begdata, begbss' and so on are its operands. The label is an identifier followed by a colon, for example 'begtext:' on line 6. However, there is no need to take a colon when referring to a label.

Usually an assembler supports many different pseudo-operators, but the following only describes the commonly used as86 pseudo-operators used in the Linux system bootsect.s and setup.s assembly language programs.

The ’.globl’ pseudo-operator is used to define that subsequent label identifiers are external or global and are mandatory to introduce even if they are not used.

In addition to the three labels defined on lines 5 to 11, three pseudo operators are defined: '.text', '.data', and '.bbs'. They respectively correspond to the assembler program to generate 3 segments in the target file, namely the text segment, the data segment, and the uninitialized data segment. '.text' is used to identify the start position of the text segment and switch to the text segment; '.data' is used to identify the starting position of the data segment, and the current segment is switched to the data segment; and '.bbs' is used Identifies the beginning of an uninitialized data segment and changes the current segment to the bbs segment. So lines 5--11 are used to define a label in each segment, and then switch to the text segment to start writing the following code. Here, all three segments are defined in the same overlapping address range, so the sample program is not actually segmented.

Line 12 defines an assignment statement ″BOOTSEG = 0x07c0″. The equal sign ’=’ (or the symbol ’EQU’) is used to define the value represented by the identifier BOOTSEG, so this identifier can be referred to as a symbolic constant. This value, like the wording in C, can be used in decimal, octal, and hexadecimal.

The identifier 'entry' on the 14th line is a reserved key for forcing the linker ld86 to include in the generated executable file a label 'start' designated thereafter. Usually when linking multiple object files to generate an executable file, you should specify an entry label in the assembler with the keyword entry for debugging purposes. But in our example and in the Linux kernel boot/bootsect.s and boot/setup.s assembler we can omit this keyword because we don't want to include any symbol information in the generated pure binary executable file.

On line 16 is an inter-segment far jump statement, which jumps to the next instruction. When the BIOS loads the program into physical memory at 0x7c00 and jumps to it, the default value of all segment registers (including CS) is 0, that is, CS:IP=0x0000:0x7c00. Therefore, the inter-segment jump statement is used here to assign the segment value 0x7c0 to CS. After the statement is executed, CS:IP = 0x07C0:0x0005. The next two statements assign values to the DS and ES segment registers, respectively, so that they point to the 0x7c0 segment. This makes it easy to address data (strings) in the program.

The MOV instruction on line 20 is used to store the high byte (0x07) of the 0x7c0 segment value in the ah register to the last '.' position in the memory string msg1. This character will cause the BIOS interrupt to beep when the string is displayed. This statement is mainly used to illustrate the use of indirect operands. In as86, indirect operands require square bracket pairs. Some other addressing methods have the following:

```
! Direct register addressing. Jump to the address specified by bx, that is, copy bx to the IP.
    mov     bx,ax
    jmp     bx
! Indirect register addressing. The bx specifies the memory location as the address of the jump.
    mov     [bx],ax
    jmp     [bx]
! Put the immediate number 1234 into ax. Put the msg1 address value in ax.
    mov     ax,#1234
    mov     ax,#msg1
! Absolute addressing. Put the contents of the memory address 1234 (msg1) into ax.
    mov    ax,1234
    mov    ax,msg1
    mov    ax,[msg1]
! Index addressing. Put the value at the memory location indicated by the second operand into ax.
    mov    ax,msg1[bx]
    mov    ax,mgs1[bx*4+si]
```

The statements on lines 21-25 are used to put immediate data in the appropriate registers. The # must be preceded by an immediate number, otherwise it will be used as a memory address to make the statement an absolute addressing statement. See the example above. In addition, when putting the address value of a label (such as msg1) into a register, it must be preceded by a '#', otherwise it will become the register at the address of msg1!

Line 26 is the BIOS screen display interrupt call int 0x10. Its function 19, sub-function 1 is used here. The purpose of this interrupt is to write a string (msg1) to the screen at the specified location. The register cx is a string length value, dx is a display position value, bx is a display used character attribute, and es:bp points to a character string.

Line 27 is a jump statement that jumps to the current instruction. So this is an endless loop statement. The endless loop statement is used here to allow the displayed content to stay on the screen without being deleted. Dead-loop statements are commonly used when debugging assembler programs.

Lines 28-29 define the string msg1. Defining a string requires the use of the pseudo-operator '.ascii' and enclosing the string in double quotes. The pseudo operator '.asciiz' also automatically adds a NULL(0) character after the string. In addition, line 29 defines carriage return and line feed (13, 10) characters. Defining characters requires the use of the pseudo-operator '.byte' and enclosing characters in single quotes. For example: "'D'". Of course, we can write the ASCII code of characters directly as in the example.

The pseudo-operator statement '.org' on line 30 defines the location of the current assembling. This statement will adjust the position counter value of the current segment during compilation of the assembler to the value given on the pseudo-operator statement. For this example program, this statement sets the location counter to 510 and places a valid boot sector flag word 0xAA55 here (line 31). The pseudo-operator '.word' is used to define a double-byte memory object (variable) at the current location, which can be followed by a number or an expression. Since there is no code or data, we can determine from this that the executable compiled by boot.s should be exactly 512 bytes.

Lines 32--37 place three more labels in each of the three segments. Used to indicate the end position of three segments. This setting can be used to distinguish between the start and end of each segment in each module when linking multiple target modules. Because both the bootsec.s and setup.s programs in the kernel are separately compiled and linked, each expecting to generate a pure binary file does not link with other object module files. Therefore, the pseudo program for each segment is declared in the sample program. The characters (.text, .data, and .bss) can all be omitted. That is, the lines 4 - 11 and 32 - 37 of the program can all be deleted and the link can be compiled to produce the correct result.

3.1.3 as86 assembly language program compilation and link

Now we show how to compile the link sample program boot.s to generate the boot sector program we need to boot. Compiling and linking the above example program requires the following first two commands:

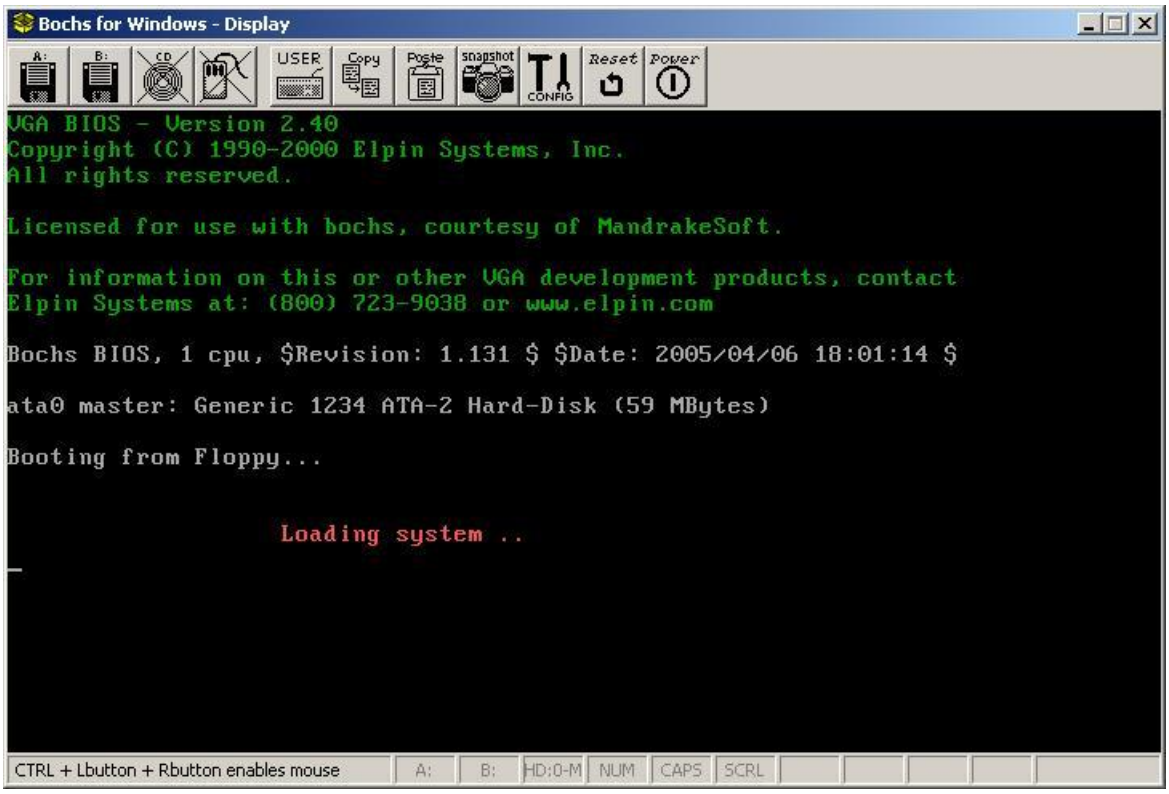
```
[/root]# as86 -0 -a -o boot.o boot.s                  // Compile. Generate the target file.
[/root]# ld86 -0 -s -o boot boot.o                    // link. Remove symbol information.
[/root]# ls -l boot*
-rwx--x--x   1 root     root          544 May 17 00:44 boot
-rw-------   1 root     root          249 May 17 00:43 boot.o
-rw-------   1 root     root          767 May 16 23:27 boot.s
[/root]# dd bs=32 if=boot of=/dev/fd0 skip=1          // Write to a floppy disk or Image file.
16+0 records in
16+0 records out
[/root]# _
```

Among them, the first one uses the as86 assembler to compile the boot.s program to generate the boot.o object file. The second command uses a linker ld86 to perform a link operation on the target file, and finally generates a MINIX structure executable file boot. The option '-0' is used to generate the 8086 16-bit target program; '-a' is used to specify that code that is compatible with the GNU as and ld parts is generated. The '-s' option is used to tell the linker to remove the symbol information from the last generated executable. '-o' specifies the name of the generated executable file.

As can be seen from the filenames listed above using the ls command, the last generated boot program is not exactly 512 bytes as stated earlier, but is 32 bytes long. These 32 bytes are the structure of the MINIX executable's header (see the "Creating a Kernel Component" chapter for a detailed structure description). In order to use this program to boot the machine, we need to manually remove the 32 bytes. There are several ways to remove this header structure:

- Use the binary editor to delete the first 32 bytes of the boot program and save it;
- Using the as86 compile linker on current Linux systems (eg RedHat 9), which have the option of generating a pure binary executable without the MINIX header structure, please refer to the online user manual (man as86) of the relevant system.
- Use the Linux system's dd command.

The third command listed above is to use the dd command to remove the first 32 bytes of the boot, and write the output directly to the floppy disk image file of the floppy disk or Bochs simulation system. (Please use the Bochs PC analog system. Refer to the last chapter). If we run this program in the Bochs simulation system, we can get the screen shown in Figure 3-1.


Bochs for Windows - Display
A:
B:
CD
USER
Copy
Paste
snapshot
CONFIG
Reset
Power
CTRL + Lbutton + Rbutton enables mouse
A:
B:
HD:0-M
NUM
CAPS
SCRL

```
VGA BIOS - Version 2.40
Copyright (C) 1990-2000 Elpin Systems, Inc.
All rights reserved.

Licensed for use with bochs, courtesy of MandrakeSoft.

For information on this or other VGA development products, contact
Elpin Systems at: (800) 723-9038 or www.elpin.com

Bochs BIOS, 1 cpu, $Revision: 1.131 $ $Date: 2005/04/06 18:01:14 $

ata0 master: Generic 1234 ATA-2 Hard-Disk (59 MBytes)

Booting from Floppy...


                  Loading system ..

_
```

Figure 3-1 Running the boot program in the Bochs simulation system

3.1.4 as86 and ld86 usage methods and options

The usage methods and options of as86 and ld86 are as follows:

```
Usage and options of as:
as [-03agjuw] [-b [bin]] [-lm [list]] [-n name] [-o objfile] [-s sym] srcfile

Default settings (Other than the defaults below, other options default to off or none; if you do not
specify the a flag, there will be no output):
-3    Use the 80386 32-bit output;
list  Display on standard output;
name  The basic name of the source file (that is, does not include the extension after '.');

The meaning of each option:
-0    Use 16-bit code segments;
-3    Use 32-bit code segments;
-a    Open some compatibility options with GNU as, ld;
-b    Generate binary files, followed by the file name;
-g    Only global symbols are stored in the object file;
-j    Make all jump statements long jumps;
-l    Generate a list file, followed by the list file name;
-m    Extend the macro definition in the list;
-n    Followed by the module name (in place of the source file name into the target file);
-o    Produce the target file, followed by the target file name (objfile);
-s    Produce a symbol file followed by a symbol file name;
-u    The undefined symbol as the symbol of the input unspecified segment;
-w    No warning message is displayed;
```

```
ld linker usage syntax and options:
For version of generating Minix a.out format:
ld [-03Mims[-]]  [-T textaddr] [-llib_extension] [-o outfile] infile...

For version of generating GNU-Minix a.out format:
ld [-03Mimrs[-]] [-T textaddr] [-llib_extension] [-o outfile] infile...

Default settings (except for the defaults below, other options are off or none by default):
-03       32-bit output;
outfile   a.out format output;

-0   Generate a header structure with 16-bit magic numbers, use i86 subdirectory for -lx option;
-3   Generate a header structure with a 32-bit magic number, use i386 subdirectory for -lx option;
-M   Display linked symbols on standard output devices
-T   Followed by the text base address (using the format suitable for strtoul);
-i   Separate instruction and data segment (I&D) output;
-lx  Add the library /local/lib/subdir/libx.a to the linked file list;
-m   Display linked modules on standard output devices
-o   Specify the output file name;
-r   Generate output suitable for further relocations;
-s   Remove all symbols in the target file.
```

3.2 GNU as assembler

The as86 assembler introduced in the previous section is only used to compile the boot/bootsect.s boot sector program in the kernel and the boot/setup.s setup program in real mode. All other assembly language programs in the kernel (including those generated by the C language) are compiled with gas and linked with the modules generated by the C language program. This section describes the use of the assembler syntax and the GNU as assembler (as assembler) in the Linux kernel based on the 80X86 CPU hardware platform. We first introduce the syntax of as assembly language programs, and then give the meaning and use of common assembly directives (indicators). As assembly language program examples with detailed instructions will be given at the end of the next chapter.

Because many key code requirements of the operating system require high execution speed and efficiency, about 10% of the key assembly language programs are usually included in an operating system source code. The Linux operating system is no exception. Its 32-bit initialization code, all interrupt and exception handling interface programs, and many macro definitions all use as assembly language programs or extended embedded assembly statements. Whether or not you can understand the functions of these assembly language programs will undoubtedly become one of the key points for understanding the concrete implementation of an operating system.

When compiling a C program, the GNU gcc compiler first outputs an as assembly language file as an intermediate result, and then gcc calls the as assembler to compile the temporary assembly language program into a target file. That is, as the assembler was originally designed to assemble the intermediate assembly language program generated by gcc, rather than being used as a standalone assembler. Therefore, as assembler also supports many C language features, including characters, numbers, and constant representation methods as well as expression forms.

The GNU as assembler was originally developed following the assembler of BSD 4.2. The current as assembler can be configured to generate many different formats of object files. Although the compiled assembly language program is not related to the target file that is used or generated in any format, if the target file format is involved in the following description, we will describe the a.out target file format adopted in the Linux 0.12 system. .

3.2.1 Compiling as assembly language program

The basic command line format for compiling an assembler program using the as assembler is as follows:

```
    as [ options ] [ -o objfile ] [ srcfile.s ...]
```

Where objfile is the target file name for the as compile output, and srcfile.s is the input assembly language program name for as. If you do not use the output file name, then as compiles the default destination file named a.out. After the as program name, the command line can include compilation options and file names. All options are free to place, but the result of compiling the filenames is closely related.

A program's source code can be placed in one or more files. How the program's source code is split across several files does not change the program's semantics. The source code of the program is the combined result of all these files in order. Every time you run the as compiler, it compiles only one source program. But a source program can consist of multiple text files (the terminal's standard input is also a file).

We can give zero or more input file names on the as command line. As will read the contents of these input files from left to right. If the parameters at any position on the command line do not have a specific meaning, they will be treated as an input file name. If no filename is given on the command line, as will attempt to read the input file content from the terminal or console standard input. In this case, if there is no content to input, you need to manually type Ctrl-D key combination to tell the as assembler. If you want to explicitly specify the standard input as an input file on the command line, you need to use the parameter '--'.

The output file of as is the binary data file compiled by the input assembly language program, ie the target file. Unless we specify the name of the output file with the option '-o', as will produce an output file named a.out. The target file is mainly used as an input file for the linker ld. The object file contains compiled program code, information that assists ld in producing an executable program, and possibly debugging symbol information. The a.out object file format used in the Linux 0.12 system will be described later in this chapter.

If we want to compile the boot/head.s assembler separately, we can type the following command on the command line:

```
[/usr/src/linux/boot]# as  -o head.o head.s
[/usr/src/linux/boot]# ls -l head*
-rw-rwxr-x   1 root     root        26449 May 19 22:04 head.o
-rw-rwxr-x   1 root     root         5938 Nov 18  1991 head.s
[/usr/src/linux/boot]#
```

3.2.2 as assembly syntax

In order to maintain compatibility with the gcc output assembler, the assembler uses the AT&T System V assembler syntax (hereinafter referred to as AT&T syntax). This syntax is very different from the syntax used by the Intel assembler (Intel syntax for short), and there are several major differences between them:

- In the AT&T syntax, an immediate value is preceded by a character '$'; the register operand name is preceded by the character percent sign '%'; absolute jump/invoke (relative to the program counter's jump/invoke) operands To add asterisk '*'. Intel assembly syntax does not have these limitations.

- The order of the source and destination operands used by AT&T syntax and Intel syntax is exactly the opposite. AT&T's source and destination operands are "source, destination" from left to right. For example Intel's statement 'add eax, 4' corresponds to AT&T's 'addl $4, %eax'.
- The length (width) of the memory operand in AT&T syntax is determined by the last character of the opcode. The operand suffixes 'b', 'w', and 'l' indicate that the memory reference width is 8 bits byte, 16-bit words, and 32-bit long words, respectively. Intel syntax achieves the same purpose by using the prefixes 'byte prt', 'word ptr', and 'dword ptr' before the memory operands. Therefore, Intel's statement 'mov al, byte ptr foo' corresponds to AT&T's statement 'movb $foo, %al'.
- In the AT&T syntax, immediate and far-form calls in the immediate form are 'ljmp/lcall $section, $offset', while Intel's is 'jmp/call far section:offset'. Similarly, in the AT&T syntax, the far return instruction 'lret $stack-adjust' corresponds to Intel's 'ret far stack-adjust'.
- The AT&T assembler does not provide support for multi-segment programs because UNIX-like operating systems require that all code be in one segment.

3.2.2.1 Assembler preprocessing

as assembler has a built-in simple preprocessing function for assembly language programs. This preprocessing function adjusts and removes extra space characters and tabs; removes all comment statements and replaces them with a single space or some newline character; converts character constants to their corresponding values. However, this preprocessing function does not process the macro definition nor handle the function of the include file. If this function is needed, then the assembly language program can use the uppercase suffix '.S' to make asc use the gcc CPP preprocessing function.

Since the as assembly language program uses C comment statements (that is, '/*' and '*/'), it also uses the hash symbol '#' as a single-line comment start character, so if the program is not preprocessed before assembly, then All indicators or commands included in the program that begin with the hash sign '#' are treated as part of the comment.

3.2.2.2 Symbols, Statements, and Constants

Symbols are identifiers composed of characters, and the valid characters that make up the symbol are taken from the uppercase and lowercase character sets, numbers, and the three characters '_.$'. Symbols are not allowed to start with numeric characters and the capitalization is different. There is no limit to the symbol length in as assembler, and all characters in the symbol are valid. Symbols use other characters (such as spaces, line breaks) or the beginning of the file to define the beginning and ending points.

The statement ends with a line break or a line break character (';'). The final statement of the file must end with a newline character.

If you use the backslash character '\' (before the newline) at the end of a line, you can use multiple lines for a statement. When as reads a backslash plus a newline, it ignores the two characters.

Statements start with zero or more labels, followed by a key symbol that determines the type of statement. The label consists of a symbol followed by a colon (':'). The key symbols determine the semantics of the rest of the statement. If the key symbol begins with a '.', the current statement is an assembly command (or directive, indicator). If the key symbol begins with a letter, the current statement is an assembly language instruction statement. So the general format of a statement is:

```
label:   .directive    followed by optional some comments
another_label:         # This is an empty statement.
         instruction   operand_1, operand_2, ...
```

A constant is a number that can be divided into character constants and numeric constants. Character constants can also be divided into strings and single characters; numeric constants can be divided into integers, large numbers, and floating-point numbers.

Strings must be enclosed in double quotes, and they can be escaped with special characters by using a backslash '\'. For example, '\\' indicates a backslash character. The first backslash is an escape indicator, which indicates that the second character is treated as a normal backslash character. The common escape sequences are shown in Table 3-1. If the backslash is followed by another character, the backslash will not work and the assembler will issue a warning message.

When using a single character constant in the assembler, you can write a single quotation mark before the character. For example, "'A' indicates the value 65, and"C" indicates the value 67. The escape codes in Table 3-1 can also be used for single character constants. For example, "\\" indicates a common backslash character constant.

Table 3-1 As assembler supports escaped character sequences

Escape code	Description
\b	Backspace, value is 0x08
\f	Formfeed, value 0x0C
\n	Newline, value 0x0A
\r	Carriage-Return value is 0x0D
\NNN	Character code represented by 3 octal numbers
\xNN...	Hexadecimal number character code
\\	Represents a backslash character
\"	Represents a double quote in a string ""

Integer numeric constants are represented in four ways, ie binary numbers starting with '0b' or '0B' ('0-1'); octal numbers starting with '0' ('0-7'); non-zero The decimal number starting with the digit ('0-9') and the hexadecimal number starting with '0x' or '0X' ('0-9a-fA-F'). To represent a negative number, just precede the negative '-'.

A Bignum is a number of bits more than 32 bits, which means that the method is the same as the integer. The representation of floating-point constants in assembler is basically the same as in C language. Since almost no floating point numbers are used in the kernel code, it is not described here.

3.2.3 Instruction statements, operands, and addressing

The instruction is the operation performed by the CPU. Usually the instruction is also called the opcode. The operand is the object of the instruction operation. The address is the position of the specified data in the memory. An instruction statement is a statement executed at the execution time of a program. It can usually consist of four components:

- Label (optional);
- Opcode (instruction mnemonic);
- Operands (specified by specific instructions);
- Comments

An instruction statement can contain zero or up to three comma separated operands. For an instruction statement with two operands, the first is the source operand and the second is the destination operand, ie the result of the instruction operation is stored in the second operand.

The operand can be an immediate value (that is, an expression whose value is a constant value), a register (value in the CPU's register), or memory (value in memory). An indirect operand (Indirect operand) contains the address value of the actual operand value. The AT&T syntax specifies an indirect operand by prepending the operand with a '*' character. Indirect operands can only be used by redirection/call instructions. See description of jump instructions below.

- A '$' character prefix is required before immediate operands;
- A '%' character prefix needs to be preceded by a register name;
- The memory operand is specified by a variable name or a register containing the address of the variable. The variable name implicitly indicates the address of the variable and instructs the CPU to reference the contents of the memory at that address.

3.2.3.1 Name the instruction opcode

The last character of the instruction opcode name (ie, the instruction mnemonic) in AT&T syntax is used to indicate the width of the operand. The characters 'b', 'w', and 'l' specify byte, word, and long operands, respectively. If the instruction name does not have such a character suffix and the instruction statement does not contain a memory operand, as will try to determine the operand width based on the destination register operand. For example, the instruction statement 'mov %ax, %bx' is equivalent to 'movw %ax, %bx'. Similarly, the statement 'mov $1, %bx' is equivalent to 'movw $1, %bx'.

The names of almost all instruction opcodes in AT&T and Intel syntax are the same, but there are still a few exceptions. Both symbolic extensions and zero-extend instructions require two widths to indicate that the width needs to be specified for the source and destination operands. AT&T syntax is done by using two opcode suffixes. The basic opcode names for symbol expansion and zero extension in AT&T syntax are 'movs...' and 'movz...', respectively, in Intel are 'movsx' and 'movzx' respectively. Two suffixes are attached to the basic name of the opcode. For example, an AT&T statement that uses symbolic extensions to move from %al to %edx is 'movsbl %al, %edx', which is bl from byte to long, bw from byte to word, and wl from word to long. The correspondence between AT&T syntax and conversion instructions in Intel syntax is shown in Table 3-2.

Table 3-2 Correspondence between conversion command in AT&T syntax and Intel syntax

AT&T	Intel	Description
cbtw	cbw	Extend the byte value in %al to %ax
cwtl	cwde	Extend the %ax sign to %eax
cwtd	cwd	Extend the %ax sign to %dx:%ax
cltd	cdq	Extend the %eax sign to %edx:%eax

3.2.3.2 Instruction opcode prefix

The opcode prefix is used to modify subsequent opcodes. They are used to repeat string instructions, provide area overrides, perform bus lock operations, or specify operands and address widths. Normally, the opcode prefix can be used as an exclusive line of instructions without an operand and must be located directly before the affected instruction, but it is best to place it on the same line as the instruction it modifies. For example, the string scan command 'scas' uses prefixes to perform repeated operations:

```
    repne scas %es:(%edi), %al
```

Some operand prefixes are listed in Table 3-3.

Table 3-3 Opcode prefix list

Opcode prefix	Description
cs, ds, ss, es, fs, gs	Section overrides the opcode prefix. Using the section:memory operands by specifying memory prefixes automatically adds this prefix.
data16, addr16	Operand/address width prefix. These two prefixes will change the 32-bit operand/address to a 16-bit operand/address. However, please note that as does not support 16-bit addressing.
lock	Bus latching prefix. Used to disable interrupts during instruction execution (only valid for some instructions, see the 80X86 manual).
wait	Coprocessor instruction prefix. Wait for the coprocessor to complete the execution of the current instruction. This prefix is not needed for the 80386/80387 combination.
rep, repe, repne	The prefix of the string instruction causes the string instruction to repeat the specified number of times in %ecx.

3.2.3.3 Memory reference

```
    Indirect memory reference form of Intel syntax: section:[base + index*scale + disp]
    Corresponds to the following AT&T syntax: section:disp(base, index, scale)
```

Base and index are optional 32-bit base registers and index registers, and disp is an optional offset value. Scale is a scale factor and its range is 1, 2, 4 and 8. Scale is multiplied by index to calculate operand address. If no scale is specified, the scale defaults to 1. Section specifies an optional segment register for the memory operand and overrides the current default segment register used by the operand. Note that if the specified section overwrite register is the same as the default operation section register, as does not output the same section prefix for the assembled instructions. The following are examples of memory references in several AT&T and Intel syntax forms:

```
movl var, %eax                   # Put the contents at memory address var in the register %eax.
movl %cs:var, %eax               # Put the contents at var in the code segment into %eax.
movb $0x0a,%es:(%ebx)            # Save byte value 0x0a to offset specified by %ebx in es segment.
movl $var, %eax                  # Put the address of var in %eax.
movl array(%esi), %eax           # Put contents at address determined by array+%esi into %eax.
movl (%ebx, %esi, 4), %eax       # Put contents at address determined by %ebx+%esi*4 in %eax.
movl array(%ebx, %esi, 4), %eax  # Put contents at address of array + %ebx+%esi*4 into %eax.
movl -4(%ebp), %eax              # Put contents at %ebp -4 in %eax, using the default segment %ss.
movl foo(,%eax,4), %eax          # Put contents at foo+eax*4 intp %eax, using default seg %ds.
```

3.2.3.4 Jump instruction

Jump instructions are used to move the execution point to another location in the program and continue execution. The destination of these jumps is usually represented by a label. When generating the object code file, the assembler will determine the address of all tagged instructions and encode the address of the jumped instruction into the jump instruction. Jump instructions can be divided into unconditional jumps and conditional jumps. The conditional jump instruction will depend on the state of a related flag in the flag register when the instruction is executed to determine whether to jump, and the unconditional jump does not depend on these flags.

JMP is an unconditional jump instruction and can be divided into two types: direct jump and indirect jump, whereas conditional jump instructions only have the form of a direct jump. For a direct jump instruction, the address of the jumped target instruction is directly encoded into the jump instruction as part of the jump instruction; for an indirect jump instruction, the jump destination is taken from a register or a Memory locations. The direct jump statement is written to give the label at the jump target; the indirect jump statement is written using a star character '*' as the prefix character of the operation indicator, and the operation indicator uses the same syntax as the movl instruction. . The following are some examples of direct and indirect jumps.

```
jmp NewLoc        # Jump directly. Unconditionally jump to label NewLoc to continue execution.
jmp *%eax         # Indirect jump. The value of register %eax is the jump destination.
jmp *(%eax)       # Indirect jump. Read the jump destination from the address indicated by %eax.
```

Similarly, indirect call operands that are independent of the instruction counter PC must also have a '*' as the prefix character. If the '*' character is not used, the as assembler will select the jump label associated with the instruction count PC. Also, any other instruction that has a memory operand must use an opcode suffix ('b', 'w', or 'l') to indicate the size of the operand (byte, word, or long).

3.2.4 Sections and Relocation

Sections (also called segments) are used to represent an address range, and the operating system will treat and process the data information in that address range in the same way. For example, there may be a "read only" area, and we can only read data from this area and cannot write it. The concept of a zone is mainly used to indicate different information areas in a target file (or executable program) generated by a compiler, such as a text area or a data area in a target file. To properly understand and compile an assembly language program, we need to understand the format of the output object file produced by as. A detailed description of the a.out format object file format used by the Linux 0.12 kernel is given later in this chapter. Here, a brief introduction to the basic concepts of the zone is provided to understand the basic structure of the object file produced by the assembler.

The linker ld will combine the contents of the input object file according to a certain rule to generate an executable program. When the as assembler outputs a target file, the code in the target file is set by default to start at address 0. After that, ld will allocate different final address locations for each part of the different target files during the linking process. Ld moves the block of bytes in the program to the address where the program was run. These blocks are moved as fixed units. Their length and byte order will not be changed. Such a fixed unit is called a zone (or segment, part). The operation of allocating the address of the runtime for a zone is referred to as a relocation operation, which includes adjusting the addresses recorded in the target file so that they correspond to the appropriate runtime address.

The as-assembler creates and outputs an object file with at least 3 fields, which are called the text, data, and bss areas. Each district may be empty. If you do not use the assembler command to place the output in the '.text' or '.data' zone, these zones will still exist but the content is empty. In a target file, its text area starts at address 0, followed by the data area, followed by the bss area.

When a section is relocated, in order for the linker ld to know what data will change and how to modify the data, the assembler will also write the required relocation information to the target file. In order to perform a relocation operation, ld must know each time an address in the target file is involved:

- Where did the reference to an address in the target file come from?
- What is the length of the quoted byte?
- Which section is referenced by this address? What is the value of (address)-(start address of section)?
- Is the reference to the address related to the program counter PC (Program-Counter)?

In fact, all the addresses used by as can be expressed as: (section) + (offset in the section). In addition, most of the expressions evaluated by as have such zone-related characteristics. In the following description, we use the notation "{secname N}" to indicate the offset N in the secname of the zone.

In addition to the text, data, and bss areas, we also need to understand the absolute address area (absolute area). When the linker combines the various object files, the address in the absolute area will always be the same. For example, ld will "relocate" the address {absolute 0} to address 0 at runtime. Although the linker will never arrange the data areas in the two target files as overlapping addresses after linking, the absolute area in the target file must overlap and be overwritten.

There is also an Undefined section. It is not possible to determine at assembly that any address in the area is set to {undefined U}, where U will be filled in later. Because the value is always defined, the only way to present an undefined address involves only undefined symbols. A reference to a common block is such a symbol: Its value is unknown at assembly time, so it is in the undefined area.

Similarly, the section name is also used to describe the group of sections in the linked program. The linker ld will put the text section in all object files of the program at the adjacent address. The text area of the program that we are accustomed to refer to actually refers to the entire address area formed by the combination of the text sections of all of its object files. The same is true for the understanding of the data and bss sections in the program.

3.2.4.1 Linker involved sections

The linker ld only involves the following 4 types of sections:

- Text section, data section -- These two areas are used to save programs. As and ld treat them independently and equally. The description of the text section is also suitable for the data section. However, when the program is running, the usual text section will not change. The text section is usually shared by the process and contains the instruction code and constants. The contents of the data section usually change when the program is running. For example, C variables are usually stored in the data section.
- bss section -- This area contains 0 bytes when the program starts running. This area is used to store uninitialized variables or as a common variable storage space. Although the length information of the bss section of each target file of the program is very important, since the area stores zero-value bytes, there is no need to save the bss section in the target file. The purpose of setting the bss area is to explicitly exclude zero-value bytes from the target file.
- Absolute section -- The address 0 of this area is always "relocated" to the address 0 of the runtime. Use this section if you do not want ld to change the address you are referencing when relocating. From this point of view, we can refer to absolute addresses as "non-relocatable": they do not change during relocation operations.
- undefined section -- A reference to an object that is not in each of the previously mentioned sections belongs to this section.

An example of 3 idealized relocatable sections is shown in Figure 3-2. This example uses the traditional section names: '.text' and '.data'. The horizontal axis indicates the memory address. The specific operation of the ld linker will be described in detail later in this section.

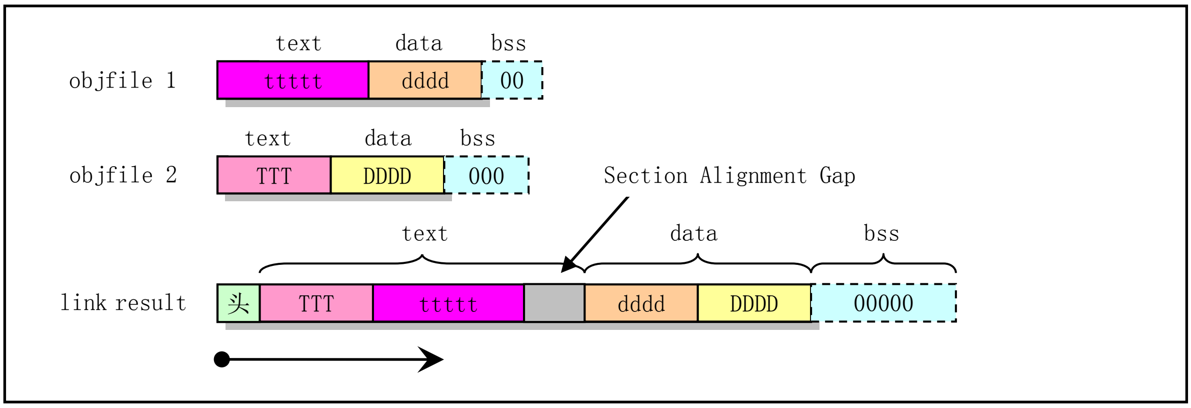


Figure 3-2 Example of linking two object files to generate a linked program

3.2.4.2 Subsection

The byte data that is assembled is usually located in the text or data section. Sometimes there may be some non-adjacent data groups in an area of an assembler source program, but you may want them to be stored together after assembly. The as assembler allows you to use subsections for this purpose. In each section, there may be a sub-area numbered 0--8192. Objects that are programmed in the same subsection are put together in the target file with other objects in that subsection. For example, the compiler may want to store constants in the text area, but do not want these constants to be scattered throughout the program being assembled. In this case, the compiler can use the '.text 0' subsection before each code area that is output, and use the '.text 1' subsection before each set of constants.

Using subsections is optional. If you do not use subsections, all objects are placed in subsection 0. Sub-sections appear in the destination file in the order of their numbers, but the destination file does not contain any information that represents sub-sections. The ld and other programs that process the target file do not see the traces of the subsections; they only see the text section consisting of all the text subsections; the data section consisting of all the data subsections. In order to specify which subsection area the subsequent statement is assembled into, you can use numeric parameters in '.text expression' or '.data expression'. The result of the expression should be an absolute value. If only '.text' is specified then '.text 0' is used by default. Similarly, '.data' indicates that '.data 0' is used.

Each section has a Location Counter that counts each byte that is assembled into the section. Because the subsections are only set up for ease of use by the assembler, there is no subsection counter. Although there is no direct way to manipulate a position counter, the assembly command '.align' can change its value, and any label definition will take the current value of the position counter. The location counter of the zone that is executing statement assembly processing is called the current activity counter.

3.2.4.3 bss section

The bss section is used to store local public variables. You can allocate space in the bss section, but you can't put data in it before the program runs. Because when the program starts executing, all bytes in the bss section will be cleared. The '.lcomm' assembly command is used to define a symbol in the bss section; '.comm' can be used to declare a public symbol in the bss section.

3.2.5 symbol

In the process of program compilation and linking, Symbol is an important concept. Programmers use symbols to name objects, linkers use symbols for link operations, and debuggers use symbols for debugging.

Label is a symbol followed by a colon. At this point the symbol represents the current value of the active position counter and can, for example, be used as the operand of the instruction. We can use the equal sign '=' to assign an arbitrary value to a symbol.

The symbol name starts with one of the letters or '._' characters. Local symbols are used to help compilers and programmers use names temporarily. A total of 10 local symbol names ('0'...'9') are available for reuse in a program. To define a local symbol, simply write a label of the form 'N:' (where N represents any number). If you refer to the previously defined symbol, you need to write 'Nb'; if you want to refer to the next defined local label, you need to write 'Nf'. Where 'b' means backwards and 'f' means forwards. There are no restrictions on the use of local labels, but at any time we can only refer to the furthest 10 local labels forward/backward.

3.2.5.1 Special point symbol

The special symbol '.' indicates as the current address of the assembly. So the expression 'mylab: .long .' will define mylab to contain its own address value. Assigning a value to '.' is the same as the assembly command '.org'. So the expression '.=.+4' is exactly the same as '.space 4'.

3.2.5.2 Symbol attributes

In addition to the names, each symbol has the "value" and "type" attributes. Depending on the format of the output, symbols can also have auxiliary attributes. If a symbol is used without definition, as will assume all its attributes are 0. This indicates that the symbol is an externally defined symbol.

The value of the symbol is usually 32 bits. For a symbol that marks a position in the text, data, bss, or absolute area, the value is the address value from the beginning of the area to the label. For the text, data, and bss areas, the value of a symbol will usually change during the linking process due to the change of the base address of the area, and the value of the symbol in the absolute area will not change. This is why they are called absolute symbols.

ld deals with the value of undefined symbols. If the value of the undefined symbol is 0, it means that the symbol is not defined in the assembler source program; ld will try to determine its value from other linked files. A symbol is generated when the program uses a symbol but does not define the symbol. If the value of an undefined symbol is not 0, then the symbol value represents the public memory space length that is required by the .comm public declaration. The symbol points to the first address of this memory space.

The type attribute of the symbol contains relocation information for the linker and the debugger, a flag indicating that the symbol is external, and some other optional information. For object files in a.out format, the symbol's type attribute is stored in an 8-bit field (n_type bytes). See the description of the include/a.out.h file for its meaning.

3.2.6 as assembler directives

Assembler directives are persudo instructions that indicate the way an assembler operates. Assembler directives are used to require the assembler to allocate space for variables, determine the program start address, specify the current assembly sections, modify the position counter value, and so on. All assembler directives begin with a '.', the rest are characters, and the case is irrelevant. However, lowercase characters are generally used. Below we give a description of some common assembler instructions.

3.2.6.1 .align abs-expr1, abs-expr2, abs-expr3

.align is a storage-align assembler directive that sets (increments) the position counter value to the next specified memory boundary in the current subsection. The first absolute value expression abs-expr1 (absolute expression) specifies the required boundary alignment value. For an 80X86 system that uses a.out format object files, the value of this expression is the number of zero-valued bits on the rightmost binary value of the position counter after it has been incremented, that is, a power of two. For example, '.align 3' means to increase the position counter value to a multiple of 8. If the position counter value itself is a multiple of 8, then there is no need to change it. But for 80X86 systems that use the ELF format, the expression value is directly the number of bytes required for it. For example, '.align 8' is to increase the position counter value to a multiple of 8.

The second expression gives the byte value to use for alignment and padding. This expression and its preceding comma can be omitted. If omitted, the padding byte value is 0. The third optional expression, abs-expr3, is used to indicate the maximum number of bytes allowed for padding to be skipped by an alignment operation. If the number of bytes skipped by the alignment operation is greater than this maximum value, the alignment operation is canceled. If you want to omit the second parameter, you can use two commas between the first and third parameters.

3.2.6.2 .ascii "string"...

Allocate space for the string from the current location of the location counter and store the string. Multiple strings can be written separately using commas. For example, '.ascii "Hellow world!", "My assembler"'. The assembler instruction will have as assemble these strings at consecutive address locations, with no 0 (NULL) bytes added after each string.

3.2.6.3 .asciz "string"...

This assembler directive is just like '.ascii', but each string is followed by a zero value byte. The "z" in '.asciz' stands for "zero".

3.2.6.4 .byte expressions

This directive expects zero or more expressions, separated by commas. Each expression is combined into the next byte.

3.2.6.5 .comm symbol, length

Declare a named public area in the bss section. During the ld link, a common symbol in one object file is merged with the common symbol with the same name in other object files. If ld does not find a symbol definition but only one or more common symbols, then ld will allocate uninitialized memory of length length bytes. Length must be an absolute value expression. If ld finds multiple common symbols with the same length but different names, ld allocates the space with the largest length.

3.2.6.6 .data subsection

This assembler directive tells as to assemble the following statements into the data subsection numbered subsection. If you omit the number, the number 0 is used by default. The number must be an absolute value expression.

3.2.6.7 .desc symbol, abs-expr

This directive sets the descriptor of the symbol to the low 16 bits of an absolute expression. It is only for a.out or b.out object format. See the description of the include/a.out.h file.

3.2.6.8 .fill repeat, size, value

This assembler directive will generate a repeat (repeat) of N bytes in size. The size value can be 0 or some value, but if size is greater than 8, it is limited to 8. The conetnts of each repeat bytes is taken from an 8-byte number. The highest order 4 bytes are zero, and the lowest 4 bytes are numeric values. The three parameter values are absolute values, size and value are optional. If the second comma and value are omitted, value defaults to 0; if the latter two parameters are omitted, size defaults to 1.

3.2.6.9 .global symbol (.globl symbol)

This assembler instruction will cause the linker ld to see the symbol. If symbol is defined in our object file, its value will be used by other object files in the link process. If the symbol is not defined in the object file, its attributes will be obtained from the symbol of the same name in other object files in the linking process. This is done by setting the external bit N_EXT in the symbol symbol type field. See the description in the include/a.out.h file.

3.2.6.10 .int expressions

The assembler directive sets 0 or more integer values in a certain area (80386 system is 4 bytes, same as .long). The value of each comma-separated expression is the value of the runtime. For example, .int 1234, 567,0x89AB.

3.2.6.11 .lcomm symbol, length

The local common area specified for the symbol reserves space of length bytes. The value of the area and symbol that is located is the value of the new local common block. The allocated address is in the bss section, so these byte values are cleared at runtime. Since the symbol symbol is not declared global, the linker ld is invisible.

3.2.6.12 .long expressions

Its meaning is the same as .int.

3.2.6.13 .octa bignums

This assembly dircetive specifies zero or more comma-separated 16-byte large numbers (.byte, .word, .long, .quad, .octa correspond to 1, 2, 4, 8 and 16 bytes, respectively).

3.2.6.14 .org new_lc, fill

This assembler directive sets the current section's location counter to the value new_lc. New_lc is an absolute value (expression), or an expression that has the same section as a subsection, ie it cannot use .org to span sections. If the section of new_lc is not correct, then .org will not work. Please note that the position counter is section-based, ie each section is used as a starting point for counting.

When the position counter value increases, the skipped bytes will be filled with the value fill. This value must be absolute. If you omit commas and fill, fill defaults to 0.

3.2.6.15 .quad bignums

This assembler directive specifies zero or more comma separated 8-byte large-number bignums. If the large number does not fit into 8 bytes, then take the lower 8 bytes.

3.2.6.16 .short expressions （same as .word expressions）

This assembler directive specifies zero or more comma separated 2-byte numbers in a section. For each expression, a 16-bit value is generated at runtime.

3.2.6.17 .space size, fill

The assembler directive generates size bytes, each of which is filled with fill. This parameter is an absolute value. If commas and fill are omitted, the default value of fill is 0.

3.2.6.18 .string "string"

Define one or more comma-separated strings. Escape characters can be used in strings. Each string is automatically appended with a null-terminated character. For example, .string "\n\nStarting", "other strings".

3.2.6.19 .text subsection

The notification as compiles the following statements into a subsection numbered subsection. If the number subsection is omitted, the default number value 0 is used.

3.2.6.20 .word expressions

For 32-bit machines, this assembly instruction has the same meaning as .short.

3.2.7 Writing 16-bit code

Although GNU as is usually used to write pure 32-bit 80X86 code, it also has limited support for writing code that runs in real mode or 16-bit protected mode after 1995. In order for as compile to generate 16-bit code, it is necessary to add the assembly instruction '.code16' before the instruction statement that is running in 16-bit mode, and use the assembly instruction '.code32' to switch the as-assembler back to 32-bit code assembly mode.

as does not distinguish between 16-bit and 32-bit assembler statements. Each instruction in 16-bit and 32-bit mode functions exactly the same way regardless of the mode. As always generates 32-bit instruction code for assembly statements regardless of whether the instruction will run in 16-bit or 32-bit mode. If the assembly instruction '.code16' is used to put as in 16-bit mode, then as will automatically add a necessary operand-width prefix to all instructions and let the instruction run in 16-bit mode. Note that since as adds extra address and operand-width prefixes for all instructions, the resulting code length and performance of the assembly will be affected.

as assembler did not support 16-bit code when developing the Linux kernel 0.12 in 1991, the as86 assembler described earlier was used when writing and assembling the boot startup code and initializing assembler in the 0.12 kernel real mode.

3.2.8 as assembler command line options

-a   Turn on program listings.

-f   Fast operation, skip whitespace and comment preprocessing.

-o objfile Name the object-file output from as *objfile*

-R   Fold the data section into the text section.

-W   Suppress warning messages.

3.3 C language program

GNU gcc has made some extensions to the C language described in ISO standard C89, some of which have been included in the ISO C99 standard. This section gives a description of some of the gcc extensions that are often used in the kernel. A brief explanation of the extended statement encountered will also be given at any time in the following section of program comments.

3.3.1 C program compiling and linking

The use of the gcc assembler to compile C programs usually goes through four stages of processing: the preprocessing stage, the compilation stage, the assembly stage, and the linking stage, as shown in Figure 3-3.

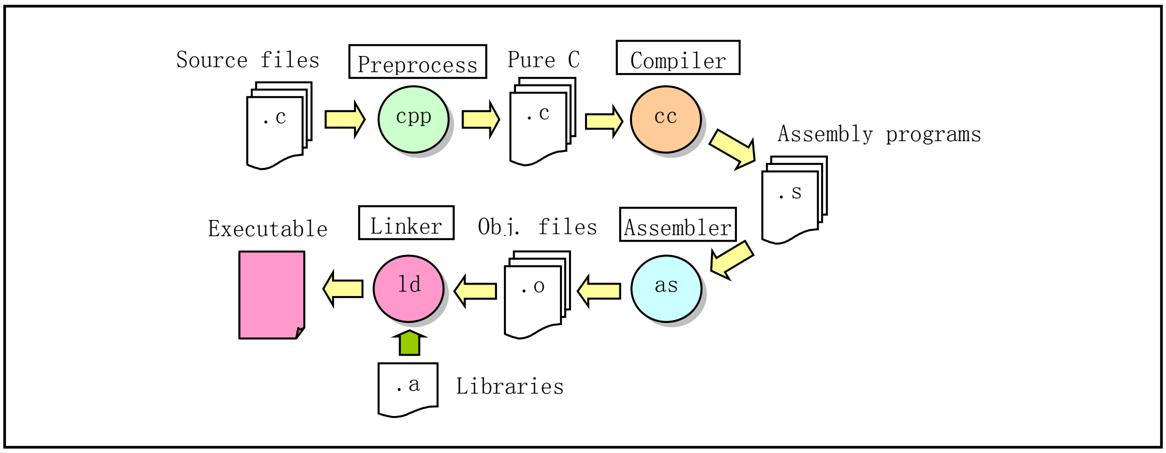


Figure 3-3 CC program compilation process

In the pre-processing stage, gcc passes the C program to the C preprocessor CPP, replaces the indicators and macros in the C language program, and outputs the plain C language code; at the compile stage, gcc compiles the C language program to generate the corresponding Machine-related assembly language code; Assembler stage, assembler will convert the assembly code into machine instructions, and output in a specific binary format and save it in the target file; Finally, the GNU ld linker links the program's related target file combination Together, the program's executable image file is generated. The command line format for calling gcc is similar to the format for compiling assembly language:

```
    gcc [ options ] [ -o outfile ] infile ...
```

Where infile is the input C language file; outfile is the compiled output file. For a compilation process, it is not necessary to execute all four stages. Using the command line option can cause the gcc compilation process to stop executing after a certain processing stage. For example, using the '-S' option allows gcc to stop after outputting the assembly language program for the C program; using the '-c' option allows gcc to only generate the target file without performing link processing, as shown below.

```
    gcc -o hello hello.c       // Compile the hello.c to generate the execution file hello.
    gcc -S -o hello.s hello.c  // Compile hello.c to generate corresponding assembly hello.s.
    gcc -c -o hello.o hello.c  // Compile hello.c to generate target file hello.o without linking.
```

When compiling a large program such as the Linux kernel containing many source program files, the make tool is usually used to automatically manage the entire program's compilation process. See the description below.

3.3.2 Inline assembly language

This section describes inline assembly statements that are exposed in the kernel C language program. Since we usually use inline assembly code rarely used in the preparation of C programs, it is necessary here to explain the basic format and usage. In an assembler instruction using asm, you can specify the operands of the instruction using C expressions. This means we need not guess which registers or memory locations will contain the data you want to use, and we must specify an assembler instruction template, as it appears in the machine description, and specify an operand constraint string for each operand.The basic format of an inline assembly statement with input and output parameters is as following:

```
    asm( “Assembly language statement”
        : Output register operands
        : Input register operands
        : Registers of clobbered or modified);
```

With the exception of the first line, lines with a colon after them can be omitted if they are not used. Among them, "asm" is an inline assembly statement keyword; "assembly statement" is where you write assembly instructions; "output register" indicates which registers are used to store output data after this embedded assembly is executed. Here, these registers correspond to a C expression value or a memory address, respectively; "input register" indicates the input value that should be stored in some of the registers specified here when the assembly code is started. They also correspond to a C variable, or constant value respectively. "Clobbered or Modified registers" means that you have changed the values in the registers listed there, and the gcc compiler can no longer rely on the values it originally loaded on these registers. Gcc needs to reload these registers if necessary. Therefore we need to list those register names that are not listed in the output/input registers section, but that are explicitly used or implicitly used in assembly statements.

For example, here is a fictitious ‘combine’ instruction:

```
    asm( “combine  %1, %0”  : “=r” (result)  : “r”  (length));
```

Here length is the C expression for the input operand while result is that of the output operand. Each has “r” as its operand constraint, saying that a dynamically allocated general register is required. The ‘=’ in ‘=r’ indicates that the operand is an output; all output operands’ constraints must use ‘=’. The constraints use the same language used in the machine description as described in the gcc manual (section Operand Constraints in Chapter 16 Machine Descriptions).

As shown in the above example, each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand and another separates the last output operand from the first input, if any. Commas separate the operands within each group. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater. If there are no output operands but there are input operands, you must place two consecutive colons surrounding the place where the output operands.

Below we use a more detailed example to illustrate the use of inline assembly statements. Here is a block of code starting at line 22 in the kernel/traps.c file. In order to see more clearly, we have rearranged and numbered this code.

```
01  #define get_seg_byte(seg,addr) \
02  ({ \
03  register char __res; \                 // A register variable __res is defined.
04  __asm__(″push %%fs;  \                 // First save original value of fs (seg selector).
05           mov %%ax,%%fs;  \             // Then use seg to set fs.
06           movb %%fs:%2,%%al;  \         // Take 1 byte of seg:addr into the al register.
07           pop %%fs″  \                  // Restore the original contents of fs register.
08           :″=a″ (__res)  \              // Output register lists and constraints.
09           :″0″ (seg),″m″ (*(addr))); \  // Input register lists and constraints.
10  __res;})
```

This 10 lines of code defines an inline assembly language macro function. The most convenient way to use assembler statements is to place them in a macro. A compound statement enclosed in parentheses (a statement in curly braces): "({})" can be used as an expression, where the variable __res (line 10) on the last line is the output value of the expression, see The next section explains.

Because macro statements need to be defined on one line, these statements are concatenated using a backslash '\' here. This macro definition will be substituted into the program where the macro name is referenced. The first line defines the name of the macro, which is the macro function name get_seg_byte(seg,addr). Line 3 defines a register variable __res. This variable will be saved in a register for quick access and operation. If you want to specify a register (such as eax), then we can write this sentence as "register char __res asm ("ax");", where "asm" can also be written as "__asm__". The "__asm__" on line 4 indicates the beginning of the embedded assembly statement. The 4 statements from line 4 to line 7 are AT&T format assembler statements. In addition, in order to have a percent sign "%" in front of the register name in an assembly language program generated by gcc, two percent sign "%%" must be written before embedding the assembly statement register name.

The eighth line is the output register. The meaning of this sentence is to place the value of the register represented by eax in the __res variable after the end of this code run, as the output value of this function, "a" in "=a" Called load code, "=" indicates that this is an output register and the value in it will be replaced by the output value. Load code is a shorthand letter code for CPU registers, memory addresses, and some numeric values. Table 3-4 shows some of the register loading code and its specific meaning that we often use. Line 9 indicates that seg is placed in the eax register when this code starts to run, and "0" means that the same register as the one above is output. (*(addr)) represents a memory offset address value. In order to use this address value in the above assembler statement, the embedded assembler program specifies that the output and input registers are numbered sequentially, starting from the left and right top to bottom of the output register sequence starting with "%0", respectively, denoted as % 0, %1, ...%9. Therefore, the output register number is %0 (here only one output register), the first part of the input register ("0" (seg)) has the number %1, and the latter part has the number %2. The %2 above line 6 above represents (*(addr)) this memory offset.

Table 3-4 Common register load code description

Code	Description	Code	Description
a	Use register eax	m	Use memory address, any memory operand is allowed.
b	Use register ebx	o	Use memory address, and can add offset value
c	Use register ecx	I	Use constants 0-31
d	Use register edx	J	Use constants 0-63
S	Use register esi	K	Use constants 0-255
D	Use register edi	L	Use constants 0-65535
q	Use dynamically allocated byte addressable registers (eax、ebx、ecx 或 edx)	M	Use constants 0-3
r	Use any dynamically allocated register	N	Use 1 byte constant (0-255)
g	Any general register, memory or immediate integer operand is allowed （eax、ebx、ecx、edx or memory variable）	O	Use constants 0-31
A	Combine eax with edx (64-bit)	=	Output operands. The output value will replace the previous value
+	Indicates that the operand is readable and writable	&	Early-clobber operands. Indicates that the content will be modified before the operands are used

Now let's examine the function of code on lines 4-7. The first sentence puts the contents of the fs segment register on the stack; the second sentence assigns the segment value in eax to the fs segment register; the third sentence puts the byte specified by fs:(*(addr)) into the al register. . When the assembly statement is executed, the value of the output register eax will be put into __res as the return value of the macro function (block structure expression). It's simple, isn't it?

From the above analysis, we know that seg in the macro name represents a specified memory segment value, and addr represents a memory offset address amount. Until now, we should be very clear about the function of this program! The function of this macro function is to fetch one byte from the memory address of the specified segment and offset value. Then look at the next example.

```
01  asm(″cld\n\t″
02      ″rep\n\t″
03      ″stol″
04      : /* No output register */
05       : ″c″(count-1), ″a″(fill_value), ″D″(dest)
06       : ″%ecx″, ″%edi″);
```

The 1-3 lines are the usual assembler statements to clear the direction bit and repeatedly store the value. The characters "\n\t" in the first two lines are used to neatly set the gcc preprocessor output program list. The meaning of the characters is the same as in the C language. That is, the operation mode of gcc is to generate the assembler corresponding to the C program, and then call the assembler to compile it to generate the target code. If you want to look at the assembler corresponding to C when you write the program and debug the program, you need to get the pre The output of the assembler program that processes the program (this is commonly used when writing and debugging efficient code). In order to preprocess the assembler output in a neat format, you can use the two "\n\t" format symbols.

Line 4 shows that the inline assembler does not use the output register. The meaning of line 5 is: Load the count-1 value into the ecx register (the loading code is "c"), fill_value is loaded into eax, and dest is placed into edi. Why do we have to make the gcc compiler to load such register values without letting us do it ourselves? Because gcc can perform some optimization work while it is registering. For example, the fill_value value may already be in eax. If it is in a loop statement, gcc may keep eax in the entire loop operation, so you can use a movl statement in each loop.

The last line is to tell gcc that the values in these registers have changed. After gcc knows what you are doing with these registers, it can help with gcc's optimization. The following example does not allow you to specify which variable to use which register, but let gcc choose it for you.

```
asm(″leal (%1, %1, 4), %0″
     : ″=r″(y)
     : ″0″(x));
```

The instruction "leal" is used to calculate the effective address, but it is used here for some simple calculations. The first assembler statement "leal (r1, r2, 4), r3" indicates r1+r2*4 => r3. This example can multiply x by 5 very quickly. Among them, "%0" and "%1" refer to the register that gcc automatically allocates. Here "%1" represents the register into which the input value x is to be placed, and "%0" represents the output value register. Be sure to add the equal sign before outputting the register code. If the code of the input register is 0 or is empty, then the same register as the corresponding output is used. So, if gcc specifies r as eax, then the meaning of the above assembly statement is:

```
    ″leal (eax,eax,4), eax″
```

Note that when executing the code, if you do not want the assembly statement to be modified by GCC optimization, you need to add the keyword volatile after the asm symbol, as shown below. The difference between these two declarations lies in the aspect of program compatibility. It is recommended to use the latter way of declaration.

```
    asm volatile (......);
Or a more detailed explanation is:
    __asm__ __volatile__ (......);
```

The keyword volatile can also be placed before the function name to decorate the function to inform the gcc compiler that the function will not return. This will allow gcc to produce better code. In addition, for functions that do not return, this keyword can also be used to prevent gcc from generating false warning messages. For example, the following statement in mm/memory.c shows that the functions do_exit() and oom() no longer return to the caller code:

```
volatile void do_exit(long code);

static inline volatile void oom(void)
{
        printk("out of memory\n\r");
        do_exit(SIGSEGV);
}
```

Here is a longer example. If you can read it, it means that inline assembly code is basically OK for you. This code is taken from the include/string.h file and is an implementation of the string comparison function for strncmp(). Similarly, the "\n\t" in each of these lines is set for the gcc preprocessor output list to look good.

```
//// String1 is compared with string2 in the first count characters.
// Paras: cs - Strings1,ct  - String2,count - The number of characters to compare.
// %0 - eax(__res) return,%1 - edi(cs) String1 ptr,%2 - esi(ct)String2 ptr, %3 - ecx(count)。
// Return: If string1 > string2, ret 1; string1 == string2, ret 0; string1<string2, then ret -1.
extern inline int strncmp(const char * cs,const char * ct,int count)
{
register int __res ;                  // __res is a register variable.
__asm__("cld\n"                       // Clear direction.
         "1:\tdecl %3\n\t"            // count--。
         "js 2f\n\t"                  // If count<0, go forward to label 2.
         "lodsb\n\t"                  // Take string 2 character ds:[esi]=>al, and esi++.
         "scasb\n\t"                  // Compare char in al and in string1 es:[edi] and edi++.
         "jne 3f\n\t"                 // If they are not equal, go forward to label 3.
         "testb %%al,%%al\n\t"        // Is this character a NULL character?
         "jne 1b\n"                   // No, go backward to label 1 and continue comparing.
         "2:\txorl %%eax,%%eax\n\t"   // If it is a NULL char, eax is cleared (return value).
         "jmp 4f\n"                   // Go forward to label 4 and end.
         "3:\tmovl $1,%%eax\n\t"      // eax is set to 1.
         "jl 4f\n\t"                  // If the string2 chars <string1 chars, return 1 and end.
         "negl %%eax\n"               // Otherwise eax = -eax returns a negative value, ends.
         "4:"
         :"=a" (__res):"D" (cs),"S" (ct),"c" (count):"si","di","cx");
return __res;                         // Return the comparison result.
}
```

3.3.3 Combination statements in parentheses

The braces pair "{...}" is used to combine variable declarations and statements into a compound statement (combination statement) or a statement block so that these statements are semantically equivalent to a single statement. There is no need to use a semicolon after the closing brace of a compound statement. Combination statements in parentheses, ie statements of the form "({...})", can be used as an expression in GNU C. This allows loops, switch statements, and local variables to be used in expressions, so this form of statement is often called a statement expression. The statement expression has the following example form:

```
({ int y = foo(); int z;
 if (y > 0) z = y;
 else z = -y;
 3 + z; })
```

The last statement in a compound statement must be an expression followed by a semicolon. The value of this expression ("3 + z") is used as the value enclosed by the entire parenthesis. If the last statement is not an expression, the entire statement expression has a void type and therefore has no value. In addition, any local variables declared by statements in such an expression will expire after the entire block statement ends. This sample statement can be used like the following form of assignment statement:

```
res = x + ({Omit...}) + b;
```

Of course, people usually don't write statements like the above, which are usually used to define macros. For example, the macro definition for reading CMOS clock information in the kernel source code init/main.c program:

```
#define CMOS_READ(addr) ({ \
outb_p(0x80|addr,0x70); \          // First, output the addr to the I/O port 0x70.
inb_p(0x71); \                     // Then read the value from port 0x71 as the return value.
})
```

Look again at the macro definition of the read I/O port in the include/asm/io.h header file, where the value of the last variable _v is the return value of inb().

```
#define inb(port) ({ \
unsigned char _v; \
__asm__ volatile ("inb %%dx,%%al":"=a" (_v):"d" (port)); \
_v; \
})
```

3.3.4 Register variables

Another extension of GNU to the C language allows us to put some variable values into the CPU registers, the so-called register variables. In this way, the CPU does not need to spend a long time to access the memory for value. There are two types of register variables: global register variables and local register variables. Global register variables hold registers dedicated to several global variables throughout the program's operation. In contrast, local register variables do not retain the specified registers, and special registers are used only as input or output operands in the inline asm assembly statement. The gcc compiler's data flow analysis capability is inherently capable of determining when a specified register has a value in use and when it can dispatch other fields. When the gcc data flow analysis function is considered to be stored when a local register variable value is useless, it may be deleted, and references to local register variables may also be deleted, moved, or simplified. Therefore, if you do not want gcc to make these optimization changes, it is best to add volatile keywords in the asm statement.

If you want to write the output of the assembler instruction directly to the specified register in an inline assembler statement, it is convenient to use local register variables at this time. Since the Linux kernel usually only uses local register variables, we will only discuss the use of local register variables here. In GNU C programs we can define a local register variable in a function like this:

```
    register int res __asm__(″ax″);
```

Here ax is the register that the variable res wants to use. Defining such a register variable does not specifically reserve this register for no other purpose. During program compilation, when the gcc data flow control determines that the value of a variable is no longer in use, the register may be dispatched for other purposes, and references to it may be deleted, moved, or simplified. In addition, gcc does not guarantee that the compiled code will keep the variable in the specified register. Therefore, it is better not to refer to this register explicitly in the portion of the instruction that is embedded in the assembly and it is assumed that the register must refer to this variable value. However, using this variable as an operand to asm ensures that the specified register is used as the operand.

3.3.5 Inline function

In a program, by declaring a function as an inline function, you can have gcc integrate the code of the function into the code that calls the function. This processing can remove the overhead of the entry/exit time when the function call is invoked, thus definitely speeding up execution. Therefore, the main purpose of declaring a function as an inline function is to be able to execute the function body as quickly as possible. In addition, if there is a constant value in an inline function, gcc may use it to perform some simplification during compilation, so not all inline function code will be embedded. The inline function method has no obvious effect on the length of the program code. Programs compiled using inline functions may generate longer or shorter target code, depending on the circumstances.

The operation of embedding an inline function in the caller's code is an optimization operation, so code embedding processing is performed only when an optimized compilation is performed. If the optimization option "-O" is not used during compilation, the code of the inline function is not actually embedded in the caller's code, but is only handled as an ordinary function call. The way to declare a function as an inline function is to use the keyword "inline" in the function declaration, such as the following function in the kernel file fs/inode.c:

```
inline int inc(int *a)
{
        (*a)++;
}
```

The use of some of the statements in a function may prevent the replacement of an inline function from working properly, or may not be suitable for replacement operations. For example, variable parameters, memory allocation functions malloca(), variable-length data type variables, non-local goto statements, and recursive functions are used. Compiler can use the option -Winline to make gcc give warning information for functions marked as inline but cannot be replaced, and why they cannot be replaced.

When using both the inline keyword and the static keyword in a function definition, ie the definition of an inline function in the file fs/inode.c below, then all calls to the inline function are replaced if they are replaced. In the caller code, and the program does not refer to the address of the inline function, the assembly code of the inline function itself will not be referenced. In this case, unless we use the option -fkeep-inline-functions during compilation, gcc will no longer generate actual assembly code for the inline function itself. For some reason, some calls to inline functions cannot be integrated into functions. In particular, the calling statement before the definition of the inline function is not replaced by the integration and cannot be a function defined by recursion. If there is a call that cannot be replaced by an integration, the inline function is compiled into assembly code as usual. Of course, if the program has a statement that references the address of an inline function, the inline function is also compiled into assembly code as usual. Because references to inline function addresses cannot be replaced.

```
static inline void wait_on_inode(struct m_inode * inode)
{
        cli();
        while (inode->i_lock)
                sleep_on(&inode->i_wait);
        sti();
}
```

Please note that inline function functions have been included in ISO standard C99, but the inline functions defined by this standard are quite different from those defined by gcc. The semantic definition of the inline function of the ISO standard C99 is equivalent to the definition of the combination keyword inline and static, which means “eliminating” the keyword static. If you need to use the C99 standard semantics in your program, you need to use the compile option -std=gnu99. However, in order to be compatible, it is still best to use inline and static combinations in this case. After that gcc will eventually use the definition of C99 by default. If you want to still use the semantics defined here, you need to use the option -std=gnu89 to specify.

If the definition of an inline function does not use the keyword static, then gcc will assume that there is also a call to this function in other program files. Because a global symbol can only be defined once, the function can no longer be defined in other source files. Therefore, calls to inline functions cannot be replaced by integration here. Therefore, a non-static inline function is always compiled with its own assembly code. In this regard, the ISO standard C99 definition of an inline function that does not use the static keyword is equivalent to using the static keyword definition here.

If both inline and extern keywords are specified when defining a function, the function definition is only used for inline integration, and the function's own assembly code is not generated separately in any case, even if the function is explicitly referenced. The address will not be generated either. Such an address becomes an external reference, just as if you just declared a function without defining a function.

The combination of inline and extern is almost identical to a macro definition. Using this combination method is to put a function definition with a combination keyword in the .h header file, and put another definition of the same function without a keyword in a library file. The definition in the header file at this time causes most of the calls to the function to be embedded by substitution. If there is a call to the function that has not been replaced, then a copy of the program file or library is used (referenced). The file include/string.h, lib/strings.c in the Linux 0.1x kernel source code is an example of this use. For example, the following function is defined in string.h:

```
// Copy the string (src) to another string (dest) until it encounters a NULL character.
// Paras: dest - dest string ptr, src - source string ptr. %0 - esi(src),%1 - edi(dest).
extern inline char * strcpy(char * dest,const char *src)
{
__asm__("cld\n"                                   // Clear direction.
        "1:\tlodsb\n\t"                           // Load DS: [esi] 1 byte => al and update esi.
        "stosb\n\t"                     // Store byte al=>ES:[edi] and update edi.
        "testb %%al,%%al\n\t"           // Just stored byte al is 0?
        "jne 1b"                        // If not, go back to label 1 or end.
        ::"S" (src),"D" (dest):"si","di","ax");
return dest;                            // Returns the destination string pointer.
}
```

In the kernel library directory, the lib/strings.c file defines the keywords inline and extern as empty, as shown below. Therefore, in fact, the kernel library contains a copy of all such functions in the string.h file, which in turn redefines these functions once and "eliminates" the effect of the two keywords.

```
#define extern                          // Defined as empty.
#define inline                          // Defined as empty.
#define __LIBRARY__
#include <string.h>

```

The above-defined strcpy() function in the library function now becomes the following:

```
char * strcpy(char * dest,const char *src)    // Removed keywords inline and extern.
{
__asm__("cld\n"                               // Clear direction.
        "1:\tlodsb\n\t"                       // Load DS: [esi] 1 byte => al and update esi.
        "stosb\n\t"                           // Store byte al=>ES:[edi] and update edi.
        "testb %%al,%%al\n\t"                 // Just stored byte al is 0?
        "jne 1b"                              // If not, go back to label 1 or end.
        ::"S" (src),"D" (dest):"si","di","ax");
return dest;                                  // Returns the destination string pointer.
}
```

3.4 Interworking between C and Assembly language

In order to improve the efficiency of code execution, some parts of the kernel source code directly use the assembly language. This will involve the invocation problem between two language programs. This section first describes the invocation mechanism of C language functions, and then uses an example to illustrate the calling method between the two functions.

3.4.1 C function call mechanism

After the Linux kernel program boot/head.s performs basic initialization operations, it will jump to execute the init/main.c program. So how does the head.s program transfer its execution control to the init/main.c program? That is how the assembler calls to execute C language programs? Here we first describe the C function call mechanism, control transfer mode, and then explain the head.s program jumps to the C program.

Function call operations include bidirectional data transfer and execution control transfer from one piece of code to another. Data passing is done through function parameters and return values. In addition, we also need to allocate storage space for the function's local variables when entering the function, and reclaim this space when exiting the function. The Intel 80x86 CPU provides simple instructions for control transfer, while the transfer of data and the allocation and recovery of local variable storage space are achieved through stack operations.

3.4.1.1 Stack frame structure and control transfer method

Most program implementations use the stack to support function call operations. The stack is used to transfer function parameters, store return information, temporarily save the original values of the registers for recovery and to store local data. The portion of the stack used by a single function call operation is called the stack frame structure. Its general structure is shown in Figure 3-4. Both ends of the stack frame structure are specified by two pointers. The register ebp is usually used as a frame pointer, and esp is used as a stack pointer. During the execution of the function, the stack pointer esp moves with the data being pushed onto the stack. Therefore, most data accesses in the function are based on the frame pointer ebp.

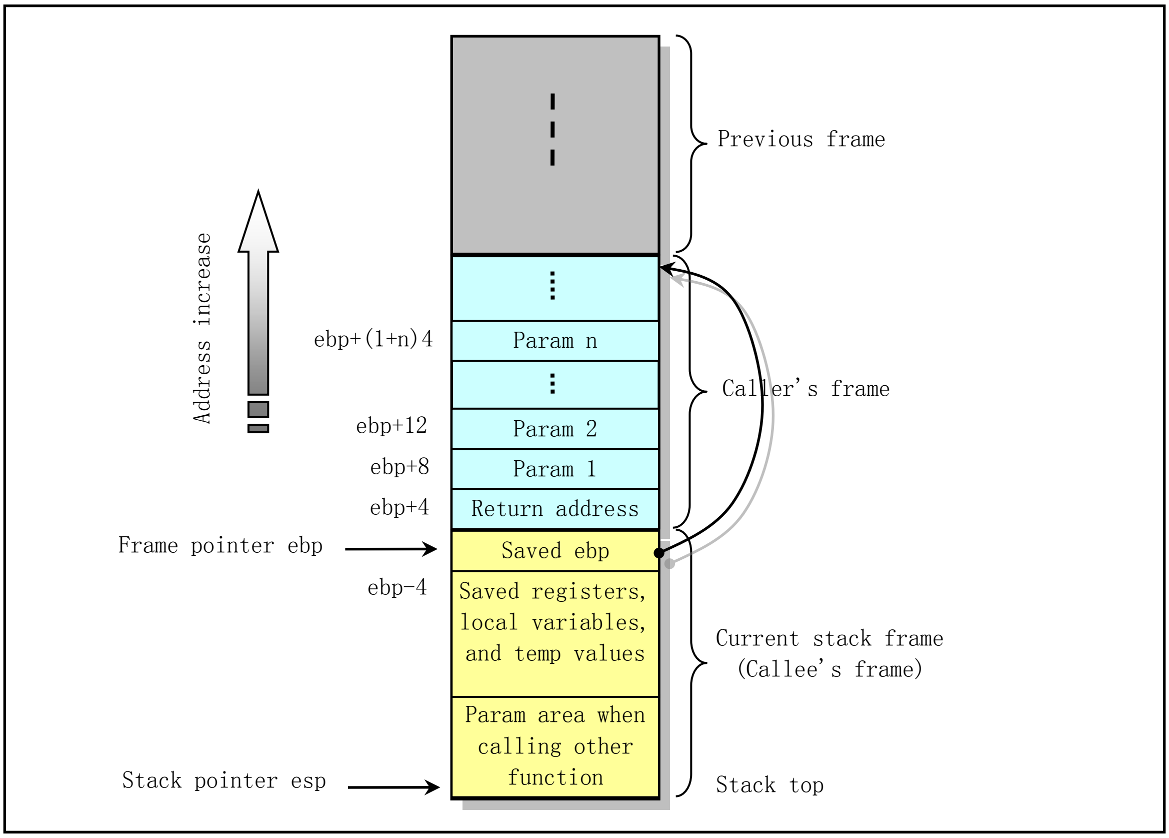


Figure 3-4 Frame structure in the stack

For the case where function A calls function B, the parameters passed to B are contained in A's stack frame. When A calls B, the return address of Function A (the address of the instruction that continues execution after the call is returned) is pushed onto the stack. This position on the stack also explicitly indicates the end of the A stack frame. The stack frame of B starts from the subsequent stack section, where the frame pointer (ebp) is stored. It is then used to store any saved register values and temporary values of the function.

The B function also uses the stack to hold local variable values that cannot be placed in registers. For example, because the normal CPU has a limited number of registers and cannot store all the local data of the function, or some local variables are arrays or structures, they must be accessed using an array or a structure reference. There is also the C language address operator '&' is applied to a local variable, we need to generate an address for the variable, which allocates a space for the address pointer of the variable. Finally, the B function will use the stack to save the parameters that call any other function.

The stack is expanded toward the low (small) address, while the esp points to the element at the top of the current stack. By using the push and pop instructions we can push data onto the stack or pop it off the stack. For storage space that does not specify initial data, we can do this by decrementing the stack pointer by an appropriate value. Similarly, we can reclaim the allocated space on the stack by increasing the stack pointer value.

The instructions CALL and RET are used to handle function invocation and return operations. The effect of the instruction CALL is to push the return address onto the stack and jump to the beginning of the called function. The return address is the address of an instruction immediately following the instruction CALL in the program. So when the called function returns, it will continue from that position. The return instruction RET is used to pop up the address at the top of the stack and jump to that address. Before using this instruction, the contents of the stack should be processed correctly so that the current stack pointer is the same as the one returned by the previous CALL instruction. In addition, if the return value is an integer or a pointer, the register eax will be used by default to pass the return value.

Although only one function is executed at a time, we still need to make sure that when a function (caller) calls another function (the callee), the callee will not modify or overwrite the register contents that the caller will use in the future. Therefore, the Intel CPU adopts the uniform usage of registers that all functions must comply with. This convention indicates that the contents of registers eax, edx, and ecx must be held by the caller themselves. When function B is called by A, function B can use them arbitrarily without saving the contents of these registers without destroying any data needed by function A. In addition, the contents of registers ebx, esi, and edi must be protected by callee B. When the callee needs to use any of these registers, it must first save its contents on the stack and restore the contents of these registers on exit. Because caller A (or some higher-level function) is not responsible for saving these register contents, it may need to use the original value in future operations. There are also registers ebp and esp that must follow the second convention usage.

3.4.1.2 Function call example

As an example, let's observe the processing of the function call in the following C program exch.c. The program exchanges the values in the two variables and returns their difference.

```
void swap(int * a, int *b)
{
        int c;
        c = *a; *a = *b; *b = c;
}

int main()
{
        int a, b;
        a = 16; b = 32;
        swap(&a, &b);
        return (a - b);
}
```

The function swap() is used to exchange the values of two variables. The main program in the C program, main(), is also a function (described below). It returns the swapped result after calling swap(). The stack frame structure of these two functions is shown in Figure 35. As you can see, the function swap() gets its parameters from the caller's (main()) stack frame. The position information in the figure is relative to the frame pointer in the register ebp. The number to the left of the stack frame indicates the address offset value relative to the frame pointer. In debuggers such as gdb, these values are represented in 2's complement. For example, '-4' is represented as '0xFFFFFFFC', and '-12' is represented as '0xFFFFFFF4'.

The stack frame structure of the caller main() includes the storage space for the local variables a and b, which are located at -4 and -8 offsets with respect to the frame pointer. Since we need to generate addresses for these two local variables, they must be stored on the stack and not simply stored in registers.

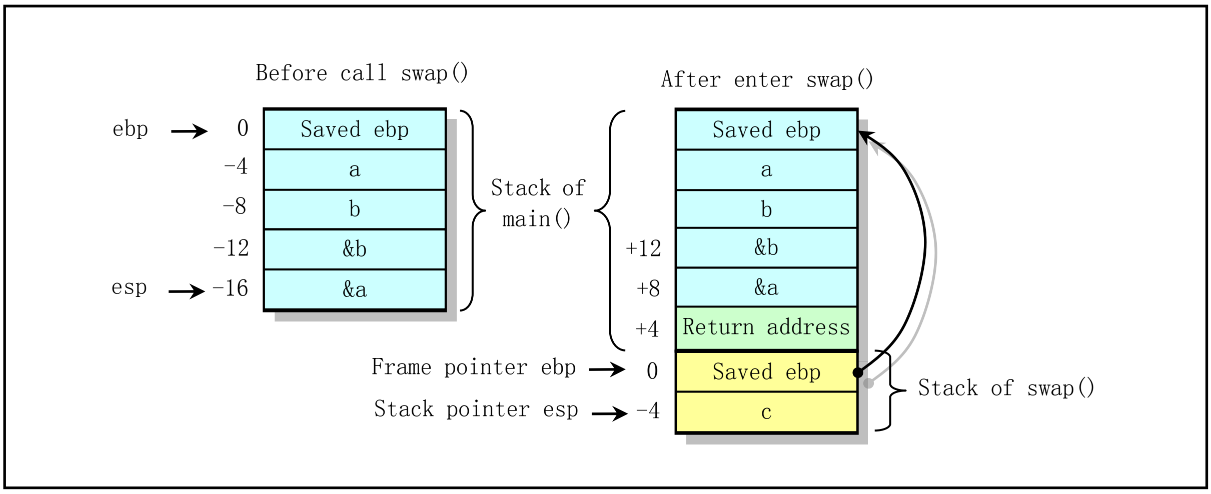


Figure 3-5 Stack frame structure when calling function main and swap

Use the command "gcc -Wall -S -oexch.s exch.c" to generate the assembler exch.s code for this C language program, as shown below (remove several lines of directives that are not relevant to the discussion).

```
.text
_swap:
       pushl %ebp                 # Save original ebp, set current function's frame pointer.
       movl %esp,%ebp
       subl $4,%esp               # Allocates space within the stack for the local variable c.
       movl 8(%ebp),%eax          # Get 1st argument, which is a pointer to an integer value.
       movl (%eax),%ecx           # Store value pointed by the pointer into variable c.
       movl %ecx,-4(%ebp)
       movl 8(%ebp),%eax          # Take 1st parameter again, and then take 2nd parameter.
       movl 12(%ebp),%edx
       movl (%edx),%ecx           # Put the content of the 2nd parameter into the place
       movl %ecx,(%eax)           # pointed by the 1st parameter.
       movl 12(%ebp),%eax         # Take the second parameter again,then place the content
       movl -4(%ebp),%ecx         # of variable c at the position pointed by this pointer.
       movl %ecx,(%eax)
       leave                      # Restore the original ebp and esp.
       ret
_main:
       pushl %ebp                 # Save original ebp, set current function's frame pointer.
       movl %esp,%ebp
       subl $8,%esp               # Allocates space in stack for local variables a and b.
       movl $16,-4(%ebp)          # Assign initial values to variables (a=16, b=32).
       movl $32,-8(%ebp)
       leal -8(%ebp),%eax         # To call the swap(), push the address of variable b onto
       pushl %eax                 # stack. That is, push the second parameter first.
       leal -4(%ebp),%eax         # Then push address of variable a as the first parameter.
       pushl %eax
```

```
28        call _swap                  # Call the function swap().
29        movl -4(%ebp),%eax          # Take the value of a - b.
30        subl -8(%ebp),%eax
31        leave                       # Restore the original ebp and esp.
32        ret
```

These two functions can be divided into three parts independently: "set" to initialize the stack frame structure; "body" to perform the actual calculation of the function; "end" to restore the stack state and return from the function. For the swap() function, the setting part code is 3--5 lines. The first two lines are used to set the caller's frame pointer and set the function's stack frame pointer. Line 5 allocates space for the local variable c by moving the stack pointer esp down by 4 bytes. Line 6-15 is the main part of the swap function. Lines 6 - 8 are used to retrieve the caller's first parameter, &a, and use this parameter as an address to fetch the contents of the memory into the ecx register and save it to the space allocated for the local variable (-4 (%ebp)) . Lines 9-12 are used to fetch the second parameter, &b, and take the parameter value as the address and take its contents to the address specified by the first parameter. Lines 13-15 store the value stored in the temporary local variable c at the address specified by the second parameter. The last 16-17 lines are the end of the function. The leave instruction is used to process the contents of the stack in preparation for return. Its role is equivalent to the following two instructions:

```
      movl %ebp,%esp              # Restores original esp (to beginning of stack frame).
      popl %ebp                   # Restores original ebp (usually the caller's frame ptr).
```

This two lines of code restore the original values of the registers esp and ebp when entering the swap() function and executes the return instruction ret.

Lines 19-21 are the set part of the main() function. After saving and resetting the frame pointer, main() allocates space for the local variables a and b on the stack. Lines 22-23 assign values to these two local variables. You can see from lines 24-28 how main() calls the swap() function. The first step is to use the leal instruction (fetching an effective address) to get the addresses of the variables b and a and push them onto the stack, and then call the swap() function. The order in which the variable addresses are pushed onto the stack is exactly the opposite of the order of the parameters declared by the function. That is, the last parameter of the function is pushed onto the stack first, and the first parameter of the function is pushed onto the stack before the call to the function instruction call. Lines 29--30 subtract the two already exchanged numbers and place them in the eax register as the return value.

From the above analysis we can see that when C calls a function, it temporarily stores the value of the transferred function parameter on the stack. That is, C language is a value-based language. There is no direct method to modify the caller variable in the called function. value. Therefore, in order to achieve the purpose of modification, you need to pass a pointer to the variable (ie, the address of the variable) to the function.

3.4.1.3 Main() is also a function

The above assembler code is compiled using gcc 1.40. It can be seen that there are a few lines of extra code. It can be seen that the gcc compiler at that time could not produce the most efficient code. This is one of the reasons why some key code needs to be compiled directly in assembly language. In addition, the main program of the C program mentioned above is also a function. This is because it will be called as a function of the crt0.s assembler program when the link is compiled. crt0.s is a stub program. The name "crt" is an abbreviation for "C run-time". The program's target file will be linked at the beginning of each user's execution program, mainly used to set some initialization global variables. The crt0.s assembler program in Linux 0.12 is shown below. The global variable _environ is created and initialized for use by other modules in the program.

```
.text
.globl _environ                   # Declare the global variable _environ. (correspond to
                                  # the environ variable in C program).
__entry:                          # Code entry label.
        movl 8(%esp), %eax        # Get environment variable pointer envp, save in _environ.
        movl %eax, _environ       # envp is set by execve() when executable file is loaded.
        call _main                # Call main program. Its return status is in eax register.
        pushl %eax                # Push return value as an argument to exit() and call it.
1:      call _exit
        jmp 1b                    # Control should not arrive here.
.data
_environ:                         # Define variable _environ and assign it a long word space.
        .long 0
```

When gcc is used to compile and link the executable file, gcc will automatically link the code of crt0.s as the first module in the executable program. Use the show details option '-v' at compile time to clearly see the linking process:

```
[/usr/root]# gcc -v -o exch exch.s
gcc version 1.40
 /usr/local/lib/gcc-as -o exch.o exch.s
 /usr/local/lib/gcc-ld -o exch /usr/local/lib/crt0.o exch.o /usr/local/lib/gnulib -lc
 /usr/local/lib/gnulib
[/usr/root]#
```

So in the normal compilation process we don't need to specify the stub module crt0.o, but if we want to use the ld(gld) to generate the executable exch from the exch.o module manually from the assembly program given above, then we need to The crt0.o module is specified on the command line, and the link order should be "crt0.o, all program modules, library files."

In order to use ELF format object files and create a shared library module file, the current gcc compiler (2.x) has extended this crt0 into several modules: crt1.o, crti.o, crtbegin.o, crrtend.o, and Crtn.o. The link order of these modules is "crt1.o, crti.o, crtbegin.o (crtbeginS.o), all program modules, crrtend.o (crtendS.o), crtn.o, library module files". The gcc configuration file specfile specifies this link order. Where ctr1.o, crti.o, and crtn.o are provided by the C library, which is the C program's "boot" module; crtbegin.o and crentend.o are the C++ language startup modules provided by the compiler gcc; and crt1.o It is similar to the effect of crt0.o, and is mainly used to do some initialization work before calling main(). The global symbol _start is defined in this module.

crtbegin.o and crentend.o are mainly used in C++ languages to implement global constructors and destructor functions in the .ctors and .dtors sections. The roles of crtbeginS.o and crrtendS.o are similar to those of the first two, but they are used to create shared modules. crti.o is used to execute the initialization function init() in the .init section. The .init section contains the initialization code for the process, ie when the program starts executing, the system executes the code in .init before calling main(). Crtn.o is used to execute the process in the .fini area to terminate the processing function fini(), that is, when the program exits normally (main() returns), the system will arrange to execute the code in .fini.

In the kernel, lines 136--140 in the boot/head.s program are used to make preparations for jumping to the main() function in init/main.c. The instruction on line 139 pushes the return address on the stack, while line 140 presses the address of the main() function code. When head.s finally executes the ret instruction on line 218, the address of main() is popped up, and control is transferred to the init/main.c program.

3.4.2 Call C function in assembler code

The method of calling a C language function from an assembly code is actually given above. In the above assembler code for the C language example, we can see how the assembler statement calls the swap() function. Now we make a summary of the calling method.

When calling a C function in assembly code, you first need to push the function parameters into the stack in reverse order. That is, the last (rightmost) parameter of the function is pushed on the stack first, and the leftmost first parameter is pushed before the last instruction is called. See Figure 3-6. Then execute the CALL instruction to execute the called function. After the calling function returns, the program needs to clear the function parameters that were previously pushed onto the stack.

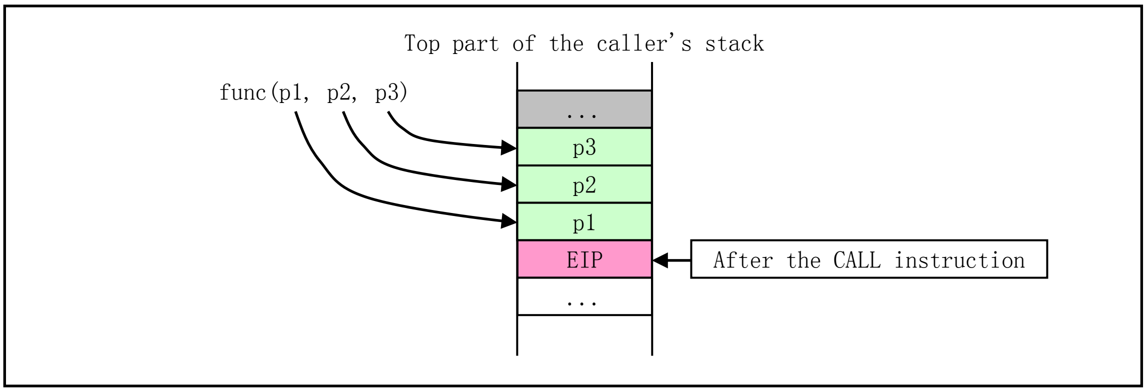


Figure 3-6 The parameters pushed into the stack when the function is called

When the CALL instruction is executed, the CPU pushes the address of the next instruction of the CALL instruction onto the stack (see EIP in the figure). If the call also involves code privilege level changes, the CPU will also perform a stack switch and push the current stack pointer, segment descriptor, and call parameters into the new stack. Since the Linux kernel uses only interrupt gates and trapdoors to handle privilege level changes during the call, and does not use the CALL instruction to handle privilege level changes, the use of the CALL instruction at the time of privilege level change is not described here. .

Calling C functions in assembly is relatively "free." As long as it is in the proper place in the stack, it can be used as a parameter for C functions. Here is still taking the function call with 3 parameters in Figure 3-6 as an example. If we do not call it directly by pushing the argument with func(), the func() function will still store the EIP position. The rest of the stack is used as its own parameter. If we explicitly press the first and second parameters for func() call, then the third parameter p3 of the func() function will directly use the contents of the stack before p2. There are several places in this Linux 0.1x kernel code. For example, the copy_process() function (line 68 in kernel/fork.c) is called on line 231 in the kernel/sys_call.s assembly program. Although only five parameters are pushed onto the stack in the assembly function _sys_fork, copy_process() has a total of up to 17 parameters, as shown below:

```
// kernel/sys_call.s partial program: _sys_fork
```

```
        push %gs
        pushl %esi
        pushl %edi
        pushl %ebp
        pushl %eax
        call _copy_process         # Call the C function copy_process() (kernel/fork.c, 68).
        addl $20,%esp              # Discard all the pushed content here.
1:      ret
```

```
   // kernel/fork.c partial program.
68 int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
69                 long ebx,long ecx,long edx, long orig_eax,
70                 long fs,long es,long ds,
71                 long eip,long cs,long eflags,long esp,long ss)
```

We know that the later the parameter is pushed onto the stack, the closer it is to the left of the C function parameter. Therefore, the five register values that were actually pushed before the copy_process() was called are the five leftmost parameters of the copy_process() function. In order, they correspond to the values of the eax(nr), ebp, edi, esi, and register gs that are stacked. The rest of the following parameters actually correspond directly to what is already on the stack. These contents are the values of various registers that are gradually added to the stack when the system call interrupt processing process is started until the system call process is called.

The parameter none is the return address of the next instruction when _sys_fork is called from the address jump table on line 99 of the sys_call.s program. The address jump table sys_call_table[] is defined in line 93 of the header file include/linux/sys.h. The following parameters are the registers ebx, ecx, edx, the original eax, and the segment registers fs, es, and ds that were pushed onto the stack at lines 85-91 just after entering system_call. The last five parameters are the CPU execution interrupt instruction push return address eip and cs, flag register eflags, user stack address esp and ss. Because the system call involves a change in program privilege level, the CPU pushes the flag register value and the user stack address onto the stack. After calling C function copy_process() returns, _sys_fork discards only 5 parameters pressed by itself, and the rest of the stack is also saved. Other functions that use the above usage include do_signal() in kernel/signal.c, do_execve() in fs/exec.c, etc. Please analyze it yourself.

In addition, we say that the assembly calls the C function is relatively free because we can use the JMP instruction to achieve the same purpose of calling the function without the CALL instruction. The method is to put the address of the instruction to be executed next into the stack manually after the parameter is pushed into the stack, and then directly use the JMP instruction to jump to the start address of the called function to execute the function. Afterwards, when the execution of the function is completed, the RET instruction will be executed to pop up the address of the next instruction we push manually into the stack, as the address returned by the function. There are also many ways to call this function in the Linux kernel, such as the case where the 62nd line of the kernel/asm.s program calls the do_int3() function in traps.c.

3.4.3 Call assembly function in C program

Calling an assembly function from a C program is the same as calling an C function in an assembler, but it is not often used in Linux kernel programs. The focus of the calling method is still on the determination of the location of function parameters in the stack. Of course, if the calling assembly language program is relatively short, it can be directly implemented in the C program using the inline assembly statement described above. Below we use an example to illustrate how to program this kind of program. The assembler calle.s containing two functions is shown below.

```
/*
  This assembly language program uses the system call sys_write() to implement the display
  function int mywrite(int fd, char * buf, int count).
  The function int myadd(int a, int b, int * res) is used to perform the a+b = res operation.
  If the function returns 0, it means overflow.
  Note: If you compile under the current Linux system (such as RedHat 9), remove the
  underscore '_' before the function name.*/
SYSWRITE = 4                        # Sys_write() system call number.
.globl _mywrite, _myadd
.text
_mywrite:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        movl    8(%ebp), %ebx       # Take the first argument of the caller: file descriptor fd.
        movl    12(%ebp), %ecx      # The second parameter: buffer pointer.
        movl    16(%ebp), %edx      # The third parameter: display character number.
        movl    $SYSWRITE,%eax      # Put system call number 4 in %eax.
        int     $0x80               # Execute the system call.
        popl    %ebx
        movl    %ebp, %esp
        popl    %ebp
        ret
_myadd:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax       # Get the first parameter a.
        movl    12(%ebp), %edx      # Get the second parameter b.
        xorl    %ecx, %ecx          # If %ecx is 0, the calculation overflows.
        addl    %eax, %edx          # Perform additions.
        jo      1f                  # Jump if it overflows.
        movl    16(%ebp), %eax      # Take the third parameter pointer.
        movl    %edx, (%eax)        # Put the result in the position of the pointer.
        incl    %ecx                # No overflow occurred, so set no overflow return value.
1:      movl    %ecx, %eax          # %eax is the function return value.
        movl    %ebp, %esp
        popl    %ebp
        ret
```

The first function mywrite() in the assembly file uses the system interrupt 0x80 to call the system call sys_write(int fd, char *buf, int count) to display the information on the screen. The corresponding system call function number is 4 (see include/unistd.h). The three parameters are the file descriptor, the display buffer pointer, and the number of display characters. Before executing int 0x80, the caller function number (4) needs to be placed in the register %eax, and the registers %ebx, %ecx, and %edx should be stored as fd, buf, and count, respectively, according to the calling rules. The function argument mywrite() uses exactly the same number of parameters and uses as sys_write().

The second function myadd(int a, int b, int *res) performs an addition operation. The parameter res is the result of the operation. The function return value is used to determine if an overflow has occurred. If the return value is 0, the calculation has overflowed and the result is not available. Otherwise the result of the calculation will be returned to the caller via the parameter res.

Note that if you compile callee.s under the current Linux system (eg RedHat 9), remove the underscore '_' before the function name. The C program caller.c that calls these two functions is shown below.

```
/*
   Call assembly function mywrite(fd, buf, count) to display information;
   Call myadd (a, b, result) to perform addition. If myadd() returns 0, it indicates
   overflow. First, the start calculation information is displayed, and then the operation
   result is displayed.
*/
int main()
{
        char buf[1024];
        int a, b, res;
        char * mystr = "Calculating...\n";
        char * emsg  = "Error in adding\n";

        a = 5; b = 10;
        mywrite(1, mystr, strlen(mystr));
        if (myadd(a, b, &res)){
                sprintf(buf, "The result is %d\n", res);
                mywrite(1, buf, strlen(buf));
        } else {
                mywrite(1, emsg, strlen(emsg));
        }
        return 0;
}
```

The program first uses the assembly function mywrite() to display the information "Calculating..." on the screen, and then calls the addition calculation function myadd() to operate on the two numbers a and b, and on the third parameter res Returns the result of the calculation. Finally, use the mywrite() function to display the formatted result information string on the screen. If the function myadd() returns 0, it means that the overflow function has overflowed and the result of the calculation is invalid. The compilation and running results of these two files are shown below:

```
[/usr/root]# as -o callee.o callee.s
[/usr/root]# gcc -o caller caller.c callee.o
[/usr/root]# ./caller
Calculating...
The result is 15
[/usr/root]#
```

3.5 Linux 0.12 Object file format

To generate the kernel code, Linux 0.12 uses two compilers. The first is the assembler as86 and the corresponding linker (or linker) ld86. They are used exclusively for compiling and linking the 16-bit kernel boot sector program bootsect.s and the setup program setup.s running in real-address mode. The second is the GNU assembler as(gas) and the C compiler gcc and the corresponding linker gld. The compiler is used to generate the corresponding binary code and data object file for the source program file. The linker is used to combine all related object files to form a target file that can be loaded by the kernel, ie, an executable file.

This section begins with a brief description of the compiler-generated object file structure, and then describes how the linker combines the object file modules that need to be linked together to generate a binary executable image file or a large module file. Finally, it explains the generation principle and process of the Linux 0.12 kernel binary code file Image. This gives information on the a.out object file format supported by the Linux 0.12 kernel. As86 and ld86 generate the MINIX-specific object file format, which we will present in the chapter on kernel creation tools that deal with this format. Because the MINIX object file structure is similar to the a.out object file format, it will not be described here. The basic working principle of the object file and linker program can be found in the book "Linkers & Loaders" by John R. Levine.

For convenience of description, the object file generated by the compiler is called an object module file (abbreviated as a module file), and the executable object file generated by the link program is called an executable file. And all of them are collectively referred to as object files.

3.5.1 Object file format

In the Linux 0.12 system, the UNIX module's traditional a.out format is used by both the GNU gcc or gas compiler output object module file and the linker generated executable file. This is an object file format called Assembly & Linker Editor Output. For a system with a memory paging mechanism, this is a simple and effective object file format. The a.out format file consists of a file header and subsequent code sections (also called text sections), initialized data sections (also called data sections), relocation information section, symbol tables and symbol names String composition, as shown in Figure 3-7. The code section and the data section are usually also referred to as a text segment (code segment) and a data segment, respectively.

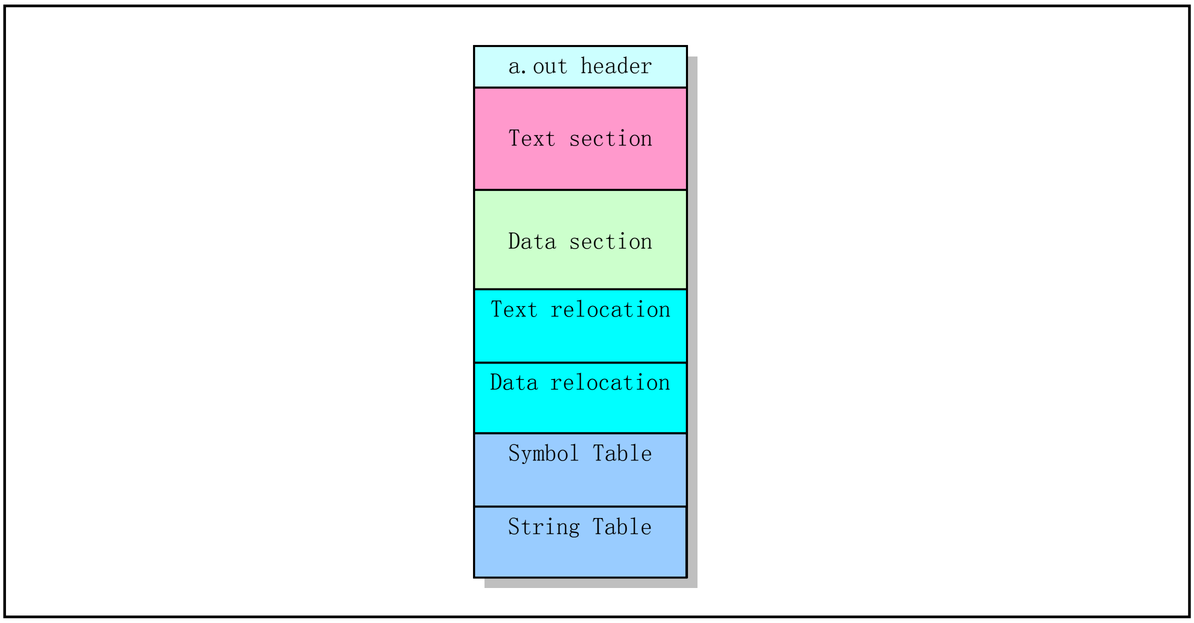


Figure 3-7 a.out format object file

The basic definitions and uses of the 7 sections of the a.out format are:

- Exec header. Execute file header section. This section contains some parameters (exec structure), which is the overall structure information about the target file. For example, the length of the code and data area, the length of the uninitialized data area, the corresponding source file name, and the target file creation time. The kernel uses these parameters to load execution files into memory and execute them, and the linker (ld) uses these parameters to combine some of the module files into an executable file. This is the only necessary part of the target document.

- Text segment. The binary instruction code and data information generated by the compiler or assembler contains the instruction code and related data that are loaded into memory when the program is executed. Can be loaded as read-only.
- Data segment. Binary instruction code and data information generated by a compiler or assembler. This section contains data that has already been initialized and is always loaded into readable and writable memory.
- Text relocations. This section contains record data for use by the linker. Used to locate a pointer or address in a code segment when combining object module files. When the linker needs to change the address of the target code, it needs to be corrected and maintained.
- Data relocations. Similar to the role of the code relocation section, but used for relocation of pointers in the data segment.
- Symbol table. This section also contains record data for use by the linker. These record data hold the global symbols defined in the module file and the symbols that need to be input from other module files, or the symbols defined by the linker, used to cross named variables and functions (symbols) between module files. References.
- String table. This section contains the string corresponding to the symbol name. Used to debug program debugging target code, regardless of the linking process. This information can include source code and line numbers, local symbols, and data structure description information.

For a given target file, not all of the above information is necessarily included. Since the Linux 0.12 system uses the memory management function of the Intel CPU, it allocates a separate 64MB address space (logical address space) for each execution program. In this case, because the linker has processed the execution file to start from a fixed address, the relocation information is no longer needed in the relevant executable file. Below we explain some of the important areas or parts.

3.5.1.1 Executive header

In the header portion of the target file, there is a 32-byte exec data structure, commonly referred to as a file header structure or an execution header structure. Its definition is as follows. For more information about the a.out structure, see the introduction after the include/a.out.h file.

```
struct exec {
  unsigned long a_magic;        /* Use macros N_MAGIC, etc for access */
  unsigned a_text;              /* length of text, in bytes */
  unsigned a_data;              /* length of data, in bytes */
  unsigned a_bss;               /* length of uninitialized data area for file, in bytes */
  unsigned a_syms;              /* length of symbol table data in file, in bytes */
  unsigned a_entry;             /* start address */
  unsigned a_trsize;            /* length of relocation info for text, in bytes */
  unsigned a_drsize;            /* length of relocation info for data, in bytes */
}
```

According to the value of the magic number field of the header structure in the a.out file, we can divide the a.out file into several types. The Linux 0.12 system uses two types: The module object file uses the OMAGIC (Old Magic) type of a.out format, which indicates that the file is an object file or an impure executable file. The magic number is 0x107 (octal 0407). The executable file uses the ZMAGIC a.out format, which indicates that the file is an executable file for demand-driven (demang-paging, ie, load on demand) loading. The magic number is 0x10b (octal 0413). The main difference between these two formats is the way they allocate storage to each part. Although the total length of the structure is only 32 bytes, for a ZMAGIC type executable file, the beginning portion of the file requires a space of 1024 bytes for the head structure. Except 32 bytes occupied by the header structure, the rest is 0. The text segment and data segment of the program are only placed after 1024 bytes. For an OGMIC type .o module file, the 32-byte header structure at the beginning of the file is followed by the code area and the data area.

The a_text and a_data fields in the execution header structure indicate the byte lengths of the read-only code segment and the read-write data segment, respectively. The a_bss field indicates the length of the uninitialized data area (bss section) following the data segment when the kernel loads the target file. Since Linux automatically zeros memory when allocating memory, the bss section does not need to be included in a module file or an executable file. In order to visually represent that the target file logically has a bss segment, a dashed box will be used in the later illustration to represent the bss segment in the target file.

The a_entry field specifies the address at which the program code begins execution, while the a_syms, a_trsize, and a_drsize fields describe the size of the relocation information for the symbol table, code, and data segments after the data segment, respectively. Symbol tables and relocation information are not required for executable files, so unless the linker includes symbol information for debugging purposes, the fields in the execution file are usually zero.

3.5.1.2 Relocation information section

The Linux 0.12 system's module files and executable files are all object files in the a.out format, but only the compiler-generated module files contain relocation information for linking programs. The relocation information of the code segment and the data segment is composed of relocation records (items). The length of each record is 8 bytes. Its structure is as follows.

```
struct relocation_info
{
  /* Address (within segment) to be relocated.  */
  int r_address;
  /* The meaning of r_symbolnum depends on r_extern.  */
  unsigned int r_symbolnum:24;
  /* Nonzero means value is a pc-relative offset
     and it should be relocated for changes in its own address
     as well as for changes in the symbol or section specified.  */
  unsigned int r_pcrel:1;
  /* Length (as exponent of 2) of the field to be relocated.
     Thus, a value of 2 indicates 1<<2 bytes.  */
  unsigned int r_length:2;
  /* 1 => relocate with value of symbol.
          r_symbolnum is the index of the symbol
          in file's the symbol table.
     0 => relocate with the address of a segment.
          R_symbolnum is N_TEXT, N_DATA, N_BSS or N_ABS
          (the N_EXT bit may be set also, but signifies nothing).  */
  unsigned int r_extern:1;
   /* Four bits that aren't used, but when writing an object file
      it is desirable to clear them.  */
  unsigned int r_pad:4;
};
```

There are two functions for relocating items. One is that when the code segment is relocated to a different base address, the relocation item is used to indicate where it needs to be modified. Second, when there is a reference to an undefined symbol in the module file, the linker can use the corresponding relocation item to correct the value of the symbol when the undefined symbol is finally defined. It can be seen from the structure of the above relocation record items that each record item contains an address in the code section (code segment) and data section (data segment) of the module file that needs to be relocated at a length of 4 bytes and specifies how to weigh Positioning operation information. The address field r_address refers to the offset value of the relocatable item from the beginning of the code segment or data segment. The 2-bit length field r_length indicates the length of the relocated entry, and 0 to 3 indicates that the width of the relocated entry is 1 byte, 2 bytes, 4 bytes, or 8 bytes, respectively. The flag r_pcrel indicates that the relocated item is a "PC related" item, ie it is used as a relative address in the instruction. The external flag r_extern controls the meaning of r_symbolnum and indicates whether the relocation entry refers to a segment or a symbol. If the flag value is 0, the relocation entry is a normal relocation entry, and the r_symbolnum field specifies in which segment the positioning is addressed. If the flag is 1, then the relocation entry is a reference to an external symbol. In this case, r_symbolnum specifies a symbol in the symbol table in the target file and needs to be relocated using the value of the symbol.

3.5.1.3 Symbol Table and String Section

The last part of the target file is the symbol table and the related string table. The structure of the symbol table entry is as follows.

```
struct nlist {
  union {
    char         *n_name;          // String pointer,
    struct nlist *n_next;          // Or a pointer to another symbolic item structure,
    long          n_strx;          // Or the byte offset value of the symbol name in the table.
  } n_un;
  unsigned char n_type;            // This byte is divided into 3 fields. see a.out.h file.
  char          n_other;           // Usually not used.
  short         n_desc;            //
  unsigned long n_value;           // Symbol’s value.
};
```

Since the GNU gcc compiler allows arbitrary-length identifiers, the identifier strings are located in a string table after the symbol table. Each symbol table entry has a length of 12 bytes, where the first field gives the symbol name string (null-terminated) offset from the string table. The type field n_type indicates the type of symbol. The last bit of this field is used to indicate if the symbol is external (global). If this bit is 1, then the symbol is a global symbol. The linker does not need local symbol information, but it can be used by the debugger. The remaining bits of the n_type field are used to indicate the symbol type. The a.out.h header file defines these types of value constant symbols. The main types of symbols include:

- text, data, or bbs indicates the symbols defined in this module file. The value of the symbol at this time is the relocatable address of the symbol in the module.
- abs indicates that the symbol is an absolute (fixed) non-relocatable symbol. The value of the symbol is the fixed value.
- undef indicates that it is an undefined symbol in this module file. The value of the symbol at this time is usually 0.

However, as a special case, the compiler can use an undefined symbol to ask the linker to reserve a memory space for the specified symbolic name. If an undefined external (global) symbol has a non-zero value, then for the linker the value is the size of the memory that the program wishes to specify for symbolic addressing. During the linking operation, if the symbol is really not defined, then the linker creates a memory space for the symbol name in the bss section. The size of the space is the largest value of the symbol in all linked modules. This is the so-called Common block definition in the bss section. It is mainly used to support uninitialized external (global) data. For example, an uninitialized array defined in the program. If the symbol has already been defined in any module, the linker will use this definition and ignore the value.

3.5.2 Target file format in Linux 0.12

In the Linux 0.12 system, we can use the objdump command to view the specific values of the file header structure in the module file or executable file. For example, the following lists the specific values of the headers in the hello.o object file and its executable file.

```
[/usr/root]# gcc -c -o hello.o hello.c
[/usr/root]# gcc -o hello hello.o
[/usr/root]#
[/usr/root]# hexdump -x hello.o
0000000    0107    0000    0028    0000    0000    0000    0000    0000
0000010    0024    0000    0000    0000    0010    0000    0000    0000
0000020    6548    6c6c    2c6f    7720    726f    646c    0a21    0000
0000030    8955    68e5    0000    0000    e3e8    ffff    31ff    ebc0
0000040    0003    0000    c3c9    0000    0019    0000    0002    0d00
0000050    0014    0000    0004    0400    0004    0000    0004    0000
0000060    0000    0000    0012    0000    0005    0000    0010    0000
0000070    0018    0000    0001    0000    0000    0000    0020    0000
0000080    6367    5f63    6f63    706d    6c69    6465    002e    6d5f
0000090    6961    006e    705f    6972    746e    0066
000009c
[/usr/root]# objdump -h hello.o
hello.o:
magic: 0x107 (407)machine type: 0 flags: 0x0 text 0x28 data 0x0 bss 0x0
nsyms 3 entry 0x0 trsize 0x10 drsize 0x0
[/usr/root]#
[/usr/root]# hexdump -x hello | more
0000000    010b    0000    3000    0000    1000    0000    0000    0000
0000010    069c    0000    0000    0000    0000    0000    0000    0000
0000020    0000    0000    0000    0000    0000    0000    0000    0000
*
0000400    448b    0824    00a3    0030    e800    001a    0000    006a
0000410    dbe8    000d    eb00    00f9    6548    6c6c    2c6f    7720
0000420    726f    646c    0a21    0000    8955    68e5    0018    0000
......
--More--q
[/usr/root]#
[/usr/root]# objdump -h hello
hello:
magic: 0x10b (413)machine type: 0 flags: 0x0 text 0x3000 data 0x1000 bss 0x0
nsyms 141 entry 0x0 trsize 0x0 drsize 0x0
```

```
[/usr/root]#
```

It can be seen that the magic number of the hello.o module file is 0407 (OMAGIC), and the code segment immediately follows the header structure. In addition to the header structure, a code segment of 0x28 bytes in length and a symbol table with 3 symbol items and code segment relocation information of 0x10 bytes in length are included. The rest of the segments are 0 in length. The magic number of the corresponding execution file hello is 0413 (ZMAGIC), and the code segment is stored from the file offset position 1024 bytes. The lengths of the code and data segments are 0x3000 and 0x1000 bytes, respectively, with a symbol table containing 141 items. We can use the command strip to delete the symbol table information in the execution file. For example, below we delete the symbol information in the hello execution file. It can be seen that the length of the symbol table of the hello execution file becomes 0, and the length of the hello file is also reduced from the original 20591 bytes to 17412 bytes.

```
[/usr/root]# ll hello
-rwx--x--x   1 root     4096        20591 Nov 14 18:30 hello
[/usr/root]# objdump -h hello
hello:
magic: 0x10b (413)machine type: 0flags: 0x0text 0x3000 data 0x1000 bss 0x0
nsyms 141 entry 0x0 trsize 0x0 drsize 0x0

[/usr/root]# strip hello
[/usr/root]# ll hello
-rwx--x--x   1 root     4096        17412 Nov 14 18:33 hello
[/usr/root]# objdump -h hello
hello:
magic: 0x10b (413)machine type: 0flags: 0x0text 0x3000 data 0x1000 bss 0x0
nsyms 0 entry 0x0 trsize 0x0 drsize 0x0
[/usr/root]#
```

Figure 3-8 shows the correspondence between the areas of the a.out executable file in the process logical address space on the disk. The logical space size of a process in a Linux 0.12 system is 64 MB. For the ZMAGIC a.out executable file, its code area is an integer multiple of the memory page size. Since the Linux 0.12 kernel uses the Demand-paging technique, which means that a page of code is actually loaded into a physical memory page, it is only set for the fs/execve() function of the load operation. The paging mechanism of page directory entries and page table entries, so demand page technology can speed up the loading process.

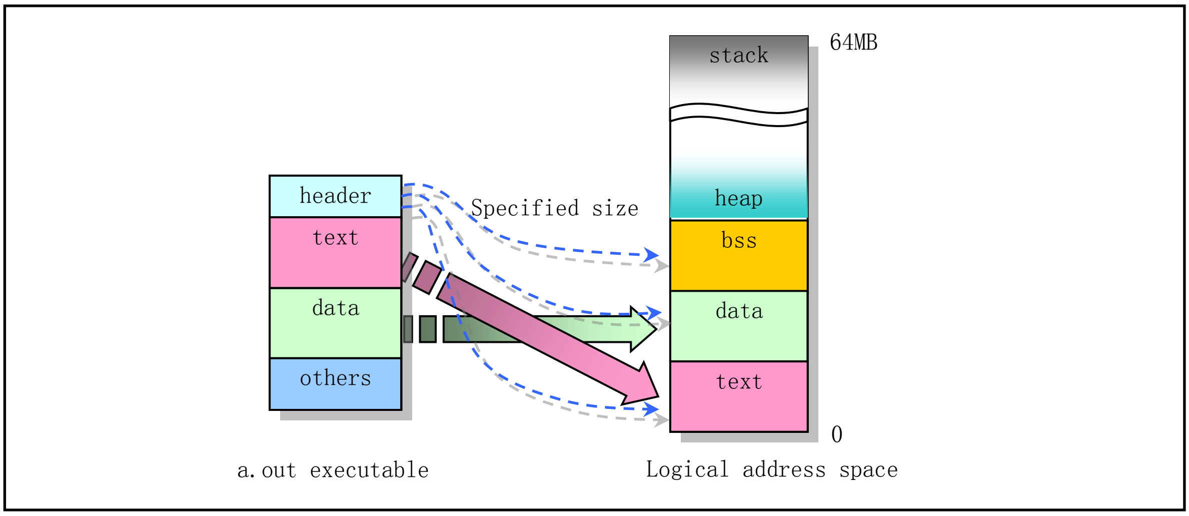


Figure 3-8 a.out execution file maps to process logical address space

In the figure, bss is the uninitialized data area of the process and is used to store static uninitialized data. The first page of bss memory will be set to all 0s at the beginning of program execution. The heap in the figure is a heap space area, which is used to allocate the memory space dynamically requested by the process during execution.

3.5.3 Linker output

The linker processes one or more input object files and related library function objects, and ultimately generates a corresponding binary execution file or a large module file composed of all modules. In this process, the link program's primary task is to perform storage space allocation operations for the execution file (or output module file). Once the storage location is determined, the linking program can continue to perform symbol-bonding operations and code revision operations. Because most of the symbols defined in the module file are related to the storage location in the file, there is no way to resolve the symbol before the corresponding position of the symbol is determined.

Each module file includes several types of segments. The second task of the linker is to join together the segments of the same type in all modules and form a single segment for the specified segment type in the output file. For example, the linker needs to merge the code segments from all the input module files into a single segment and place it in the output executable file.

For a.out format module files, since the segment types are known in advance, the linker can easily store and allocate a.out format module files. For example, for the case of having two input module files and the need to connect a library function module, its storage allocation is shown in Figure 3-9. Each module file has a code section, a data section, and a bss section. There may also be some common blocks that appear to be external (global) symbols. The linker collects the size of each of the module files, including code segments, data segments, and bss segments in any library function module. After all the modules have been read in and processed, any unresolved external symbols with a non-zero value will be treated as common blocks and their allocations will be stored at the end of the bss section.

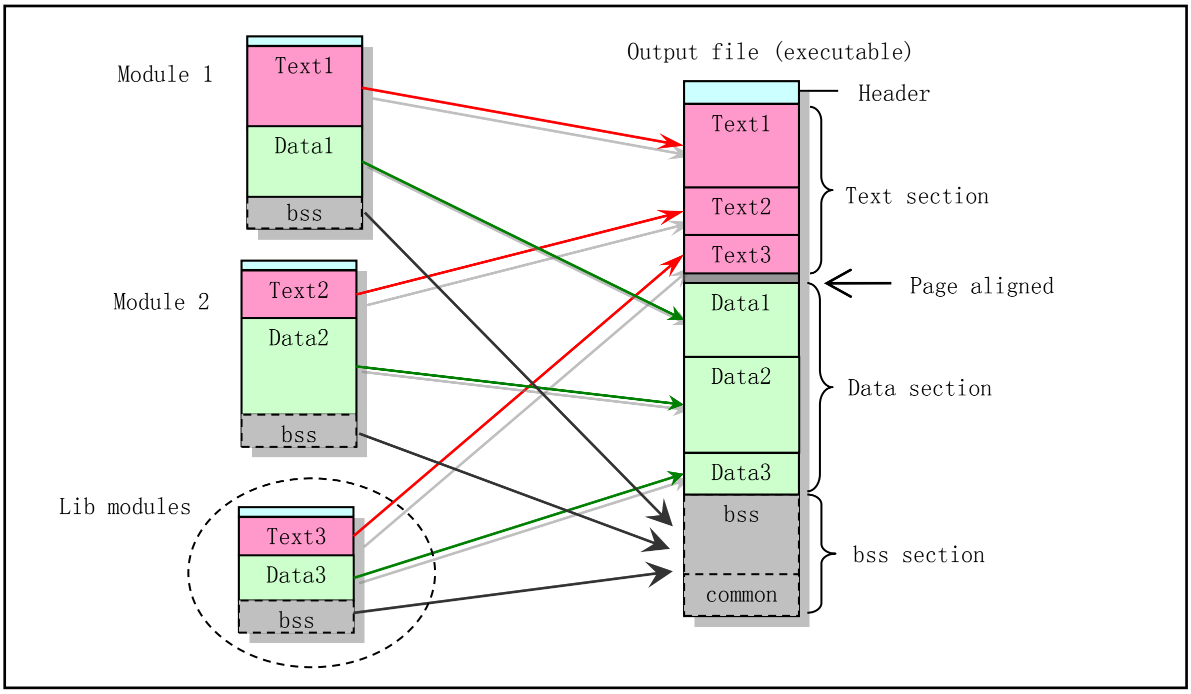


Figure 3-9 Object files link operation

The linker can then assign addresses to all segments. For the ZMAGIC type a.out format used in the Linux 0.12 system, the code segment in the output file is set to start at fixed address 0. The data segment starts from the next page boundary after the code segment. The bss section immediately follows the data segment. Within each segment, the linker stores the same type of segments in the input module file and aligns them by word.

When Linux 0.12 kernel loads an executable file, it will first determine whether the file is a suitable executable file based on the information in the file header structure, that is, if the magic number type is ZMAGIC, then the system is at the top of the user mode stack. The program sets the environmental parameters and parameter information blocks entered on the command line and builds a task data structure for it. Then use the stack return technique to execute the program after setting some related register values. The code and data in the execution program image file will be dynamically loaded into memory using Load on Demand when actually executed or used.

For the Linux 0.12 kernel compilation process, it is done using the make command to command the compiler and linker operations based on the kernel configuration file Makefile. In the build process, make also uses the build.c program in the kernel source code in the tools/ directory to compile and build a temporary tool program build that is used to combine all modules. Since the kernel is loaded into memory by the boot-up program using the ROM BIOS interrupt call, the execution header structure in the compiled kernel modules needs to be removed. The main function of the utility program build is to remove the execution header structures in the bootsect, setup, and system files, and then combine them sequentially to produce a kernel image file named Image.

3.5.4 Linker Predefined Variables

During linking, the linker ld and ld86 use variables to record the logical address of each segment in the execution program. Therefore, in the program, you can access the external variables to obtain the position of the program's middle segment. The linker's predefined external variables are usually at least etext, _etext, edata, _edata, end, and _end.

The addresses of the variable names _etext and etext are the first address after the program text segment ends; the addresses of _edata and edata are the first address after the initial data area; the addresses of _end and end are after the uninitialized data area (bss) The first address location. Names prefixed with the underscore '_' are equivalent to the underlined counterparts. The only difference between them is that the symbols etext, edata, and end are not defined in the ANSI, POSIX, and other standards.

When the program just begins to execute, its brk location is in the same position as _end. But system calls sys_brk(), memory allocation functions malloc(), and standard input/output operations will change this position. Therefore, the current brk position of the program needs to be obtained using sbrk(). Note that these variable names must be treated as addresses. Therefore, when accessing them, you need to use the address prefix '&', such as &end. E.g:

```
    extern int _etext;
    int et;

    (int *) et = &_etext;          // et contains the address after the end of text segment.
```

The following program predef.c can be used to display the addresses of these variables. It can be seen that the address value is the same for the band and the underscore ’_’ symbol.

```
/*
  Print the symbols predefined by linker.
*/
extern int end, etext, edata;
extern int _etext, _edata, _end;
int main()
{
        printf(″&etext=%p,  &edata=%p,  &end=%p\n″,
                &etext, &edata, &end);
        printf(″&_etext=%p, &_edata=%p, &_end=%p\n″,
                &_etext, &_edata, &_end);
        return 0;
}
```

Running this program on a Linux 0.1X system gives the following results. Please note that these addresses are logical addresses in the program's address space, which is the address from when the execution program was loaded into the memory location.

```
[/usr/root]# gcc -o predef predef.c
[/usr/root]# ./predef
&etext=4000,  &edata=44c0,  &end=48d8
&_etext=4000, &_edata=44c0, &_end=48d8
[/usr/root]#
```

If you run this program on a modern Linux system (such as RedHat 9 or later), the following results can be obtained. We know that the program code in the Linux system is now stored from its logical address 0x08048000, so we can see that the program's code segment length is 0x41b bytes.

```
[root@plinux]# ./predef
&etext=0x804841b,  &edata=0x80495a8,  &end=0x80495ac
&_etext=0x804841b, &_edata=0x80495a8, &_end=0x80495ac
[root@plinux]#
```

When the Linux 0.1x kernel initializes the block device cache (fs/buffer.c), it uses the variable name _end to obtain the location of the kernel image file Image in memory at the end of the file, and from this position, the high speed is set. Buffer.

3.5.5 System.map file

When running the GNU linker gld(ld), if the '-M' option is used, or the nm command is used, the link map information is printed on the standard output device (usually the screen). The target program memory address map information generated by the linker. It lists the location information of the program segment loaded into memory. Specifically, there is the following information:

- Locations of object files and symbol information mapped in memory;
- How public symbols are placed;
- All file members included in the link and their referenced symbols.

Usually we will redirect the link image information sent to the standard output device to a file (eg System.map). When compiling the kernel, the System.map file generated by the linux/Makefile file is used to store kernel symbol table information. The symbol table is a list of all kernel symbols and their corresponding addresses. Of course, it also includes address information of symbols such as _etext, _edata, and _end described above. With each kernel compilation, a new corresponding System.map file is generated. When an error occurs in the kernel, the variable name corresponding to an address value can be found by parsing the symbol table in the System.map file, or vice versa.

By using the System.map symbol table file, we can get more easily identified information when a kernel or related program fails. Examples of symbol tables are as follows:

```
c03441a0 B dmi_broken
c03441a4 B is_sony_vaio_laptop
c03441c0 b dmi_ident
c0344200 b pci_bios_present
c0344204 b pirq_table
```

Each line describes a symbol, the first column indicates the symbol value (address), the second column is the symbol type, indicates which section of the target file the symbol is located in or its attributes, and the third column is the corresponding symbol name.

The symbol type indicators in the second column usually have the types shown in Table 3-5, and there are some related to the target file format adopted. If the symbol type is a lowercase character, the symbol is local; if it is an uppercase character, the symbol is global (external). See the definition of the nlist{} structure n_type field in the file include/a.out.h (lines 110-185).

Table 3-5 The symbol type in the target file symbol list file

Symbol type	Name	Description

A	Absolute	The value of the symbol is an absolute value and will not be changed during further linking.
B	BSS	The symbols are in the uninitialized data section, ie in the BSS section.
C	Common	The symbol is public. Public symbols are uninitialized data. When linking, multiple public symbols may have the same name. If the symbol is defined elsewhere, the public symbol is treated as an undefined reference.
D	Data	The symbol is in the initialized data section.
G	Global	The symbol is in the initialized data section of the small object. The format of some object files allows more efficient access to small data objects such as a global integer variable.
I	Inderect	A symbol is an indirect reference to another symbol.
N	Debugging	The symbol is a debugging symbol.
R	Read only	The symbol is in a read-only data section.
S	Small	symbol in a small object's uninitialized data section.
T	Text	Symbols in code sections.
U	Undefined	The symbol is external and its value is 0 (undefined).
-	Stabs	The symbol is a stab symbol in the a.out object file and is used to save debugging information.
?	Unknwon	The type of symbol is unknown or related to a specific file format.

It can be seen that the variable named dmi_broken is located at kernel address 0xc03441a0.

System.map is where the software that uses it (such as the kernel logging daemon klogd) can find it. When the system is started, klogd will search System.map in three places if the location of System.map is not given as klogd in the form of a parameter. as followed:

```
/boot/System.map
/System.map
/usr/src/linux/System.map
```

Although the kernel itself does not actually use System.map, other programs, such as klogd, lsof, ps, and other software like dosemu, require a correct System.map file. Using this file, these programs can find the corresponding kernel variable name based on the known memory address to facilitate debugging of the kernel.

3.6 Make Command and Makefile

As you can see from the examples given above, when creating an executable file that is generated by one or a few source programs, you only need to type a few simple lines of commands. But for a large program such as a kernel that consists of hundreds or even thousands of source files, compiling all code files by manually typing lines of commands is extremely complicated. The make command is the best tool designed to automatically handle this type of situation. Its main purpose is to be able to automatically determine which files need to be recompiled in a large project containing many source files and issue these files with a recompilation command. Let's take the compiling C program as an example to illustrate how to use Make briefly, but you can apply it to any programming language that can be compiled using shell commands. For detailed usage of make, please refer to "GNU make manual".

In order to use the make tool, we need to write a text file named Makefile (or makefile) for make execution.

The Makefile mainly contains some execution rules and commands required to tell make the relationships among files and what compile and link operations are required for the source files involved to generate the corresponding target files.

When "make" recompiles the source files, each modified source file is recompiled. If the header file has changed, each source file containing the header file will be recompiled. Each compilation generates an object file that corresponds to the source file. Finally, if all source files have been recompiled, all object files, whether newly created or saved from a previous compilation, will be linked together to produce a new executable.

3.6.1 Contents of the Makefile

A Makefile can include five elements: explicit rules, implicit rules, variable definitions, directives, and comments.

- Explicit rules are used to specify when and how to recompile one or more files called rule’s targets. The rules explicitly list the other files on the target that depend on the prerequisites (or dependencies), as well as the commands for creating or updating targets.
- Implicit rules are based on the name of the target and the object to determine when and how to recompile one or more files called targets of the rule. This rule describes how the target depends on a file that is similar to the target name and will be given to create or update such a target file.
- Variable definitions define a text string for a variable on one line. This variable can be replaced in subsequent statements. For example, the variables object in the example below defines a list of all .o files.
- A directive is a command of make that indicates the specific operation that it performs when it reads a Makefile. These operations can include reading another makefile; determining whether to use or ignore a portion of the makefile and defining a variable from a string containing multiple lines.
- Comments are the parts of the text in the Makefile that begin with the '#' character. If we really need to use the '#' character, we need to escape it by adding a backslash character ('\#') before the character. Comments can appear anywhere in the Makefile. In addition, a command line script in the Makefile that begins with the TAB is passed to the shell in its entirety, and the shell determines whether it is a command or just a comment.

Once we have written an appropriate Makefile, we can simply type "make" each time we modify the source code to perform all necessary program updates. make determines which files need to be updated (recompiled) based on the contents of the Makefile and the last updated time of the file. For each file that needs to be updated, make executes the relevant commands recorded in the Makefile.

3.6.2 Rules in the Makefile File

A simple Makefile contains some of the following rules. These rules are mainly used to describe the dependencies between operating objects (source files and object files).

```
target ...: prerequisites ...
        command
        ...
        ...
```

Among them, the target object usually refers to the name of a file generated by the program, for example, it can be an executable file or an object file that ends with ".o". The target can also be the name of the activity to be taken, for example "clean".

The prerequisite is a series of files or other targets necessary to create a target. The target usually depends on multiple such necessary or target files.

The command (command) refers to the operations performed by make, which are usually shell commands that generate the target. When the last modification time of one or more files in the prerequisite condition is newer than that of the target file, the command of the rule will be executed. In addition, there can be multiple commands in one rule, each command occupies a single line in the rule. Please note that we need to type a tab character (press Tab) before writing each command!

Typically, the command exists in a rule with prerequisites and is used to create a target file when any of the prerequisites change. However, the rules do not necessarily have preconditions. For example, a rule containing a delete command related to the target "clean" does not contain prerequisites.

A rule explains how and when to remake certain files, which are the targets of some specific rules. make will execute the command based on the prerequisites to create or update the target. A rule can also explain how and when to perform an operation.

In addition to the rules, a Makefile can also contain other text. However, for a simple Makefile it is usually sufficient to include only a few rules. Some rules may be more complex than the ones given earlier, but they are basically similar.

3.6.3 **A Simple Makefile**

Below we discuss a simple Makefile that describes how to compile and link a text editor program consisting of eight C source files and three header files.

When make recompiles C files based on the contents of the Makefile, only the modified C files are recompiled. Of course, if a .h header file is modified, then in order to ensure that the program is compiled correctly, every C code file that contains the header file is recompiled. Each compilation operation produces a target file that corresponds to the source program. The net result is that if any of the modified source code files are compiled, then all .o object files that are generated (including those that were just compiled and unmodified before the source code is compiled) need to be linked together to generate a new one. Executable editor program.

The contents of the Makefile example file describe how an executable named edit depends on 8 object files, and how the 8 object files depend on 8 C source files and 3 header files. In this example, all C files contain the "defs.h" header file, but only those C files that define the edit command contain "command.h" and only the underlying C file that changes the edit buffer contains "buffer.h" "head File.

```
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
```

```
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

To use the Makefile to create the "edit" executable file, simply type “make” on the command line.

To use the Makefile to remove the compiled executable file and all object files from the current directory, just type “make clean”.

In the Makefile, the target of the rule includes the execution file "edit" and the .o object file "main.o", "kbd.o", and the like. Prerequisite (or dependent) files are source files such as "main.c" and "defs.h". In fact, we can see that each ".o" file is both a rule goal and a necessary prerequisite file for another rule. The commands include "cc -c main.c" and "cc -c kbd.c".

When the target is a file, then any dependent files in its prerequisites need to be recompiled or linked when they are modified. Of course, the file in the precondition that is itself an object should be updated first. In this example, "edit" depends on eight .o target files; and the .o target file "main.o" depends on the source file "main.c" and the header file "defs.h".

On the next line of the rule's target and prerequisites in the Makefile is the shell command. These shell commands indicate how to use the files in the prerequisite to update or generate the target object file. Note that we need to type a tab before each command line to distinguish the command line and other lines in the Makefile. What make does is execute the commands in the rule when the target needs to be updated.

The target "clean" is not a file, but just the name of an operation (activity). Because we generally do not require that this action be performed in its rules, "clean" is not a prerequisite for any other rule. The result is that make does not enforce this rule unless it is explicitly stated. Note that this rule (target) is not only a prerequisite for any other rule, it does not contain nor does it require any prerequisites. So the sole purpose of this rule is to execute the specified command. For such a rule, its target does not refer to or depend on any other file, but only indicates a specific operation. This target is called a phony target.

3.6.4 How make handles Makefile

By default, make will start from the first target in the Makefile (not including targets starting with '.'). This first goal is called the default goal of the Makefile. The ultimate goal is to make an effort to try to update the target.

In the above example, the default end goal is to update or create the execution program "edit", so we put the corresponding rules at the top of the Makefile. When we type the command make on the command line, make will read the Makefile and start processing the first rule. In the example, the first rule is to re-link to generate "edit", but before make can completely process this rule, it must first process the rules of the file that "edit" depends on. In the example, you need to first create or update those .o object files. Each .o file will be processed according to its own rules, that is, by compiling the respective source files to generate the respective .o object files. If any source or header file that is a target prerequisite is newer than the .o object files or the .o object files do not exist, recompilation is required to update or create the corresponding .o object file. .

Some of the other rules in the Makefile will also be processed if their goals (files) appear in the prerequisites of the final target. If the final goal (or any goal) does not depend on some other rule, make will not process these rules unless we actively request make to handle it. For example, when running make, we can give the target name of a specific rule in the Makefile on the command line to perform the specified update operation, such as using the command “make clean”.

Before recompiling a .o object, make first considers updating its prerequisites, source files, and header files. However, the Makefile does not specify any operation for the source and header files, ie, the source and header files are not the target of any rules, so make will not perform any processing on these source files.

After recompiling the desired .o object file, make decides whether to perform a relink to generate an updated edit program "edit". This will only be done if "edit" does not exist or if any .o target file is newer than "edit". If an .o object file has just been recompiled, it will be newer than "edit" and make will be relinked to generate a new "edit".

Therefore, if we modify the file "insert.c" and run make, make compiles the source file to update the corresponding "insert.o" and then links "edit". If we modify the header file "command.h" and run make, make will recompile the target files "kbd.o", "command.o" and "files.o" and then link to generate a new executable file "edit".

In general, make will use the contents of the Makefile to determine which .o object files need to be updated, and then determine which of the target files do need to be updated. If the .o object file is newer than all of its related files, the .o object is already up-to-date and no further updates are required. Of course, all necessary targets in the input condition (prerequisite) as the first final target are updated beforehand.

3.6.5 Variables in the Makefile

In the above example, we need to list all .o target files twice in the "edit" rule (see below):

```
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

This duplicate information is easy to make mistakes. If we add a new .o object file to the program, it is possible to add the .o object file name to a list but forget to add it in another place. By using a variable, we are likely to reduce the risk of this error, and also make the Makefile look more concise. Using variables allows us to define a text string once, which can then be replaced in several places.

For Makefiles, the typical practice is to define a variable named objects or OBJECTS to represent a list of all .o object files. We usually use the following line in the Makefile to define a variable objects:

```
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

After that, in every place where you need to list the .o object files, you can replace the value of the variable by writing "$(objects)".

3.6.6 Let make automatically deduce commands

We do not need to give the relevant commands in the rules in order to compile each C source program. Because make itself can judge it: it has an implied rule, which uses the 'cc -c' command according to the naming of the target file, and updates the corresponding .o file according to the corresponding .c file. For example, it compiles 'main.c' to 'main.o' using the command 'cc -c main.c -o main.o'. Therefore we can omit the commands in the .o object file rules.

When a .c file is used automatically in this way, it is automatically added to the prerequisites (dependencies). So we can omit the `.c' file in the rule preconditions --- Suppose we also omit the command. The following is a complete Makefile example that includes these two changes and uses variables:

```
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

clean :
        -rm edit $(objects)
```

This is how we actually write the Makefile file. Because implied rules are so convenient, it is important. We will often see using them.

3.6.7 Implicit rules in automatic variables

If one of the prerequisites (dependent objects) is searched for by searching the directory and is found in another directory, the command of the rule will be executed as scheduled. Therefore, we must carefully set the command so that the command can find the necessary prerequisites in this directory. This can be done by using automatic variables. Automatic variables that are implicit rules are variables that can be automatically replaced on the command line depending on the situation. The value of the automatic variable is set before the regular command is executed. For example, the value of the automatic variable '$^' represents all the prerequisites for the rule, including the name of the directory they are in; the value of '$<' represents the first prerequisite in the rule; '$@' represents the target object (for other automatic variables, see the make manual). Sometimes, when we don't want to specify a header file on the command line, we can include these header files in the prerequisites. At this point, the automatic variable '$<' is the first prerequisite.

```
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $< -o $@
```

The '$<' will be automatically replaced with foo.c and $@ will be replaced with foo.o.

In order for make to use idioms to update a target, we can eliminate the need for commands. Write a rule without a command or do not write a rule. At this point, make will determine which implicit rule to use based on the type of source file (file suffix).

In addition, there is an implicit rule called a suffix rule. It is an old-fashioned way of defining implicit rules for make (now that this rule is no longer used, instead using more general and clearer pattern matching rules). Since this rule is used in the Makefile of the Linux 0.1x kernel, here's a brief explanation. The following example is an application of a double suffix rule. Dual suffix rules are defined with a pair of suffixes: source suffix and target suffix. The corresponding implicit prerequisite is obtained by replacing the target suffix with the source suffix in the file name. Therefore, the following '$<' value at this time is the '*.c' file name. The meaning of the positive make rule is to compile the '*.c' program into the '*.s' code.

```
.c.s:
        $(CC) $(CFLAGS) \
        -nostdinc -Iinclude -S -o $*.s $<
```

Usually the command belongs to a rule with preconditions (dependency objects) and is used to generate a target file when any of the prerequisites change. However, the rules that specify commands for goals do not necessarily have prerequisites. For example, rules related to the target 'clean' with a delete command do not require a prerequisite. At this point, a rule explains how and when to re-create certain files, which are the targets of specific rules. Make executes commands based on prerequisites to create or update targets. A rule can also explain how and when to perform an operation.

Makefiles can also contain text other than rules, but simple Makefiles need only contain the appropriate rules. The rules may seem much more complex than the template shown above, but they are basically consistent.

The Makefile file can also contain some referenced dependencies between the files. These dependencies are used by make to determine if a target needs to be rebuilt. For example, when a header file is changed, make recompiles all '*.c' files associated with the header file with these dependencies. For an example of a dependency, refer to the Makefile in the kernel source code.

3.7 Summary

In this chapter, several executable assembly language programs are used as description objects, and the basic language and usage of as86 and GNU as assembly language are described in detail. At the same time, the C language extensions used by the Linux kernel are described in detail. For learning the operating system, the object file structure supported by system has a very important role, so this chapter describes the a.out object file format used in Linux 0.12 in detail.

In the next chapter, we will detail the operating principle of the Intel 80X86 processor running in protected mode. Given an example of a protection mode multitasking program, reading this example will give us a basic understanding of how the operating system initially "rotates" and lays a solid foundation for continuing to read the full Linux 0.12 kernel source code.

4 80X86 Protection Mode and Its Programming

The Linux operating system introduced in this book is based on a PC system consisting of an Intel 80X86 processor and related peripheral hardware.The best reference for 80X86 CPU system programming is of course the three-volume "IA-32 Intel Architecture Software Developer's Manual" released by Intel Corporation, especially the third volume: "System Programming Guide". It is an essential reference for understanding the operating principle of the 80X86 CPU or system programming. These information can be downloaded for free from the Intel Corporation website. This chapter mainly describes the architecture of the 80X86 CPU and some basic knowledge of programming in protected mode, laying a solid foundation for preparing to read the source code of Linux kernel based on 80X86 CPU. Mainly include: 1. 80X86 CPU basic knowledge; 2. Protected mode memory management; 3. Various CPU protection methods; 4. Interrupt and exception handling; 5. Task management; 6. Initialization of protection mode programming; 7. A simple multitasking Kernel example.

A simple multitasking kernel program described in the last section of this chapter is a simplified example based on the Linux 0.12 kernel. This example is used to demonstrate the implementation of memory segmentation management and task management. It does not include paging mechanism content. However, if you thoroughly understand the operating mechanism of this example, you should not encounter any major problems when you read the Linux kernel source code later. If the reader is already familiar with this part of the content, you can directly read a runnable kernel sample program given at the end of this chapter. Of course, readers can refer back to this chapter at any time when reading kernel source code.

4.1 80X86 System Registers and System Instructions

To assist the processor in performing initialization and control system operations, the 80X86 provides a flag register, EFLAGS, and several system registers. In addition to some common status flags, EFLAGS also contains several system flags. These system flags are used to control task switching, interrupt handling, instruction tracking, and access permissions. System registers are used for memory management and control processor operations. They contain the base address of the system table for segmentation and paging processing, and the bit flags that control processor operations.

4.1.1 Flag Registers

The system flags and IOPL fields in the flag register EFLAGS are used to control I/O access, maskable hardware interrupts, debugging, task switching, and virtual-8086 modes, as shown in Figure 4-1. Normally only operating system code is allowed to modify these flags. The other flags in EFLAGS are some common flags (carry CF, parity PF, auxiliary carry AF, zero flag ZF, negative SF, direction DF, overflow OF). Here we describe only the system flags in the team EFLAGS.

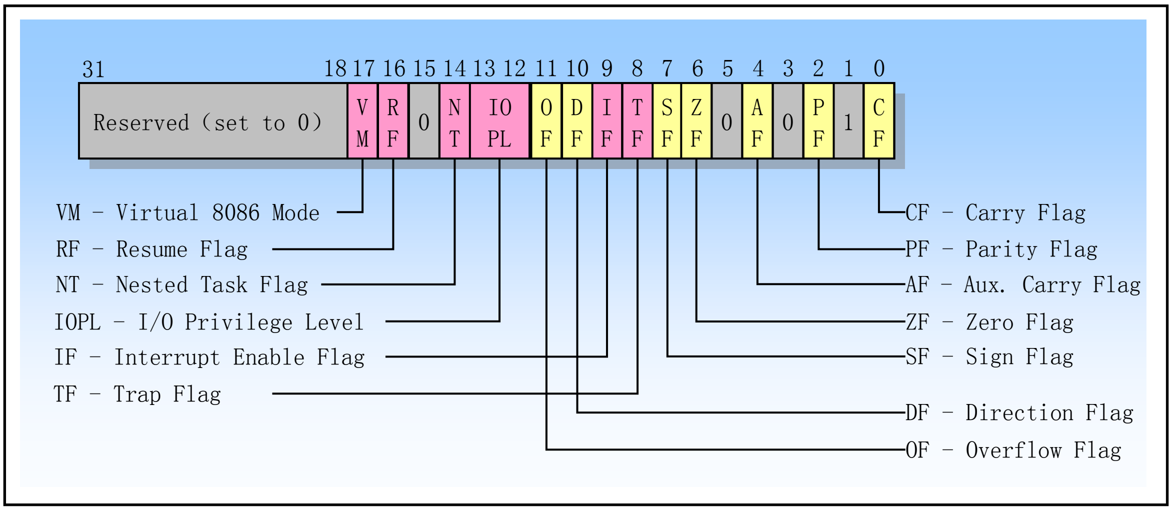


Figure 4-1 System Flags in EFLAGS

TF Bit 8 is the Trap Flag. When this bit is set, step-by-step execution can be started for the debug operation; single-step execution is prohibited when reset. In the single-step execution mode, the processor generates a debug exception after each instruction is executed, so that we can observe the state of the execute program after executing each instruction. If the program uses the POPF, POPFD, or IRET instructions to set the TF flag, the processor generates a debug exception after the subsequent instruction.

IOPL Bits 13-12 are the I/O Privilege Level field. This field indicates the I/O privilege level IOPL of the currently running program or task. The CPL currently running a program or task must be less than or equal to this IOPL to access the I/O address space. Only when the CPL is at privilege level 0 can the program use the POPF or IRET instructions to modify this field. IOPL is also one of the mechanisms that control the modification of the IF logo.

NT Bit 14 is a nested task flag. It controls the chain relationship between the interrupted task and the called task. The processor sets this flag on calls to a task initiated with a CALL instruction, an interrupt, or an exception. When returning from a task by using the IRET instruction, the processor checks and modifies this NT flag. This flag can also be modified using the POPF/POPFD instructions, but changing the state of this flag in the application can generate unexpected exceptions.

RF Bit 16 is the Resume Flag. This flag controls the processor's response to breakpoint instructions. When set, this flag temporarily disables the debug exception generated by the breakpoint instruction; when the flag is reset, the breakpoint instruction will generate an exception. The main function of the RF flag is to allow re-execution of an instruction after debugging an exception. Before the debug software uses the IRETD instruction to return to the interrupted program, the RF flag in the EFLAGS content on the stack needs to be set to prevent the instruction breakpoint from causing another exception. The processor automatically clears the flag after the instruction returns, again allowing instruction breakpoint exceptions.

VM Bit 17 is the Virtual-8086 Mode flag. When this flag is set, the virtual-8086 mode is turned on; when the flag is reset, it returns to the protected mode.

4.1.2 Memory Management Registers

The processor provides four memory management registers (GDTR, LDTR, IDTR, and TR) that specify the base address of the system table used for memory segment management, as shown in Figure 4-2. The processor provides specific instructions for loading and saving these registers. For the role of the system table, see the detailed description in the next section, "Protection Mode Memory Management."

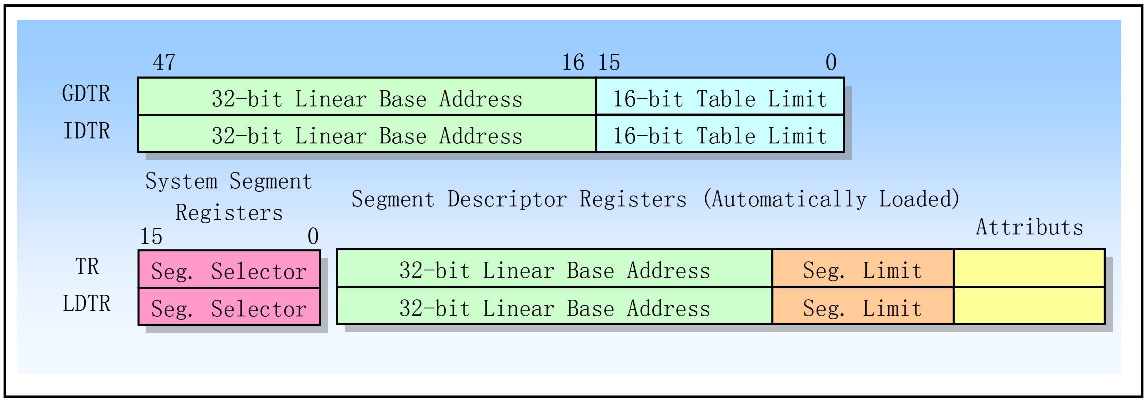

47
16 15
0
GDTR
IDTR
32-bit Linear Base Address
16-bit Table Limit
32-bit Linear Base Address
16-bit Table Limit
System Segment Registers
Segment Descriptor Registers (Automatically Loaded)
Attributs
15
0
TR
LDTR
Seg. Selector
Seg. Selector
32-bit Linear Base Address
Seg. Limit
32-bit Linear Base Address
Seg. Limit

Figure 4-2 Memory management registers

GDTR, LDTR, IDTR, and TR are segment base registers that contain important information tables for the segmentation mechanism. GDTR, IDTR, and LDTR are used to address the segment where the descriptor table is stored. TR is used to address a special task state segment TSS (Task State Segment). The TSS segment contains important information about the currently executing task. Now we explain them in detail.

1. Global Descriptor Table Register (GDTR)

The GDTR register holds the 32-bit linear base address and the 16-bit limit value of the global descriptor table GDT. The base address specifies the address of byte 0 in the GDT table in the linear address space, and the table length indicates the byte length value of the GDT table. The instructions LGDT and SGDT are used to load and set the contents of the GDTR register, respectively. After the machine has just powered up or the processor is reset, the base address is set to 0 by default and the length value is set to 0xFFFF. The GDTR must be loaded with a new value during the initialization of the protection mode.

2. Interrupt Descriptor Table Register (IDTR)

Similar to GDTR, the IDTR register is used to store the 32-bit linear base address and 16-bit table length values of the interrupt descriptor table IDT. The instructions LIDT and SIDT are used to load and set the contents of the IDTR register, respectively. After the machine has just powered up or the processor is reset, the base address is set to 0 by default and the length value is set to 0xFFFF.

3. Local Descriptor Table Register (LDTR)

The LDTR register holds the 16-bit segment selector, 32-bit linear base address, 16-bit segment limit, and descriptor attribute values of the local descriptor table LDT. The instructions LLDT and SLDT are used to load and store the segment selector part of the LDTR register, respectively. The segment containing the LDT table must have a segment descriptor entry in the GDT table. When using the LLDT instruction to load selectors containing LDT segments into the LDTR, the segment base address, segment length, and descriptor attributes of the LDT segment descriptor are automatically loaded into the LDTR. When task switching occurs, the processor automatically loads the segment selector and segment descriptor of the new task LDT into the LDTR. After the machine powers up or the processor resets, the segment selector and base address are set to 0 by default, and the segment length is set to 0xFFFF.

4. Task Register (TR)

The TR register holds the 16-bit segment selector, 32-bit base address, 16-bit segment length, and descriptor attribute values of the current task TSS segment. It references a TSS type descriptor in the GDT table. The LTR and STR instructions are used to load and save the segment selector portion of the TR register, respectively. When using the LTR instruction to load the selector into the task register, the segment base address, segment length, and descriptor attributes in the TSS descriptor are automatically loaded into the task register. When task switching is performed, the processor automatically loads the segment selector and segment descriptor of the TSS of the new task into the task register TR.

4.1.3 Control Registers

The control registers (CR0, CR1, CR2, and CR3) are used to control and determine the operating mode of the processor and the characteristics of the currently executing task, as shown in Figure 4-3. CR0 contains system control flags that control the operating mode and state of the processor; CR1 is reserved for use; CR2 contains a linear address that causes a page fault. CR3 contains the physical memory base address of the page directory table, so this register is also called the Page-Directory Base Address Register (PDBR).

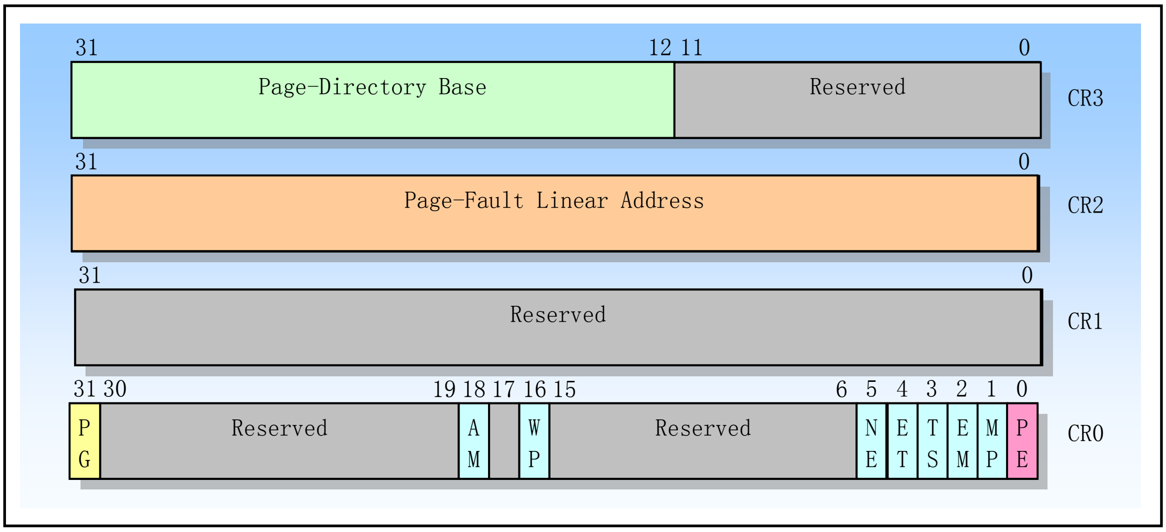


Figure 4-3 Control registers CR0--CR3

1. Coprocessor control bits in CR0

Four bits of CR0: Extended Type Bit (ET), Task Switching Bit (TS), Emulation Bit (EM), and Math Presence Bit (MP) are used to control the operation of the 80X86 floating point (math) coprocessor. For details on coprocessors, see Chapter 11. The ET bit (flag) of CR0 is used to select the protocol used to communicate with the coprocessor, ie to indicate whether the system is using the 80387 or 80287 coprocessor. The TS, MP, and EM bits are used to determine if a float instruction or WAIT instruction should generate a Device Not Available exception. This exception can be used to save and restore floating-point registers only for tasks that use floating-point operations. For tasks that do not use floating-point arithmetic, doing so can speed up the switching between them.

ET Bit 4 of CR0 is an Extension Type flag. When the flag is 1, it indicates that the system has a 80387 coprocessor and uses a 32-bit coprocessor protocol. ET=0 indicates use of the 80287 coprocessor. If simulation bit EM=1, this bit will be ignored. During a processor reset operation, the ET bit is initialized to indicate the type of coprocessor used in the system. If there is 80387 in the system, then ET is set to 1, otherwise if there is a 80287 or no coprocessor, ET is set to 0.

TS Bit 3 of CR0 is the Task Switched flag. This flag is used to postpone saving the coprocessor content of the task switch until the new task begins to actually execute the coprocessor instruction. The flag is set by the processor each time the task is switched and tested when the coprocessor instruction is executed.

If the TS flag is set and the EM flag of CR0 is 0, a Device Not Available (DNA) exception is generated before any coprocessor instructions are executed. If the TS flag is set, but the MP0 and EM flags of CR0 are not set, no device exception will not occur until the coprocessor instruction WAIT/FWAIT is executed. If the EM flag is set, the TS flag has no effect on the execution of coprocessor instructions. See Table 4-1.

When the task is switched, the processor does not automatically save the context of the coprocessor but sets the TS flag. This flag causes the processor to generate a device-existent exception when it encounters a coprocessor instruction at any time while executing a stream of new tasks. A device-existent handler can use the CLTS instruction to clear the TS flag and save the coprocessor's context. If the task has never used a coprocessor, the corresponding coprocessor context does not need to be saved.

EM Bit 2 of CR0 is an EMulation flag. When this bit is set, it means that the processor has no internal or external coprocessor. When the coprocessor instruction is executed, it will generates a device-not-available exception. When cleared, it means that the system has a coprocessor. Setting this flag forces all floating-point instructions to be simulated using software.

MP Bit 1 of CR0 is the Monitor Coprocessor or Math Present flag. Used to control the interaction between the WAIT/FWAIT instruction and the TS flag. If MP=1 and TS=1, then executing the WAIT instruction will generate a device-not-available exception; if MP=0, the TS flag will not affect the execution of the WAIT.

Table 4-1 Influence of EM, MP and TS Combinations in CR0 on Coprocessor Actions

CR0 Flags			Instruction Type	
EM	MP	TS	Floating-Point	WAIT/FWAIT
0	0	0	Execute	Execute
0	0	1	DNA Exception	Execute
0	1	0	Execute	Execute
0	1	1	DNA Exception	DNA Exception
1	0	0	DNA Exception	Execute
1	0	1	DNA Exception	Execute
1	1	0	DNA Exception	Execute
1	1	1	DNA Exception	DNA Exception

2. Protection and Control bits in CR0

PE Bit 0 of CR0 is the Protection Enable flag. When this bit is set, the protection mode is enabled; when reset, real address mode is entered. This flag only enables segment-level protection and does not enable paging. To enable the paging mechanism, both the PE and PG flags are set.

PG Bit 31 of CR0 is a paging signature. When this bit is set, the paging mechanism is enabled; when reset, the paging mechanism is disabled. At this time, all linear addresses are equivalent to physical addresses. The PE flag must be turned on before turning this flag on. That is, to enable the paging mechanism, both the PE and PG flags are set.

WP For Intel 80486 or higher CPUs, bit 16 of CR0 is the Write Proctect flag. When this flag is set, the processor prohibits the superuser program (eg, a privilege level 0 program) from performing a write operation to the user-level read-only page; when this bit is reset, it is the reverse. This flag is beneficial for UNIX-type operating systems to implement the Copy on Write technique when creating processes.

NE For Intel 80486 or higher CPUs, bit 5 of CR0 is the Numeric Error flag. When this flag is set, the internal reporting mechanism of the X87 coprocessor error is enabled; if the flag is reset, the X87 coprocessor error reporting mechanism in the form of a PC is used. When NE is in the reset state and there is a signal on the CPU's IGNNE input pin, the math coprocessor X87 error will be ignored. When NE is in the reset state and there is no signal on the CPU's IGNNE input pin, an unmasked math coprocessor X87 error will cause the processor to generate an interrupt externally through the FERR pin and perform the next wait form floating-point instruction Stop instruction execution immediately before or WAIT/FWAIT instruction. The FERR pin of the CPU is used to emulate the ERROR pin of the external coprocessor 80387, so it is usually connected to the interrupt controller input request pin. The NE flag, the IGNNE pin, and the FERR pin are used to implement an external error reporting mechanism in the form of a PC using external logic.

The Enable Protected (PE) bit (Bit 0) and the Paging bit (Bit 31) are used to control the segmentation and paging mechanisms, respectively. PE is used to control the segmentation mechanism. If PE=1, the processor operates in the context of the open segmentation mechanism, ie, operates in protected mode. If PE = 0, the processor turns off the segmentation mechanism and operates in the real address mode as in 8086. The PG is used to control the paging mechanism. If PG=1, the paging mechanism is turned on. If PG=-0, the paging mechanism is disabled and the linear address is used directly as the physical address.

If PE=0, PG=0, the processor operates in real-address mode; if PG=0, PE=1, the processor operates in protection mode without paging mechanism; if PG=1, PE=0, this Since the paging mechanism cannot be enabled because it is not in protected mode, the processor generates a general protection exception. This flag combination is invalid; if PG=1, PE=1, the processor works under the protection mode with the paging mechanism enabled. .

We must be careful when changing the PE and PG bits. We can only change the setting of the PG bit when the execution program has at least some of the code and data in the linear address space and the physical address space with the same address. At this point, this part of the code with the same address acts as a bridge between paged and non-paged worlds. Regardless of whether the paging mechanism is turned on, this part of the code has the same address. In addition, the page cache TLB must be refreshed before paging is enabled (PG=1).

After the PE bit is modified, the program must immediately use a jump instruction to flush any instructions in the processor's execution pipeline that have acquired different modes. Before setting the PE bit, the program must initialize several system segments and control registers. On power up, the processor is reset to PE=0 and PG=0 (real mode state) to allow the boot code to initialize these registers and data structures before enabling the segmentation and paging mechanism.

3. CR2 and CR3

CR2 and CR3 are used for paging mechanism. CR3 contains the physical address of the page directory table page, so CR3 is also called PDBR. Because the page directory table page is page-aligned, only the upper 20 bits of this register are valid. The lower 12 bits are reserved for use by more advanced processors, so the lower 12 bits must be set to 0 when loading a new value into CR3.

CR2 is used to report error messages when a page exception occurs. When the report page is abnormal, the processor stores the linear address that caused the exception in CR2. Therefore, the page exception handler in the operating system can determine which page in the linear address space caused an exception by checking the contents of CR2.

The use of the MOV instruction to load CR3 has the side effect of invalidating the page cache. In order to reduce the number of bus cycles required for address translation, the most recently accessed page directory and page table are stored in the page cache of the processor, which is referred to as the Translation Lookaside Buffer (TLB). The page table entry is read from memory using an extra bus cycle only if the TLB does not contain the required page table entry.

Even if the PG bit in CR0 is in the reset state (PG = 0), we can load CR3 first to allow the paging mechanism to be initialized. When switching tasks, the contents of CR3 will also change. However, if the CR3 value of the new task is the same as that of the original task, the processor does not need to refresh the page cache. This allows tasks that share page tables to execute faster.

4.1.4 System Instructions

System instructions are used to handle system-level functions. Examples include loading system registers, managing interrupts, and so on. Most system instructions can only be executed by operating system software at privilege level 0. The remaining instructions can be executed at any privilege level so applications can use it. Table 4-2 lists some of the system instructions we will use. It also indicates whether they are protected.

Table 4-2 List of commonly used system instructions

Instruction	Description	Protected?	Description
LLDT	Load LDT Register	Yes	Load LDT segment selectors and segment descriptors from memory into the LDTR register.
SLDT	Store LDT Regiter	No	Save the LDT segment selector in LDTR to internal memory or general-purpose registers.
LGDT	Load GDT Register	Yes	Load the base address and length of the GDT table from memory into GDTR.
SGDT	Store GDT Register	No	Save the base address and length of the IDT table in GDTR to memory.
LTR	Load Task Register	Yes	Load TSS segment selectors (and segment descriptors) into the task register.
STR	Store Task Register	No	Save the current task TSS segment selector in TR to the memory or general register.
LIDT	Load IDT Register	Yes	The base address and length of the IDT table are loaded from memory into the IDTR.
SIDT	Store IDT Register	No	Store the base address and length of the IDT table in IDTR in memory.
MOV CRn	Move Control Registers	Yes	Load and save control registers CR0, CR1, CR2, or CR3.
LMSW	Load Machine State Word	Yes	Load the machine status word (corresponds to CR0 bit 15--0). This instruction is for compatibility with the 80286 processor.
SMSW	Store Machine State Word	No	Save the machine status word. This instruction is for compatibility with the 80286 processor.
CLTS	Clear TS flag	Yes	Clears the task switched flag TS in CR0. There are no exceptions for handling devices (coprocessors).
LSL	Load Segment Limit	No	Load Segment Limit
HLT	Halt Processor	Yes	Stop the processor execution.

4.2 Protect Mode Memory Management

The following is a brief introduction to the definition of memory addressing, the use of segmentation and paging mechanism for the principle of the transformation between logical addresses, linear addresses and physical addresses, and the protection mechanism between tasks and privilege levels. Subsequent subsections elaborate on the working principles of each part.

4.2.1 Memory Addressing

Memory refers to an array of ordered bytes, each byte having a unique memory address. Memory addressing refers to locating the address of a specified data object stored in memory. Here, a data object is a numeric value or a string of a specified data type stored in memory. 80X86 supports multiple data types: 1-byte, 2-byte (1 word) or 4-byte (double-word or long-word) unsigned integer or signed integer, and multi-byte character strings . Usually the positioning or addressing of a certain bit in a byte can be addressed on a byte basis, so the addressing of the smallest data type is the positioning of 1-byte data (numeric values or characters). Normally, the memory address is addressed from 0. For the 80X86 CPU, the address bus width is 32 bits, so there are a total of 2^32 different physical addresses. That is, the memory physical address space has 4G, a total of 4G bytes of physical memory can be addressed. For multi-byte data types (such as 2-byte integer data types), these bytes are stored in memory. The 80X86 first stores the low-value byte and then stores the high-value byte at the address. Therefore, the 80X86 CPU is a small-endum processor.

For the 80X86 CPU, one instruction consists mainly of the opcode and the operand. The operand can be located in a register or in memory. To locate an operand in memory, memory addressing is required. The 80X86 has many instruction operands that involve memory addressing, and there are many different addressing schemes to choose from depending on the type of data being addressed. For memory addressing, the 80X86 uses an addressing technique called **Segment**. Addressing a data object in memory requires the use of a segment's start address (that is, segment address) and an intra-segment offset address. The segment address part is specified using a 16-bit segment selector, of which 14 bits can select 2^14 powers, ie 16384 segments. The intra-segment offset address portion is specified using a 32-bit value, so the intra-segment address can be 0 to 4G. That is, the maximum length of a segment can reach 4G. A 48-bit address or long pointer consisting of a 16-bit segment selector and a 32-bit offset in the segment thus forms a logical address (virtual address). It uniquely determines the segment address and segment offset address of a data object. An address specified by only a 32-bit offset address or pointer is based on the object address of the current segment. The segmentation mechanism also allows typing of segments so that the operations that may be performed on a particular type of segment can be restricted.

The 80X86 provides six segment registers for storing segment selectors: CS, DS, ES, SS, FS, and GS. Where CS is always used to address the code segment, and the stack segment specifically uses the SS segment register. The segment addressed by CS at any given moment is called the current code segment. At this time, the EIP register contains the offset address within the segment within the current code segment to be executed. Therefore, the address of the instruction to be executed can be expressed as CS:[EIP]. The inter-segment control branch instructions, which will be described later, can be used to assign new values to CS and EIP so that the execution position can be changed to other code segments, thus achieving control transfer of the program in different segments.

The segment addressed by the segment register SS is called the current stack segment. The top of the stack is specified by the contents of the ESP register. So the address at the top of the stack is SS:[ESP]. The other 4 segment registers are general segment registers. When the instruction does not specify a segment of the data to be operated, DS will be the default data segment register.

In order to specify the intra-segment offset address of a memory operand, the 80X86 instruction specifies many ways to calculate the offset, which is called instruction addressing. The instruction's offset consists of three parts: the base address register, the index register, and an offset constant. which is:

Offset address = base address + (index x scale factor) + offset

4.2.2 Address Translation

Any complete memory management system contains two key parts: protection and address translation. Providing protection prevents one task from accessing another task or operating system's memory area. Address translation allows the operating system to have flexibility in allocating memory to tasks, and because we can make certain physical addresses unmapped by any logical address, memory protection is also provided during address translation.

As mentioned above, the physical memory in the computer is a linear array of bytes, each byte has a unique physical address; the address in the program is a logical address composed of a segment selector and an offset within the segment. This kind of logical address cannot be directly used to access physical memory, but it needs to be transformed or mapped to a physical memory address using an address translation mechanism. The memory management mechanism is used to translate this logical address into a physical memory address.

In order to reduce the information needed to determine the address translation, the translation or mapping usually uses memory blocks as the operating unit. Segmentation mechanism and paging mechanism are two widely used address translation techniques. They differ in how logical addresses are organized into mapped memory blocks, how the translation information is specified, and how programmers operate. Fragmentation and paging operations use tables that reside in memory to specify their respective translation information. These tables can only be accessed by the operating system to prevent unauthorized modifications by the application.

The 80X86 uses segmentation and paging in the translation from logical addresses to physical addresses, as shown in Figure 4-4. The first stage uses a segmentation mechanism to translate the logical address into an address in the processor linear address space. The second stage uses a paging mechanism to translate linear addresses into physical addresses. In the address translation process, the first-stage segmentation mechanism is always used, and the second-stage paging mechanism is optional. If no paging mechanism is enabled, the linear address space generated by the segmentation mechanism is directly mapped to the processor's physical address space. The physical address space is defined as the address range that the processor can generate on its address bus.

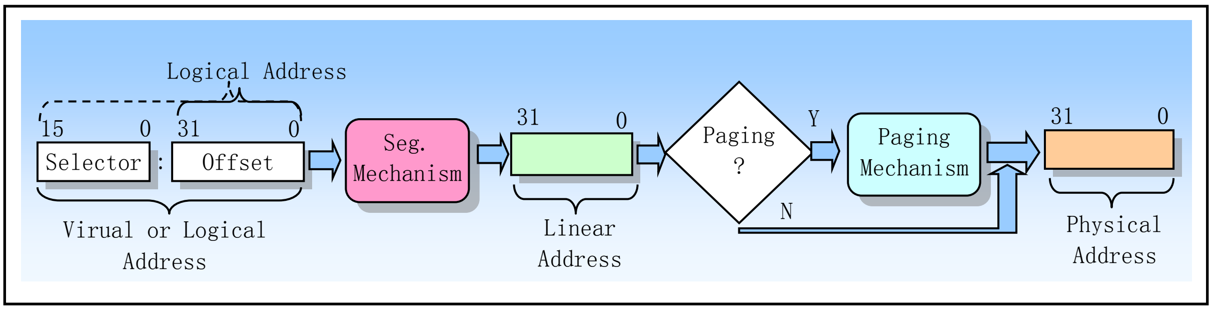


Figure 4-4 Logical address to physical address translation

1. Segmentation mechanism

Segmentation provides a mechanism to isolate individual code, data, and stack areas so that multiple programs (or tasks) can run on the same processor without interfering with each other. The paging mechanism provides an implementation mechanism for traditional demand pages and virtual memory systems. The virtual memory system is used to realize that the program code is mapped to physical memory as required. The paging mechanism can of course also be used to provide isolation between multiple tasks.

As shown in Figure 4-5, segmentation provides a mechanism for dividing the processor addressable linear address space into smaller, protected address space regions called segments. Segments can be used to hold program code, data, and stacks, or to hold system data structures (such as TSS or LDT). If there are multiple programs or tasks running in the processor, each program can allocate its own set of segments. At this point the processor can enforce the boundaries between these segments and ensure that a program does not interfere with the execution of the program by accessing segments of another program. Segmentation also allows classification of segments. In this way, operations on specific types of segments can be limited.

All used segments in a system are contained in the processor's linear address space. In order to locate a byte in a specified segment, the program must provide a logical address. The logical address includes a segment selector and an offset. The segment selector is the unique identifier of a segment. In addition, the segment selector provides the offset of a data structure (called segment descriptor) in the segment descriptor table (eg, the global descriptor table GDT). Each segment has a segment descriptor. The segment descriptor specifies the size of the segment, the access rights and the privilege level of the segment, the segment type, and the position of the first byte of the segment in the linear address space (referred to as the segment's base address). The offset of the logical address is added to the base address of the segment to locate a byte in the segment. Therefore, the base address plus the offset form the address in the processor's linear address space.

The linear address space has the same structure as the physical address space. Compared to two-dimensional logical address space, both are one-dimensional address spaces. The virtual address (logical address) space can contain up to 16K segments, and each segment can be up to 4GB, resulting in a virtual address space capacity of 64TB (2^46). Both the linear and physical address spaces are 4GB (2^32). In fact, if the paging mechanism is disabled, the linear address space is the physical address space.

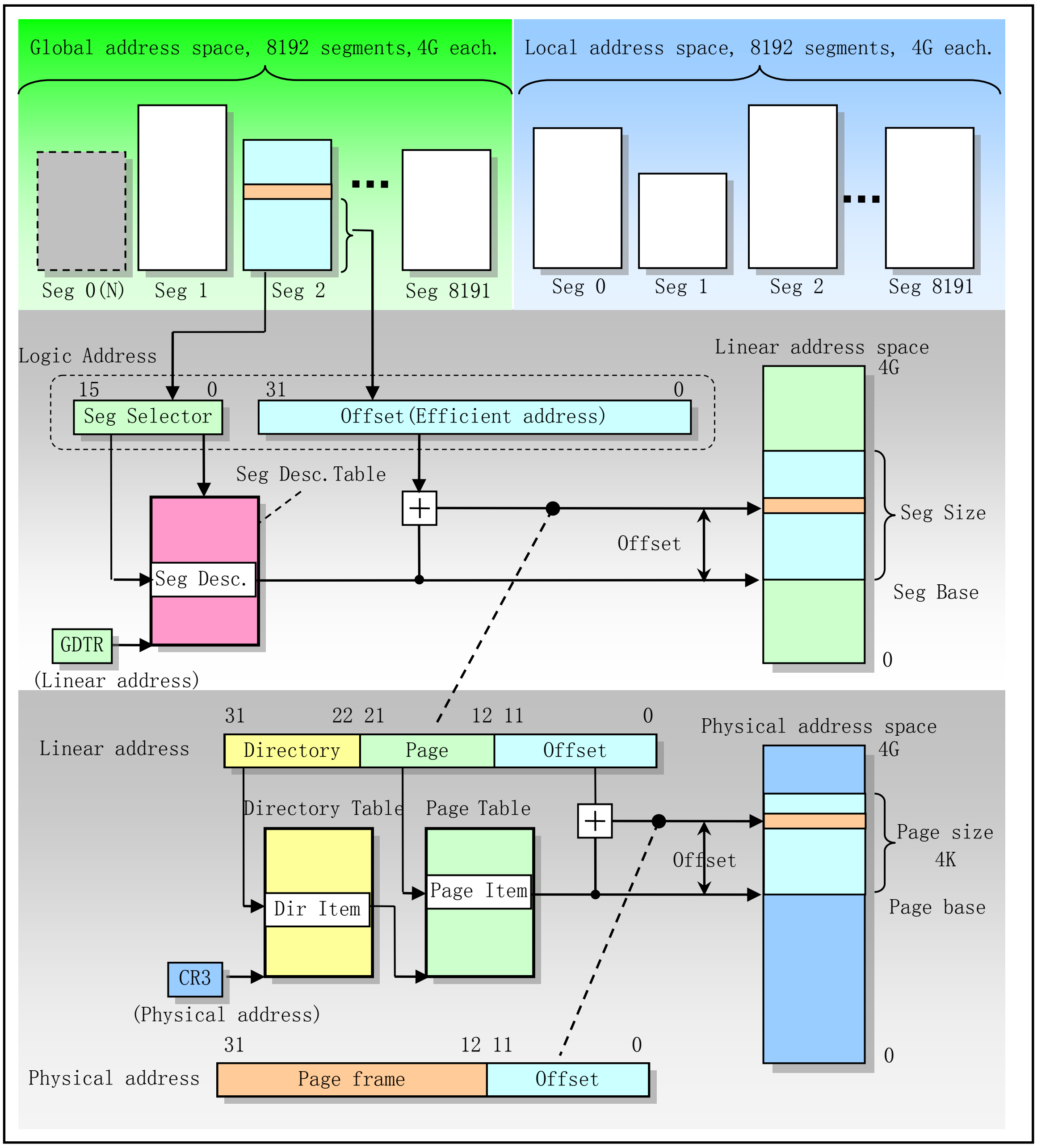


Figure 4-5 Logical, Linear, and Physical Addresses

2. Paging mechanism

Because the multi-tasking system usually defines a linear address space much larger than the physical memory it contains, it needs to use some kind of "virtualized" linear address space approach, that is, the use of virtual storage technology. Virtual storage is a memory management technology that allows programmers to create the illusion that the memory space is much larger than the actual physical memory capacity of the computer. With this illusion, we can program large programs at will without considering how much physical memory actually exists.

The paging mechanism supports virtual storage technology. In an environment using virtual storage, a large-capacity linear address space needs to be simulated using a small amount of physical memory (RAM or ROM) and some external storage space (such as a large-capacity hard disk). When paging is used, each segment is divided into pages (usually 4 KB in size per page) and the pages are stored in physical memory or on hard disk.

The operating system pays attention to these pages by maintaining a page directory and some page tables. When a program (or task) tries to access an address location in a linear address space, the processor uses the page directory and page table to convert the linear address to a physical address, and then performs the required operation at that memory location (reading Or write).

If the currently visited page is not in physical memory, the processor interrupts the execution of the program (by generating a page fault exception). The operating system can then read the page from the hard disk into physical memory and continue to execute the program that was just interrupted. When the operating system strictly implements the paging mechanism, the exchange of pages between the physical memory and the hard disk is transparent to the correctly executed program.

The 80X86 paging mechanism is best suited to support virtual storage technology. The paging mechanism uses fixed-size memory blocks, while segment management uses variable-sized blocks to manage memory. The use of fixed-size blocks for paging is more suitable for managing physical memory, whether in physical memory or on hard disks. On the other hand, the segmentation mechanism uses variable-sized blocks that are more suitable for processing logical partitions of complex systems. You can define memory cells that fit into the logical block size without being constrained by fixed-size pages. Each segment can be treated as a unit, simplifying the protection and sharing of segments.

Segmentation and paging are two different address translation mechanisms, all of which provide independent processing stages for the entire address translation operation. Although both mechanisms use a conversion table stored in memory, the table structure used is different. In fact, the segment table is stored in the linear address space, and the page table is stored in the physical address space. Therefore, the segment conversion table can be relocated by the paging mechanism without the information or cooperation of the segment mechanism. The segment conversion mechanism transforms virtual addresses (logical addresses) into linear addresses and accesses its own table in a linear address space, but it is not aware of the process of the paging mechanism converting these linear addresses to physical addresses. Similarly, the paging mechanism does not know the virtual address space where the program generates addresses. The paging mechanism simply translates linear addresses into physical addresses and accesses their own translation tables in physical memory.

4.2.3 **Protection**

The 80X86 supports two types of protection. One is to completely isolate each task by giving each task a different virtual address (logical address) space. The implementation principle is to provide each task with different logical address to physical address mapping. Another protection mechanism operates on tasks to protect the operating system memory segments and processor special system registers from being accessed by applications.

1. Protection between tasks

One important aspect of protection is to provide protection between applications' tasks. The method used by the 80X86 is to place each task in a different virtual address space and give each task a different mapping of logical addresses to physical addresses. The address translation function in each task is defined as the logical address in one task is mapped to a part of physical memory, and the logical address in another task is mapped to a different area in physical memory. In this way, all tasks are isolated because one task cannot generate parts of the physical memory that can be mapped to the corresponding logical addresses of other tasks. Just give each task a separate mapping table, and each task will have a different address translation function. In 80X86, each task has its own segment table and page table. When the processor switches to perform a new task, the key part of task switching is to switch to the new task's conversion table.

By arranging the same virtual-to-physical address mapping portion in all tasks and storing the operating system in this common virtual address space portion, the operating system can be shared by all tasks. The same portion of the virtual address space that all this task has is called the Global address space. This is exactly how modern Linux operating systems use virtual address spaces.

The part of the virtual address space that is unique to each task is called the Local address space. The local address space contains private code and data that need to be distinguished from other tasks in the system. Because there is a different local address space in each task, references to the same virtual address in two different tasks will be converted to different physical addresses. This allows the operating system to give the same virtual address to each task's memory, but still isolate each task. On the other hand, all tasks' references to the same virtual address in the global address space will be translated to the same physical address. This provides support for the sharing of common code and data (such as operating systems).

2. Privilege level protection

In a task, four Privilege Levels are defined for restricting access to segments in a task based on the sensitivity of the data contained in the segment and the degree of trust of different portions of the program in the task. The most sensitive data is given the highest privilege level and they can only be accessed by the most trusted part of the task. Less sensitive data is given lower privilege levels and they can be accessed by less privileged code in the task.

The privilege level is represented by the numbers 0 to 3, with 0 having the highest privilege level and 3 being the lowest privilege level. Each memory segment is associated with a privilege level. This privilege level restricts programs with sufficient privilege level to access a segment. We know that the processor fetches and executes the instruction from the segment specified by the CS register, the current privilege level, that is, the CPL is the privilege level of the currently active code segment, and it defines the privilege level of the currently executing program. The CPL determines which segments can be accessed by the program.

Each time a program attempts to access a segment, the current privilege level is compared to the segment's privilege level to determine if there is an access permission. Programs executed at a given CPL level allow access to data segments of the same or lower level. Any references to high-level segments are illegal and raise an exception to notify the operating system.

Each privilege level has its own program stack to avoid the protection issues associated with using the shared stack. When the program is switched from one privilege level to another, the stack segment is also changed to the new level stack.

4.3 Segmentation Mechanism

The segmentation mechanism can be used to implement a variety of system designs. These designs range from flat models that only use minimum functionality of segmentations to protect programs to multi-segmented models that use segmentation to create a robust operating environment that can reliably run multiple programs (or tasks).

The simplest memory model for a system is the basic flat model, in which the operating system and programs have access to a continuous, unsegment address space. For the most part, this basic flat model hides the architecture's segmentation mechanism from system designers and application programmers. To implement a basic flat memory model, you must create at least two segment descriptors, one for the reference code segment and one for the reference data segment. However, both segments are mapped to the entire linear address space: that is, two segment descriptors have the same segment limit of 0 and 4 GB for the same base address value.

The multi-segment model can use the segmentation mechanism to provide full protection of hardware-enhanced code, data structures, programs, and tasks. In general, each program (or task) uses its own segment descriptor table and its own segment. For programs, segments can be completely private or shared between programs. Access to all segments and each running program execution environment on the system is controlled by hardware.

Access checks can be used not only to protect references to addresses outside the boundaries of the segment, but also to prevent execution of impermissible actions in certain segments. For example, because the code segment is designed to be a read-only segment, you can use hardware to prevent writes to the code segment. The access rights information in the segment can also be used to set the protection ring or level. Protection levels can be used to protect operating system programs from unauthorized access by applications.

4.3.1 Segment definition

As mentioned in the overview of the previous section, the 80X86 provides 4GB of physical address space in protected mode. This is the address space that the processor can address on its address bus. This address space is flat and the address range is from 0 to 0xFFFFFFFF. This physical address space can be mapped to read and write memory, read-only memory, and memory-mapped I/O. The segmentation mechanism is to organize the virtual memory in the virtual address space into some variable-length memory block units called segments. The virtual address (logical address) in the 80X86 virtual address space consists of a segment portion and an offset portion. Segments are the basis for virtual address-to-linear address translation mechanisms. Each segment is defined by three parameters:

1. Base address specifies the starting address of the segment in the linear address space. The base address is a linear address and corresponds to offset 0 in the segment.
2. The segment limit is the maximum available offset within the segment in the virtual address space. It defines the length of the segment.
3. Attributes, specify the characteristics of the segment. For example, whether the segment is readable, writable, or executable as a program; the privilege level of a segment, and so on.

Segment length defines the size of the segment in the virtual address space. The segment base address and the segment limit length define the linear address range or region to which the segment is mapped. The address range from 0 to limit in the segment corresponds to the range base in the linear address to base + limit. A virtual address with an offset greater than the segment limit is meaningless and can cause an exception if used. In addition, exceptions can also result if you access a segment without permission from the segment attribute. For example, if you try to write a read-only segment, 80X86 will generate an exception. In addition, the range of multiple segments mapped into the linear address can partially overlap or cover, or even completely overlap, as shown in Figure 4-6. In the Linux 0.1x system introduced in this book, the length of the segment of a task's code segment and data segment is the same, and is mapped to the same area where the linear addresses are identical and overlap.

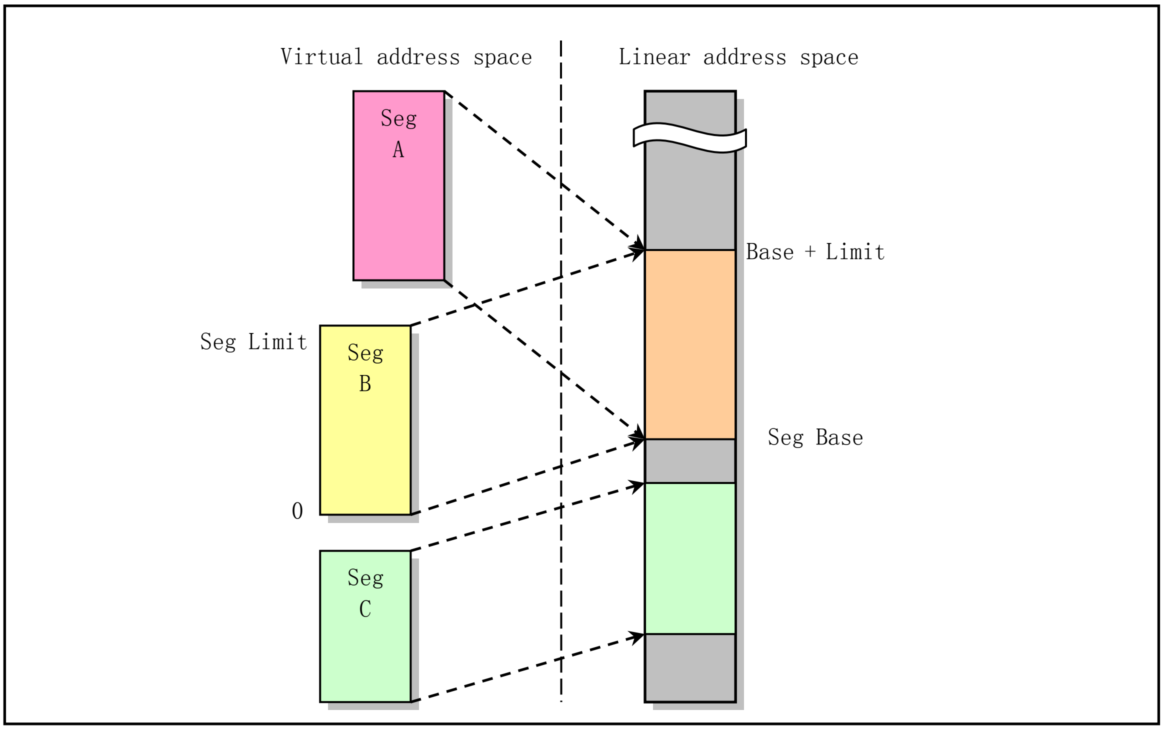


Figure 4-6 Segments in virtual map to linear address space

The base address,the limit length, and the protection attribute of a segment are stored in a structure item called a Segment Descriptor. This segment descriptor is used during the translation mapping from logical address to linear address. Segment descriptors are stored in the Descriptor table. The segment descriptor table is a simple array containing segment descriptor items. The segment selector described earlier is used to specify the corresponding segment by specifying the position of a segment descriptor in the table.

Even with the minimum functionality of the segment, each byte in the processor's address space can be accessed using the logical address. The logical address consists of a 16-bit segment selector and a 32-bit offset, as shown in Figure 4-7. The segment selector specifies the segment where the byte is located, and the offset specifies the position of the byte in the segment relative to the segment base address. The processor will convert each logical address to a linear address. A linear address is a 32-bit address in the processor's linear address space. Similar to the physical address space, the linear address space is also a flat 4GB address space with addresses ranging from 0 to 0xFFFFFFFF. The linear address space contains all the system-defined segments and system tables.

To convert the logical address into a linear address, the processor performs the following operations:

1. Use the offset value (segment index) in the segment selector to locate the corresponding segment descriptor in the GDT or LDT table. (This step is only needed if a new segment selector is loaded into a segment register.)
2. Examines the segment descriptor to check the access rights and range of the segment to insure that the segment is accessible and that the offset is within the limits of the segment.
3. 3. Add the segment base address obtained in the segment descriptor to the offset and finally form a linear address.

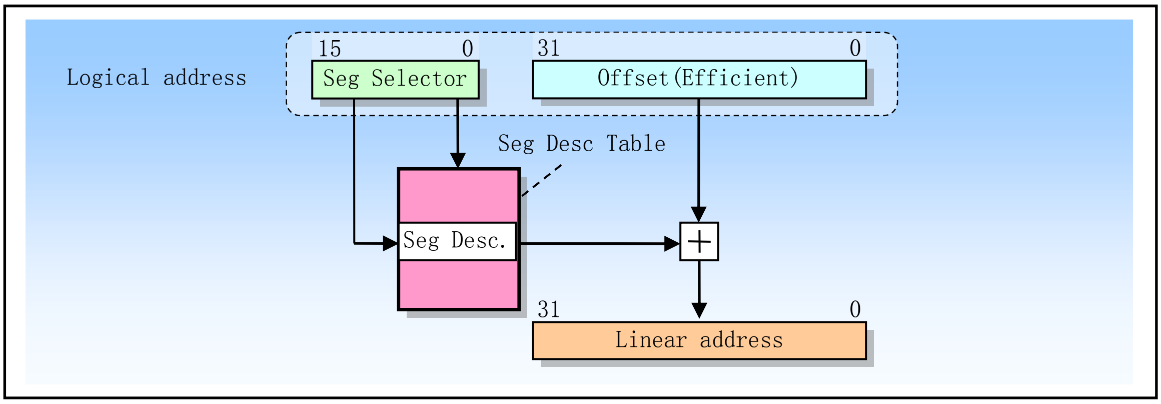


Figure 4-7 Logical address to linear address translation

If paging is not enabled, the processor directly maps the linear address to the physical address (ie, the linear address is sent to the processor's address bus). If the linear address space is paged, then a second level address translation will be used to translate the linear address into a physical address. The page conversion will be explained later.

4.3.2 Segment Descriptor Tables

The segment descriptor table is an array of segment descriptors, as shown in Figure 4-8. The descriptor table is variable in length and can contain up to 8192 8-byte descriptors. There are two descriptor tables: a global descriptor table (GDT) and a local descriptor table (LDT).

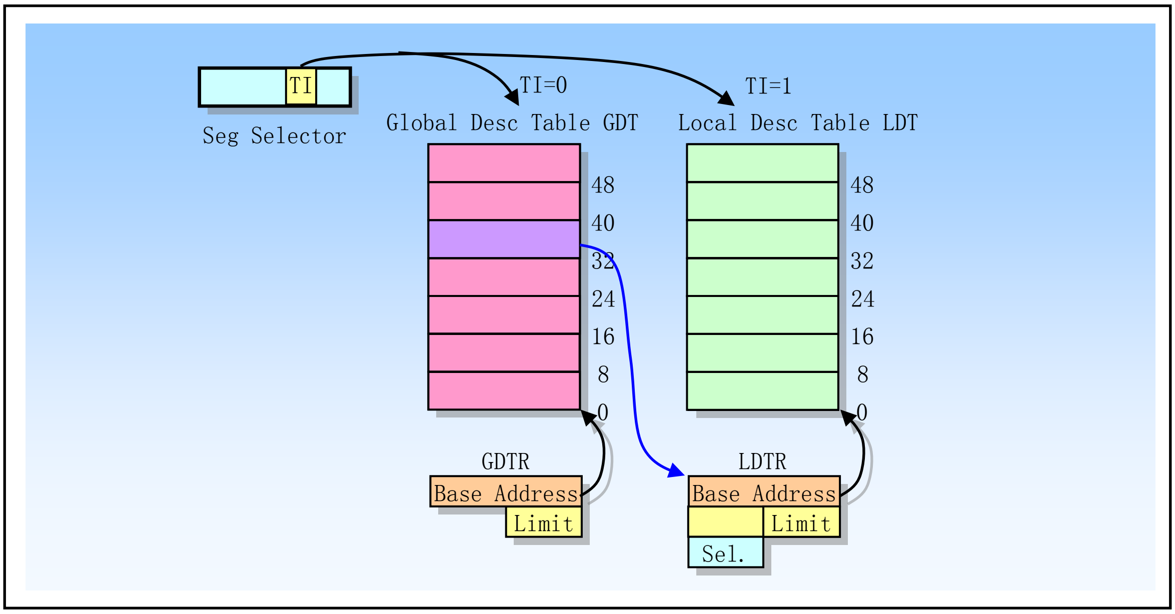


Figure 4-8 Global and Locat Descriptor Tables

Each system must have one GDT that can be used for all programs and tasks in the system. Optionally, one or more LDTs may be defined. For example, an LDT can be defined for each individual task that is running, or some or all tasks can share the same LDT.

The GDT itself is not a segment; instead, it is a data structure in the linear address space. The basic linear address and limit of the GDT must be loaded into the GDTR register. The base address of the GDT should be aligned on an eight-byte boundary to produce the best processor performance. The GDT limit is expressed in bytes. As with segmentation, limit values are added to the base address to get the address of the last valid byte. A limit of 0 results in only one valid byte. Because the segment descriptor is always 8 bytes long, the GDT limit should always be less than an integer multiple of 8 (ie, 8N-1).

The LDT table is stored in the LDT type of system segment. At this point the GDT must contain the LDT segment descriptor. If the system supports multiple LDTs, then each LDT must have a segment descriptor and segment selector in the GDT. An LDT segment descriptor can be stored anywhere in the GDT table.

Accessing LDT requires its segment selector. In order to reduce the number of address conversions when accessing the LDT, the LDT's segment selector, base address, segment length, and access rights need to be stored in the LDTR register.

When the GDTR register is stored (using the SGDT instruction), a 48-bit "pseudo-descriptor" is stored in memory. In order to avoid alignment check errors in user mode (privilege level 3), dummy descriptors should be stored at an odd word address (ie, address MOD 4 = 2). This will cause the processor to store an aligned word first, followed by an aligned doubleword (at the 4-byte alignment). User-mode programs usually do not save dummy descriptors, but you can use this alignment to avoid the possibility of an alignment check error. The same alignment is also used when using the SIDT instruction to save the IDTR register contents. However, when saving LDTR or task registers (using SLTR or STR instructions, respectively), dummy descriptors should be stored at double-word aligned addresses (ie, address MOD 4 = 0).

Descriptor tables are stored in special data structures maintained by the operating system and referenced by the processor's memory management hardware. These special structures should be stored in a protected memory area that is only accessible by the operating system software to prevent the application from modifying the address translation information in it. The virtual (logical) address space is divided into two halves of equal size. Half is mapped to a linear address by the GDT and the other half is mapped by the LDT. The entire virtual address space contains 2^14 segments: half of the space (that is, 2^13 segments) is a global virtual address space mapped by GDT, and the other half is a local virtual address space mapped by LDT. By specifying a descriptor table (GDT or LDT) and description symbols in the table, we can locate a descriptor.

When a task switch occurs, the LDT will be replaced with a new task LDT, but the GDT will not change. Therefore, half of the virtual address space mapped by the GDT is common to all the tasks in the system, but the other half of the LDT mapping is changed when the task is switched. The segments shared by all tasks in the system are mapped by the GDT. Such segments typically include a section containing the operating system and each task's own special section containing LDTs. The LDT segment can be thought of as data belonging to the operating system.

The LDT contains descriptors for segments that are dedicated to a single task. Several tasks can share a common LDT. In this case, all these tasks can use the same set of segments because they have the same LDT, and all tasks share one GDT. Both tasks can also share a segment descriptor in their respective LDTs so that sharing a segment without having to put its descriptor in the GDT is shared for all tasks. In this case, the shared segment must be handled exclusively by the operating system because it has two descriptors in two different LDTs and must be updated together.

Figure 4-9 shows how the segments in a task can be separated between GDT and LDT. There are six segments in the figure for the two applications (A and B) and the operating system. Each application in the system corresponds to a task, and each task has its own LDT. Application A runs in Task A and has LDT_A to map segments Code_A and Data_A. Similarly, application B runs in task B and uses LDT_B to map the Code_B and Data_B segments. The two segments containing the operating system kernel, Code_{OS} and Data_{OS}, are mapped using the

GDT so that they can be shared by both tasks. Two LDT segments: LDT_A and LDTB are also mapped using GDT.

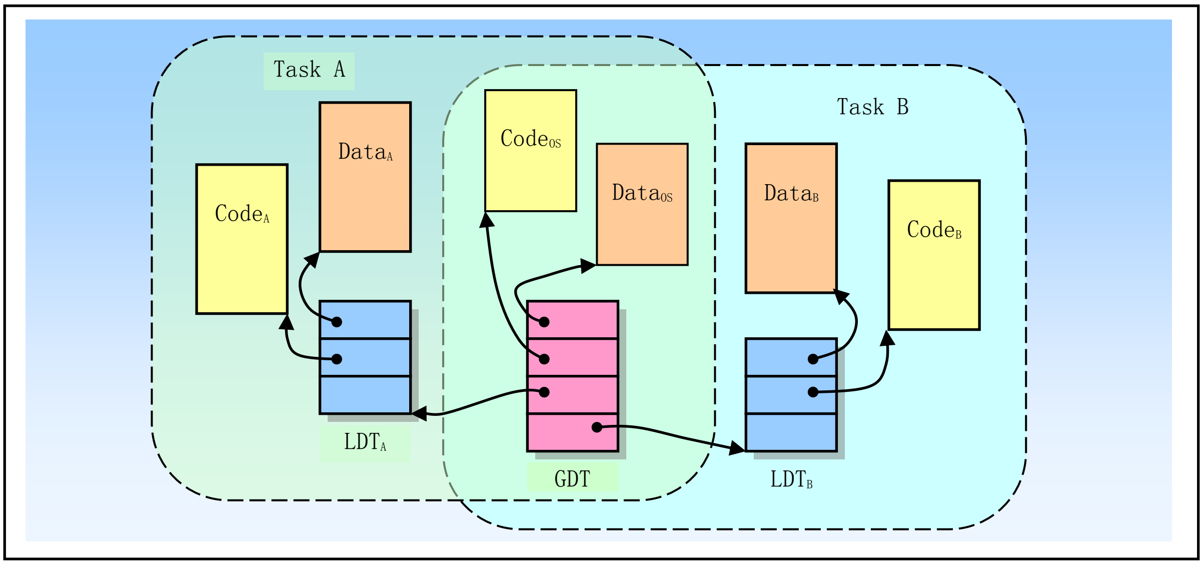


Figure 4-9 The segment types used by tasks

When task A is running, accessible segments include the $Code_A$ and $Data_A$ segments of the LDT_A map, plus the $Code_{OS}$ and $Data_{OS}$ segments of the GDT mapped operating system. When task B is running, the accessible segments include the LDT_B mapped $Code_B$ and $Data_B$ segments plus the GDT mapped segments.

This example demonstrates how virtual address space can be organized to isolate each task by having each task use a different LDT. When task A is running, the segment of task B is not part of the virtual address space, so task A cannot access task B's memory. Similarly, when task B is running, the segment of task A cannot be addressed. This method of isolating each application task using LDT is one of the key protection needs.

4.3.3 Segment Selectors

The segment selector is a 16-bit identifier for the segment, as shown in Figure 4-10. The segment selector does not point directly at the segment, but instead points to the segment descriptor that defines the segment in the segment descriptor table. The segment selector consists of 3 fields and the contents are as follows:

- Requested Privilege Level (RPL);
- Table Index (TI);
- Index.

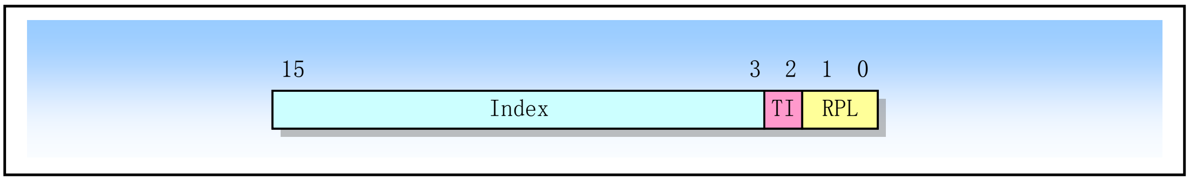
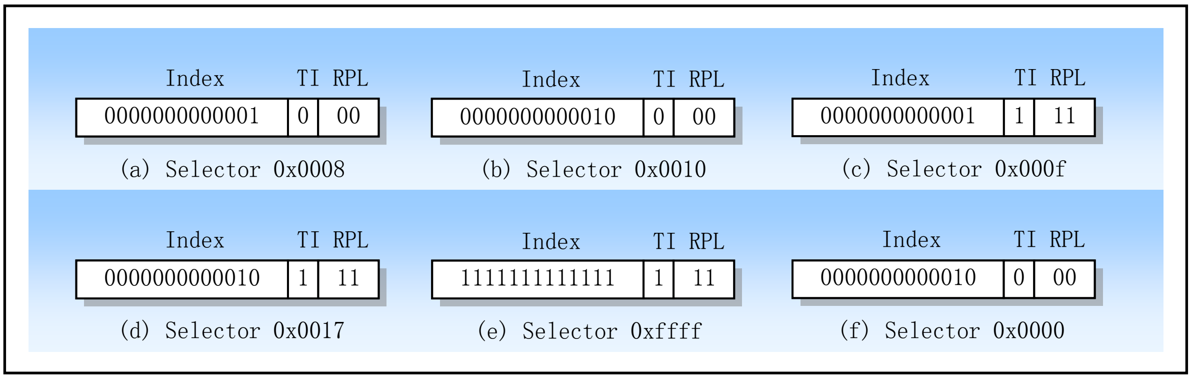


Figure 4-10 Segment selector structure

The request privilege level RPL provides segment protection information, which will be described later in detail. The table index TI is used to indicate the segment descriptor table GDT or LDT containing the specified segment descriptor. TI=0 indicates that the descriptor is in the GDT; TI=1 indicates that the descriptor is in the LDT. The index field gives the index number of the descriptor in the GDT or LDT table. It can be seen that the selector specifies a segment by locating a descriptor in the segment table, and the descriptor contains all the information for accessing a segment, such as the base address of the segment, segment length, and segment attributes.

For example, the selector (0x08) in Figure 4-11(a) specifies a segment 1 with RPL=0 in the GDT. The index field value is 1, the TI bit is 0, and the GDT table is specified. The selector (0x10) in Figure 4-11(b) specifies the segment 2 with RPL=0 in the GDT. The index field value is 2, the TI bit is 0, and the GDT table is specified. The selector (0x0f) in Figure 4-11(c) specifies segment 1 with LPL=3 in the LDT. Its index field value is 1, the TI bit is 1, and the LDT table is specified. The selector (0x17) in Figure 4-11(d) specifies segment 2 with LPL=3 in the LDT. The index field value is 2, the TI bit is 1, and the LDT table is specified. In fact, the first four selectors in Figure 4-11: (a), (b), (c), and (d) are the kernel code snippets, kernel data snippets, task code snippets, and tasks of the Linux 0.1x kernel, respectively. The data segment selector. The selector (0xffff) in Figure 4-11(e) specifies segment 8191 with RPL=3 in the LDT table. Its index field value is 0b1111111111111 (that is, 8191), the TI bit is equal to 1, and the LDT table is specified.



(a) Selector 0x0008 (b) Selector 0x0010 (c) Selector 0x000f

(d) Selector 0x0017 (e) Selector 0xffff (f) Selector 0x0000

Figure 4-11 Segment selector examples

In addition, the processor does not use the first item in the GDT table. The selector to the GDT entry (that is, the index whose index value is 0 and the TI flag is 0) is used as an "null selector", as shown in Figure 4-11(f). When an empty selector is loaded into a segment register (other than CS and SS), the processor does not generate an exception. However, an exception occurs when using a segment register containing an empty selector to access memory. An exception will be caused when an empty selector is loaded into the CS or SS segment register.

The segment selector is visible to the application as part of the pointer variable, but the value of the selector is usually set or modified by the link editor or link loader, not the application. To reduce address translation time and programming complexity, the processor provides registers that hold up to six segment selectors (see Figure 4-12), that is, segment registers. Each segment register supports a specific type of memory reference (code, data, or stack). In principle, each program must at least load valid segment selectors into the code segment (CS), data segment (DS), and stack segment (SS) registers. The processor additionally provides three auxiliary data segment registers (ES, FS, and GS) that can be used to allow the currently executing program (or task) to access several other data segments.

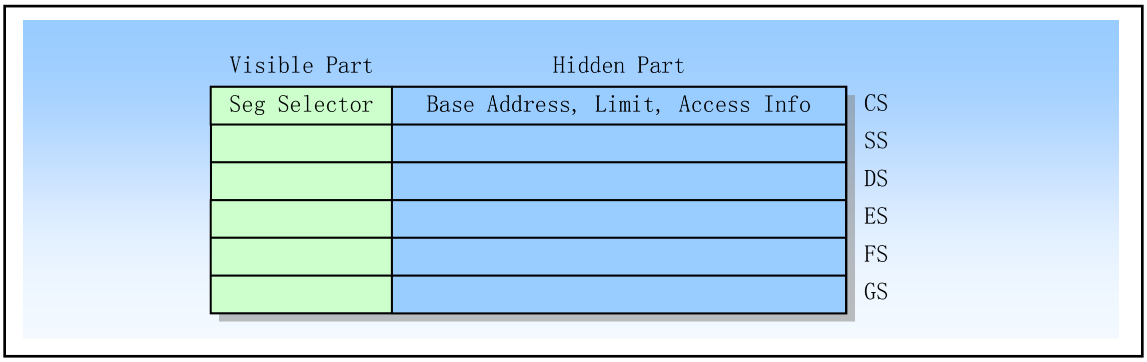


Figure 4-12 Segment register structure

For programs that access a segment, the segment selector must have been loaded into a segment register. Therefore, although a system can define many segments, only 6 segments are available for immediate access at the same time. To access other segments you need to load selectors for these segments.

In addition, to avoid reading the descriptor table every time you access the memory, to read and decode a segment descriptor, each segment register has a "visible" part and a "hidden" part (the hidden part is also called " Descriptor Buffer" or "Shadow Register"). When a segment selector is loaded into the visible portion of a segment register, the processor also loads the segment address, segment length, and access control information in the segment descriptor pointed to by the segment selector to the hidden portion of the segment register. The information buffered in the segment registers (visible and hidden portions) allows the processor to no longer spend time reading the base address and limit value from the segment descriptor when performing address translation.

Since the shadow register contains a copy of the descriptor information, the operating system must ensure that changes to the descriptor table should be reflected in the shadow register. Otherwise, the base address or limit of a segment in the descriptor table is modified, but the changes are not reflected in the shadow register. The simplest way to deal with this kind of problem is to reload 6 segment registers immediately after making any changes to the descriptors in the descriptor table. This will reload the corresponding segment information in the descriptor table into the shadow register.

There are two types of load instructions for loading segment registers:

1. Like MOV, POP, LDS, LES, LSS, LGS and LFS instructions. These instructions explicitly reference the segment register directly;
2. Implicitly loaded instructions such as CALL, JMP, and RET instructions using long pointers, IRET, INTn, INTO, and INT3 instructions. These instructions are accompanied by changes to the contents of the CS register (and some other segment registers) during operation.

The MOV instruction can of course also be used to store the contents of the visible part of the segment register in a general-purpose register.

4.3.4 Segment Descriptor

Earlier we explained that using a segment selector to locate a descriptor in the descriptor table. The segment descriptor is a data structure item in the GDT and LDT tables used to provide the processor with information about the position and size of a segment and the status of access control. Each segment descriptor is 8 bytes in length and contains three main fields: segment base address, segment length, and segment attributes. Segment descriptors are usually created by the compiler, linker, loader, or operating system, but are by no means applications. Figure 4-13 shows the general format of all types of segment descriptors.

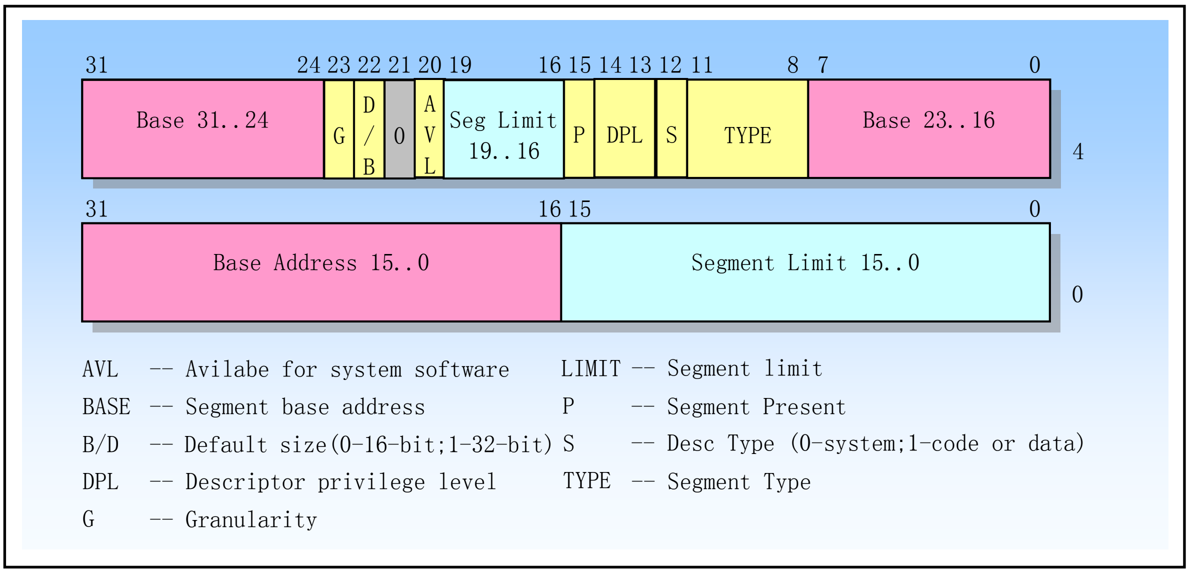


Figure 4-13 General format of segment descriptor

The meaning of fields and flags in a segment descriptor is as follows:

◆Segment limit field (LIMIT)

The segment limit field is used to specify the size of the segment. The processor will combine the two segment limit fields in the segment descriptor into a 20-bit value, and specify the actual meaning of the segment limit length Limit value according to the granularity flag G. If G=0, the segment length Limit range can be from 1 byte to 1 MB byte in units of bytes. If G=1, the segment length Limit ranges from 4KB to 4GB and the unit is 4KB.

Depending on the segment extension direction flag E in the segment type, the processor uses the segment limit Length in two different ways. For an upward-extending segment (abbreviated as an upper-extending segment), the offset value in the logical address can range from 0 to the segment-limit value Limit. An offset greater than the limit length limit will produce a general protective exception. For the segment that is extended downward (abbreviated as the lower segment), the meaning of the segment limit Limit is reversed. Depending on the setting of the default stack pointer size flag B, the offset value can range from the segment limit length to 0xFFFFFFFF or 0xFFFF. An offset value less than the limit length Limit will produce a general protective exception. For the next expanded segment, reducing the value in the segment limit field allocates new memory at the bottom of the segment address space instead of at the top. The 80X86's stack is always scaled down, so this implementation is well suited to extending the stack.

◆Base address field (BASE)

This field defines the location of a segment of byte 0 in the 4GB linear address space. The processor will combine 3 separate base address fields to form a 32-bit value. The segment base address should be aligned to a 16-byte boundary. Although this is not required, the best performance of the program can be achieved by aligning the code and data segments of the program on a 16-byte boundary.

◆Type field (TYPE)

The type field specifies the type of the segment or gate, the type of access to describe the segment, and the extension direction of the segment. The interpretation of this field depends on the descriptor type flag S, indicating whether it is an application (code or data) descriptor or a system descriptor. The encoding of the TYPE field is different for code, data, or system descriptors, as shown in Figure 4-14.

◆Descriptor type flag (S)

The descriptor type flag S indicates whether a segment descriptor is a system segment descriptor (when S=0) or a code or data segment descriptor (when S=1).

◆Descriptor privilege level (DPL)

The DPL field indicates the privilege level of the descriptor. The privilege level ranges from 0 to 3. The 0 privilege level is the highest and the 3 level is the lowest. DPL is used to control access to segments.

◆Segment present (P)

The segment presence flag P indicates whether a segment is in memory (P=1) or not in memory (P=0). When the P flag of a segment descriptor is 0, then loading the selector pointing to this segment descriptor into the segment register will result in the generation of a segment without an exception. Memory management software can use this flag to control the actual need to load that segment into memory at a given time. This feature provides virtual storage with control beyond the paging mechanism. Figure 4-15 shows the segment descriptor format when P=0. When the P flag is 0, the operating system is free to save its own data using fields that are marked as Available in the format, such as information about where the segment actually does not exist.

◆ Default operation size/default stack pointer size and/or upper bound (D/B)

According to the segment descriptor describes an executable code segment, spread data segment or a stack segment, this mark has a different function. (For 32-bit code and data segments, this flag should always be set to 1; for 16-bit code and data segments, this flag is set to 0.)

- **Executable code segment**. This flag is called the D flag at this time and is used to indicate that the instruction in this segment refers to a valid address and the default length of the operand. If the flag is set, the default value is a 32-bit address and a 32-bit or 8-bit operand; if the flag is 0, the default value is a 16-bit address and a 16-bit or 8-bit operand. The instruction prefix 0x66 can be used to select a non-default operand size; the prefix 0x67 can be used to select an address size other than the default.
- Stack segment (data segment pointed to by the SS register). At this point, this flag is called the B (Big) flag and indicates the size of the stack pointer when an implicit stack operation (such as PUSH, POP, or CALL) occurs. If this flag is set, the 32-bit stack pointer is used and stored in the ESP register; if the flag is 0, the 16-bit stack pointer is used and stored in the SP register. If the stack segment is set to a lower extended data segment, this B flag also specifies the upper bound of the stack segment.
- Expand the data segment. At this point the flag is called the B flag and is used to indicate the upper limit of the segment. If this flag is set, the upper bound of the stack segment is 0xFFFFFFFF (4GB); if this flag is not set, the upper bound of the stack segment is 0xFFFF (64KB).

◆Granularity (G)

This field is used to determine the unit of the segment-limited field value. If the granularity flag is 0, the unit of the segment limit value is bytes; if the granularity flag is set, the segment limit value uses 4 KB units. (This flag does not affect the granularity of the segment's base address. The base address's granularity is always in bytes.). If the G flag is set, the 12-bit least significant bit of the offset value is not checked when the segment length is used to check the offset value. For example, when G=1, the segment limit length of 0 indicates that the effective offset value is 0 to 4095.

◆Available and reserved bits

Bit 20 of the second double word of the segment descriptor is available for use by system software; Bit

21 is a reserved bit and should always be set to 0.

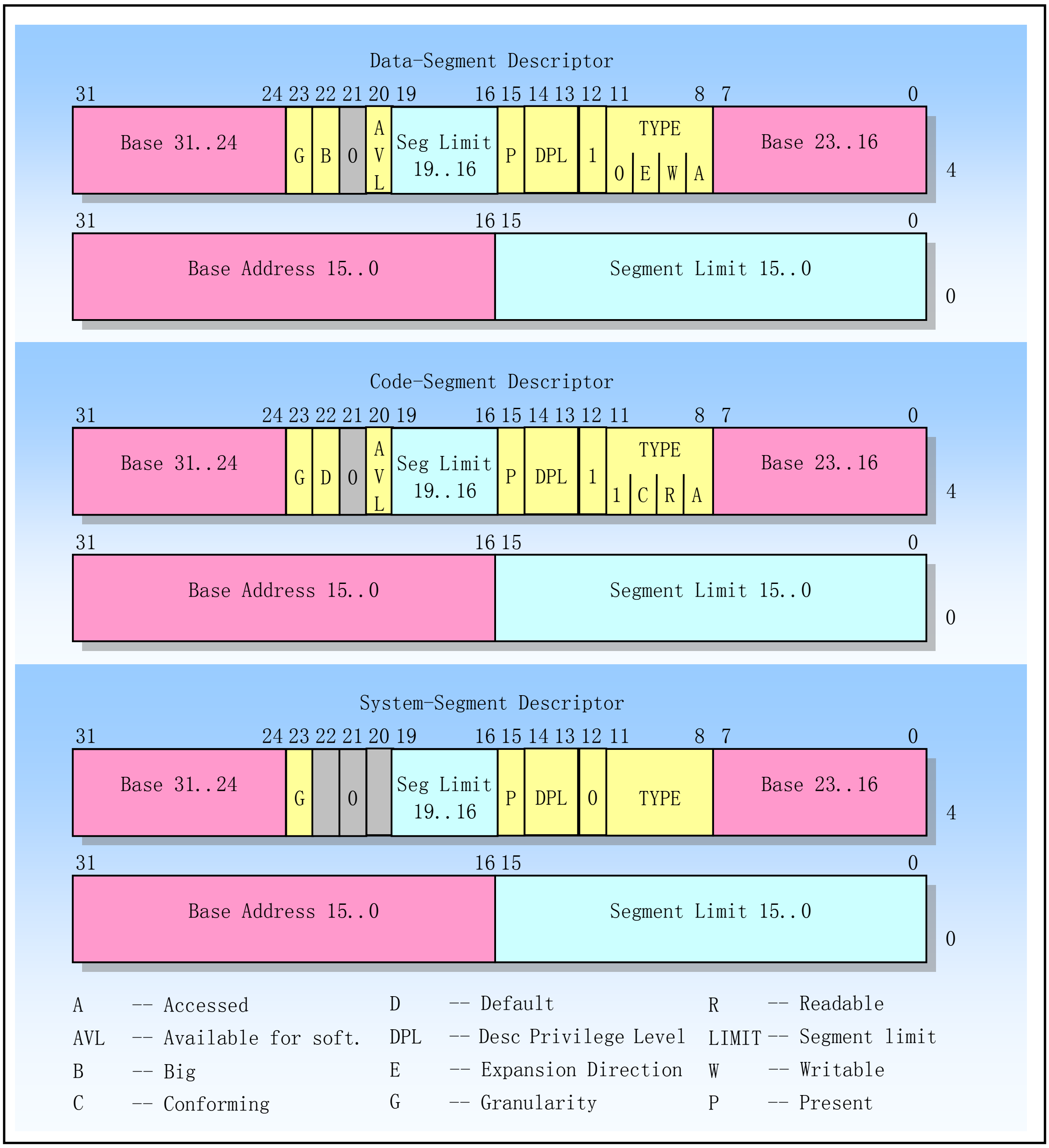


Figure 4-14 Code, data, and system segment descriptors formats

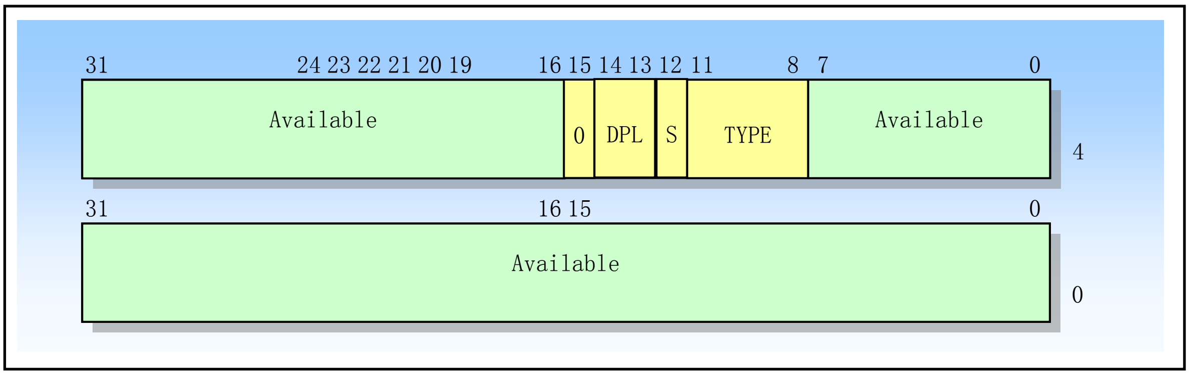


Figure 4-15 Segment Descriptor When bit P=0

4.3.5 Code and Data Segment Descriptor Types

When the S (descriptor type) flag is set in a segment descriptor, the descriptor is used for the code or data segment. At this point, the most significant bit in the type field (bit 11 of the second double word) is used to determine whether it is a data segment descriptor (reset) or a code segment descriptor (set).

For data segments, the lower 3 bits (bits 8, 9, 10) of the type field are used to indicate Accessed, Write-enable, and Expansion-direction, respectively. See Table 4-3 for descriptions of bit fields in the code and data segment type fields. According to the writable bit W setting, a data segment can be read-only or readable and writable.

Table 4-3 Code and Data Segment Descriptor Types

TYPE Field					**Descriptor Type**	**Description**
Decimal	**11**	**10**	**9**	**8**		
		E	**W**	**A**		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Expand-down, Read-Only.
5	0	1	0	1	Data	Expand-down, Read-Only, accessed
6	0	1	1	0	Data	Expand-down, Read/Write
7	0	1	1	1	Data	Expand-down, Read/Write, accessed
		C	**R**	**A**		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Conforming, Execute-Only
13	1	1	0	1	Code	Conforming, Execute-Only, accessed
14	1	1	1	0	Code	Conforming, Execute/Read-Only
15	1	1	1	1	Code	Conforming, Execute/Read-Only, accessed

Stack segment must be a read/write data segment. If a non-writable data segment selector is loaded into the

SS register, a general protection exception will result. If the length of the stack segment needs to change dynamically, the stack segment can be a data segment that is extended downwards (the expansion direction flag is set). Here, dynamically changing the segment limit will cause the stack space to be added to the bottom of the stack.

The accessed bit indicates whether a segment has been accessed since the last time the operating system resets the bit. Each time the processor loads a segment selector into the segment register, it sets this bit. This bit needs to be explicitly cleared, otherwise it remains set. This bit can be used for virtual memory management and debugging.

For code segments, the lower 3 bits of the type field are interpreted as Accessed, Read-enable, and Conforming. Depending on the setting of the readable R flag, the code segment can be execute-only or execute/read. An executable/readable code segment can be used when constants or other static data and instruction code are placed in a ROM. Here, we can read the data in the code segment by using an instruction with a CS override prefix or by loading the code segment selector for the code segment into a data segment register (DS, ES, FS, or GS). In protected mode, the code segments are not writable.

Code segments can be conforming or non-conforming. A transfer of execution control to higher privilege level conforming segment allows the program to continue execution at the current privilege level. A transfer to a non-conforming segment with a different privilege level will result in a general protection exception, unless a call gate or task gate is used- (for more information on consistent and non-conforming code segments, see "Directly Invoking or Jumping to Code segment"). System tools that do not access the protection facility and some exception types (such as errors, overflows) can be stored in the conforming segments. Tools that need to prevent access by low-privilege programs or procedures should be stored in non-conforming segments. Note that execution cannot be transferred by a call or a jump to a less-privileged (numerically higher privilege level) code segment, regardless of whether the target segment is a conforming or nonconforming code segment.

All data segments are non-conforming, meaning that they cannot be accessed by less privileged programs or procedures. However, unlike code segments, data segments can be accessed by higher privileged programs or procedures without the use of special access gates.

If a segment descriptor in the GDT or LDT is stored in the ROM, the processor will enter an infinite loop if the software or processor attempts to update (write) the segment descriptor in ROM. In order to prevent this problem, the accessed bit of all descriptors that need to be stored in the ROM should be set to the pre-set state. At the same time, delete any code in the operating system that attempts to modify the ROM segment descriptor.

4.3.6 System Descriptor Types

When the S flag (descriptor type) in a segment descriptor is in a reset state (0), the descriptor type is a system descriptor. The processor can recognize the following types of system descriptors:

- Local Descriptor Table (LDT) segment descriptor;
- Task-state segment (TSS) descriptor;
- Call-gate descriptor;
- Interrupt-gate descriptor;
- Trap-gate descriptor;
- Task-gate descriptor.

These descriptor types can be divided into two major categories: system segment descriptors and gate descriptors. The system segment descriptor points to the system segment (such as LDT and TSS segment). The gate descriptor is a "gate". For the call, interrupt or trap gate, it contains the selector of the code segment and the pointer of the program entry point in the segment; for the task gate, it contains the TSS segment selector. Table 4-4 shows the encoding of the system segment descriptor and gate descriptor type fields.

Table 4-4 System-Segment and Gate-Descriptor Types

TYPE Field					Description
Decimal	11	10	9	8	
0	0	0	0	0	Reserved
1	0	0	0	1	16-Bit TSS (Available)
2	0	0	1	0	LDT
3	0	0	1	1	16-Bit TSS (Busy)
4	0	1	0	0	16-Bit Call Gate
5	0	1	0	1	Task Gate
6	0	1	1	0	16-Bit Interrupt Gate
7	0	1	1	1	16-Bit Trap Gate
8	1	0	0	0	Reserved
9	1	0	0	1	32-Bit TSS (Available)
10	1	0	1	0	Reserved
11	1	0	1	1	32-Bit TSS (Busy)
12	1	1	0	0	32-Bit Call gate
13	1	1	0	1	Reserved
14	1	1	1	0	32-Bit Interrupt Gate
15	1	1	1	1	32-Bit Trap Gate

The use of TSS status segments and task gates will be explained in the task management section. The use of call gates will be described in the section on protection. The use of interrupts and trap gates will be used in interrupt and exception handling. Instructions are given in the section.

4.4 Paging

The paging mechanism is the second part of the 80X86 memory management mechanism. It completes the process of virtual (logical) address to physical address translation based on the segmentation mechanism. The segmentation mechanism translates logical addresses into linear addresses, while paging converts linear addresses into Physical addresses. Pagination can be used for any kind of segmented model. The processor paging mechanism divides the linear address space into which segments have been mapped, and these linear address space pages are then mapped to pages in the physical address space. Several page-level protection measures of the paging mechanism can be used in conjunction with the segment protection mechanism or replace the protection measures of the segmentation mechanism. For example, read/write protection can be enhanced on a page-based basis. In addition, on the page unit, the paging mechanism also provides user-superuser two-level protection.

The paging mechanism can be enabled by setting the PG bit in control register CR0. If PG=1, paging is enabled and the processor will use the mechanism described in this section to translate the linear address into a physical address. If PG=0, the paging mechanism is disabled, and the linear address generated by the segmentation mechanism is directly used as a physical address.

The previously described segmentation mechanism operates on various variable size memory regions. Unlike fragmentation, paging mechanisms operate on fixed-size blocks of memory (called pages). The paging mechanism divides the linear and physical address spaces into pages. Any page in the linear address space can be mapped to any page in the physical address space. Figure 4-16 shows how the paging mechanism divides the linear and physical address spaces into pages and provides an arbitrary mapping between these two spaces. The arrows in the figure correspond the pages in the linear address space to the pages in the physical address space.

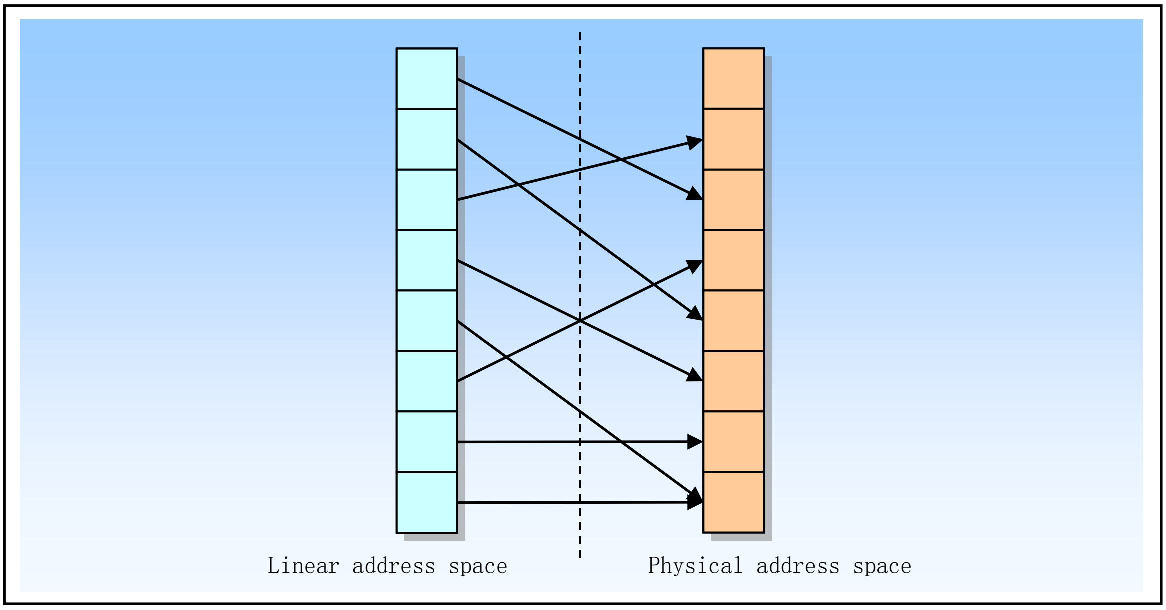


Figure 4-16 Illustration of correspondence between linear and physical address space pages

The 80X86 uses 4K (2^12) bytes of fixed-size pages and is aligned at the 4K address boundary. This means that the paging mechanism divides 2^32 bytes (4GB) of linear address space into 2^20 (1M = 1048576) pages. The paging mechanism operates by relocating pages in a linear address space into physical address space. Since a 4K page is mapped as a unit and aligned to a 4K boundary, the lower 12 bits of the linear address can be used as the in-page offset directly as the lower 12 bits of the physical address. The relocation function performed by the paging mechanism can be seen as converting the upper 20 bits of the linear address to the upper 20 bits of the corresponding physical address.

When paging is used, the processor divides the linear address space into fixed-size pages (length 4KB) that can be mapped into physical memory and/or disk storage. When a program (or task) references a logical address in memory, the processor translates the logical address into a linear address and then uses a paging mechanism to translate the linear address into a corresponding physical address. If a page containing a linear address is not currently in physical memory, the processor generates a page fault exception. A page fault exception handler usually causes the operating system to load the corresponding page into physical memory from disk (it may also write different pages in physical memory to disk during operation). After the page is loaded into physical memory, the return from the exception handler causes the instruction that caused the exception to be re-executed. The information used by the processor to translate linear addresses into physical addresses and to generate page fault exceptions (if necessary) is contained in page directories and page tables stored in memory.

The biggest difference between paging and segmentation is that paging uses fixed-length pages. If you only use segmented address translation, a data structure stored in physical memory will contain all of its parts. However, if pagination is used, one data structure can be stored in physical memory in part and another part in disk.

In order to reduce the number of bus cycles required for address translation, the most recently accessed page directory and page table are stored in a processor buffer, known as the Translation Lookaside Buffer (TLB). The

TLB can satisfy most read page directories and page table requests without using bus cycles. Only when the TLB does not contain the required page table entries will an extra bus cycle be used to read the page table entries from memory. This usually occurs when a page table entry has not been accessed for a long time.

4.4.1 Page Table Structure

The paging translation is described by a table that resides in memory. This table is called a page table and is stored in a physical address space. The page table can be seen as a simple array with 2^20 items. The linear to physical address mapping function can simply be seen as an array search. The upper 20 bits of the linear address form the index of this array and is used to select the physical (base) address of the corresponding page. The lower 12 bits of the linear address give the offset in the page, plus the base address of the page eventually forms the corresponding physical address. Since the page base address is aligned on the 4K boundary, the bottom 12 bits of the page base address must be 0. This means that the 20-bit page base address and the 12-bit offset connection are combined to get the corresponding physical address.

Each page table entry in the page table has a size of 32 bits. Since only 20 bits are needed to store the physical base address of the page, the remaining 12 bits can be used to store attribute information such as whether the page exists or not. If the linear address index page table entry is marked as existing, then the item is valid, we can get the physical address of the page. If the item indicates that it does not exist, an exception will be generated when accessing the corresponding physical page.

4.4.1.1 Two-Level Page Table Structure

A page table contains 2^20 (1M) entries, each occupying 4 bytes (32 bits). If they are only stored as one table, they will occupy up to 4MB of memory. Therefore, in order to reduce the memory footprint, the 80X86 uses two levels of tables. Thus, the conversion of a high 20-bit linear address to a physical address is also performed in two steps, using 10 bits per step.

The first level table is called a page directory. Occupies one page with 2^10 (1K) entries of 4 bytes in length. These entries point to the corresponding secondary table. The top 10 bits (bits 31 - 22) of the linear address are used as index values in the primary table (page directory) to select one of the 2^10 secondary tables.

The second level table is called a page table. Its length is also a page, and it contains at most 1K 4-byte entries. Each 4-byte table entry contains the 20-bit physical base address of the associated page. The secondary page table uses the middle 10 bits of the linear address (bits 21--12) as the index of the entry to obtain the entry containing the 20-bit physical base address of the page. The 20-bit page physical base address and the lower 12 bits (in-page offset) in the linear address are combined to obtain the output value of the page conversion process, ie, the corresponding final physical address.

Figure 4-17 shows the two-level table lookup process. The CR3 register specifies the base address of the page directory table. The upper 10 bits of the linear address are used to index this page directory table to obtain a pointer to the associated second-level page table. The central 10 bits of the linear address are used to index the secondary page table to obtain the upper 20 bits of the physical address. The lower 12 bits of the linear address are directly used as the physical address low 12 bits to form a complete 32-bit physical address.

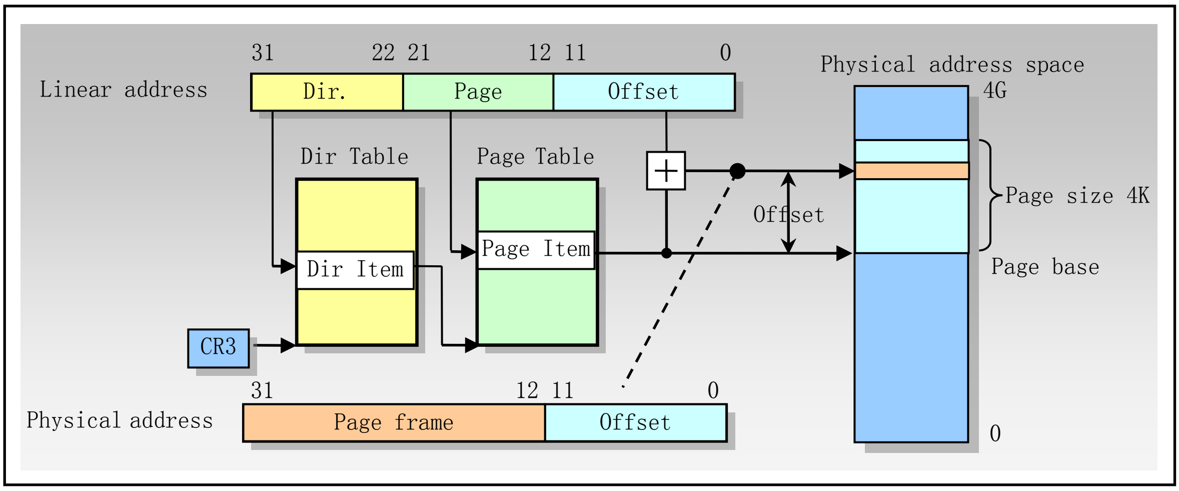


Figure 4-17 Linear Address Translation

4.4.1.2 Nonexistent Page Tables

By using the two-level table structure, page tables are allowed to be spread across the pages of memory without being stored in consecutive 4MB memory blocks. In addition, there is no need to allocate secondary page tables for portions that are not present or unused in the linear address space. Although directory pages must always exist in physical memory, secondary page tables can be redistributed as needed. This makes the size of the page table structure correspond to the actual use of the linear address space size.

Each table entry in the page directory table also has a present attribute, similar to the table entry in the page table. The presence attribute in the page directory entry indicates whether the corresponding secondary page table exists. If the directory entry indicates that the corresponding secondary page table exists, then by accessing the secondary table, the second step of the table lookup process will continue as described above. If there is a bit indicating that the corresponding secondary table does not exist, the processor will generate an exception to notify the operating system. The presence attribute in the page directory entries allows the operating system to allocate secondary page table pages based on the actual linear address range used.

The presence bits in the page directory entries can also be used to store secondary page tables in virtual memory. This means that only part of the secondary page table needs to be stored in physical memory at any time, while the rest can be stored on disk. The page directory entries corresponding to the page tables in physical memory will be marked as present to indicate that they can be paged. The page directory entry for the page table on disk will be marked as not present. An exception caused by the absence of a secondary page table informs the operating system to load the missing page table from disk into physical memory. Storing page tables in virtual memory reduces the amount of physical memory needed to save the paging translation tables.

4.4.2 Page-Directory and Page-Table Entries

The format of the page directory entry and page table entry is shown in Figure 4-18. The bits 31--12 contain the upper 20 bits of the physical address and are used to locate the physical base address of a page (also called a page frame) in the physical address space. The lower 12 bits of the table entry contain page attribute information. We have already discussed the existence attributes. Here we briefly describe the functions and uses of the remaining attributes.

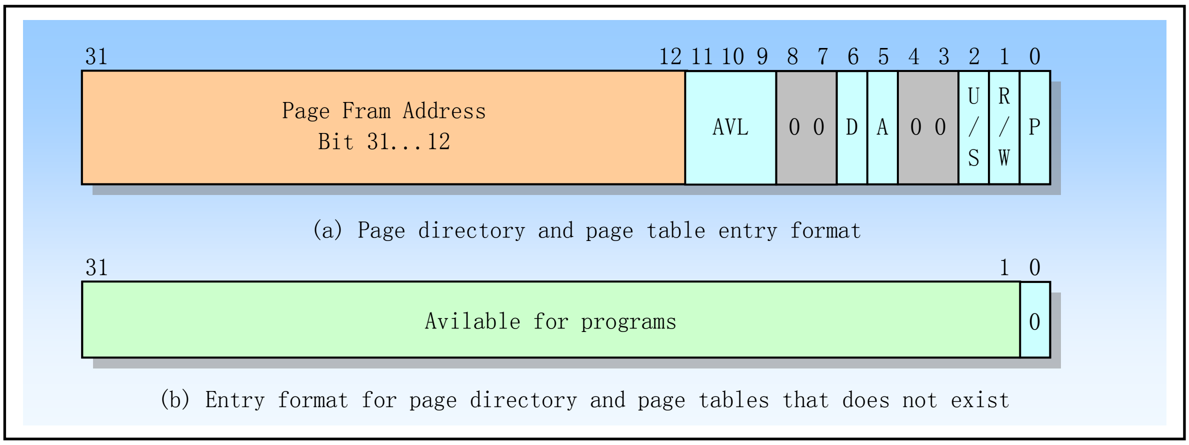

31
12 11 10 9 8 7 6 5 4 3 2 1 0
Page Fram Address
Bit 31...12
AVL
0 0
D
A
0 0
U/S
R/W
P
(a) Page directory and page table entry format
31
1 0
Avilable for programs
0
(b) Entry format for page directory and page tables that does not exist

Figure 4-18 Page directory and page table entry format

- P – Bit 0 is Present flag. It indicates whether the page or page table pointed to by the table entry is currently loaded into physical memory. After the flag is set, it indicates that the page is in physical memory and performs address translation. When the flag is cleared, it means that the page is not in memory. If the processor tries to access the page, a page fault exception will be generated. At this point the operating system can use the rest of the entry to store information such as the location of the page in the disk system.
- R/W -- Bit 1 is the Read/Write flag. If set to 1, it means the page can be read, written or executed. If 0, the page is read-only or executable. The R/W bit has no effect when the processor is running at the superuser privilege level (level 0, 1 or 2), refer to the U/S flag below. The R/W bit in the page directory entry acts on all pages it maps to.
- U/S -- Bit 2 is the User/Supervisor flag. If set to 1, then the program running on any privilege level can access the page. If 0, the page can only be accessed by programs running on the superuser privilege level (0, 1, or 2). The U/S bit in the page directory entry acts on all pages it maps to.
- A -- Bit 5 is the Accessed flag. This flag of the page table entry is set to 1 when the processor accesses the page mapped by the page table entry. This flag of the page directory entry is set to 1 when the processor accesses any page mapped by the page directory entry. The processor is only responsible for setting this flag, and the operating system can count the usage of the page by periodically resetting the flag.
- D -- Bit 6 is the page modified (Dirty) flag. When the processor performs a write operation on a page, the D flag corresponding to the page table entry is set. The processor does not modify the D flag in the page directory entry.
- AVL – Available field. This field is reserved for program use. The processor will not modify these bits, and the future upgrade processor will not.

4.4.3 **Virtual Memory**

The presence flag P in the page directory and page table entries provides the necessary support for virtual storage using paging technology. If the page in the linear address space exists in the physical memory, the flag P=1 in the corresponding entry and the corresponding physical address is included in the entry. A table whose page is not in physical memory has its flag P = 0. If the program accesses a page that does not exist in physical memory, the processor generates a page fault exception. At this point, the operating system can use this exception handling process to transfer the missing pages from the disk to the physical memory, and store the corresponding physical address in the table entry. Finally, the flag P=1 is set before the return program re-executes the instruction that caused the exception.

The accessed flag A and the modified flag D can be used to effectively implement virtual memory technology. By periodically checking and resetting all A flags, the operating system can determine which pages have not been accessed recently. These pages can be candidates for removal to disk. Suppose that when a page is read into memory from disk, its dirty flag D=0, then when the page is moved out to disk again, if the D flag is still 0, the page does not need to be written to disk. If D=1 at this time, the page content has been modified, so the page must be written to disk.

4.5 Protection

In protected mode, the 80X86 provides segment and page level protection. This protection mechanism provides access restrictions on certain segments and pages based on privilege levels (4 levels of protection and level 2 page protection). For example, operating system code and data are stored in segments that have a higher privilege level than normal applications. The processor's protection mechanism will then limit the application's access to the operating system's code and data in a controlled and regulated manner.

Protection mechanisms are required for a reliable multitasking environment. It can be used to protect individual tasks from mutual interference. Segment and page level protection can be used at any stage of software development to assist in finding and detecting design issues and errors. When the program performs an undesired reference to the error memory space, the protection mechanism can block such operations and report such events.

Protection mechanisms can be used for segmentation and paging mechanisms. The 2 bits of the processor register define the privilege level of the currently executing program, called the Current Privilege Level (CPL). During segmentation and paging address translation, the processor will verify the CPL.

By setting the PE flag (bit 0) of the control register CR0, the processor can be operated in protected mode, thus turning on the segmentation protection mechanism. Once in protected mode, there is no clear control flag in the processor to stop or enable the protection mechanism. However, the privilege-level protection mechanism part can be implicitly turned off by setting the privilege level of all segment selectors and segment descriptors to level 0. This approach can prohibit privilege-level protection barriers between segments, but other segment length and segment type checks and other protection mechanisms still work.

Setting the PG flag (bit 31) of the control register CR0 enables the paging mechanism and also enables the paging protection. Similarly, there are no associated flags in the processor to disable or enable the page level protection mechanism in the paging open condition. But by setting the read/write (R/W) flag and the user/superuser (U/S) flag for each page directory entry and page table entry, we can disable page-level protection. Setting these two flags allows each page to be arbitrarily read/write, so page-level protection is actually disabled.

For the segment level protection, the processor performs protection verification using the selectors (RPL and CPL) in the segment register and the fields in the segment descriptor. For the paging mechanism, the R/W and U/S flags in the page directory and page table entries are mainly used to implement the protection operation.

4.5.1 Segment Protection

When the protection mechanism is used, each memory reference is checked to verify that the memory reference meets various protection requirements. Because the check operation is concurrent with the address translation, processor performance is not affected. The protection checks performed can be divided into the following categories:

- Segment limit checks;
- Segment type checks;
- Privilege level checks;
- Restriction of addressable domain;

- Restriction of procedure entry-points;
- Restriction of instruction set.

All violations of protection will result in an exception. The following sections describe the protection mechanisms in protected mode.

4.5.1.1 Segment Length Limit Check

The segment limit length field of the segment descriptor is used to prevent program or process addressing to the location outside the segment. The effective value of the segment length depends on the setting state of the granularity G flag. For data segments, the segment length is also related to the flag E (extended direction) and the flag B (default stack pointer size and/or upper bound). The E flag is a bit of the type field in the segment descriptor of the data segment type.

When the G flag is cleared (byte granularity), the effective segment length is the value of the segment descriptor length field Limit of the 20-bit segment descriptor. In this case, Limit ranges from 0 to 0xFFFFF (1MB). When the G flag is set (4KB page granularity), the processor multiplies the value of the Limit field by a factor of 4K. In this case, the valid Limit range is from 0xFFF to 0xFFFFFFFF (4GB). Note that when the G flag is set, the lower 12 bits of the segment offset (address) are not checked against Limit. For example, when the segment length Limit is equal to 0, the offset values 0 to 0xFFF are still valid.

Except for the expand-down data segment, the value of the valid Limit for all other segment types is the last address allowed to be accessed in the segment, which is one byte smaller than the segment length. Any valid address range specified beyond the segment length field will result in a general protection exception.

For the expand-down data segment, the segment length has the same function, but its meaning is different. Here, the segment length specifies the last address in the segment that is not allowed to access, so in the case where the B flag is set, the effective offset range is from (valid segment offset +1) to 0xFFFF FFFF; when B is cleared The valid offset value range is from (valid segment offset +1) to 0xFFFF. When the segment length of the next expansion segment is 0, the segment will have the maximum length.

In addition to checking the segment length, the processor also checks the length of the descriptor table. The GDTR, IDTR, and LDTR registers contain a 16-bit limit value that the processor uses to prevent the program from selecting descriptors outside of the descriptor table. The limit length value of the descriptor table indicates the last valid byte in the table. Since each descriptor is 8 bytes long, the table containing N descriptor entries should have a limit value of 8N-1.

The selector can have a value of zero. Such a selector points to the first unused descriptor item in the GDT table. Although this null selector can be loaded into a segment register, any attempt to reference memory using this descriptor will result in a general protection exception.

4.5.1.2 Segment Type Checking

The segment descriptor contains type information in two places, namely the S flag in the descriptor and the type field TYPE. The processor uses this information to detect programming errors caused by illegal use of segments or gates.

The S flag is used to indicate whether a descriptor is of a system type or a code or data type. The TYPE field additionally provides 4 bits for defining various types of code, data, and system descriptors. The table in the previous section gives the encoding of the code and data descriptor TYPE fields; the other table gives the encoding of the system descriptor TYPE field.

When the segment selector and descriptor are manipulated, the processor will check the type information at any time. The type information is checked mainly in the following two cases:

1. When a segment selector is loaded into a segment register. Certain segment registers can contain only certain descriptor types, for example:

- The CS register can only be loaded with a selector for a executable code segment;
- The selector of the unreadable executable segment cannot be loaded into the data segment register;
- Only selectors of writable data segments can be loaded into the SS register.

2. When instructions access segments whose descriptors are already loaded into segment registers. Certain segments can be used by instructions only in certain predefined ways, for example:
   - No instruction can write an executable segment;
   - No instruction can write into a data segment where the writable bit is not set;
   - No instruction can read an executable segment unless the executable the readable flag is set.

4.5.1.3 Privilege Levels

The segment protection mechanism of the processor can identify 4 privilege levels (or privilege layers), 0 to 3 levels. The greater numbers mean lesser privileges. Figure 4-19 shows how these privilege levels can be interpreted as protection ring forms. The center (retained for the most advanced code, data, and stack) is used for segments that contain the most important software, usually for the core part of the operating system. The middle two rings are used for more important software. Systems using only 2 privilege levels should use privilege levels 0 and 3.

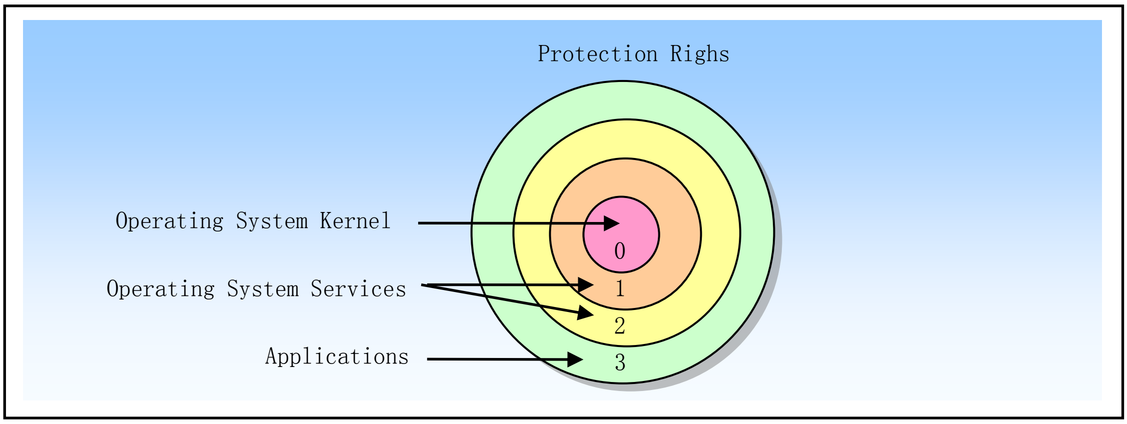


Figure 4-19 Protection Level Rings

The processor utilizes a privilege levels to prevent a program or task running at a lower privilege level from accessing a segment with a higher privilege level, except under controlled conditions. When the processor detects an operation that violates a privilege level, it generates a general protection exception.

In order to perform privilege level checking between individual code segments and data segments, the processor can recognize the following three types of privilege levels:

- **Current Privilege Level (CPL)**. The CPL is the privilege level of the currently executing program or task. It is stored in bits 0 and 1 of the CS and SS segment registers. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The processor changes the CPL when program control is transferred to a code segment with a different privilege level. The CPL is treated slightly differently when accessing conforming code segments. Conforming code segments can be accessed from any privilege level that is equal to or numerically greater (less privileged) than the DPL of the conforming code segment. Also, the CPL is not changed when the processor accesses a conforming code segment that has a different privilege level than the CPL.
- **Descriptor Privilege Level (DPL)**. The DPL is the privilege level of a segment or gate. It is stored in the DPL field of the segment or gate descriptor for the segment or gate. When the currently executing

code segment attempts to access a segment or gate, the DPL of the segment or gate is compared to the CPL and RPL of the segment or gate selector (as described later in this section). The DPL is interpreted differently, depending on the type of segment or gate being accessed:

- **Data Segment.** Its DPL indicates the numerically highest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a data segment is 1, only programs running at a CPL of 0 or 1 can access the segment.
- **Nonconforming code segment (without using a call gate)**. The DPL indicates the privilege level that a program or task must have to access the segment. For example, if the DPL of a nonconforming code segment is 0, then only programs running at CPL 0 can access this segment.
- **Call Gate**. Its DPL indicates the numerically highest privilege level at which the current executing program or task accessing the call gate can be. (This is the same as the access rule for the data segment.)
- **Conforming code segment and nonconforming code segment accessed through a call gate**. The DPL indicates the numerically lowest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a conforming code segment is 2, programs running at a CPL of 0 or 1 cannot access the segment.
- **Task status segment TSS.** Its DPL indicates the numberically highest privilege level at which the current executing program or task accessing the TSS can be. (This is the same as the access rule for the data segment.)

- **Request privilege level RPL**. The RPL is an override privilege level assigned to a segment selector, which is stored in bits 0 and 1 of the selector. The processor checks both the RPL and the CPL to determine if access to a segment is allowed. Even if a program or task has sufficient privilege level (CPL) to access a segment, access will be denied if the provided RPL privilege level is insufficient. That is, if the RPL of the segment selector has a value greater than the CPL, the RPL will overwrite the CPL (and use the RPL as the privilege level for checking comparisons), and vice versa. That is, the privilege level with the largest value in the RPL and CPL is always taken as the comparison object when accessing the segment. Therefore, RPL can be used to ensure that high privileged code does not access a segment on behalf of the application unless the application itself has access to the segment.

The privilege level check operation is performed when the segment selector of the segment descriptor is loaded into a segment register, but the check method for data access is different from the one for checking the program control transfer between code segments. Therefore, the following two access situations are considered.

4.5.2 Privilege Level Check When Accessing Data Segments

To access operands in a data segment, the segment selector for the data segment must be loaded into the data-segment registers (DS, ES, FS, or GS) or into the stack-segment register (SS).(Segment registers can be loaded with the MOV, POP, LDS, LES, LFS, LGS, and LSS instructions.) Before the processor loads a segment selector into a segment register, it performs a privilege check (refer to Figure 4-20) by comparing the privilege levels of the currently running program or task (the CPL), the RPL of the segment selector, and the DPL of the segment's segment descriptor. The processor loads the segment selector into the segment register if the DPL is numerically greater than or equal to both the CPL and the RPL. Otherwise, a generalprotection fault is generated and the segment register is not loaded.

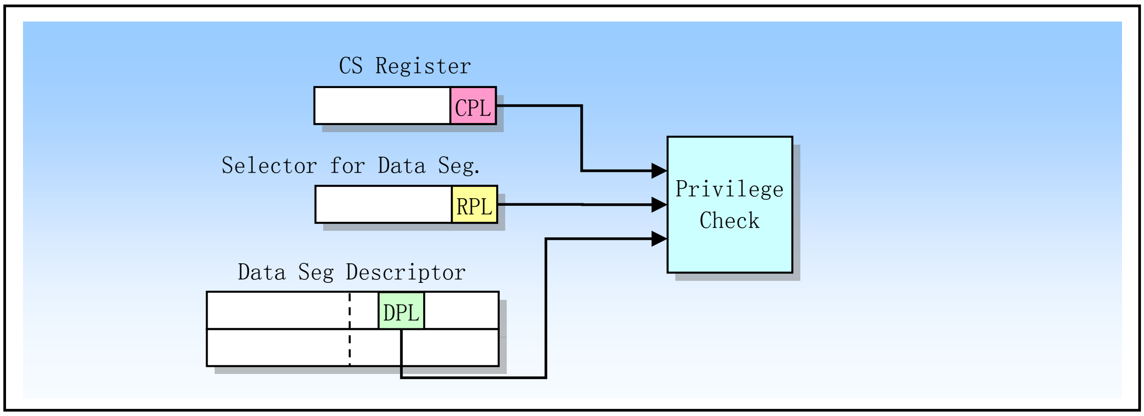


Figure 4-20 Privilege level check when accessing data segments

It can be seen that a program or task addressable area changes as its CPL changes. When the CPL is 0, the data segments at all privilege levels can be accessed at this time; when the CPL is 1, only the data segments at the privilege levels 1 to 3 can be accessed; when the CPL is 3, only the privileged level The data segment of 3 can be accessed.

In addition, it may desirable to access data structures that are contained in a code segment, for example, when the code and data are in ROM. So sometimes we will need access to the data in the code segment. At this point you can use the following methods to access the data in the code segment:

1. Load the selector of a nonconforming, readable, code segment into a data segment register.
2. Load the selector of a conforming, readable, code segment into a data segment register.
3. Use the code segment override prefix (CS) to read a readable code segment whose selector is already in the CS register.

The same rules for accessing data segments also apply to Method 1. Method 2 is always valid because the privilege level of the consistent code segment is equivalent to the CPL, regardless of the DPL of the code segment. Method 3 is also always valid because the DPL of the code segment selected by the CS register is the same as the CPL.

A privilege level check is also performed when the SS segment register is loaded using the stack segment selector. All privilege levels associated with the stack segment here must match the CPL. That is, the CPL, the RPL of the stack segment selector, and the DPL of the stack segment descriptor must all be the same. If the RPL or DPL is different from the CPL, the processor will generate a general protection exception.

4.5.3 Privilege Level Checking When Transferring Program Control Between Code Segments

For transferring program control from one code segment to another, the segment selector for the target code segment must be loaded into the code segment register (CS). As part of this loading process, the processor detects the segment descriptor for the target code segment and performs various limit, type, and privilege level checks. If these checks pass, the target code segment selector is loaded into the CS register, and control of the program is transferred to the new code segment, and the program will begin execution at the instruction pointed to by the EIP register.

The control transfer of the program is implemented using the instructions JMP, RET, INT, and IRET as well as exception and interrupt mechanisms. Exceptions and interrupts are special implementations that will be described later. This section discusses only the JMP, CALL, and RETS instructions. A JMP or CALL instruction can reference another code segment in one of four ways:

- The target operand contains the segment selector for the target code segment;
- The target operand points to a call gate descriptor, which contains the selector for the target code segment;
- The target operand points to a TSS, which contains the selector for the target code segment;
- The target operand points to a task gate that points to a TSS, which contains the selector for the target code segment;

The first two reference types are described below, the latter two of which are described in the section on task management.

4.5.3.1 Direct Calls or Jumps to Code Segments

The near forms of the JMP, CALL, and RET instructions simply performs program control transfers in the current code segment, so privilege level checks are not performed. The far forms of the JMP, CALL, or RET instructions transfer control to another code segments, so the processor must perform privilege level checks.

When transfering program control to another code segment without going through a call gate, the processor verifies four kinds of privilege level and type information as shown in Figure 4-21:

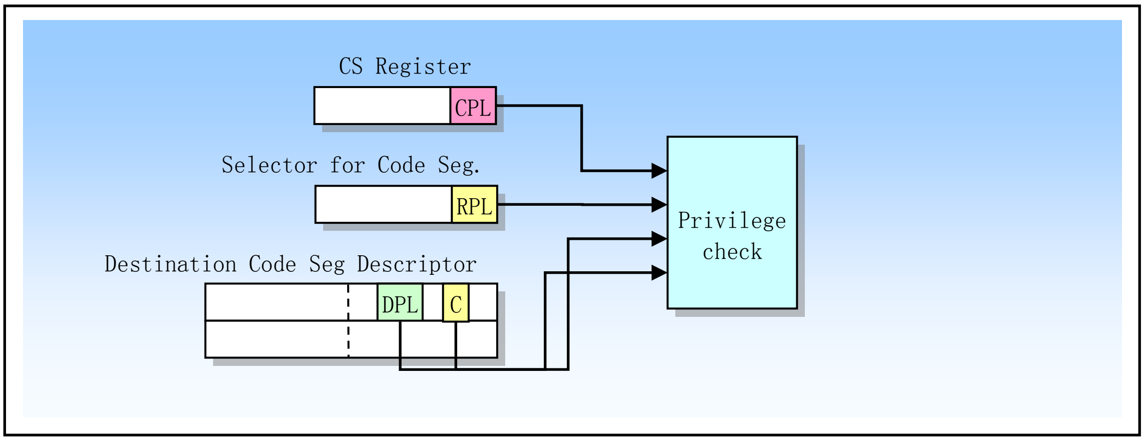


Figure 4-21 Privilege check when calling or jumping directly to another code segment

- Current privilege level CPL. (Here, CPL is the privilege level of the code segment that executes the call, that is, the code segment that executes the call or jump.)
- The descriptor privilege level DPL of the segment descriptor for the destination code segment that contains the called procedure.
- Request privilege level RPL in the segment selector of the destination code segment.
- The conforming flag C in the destination code segment descriptor. It determines whether a code segment is a non-conforming code segment or a consistent code segment.

The rules for the processor to check the CPL, RPL, and DPL depend on the setting state of the flag C. When accessing a non-conforming code segment (C=0), the CPL of the caller (program) must be equal to the DPL of the destination code segment, otherwise a general protection exception will be generated. The RPL of the segment selector pointing to a non-conforming code segment has a limited effect on the check. The RPL must be numerically less than or equal to the caller's CPL in order for the control transfer to complete successfully. When the segment selector of a non-conforming code segment is loaded into the CS register, the privilege level field does not change, ie it is still the caller's CPL. This is true even if the RPL of the segment selector is different from the CPL.

When accessing a conforming code segment (C = 1), the caller's CPL may be numerically greater than or equal to the DPL of the destination code segment. The processor will only generate a general protection exception when CPL < DPL. For access to conforming code segments, the processor ignores the check for RPL. For a conforming code segment, DPL represents the numerically lowest privilege level at which the caller can make a successful call to the code segment.

When program control is transferred to a conforming code segment, the CPL does not change, even if the DPL of the destination code segment is numerically less than the CPL. This is the only case where the CPL may not be the same as the current code segment DPL. Also, since the CPL has not changed, the stack will not switch.

Most code segments are non-conforming code segments. For these segments, control of the program can only be transferred to code segments with the same privilege level, unless the transfer is through a call gate, as explained below.

4.5.3.2 Gate Descriptors

To provide controlled access to code segments with different privilege levels, the processor provides a special set of descriptors called gate descriptors. There are four kinds of gate descriptors:

- Call Gate (TYPE=12);
- Trap Gate (TYPE=15);
- Interrupt Gate (TYPE=14);
- Task Gate (TYPE=5).

The task gate is used for task switching and will be explained later in the Task Management section. Trap gates and interrupt gates are special classes for calling gates that are used to call handlers for exceptions and interrupts, as explained in the next section. This section only describes how to use the call gate.

Call gates are used to implement controlled program control transfers between different privilege levels. They are usually only used in operating systems that use privilege-level protection mechanisms. Figure 4-22 shows the format of the call gate descriptor. The call gate descriptor can be stored in the GDT or LDT, but cannot be placed in the interrupt descriptor table IDT. A call gate has the following main functions:

- Specifies the code segment to be accessed;
- Defines an entry point for a procedure (program) in a specified code segment;
- Specifies the privilege level that the caller of the access procedure needs to have;
- If a stack switch occurs, it specifies the number of optional parameters that need to be copied between the stacks;
- Indicates whether the call gate descriptor is valid.

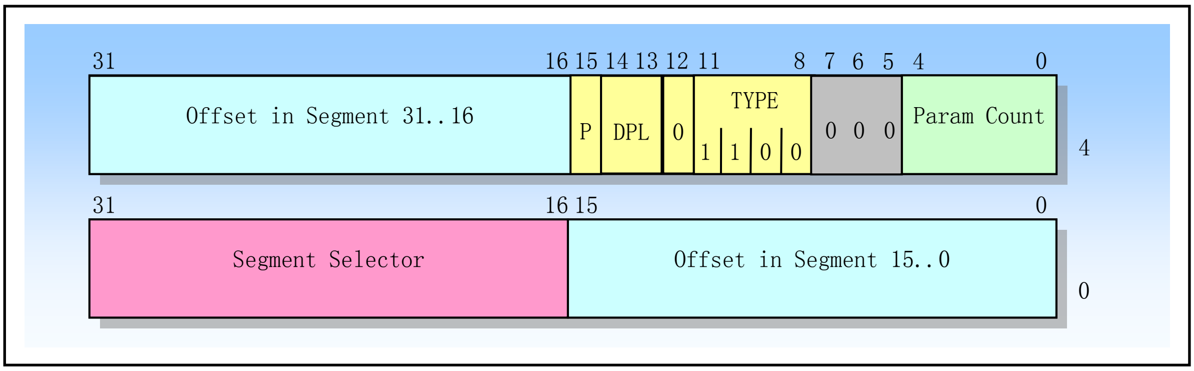


Figure 4-22 Call gate descriptor format

The segment selector field in the call gate specifies the code segment to be accessed. The Offset Value field specifies the entry point in the segment. This entry point is usually the first instruction in the specified process. The DPL field specifies the privilege level of the call gate, thereby specifying the privilege level required to access a particular procedure through the call gate. The flag P indicates whether the call gate descriptor is valid. The Parameter Count field (Param Count) indicates the number of parameters copied from the caller stack to the new stack when a stack switch occurs.

The call gate is not used in the Linux kernel. The description of the call gate is intended to prepare for the processing of interrupts and exception gates in the next section.

4.5.3.3 Accessing a Code Segment Through a Call Gate

In order to access the call gate, we need to provide a far pointer for the operand of the CALL or JMP instruction. The segment selector in this pointer is used to specify the call gate. The offset value of the pointer is needed, but the processor does not use it. This offset value can be set to any value. See Figure 4-23.

When the processor accesses the call gate, it uses the segment selector in the call gate to locate the segment descriptor for the destination code segment. The processor then combines the base address of the code segment descriptor with the offset value in the call gate to form the linear address of the specified program entry point in the code segment.

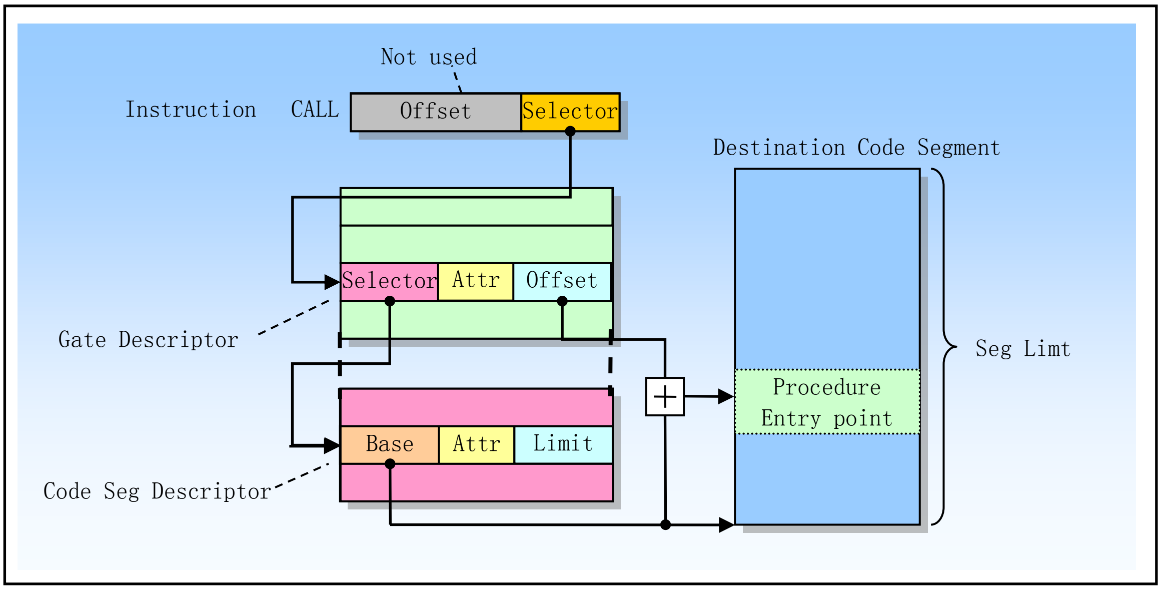


Figure 4-23 Call-gate operation process

When a program control transfer is made by a call gate, the CPU checks four different privilege levels to determine the validity of the control transfer, as shown in Figure 4-24.

- The current privilege level CPL;
- The requestor's privilege level RPL of the call gate's selector;
- The descriptor privilege level DPL of the call gate descriptor;
- The DPL of the segment descriptor of the destination code segment.

In addition, the conforming flag C in the destination code segment descriptor will also be checked.

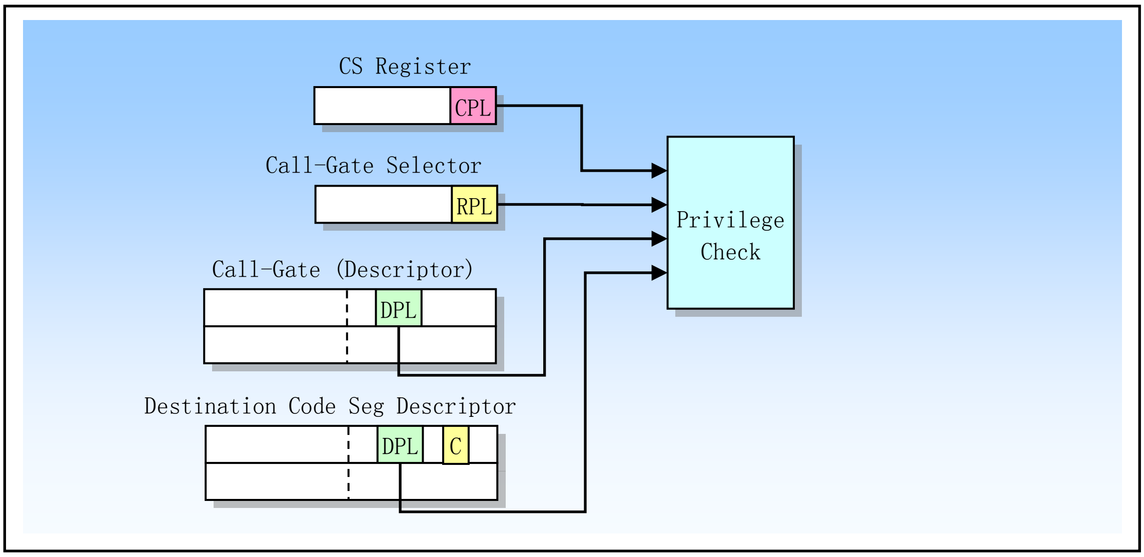


Figure 4-24 Privilege level check for control transfer with call-gate

The privilege level checking rule is different for controlling transfer using CALL instruction or JMP instruction, as shown in Table 4-5. The DPL field of the call gate descriptor indicates the numerically maximum privilege level (least privilege level) at which the caller can access the call gate. That is, in order to access the call gate, the privilege level CPL of the caller program must be less than or equal to the DPL of the call gate. The RPL of the segment selector of the call gate also needs to follow the same rules as the CPL that invokes this, ie the RPL must also be less than or equal to the DPL of the calling gate.

Table 4-5 Privilege level check rules for CALL and JMP instructions

Instruction	Privilege Check Rules (numerically)
CALL	CPL<= Call gate DPL; RPL<= Call gate DPL Destination conforming & nonconforming code segments DPL<= CPL
JMP	CPL<= Call gate DPL; RPL<= Call gate DPL Destination conforming code segment DPL<= CPL; Destination nonconforming code segment DPL=CPL

If the privilege level check between the caller and the call gate succeeds, the CPU will then compare the caller's CPL with the DPL of the code segment descriptor. In this regard, the CALL instruction and the JMP instruction check rules are different. Only the CALL instruction can use call gate to transfer program control to the more privileged (numerically lower privilege level) non-conforming code segment, that is, it can be transferred to the non-conforming code segment with a DPL less than the CPL. A JMP instruction can use a call gate only to transfer program control to a nonconforming code segment with a DPL equal to the CPL. However, both the CALL instruction and the JMP instruction can transfer control to a conforming code segment of a higher privilege level, that is, to a conforming code segment where the DPL is numerically less than or equal to the CPL.

If a call transfers control to a non-conforming code segment of a higher privilege level, the CPL is set to the DPL value of the destination code segment and causes a stack switch. But if a call or jump transfers control to a higher privileged level conforming code segment, the CPL does not change and does not cause a stack switch.

Call gate allows a procedure in a code segment to be accessed by a program of a different privilege level. For example, operating system code located in a code segment may contain code that the operating system itself and application software allow to access (such as code that handles character I/O). So you can set up a call gate for all of these procedures that all privilege level code can access. In addition, some higher privilege level call gates can be set specifically for code that is only used by the operating system.

4.5.3.4 Stack Switching

Whenever the call gate is used to transfer program control to a more privileged non-conforming code segment, the CPU automatically switches to the stack of the privilege level of the destination code segment. The purpose of the stack switch operation is to prevent more privileged programs from crashing due to insufficient stack space, and also to prevent low privilege level programs from intentionally or unintentionally interfering with high privilege level programs through the shared stack.

Each task must define up to 4 stacks. One is for application code running at privilege level 3, and the other is used for privilege levels 2, 1, and 0, respectively. If only two privilege levels of 3 and 0 are used in a system, then only two stacks need to be set for each task. Each stack is in a different segment and is specified using the segment selector and the offset value in the segment.

When the privilege level 3 program is executing, the segment selector and stack pointer of the privilege level 3 stack are stored in the SS and ESP, respectively, and are saved on the stack of the called procedure when a stack switch occurs.

The initial pointer values for the stacks of privilege levels 0, 1, and 2 are stored in the TSS segment of the currently running task. These pointers in the TSS segment are read-only values. The CPU does not modify them while the task is running. When a higher privilege level program is called, the CPU uses them to build a new stack. When returning from the calling procedure, the corresponding stack does not exist. The next time the procedure is called, a new stack is created again using the initial pointer values in the TSS.

The operating system is responsible for establishing stack and stack segment descriptors for all used privilege levels and setting the initial pointer value in the task's TSS. Each stack must be readable and writable and have enough space to hold some of the following information:

- The contents of the SS, ESP, CS, and EIP registers for the calling process;
- The parameters of the called procedure and the space required for the temporary variables;
- The EFLAGS register and error code, when implicit calls are made to an exception or interrupt handler.

Since one procedure can call other procedures and the operating system can support nesting of multiple interrupts, each stack must have enough space to accommodate multiple frames of the above information.

When a privilege level change is made by a call to a gate, the CPU performs the following steps to switch the stack and begin executing the called procedure on the new privilege level (see Figure 4-25):

1. Select the pointer to the new stack from the TSS using the DPL of the destination code segment (ie the new CPL). The segment selector and stack pointer of the new stack are read from the current TSS. Any error that violates the segment boundary will result in an invalid TSS exception during the process of reading the stack segment selector, stack pointer, or stack segment descriptor;
2. Check if the stack segment descriptor privilege level and type are valid. If invalid, an invalid TSS exception is also generated.
3. Temporarily save the current values of the SS and ESP registers, and load the segment selector and stack pointer of the new stack into the SS and ESP. Then push the temporarily saved SS and ESP content onto the new stack.
4. Copy the specified number of parameters in the call gate descriptor from the calling procedure stack to the new stack. The value of the parameter in the call gate is up to 31. If the number is 0, it means no parameter and no copy is needed.
5. Push the return instruction pointer (ie the current CS and EIP content) onto the new stack. The new

(destination) code segment selector is loaded into the CS, and the offset value (new instruction pointer) in the call gate is loaded into the EIP. Finally, the execution of the called process begins.

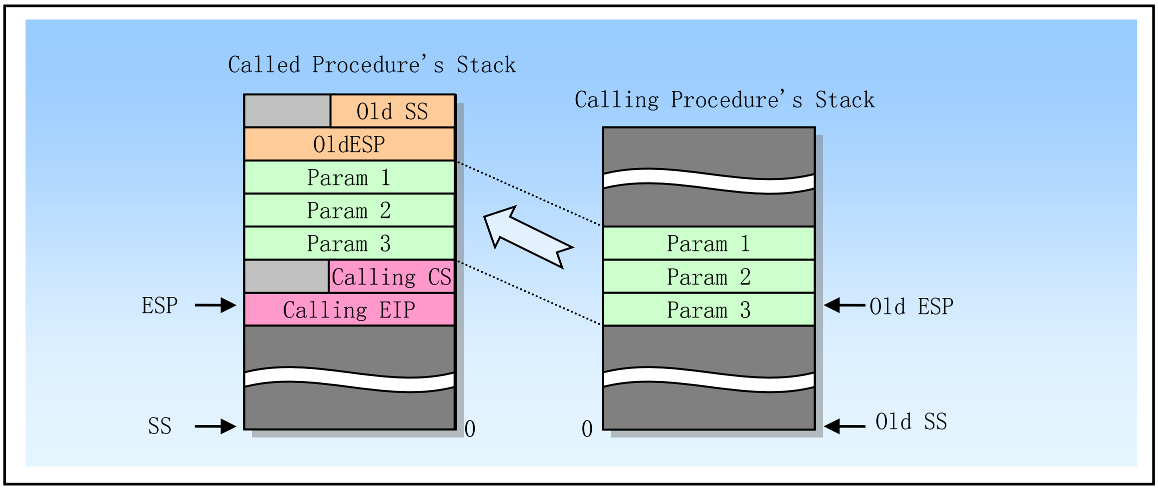


Figure 4-25 Stack switch when calling between different privilege levels

4.5.3.5 Returning from a Called Procedure

The instruction RET is used to perform near return, far return with privilege level, and far return with different privilege levels. This instruction is used to return from the procedure called with the CALL instruction. Near return only transfers program control in the current code segment, so the CPU only performs boundary checking. For far return of the same privilege level, the CPU simultaneously pops the selector of the return code segment and the return instruction pointer from the stack. Since these two pointers are normally pushed onto the stack by CALL instructions, they are valid because of this. However, the CPU still performs a privilege level check to cope with situations where the current process may modify the pointer value or if there is a problem with the stack.

The far return that would result in a privilege level change is only allowed to return to the low privilege level program, ie the code segment DPL returned is numerically greater than the CPL. The CPU uses the RPL field of the selector in the CS register to determine if a low privilege level is required. If the value of the RPL is larger than the CPL, a return operation between privilege levels is performed. When the execution returns far to a calling process, the CPU performs the following steps:

1. Check the RPL field value in the saved CS register to determine if the privilege level needs to be changed on return.
2. Pop up and load the CS and EIP registers using the values on the called procedure stack. The privilege level and type checking of the code segment descriptor and the code segment selector RPL are performed during this process.
3. If the RET instruction contains a parameter count operand and the return operation changes the privilege level, then the parameter count value is added to the ESP register after the CS and EIP values in the pop-up stack to skip the caller stack's parameter. At this point the ESP register points to the pointer SS and ESP of the originally saved caller stack.
4. Load the saved SS and ESP values into the SS and ESP registers to switch back to the caller's stack. At this time, the SS and ESP values of the caller stack are discarded.
5. If the RET instruction contains a parameter number operand, the parameter value is added to the ESP register value to skip (discard) the parameters on the caller stack.

6. Check the contents of the segment registers DS, ES, FS and GS. If there is a segment pointing to a DPL that is smaller than the new CPL (except for the consistent code segment), then the CPU loads the segment register with the NULL selector.

4.5.4 Page-Level Protection

The read/write flags R/W and the user/supervisor flag U/S in the page directory and page table entries provide a subset of the segmentation mechanism protection attributes. The paging mechanism only recognizes two levels of permissions. Privilege levels 0, 1, and 2 are classified as superuser level, while privilege level 3 is classified as a normal user level. Normal user level pages can be marked as read only/executable or readable/writable/executable. Superuser-level pages are always readable/writable/executable for superusers, but are not accessible to ordinary users, as shown in Table 4-6.

For the segmentation mechanism, programs executed at the outermost user level can only access user-level pages, but programs executed at any super user level (0, 1, 2) can not only access the user layer's page, but also access the super. User layer page. Unlike the segmentation mechanism, programs executed at the inner superuser level have readable/writable/executable permissions on any page, including those that are marked as read-only/executable at the user level.

Table 4-6 Normal and super user access restrictions on the page

U/S	R/W	User Access Rights	Supervisor Access Rights
0	0	None	Read/Write/Execute
0	1	None	Read/Write/Execute
1	0	Read/Execute	Read/Write/Execute
1	1	Read/Write/Execute	Read/Write/Execute

Just as the paging mechanism is implemented after the segmentation mechanism in the entire 80X86 address translation mechanism, page-level protection also plays a role in the protection after the the segmentation mechanism. First, all segment level protection is checked and tested. If you pass the check, the page level protection check will be performed. For example, a byte in memory can be accessed by a program on level 3 only when a byte is in a segment accessible by the program on level 3 and is marked as a user-level page. A write to a page can only be performed when both segmentation and paging are allowed to be written. If a segment is a read/write type segment, but the corresponding page corresponding to the address is marked as read-only/executable, then the page cannot be written. If the type of the segment is read-only/executable, the page always has no write permission regardless of the protection attribute given to the corresponding page. It can be seen that the protection mechanism for segmentation and paging is like a serial line in an electronic circuit, in which the switch does not open without a connection.

Similarly, the protection attribute of a page consists of the "serial" or "and operation" of the table entry and the entries in the page table, as shown in Table 4-7. The U/S flag and the R/W flag in the page table entry are applied to a single page of the entry mapping. The U/S and R/W flags in the page directory entry act on all pages mapped to the directory entry. The combined protection attribute of the page directory and the page table is composed of the AND operation of the two attributes, so the protection measures are very strict.

Table 4-7 Page directory and page table entries combined protection of the page

Dir Entry U/S	Page Entry U/S	Combined U/S	Dir Entry R/W	Page Entry R/W	Combined R/W
0	0	0	0	0	0

0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	1	1	1

4.5.4.1 Software issues for modifying page table entries

To avoid having to access the page table that resides in memory each time a memory reference is accessed, the most recently used linear to physical address translation information is stored in the page conversion cache within the processor. The processor first uses the information in the buffer cache before accessing the page table in memory. The processor searches the page directory and page table in memory only when the necessary conversion information is not in the cache. Another term for page conversion caching is called Translation Lookaside Buffer (TLB).

The 80X86 processor does not maintain the dependency of the page conversion cache and the data in the page table, but requires operating system software to ensure they are consistent. That is, the processor does not know when the page table has been modified by the software. Therefore, the operating system must refresh the cache after changing the page table to ensure that the two are consistent. By simply reloading register CR3, we can complete the refresh operation on the cache.

There is a special case where modifying the page table entry does not require refreshing the page conversion cache. That is, when the entry of the non-existing page is modified, even if the P flag is changed from 0 to 1 to mark the entry to be effective for page conversion, there is no need to refresh the cache. Because invalid entries are not stored in the cache, we don't need to refresh the page conversion cache when we call a page from disk into memory to make the page exist.

4.5.5 Combining page and segment protection

When the paging is enabled, the CPU first performs segment-level protection before processing page-level protection. If the CPU detects a protection violation error at any level, it will discard the memory access and generate an exception. If it is an exception generated by the segment mechanism, then no more page exception will be generated.

Page level protection cannot be used to replace or override segment level protection. For example, if a code segment is set to be non-writable, then after the code segment is paged, the page will not be writable even if the page's R/W flag is set to be readable and writable. At this point the segment protection check will block any attempt to write to the page. Page level protection can be used to enhance segment level protection. For example, if a readable and writable data segment is paged, the page level protection mechanism can be used to write protect individual pages.

4.6 Interrupt and Exception Handling

Interrupts and Exceptions are events that occur somewhere in the system, processor, or current executor (or task) that need to be processed by the processor. Often, such events can cause execution control to be forced from the currently running program to a special software function or task called interrupt handler or exception handler. The action taken by the processor in response to an interrupt or exception is called an interrupt/exception service (processing).

Typically, an interrupt occurs at a random time in the execution of the program in response to a signal from the hardware. The system hardware uses interrupts to handle external events, such as requiring service to external devices. Of course, the software can also generate interrupts by executing the INT n instruction.

An exception occurs when the processor executes an instruction and an error condition is detected, such as an error condition divided by zero. The processor can detect various error conditions, including violations of protection mechanisms, page faults, and internal machine errors.

For applications and operating systems, the 80X86 interrupt and exception handling mechanisms transparently handle interrupts and exceptions that occur. When an interrupt is received or an exception is detected, the processor automatically suspends the currently executing program or task and begins running the interrupt or exception handler. When the handler completes, the processor resumes and continues executing the interrupted program or task. The recovery process of the interrupted program does not lose the continuity of program execution unless it is impossible to recover from the exception or the interrupt causes the current program to be terminated. This section describes the processing mechanisms for processor interrupts and exceptions in protected mode.

4.6.1 Sources of Interrupts

The processor receives interrupts from two places:

- External (hardware generated) interrupts;
- Software generated interrupts.

External interrupts are received by two pins (INTR and NMI) on the processor chip. When the pin INTR receives an external interrupt signal, the processor reads the interrupt vector number provided by the external interrupt controller (such as the 8259A) from the system bus. When the pin NMI receives a signal, it generates a non-maskable interrupt. It uses a fixed interrupt vector number of 2. Any external interrupt received through the INTR pin of the processor is referred to as a maskable hardware interrupt, including interrupt vector numbers 0 through 255. The IF flag in the flag register EFLAGS can be used to mask all of these hardware interrupts.

The INT n instruction can be used to generate an interrupt from software by providing the interrupt vector number in the instruction operand. For example, the instruction INT 0x80 will execute the Linux system interrupt call interrupt 0x80. Any of the vectors from 0 to 255 can be used as a parameter in this instruction. However, if a processor predefined NMI vector is used, the processor's response to it will be different from the normal-generated NMI interrupt. If the NMI vector number 2 is used for the INT instruction, the NMI interrupt handler is called, but the processor's NMI processing hardware is not activated at this time.

Note that interrupts generated in software using the INT instruction cannot be masked by the IF flag in EFLAGS register.

4.6.2 Sources of Exceptions

There are also two sources of exceptions received by the processor:

- Processor-detected program-error exceptions;
- Software-generated exceptions.

One or more exceptions are raised if the processor detects a program error during the execution of an application or operating system. The 80X86 processor defines a vector for each exception it detects. Exceptions can be further classified as faults, traps, and aborts, as explained later.

The INTO, INT 3, and BOUND instructions can be used to generate exceptions from software. These instructions check for special exception conditions that are executed at specified points in the instruction stream. For example, the INT 3 instruction will generate a breakpoint exception.

The INT n instruction can be used to simulate a specified exception in software, with one limitation. If the operand n in the INT instruction is one of the 80X86 exception vector numbers, then the processor will generate an interrupt for the vector that will execute the exception handler associated with the vector. But since this is actually an interrupt, the processor does not push an error number onto the stack, even if the vector-related interrupt generated by the hardware usually produces an error code. For exceptions that generate an error code, the exception handler will attempt to pop the error code from the stack while processing. Therefore, if an INT instruction is used to emulate the generation of an exception, the handler will pop off and discard the EIP (just at the missing error code location) onto the stack, causing a return position error.

4.6.3 Exception Classifications

Exceptions can be subdivided into Faults, Traps, and Aborts depending on how the exception is reported and whether the instruction that caused the exception can be re-executed with no loss of program or task continuity.

- A Fault is an exception that can usually be corrected and can continue to run once it has been corrected. When a Fault occurs, the processor will restore the state of the machine to the state it was in prior to the instruction that generated the Fault. At this point, the return address of the exception handler will point to the instruction that generated the Fault, not the one that follows. Therefore, the instruction that generated the fault after returning will be re-executed.
- Trap is an exception that causes a trap to be reported immediately after the execution of the trapping instruction. Trap also enables programs or tasks to execute without loss of continuity. The return address for the trap handler points to the next instruction, so the next instruction is executed after the return.
- Abort is an exception that does not always report the exact location of the instruction that caused the exception, and does not allow the program that caused the exception to resume execution. Abort is used to report serious errors such as hardware errors and inconsistencies or illegal values in the system tables.

4.6.4 Exception and Interrupt Vectors

To help handle exceptions and interrupts, each defined exception and interrupt condition that needs to be specially processed by the processor is given an identification number called a vector. The processor uses the vector as an index number in the Interrupt Descriptor Table (IDT) to locate an exception or interrupt handler entry point location.

The allowed vector number ranges from 0 to 255. 0 to 31 are reserved for the exceptions and interrupts defined by the 80X86 processor, but currently the vector numbers in this range are not defined for each function, and the vector number of the undefined function will be reserved for future use.

Vector numbers ranging from 32 to 255 are used for user-defined interrupts. These interrupts are typically used for external I/O devices so that they can send interrupts to the processor through an external hardware interrupt mechanism.

The vectors assigned to the exceptions and NMI interrupts defined for 80X86 are given in Table 4-8. For each exception, the table gives the exception type and whether an error code is generated and saved on the stack. Each pre-defined exception and NMI interrupt source is also given.

Table 4-8 Exceptions and interruptions in protected mode

Vector No.	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug	Fault/Trap	No	Any code or data reference or the INT 1 instruction.
2	--	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.

5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined)	Fault	No	UD2 instruction or reserved (new for P6)
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes(Zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9	--	Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction (not for CPU after 386)
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	--	(Intel reserved. Do not use.)		No	
16	#MF	Floating-Point Error (Math Fault)	Fault	No	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes(Zero)	Any data reference in memory.
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.
19	#XF	Streaming SIMD Extensions	Fault	No	SSE and SSE2 floating-point instructions. (for PIII cpu)
20-31	--	Intel reserved. Do not use.			
32-255	--	User Defined (Nonreserved) Interrupts	Interrupt		External interrupt or INT n instruction.

4.6.5 Program or Task Restart

In order for a program or task to resume execution after an exception or interrupt has been processed, all exceptions except Abort can report the exact instruction location, and all interrupt guarantees occur on the instruction boundary.

For fault class exceptions, the return instruction pointer saved when the processor generates an exception points to the faulting instruction. Therefore, when a program or task is restarted after the fault handler returns, the original faulting instruction is re-executed. Re-execution of the instruction that caused the fault is usually used to handle the case where the access instruction operand is blocked. The most common example of a Fault is a page fault exception. This exception occurs when a program references an operand that is not on a page in memory. When a page fault exception occurs, the exception handler can load the page into memory and resume program execution by re-execution of the faulting instruction. To ensure that re-execution is transparent to the current execution program, the processor saves the necessary register and stack pointer information so that it can return to the state it was in prior to executing the faulting instruction.

For a Trap class exception, the return pointer saved when the processor generates an exception points to the next instruction that caused the trap operation. If a Trap is detected during the execution of an instruction that performs a control transfer, the return instruction pointer reflects the transition of the control. For example, if a Trap exception is detected while executing a JMP instruction, the return instruction pointer points to the target location of the JMP instruction, not to the next instruction of the JMP instruction.

The abort class exceptions do not support reliable restarting of programs or tasks. A handler that aborts an exception is typically used to collect diagnostic information about the state of the processor when the exception occurred, and to close the program and system as appropriately as possible.

Interrupts strictly support restarting of interrupted programs without losing any continuity. The return instruction pointer saved for an interrupt points to the next instruction boundary that will be executed when the processor acquires the interrupt. If the instruction just executed has a repeat prefix, the interrupt will occur when the current iteration ends and the register has been set for the next iteration.

4.6.6 Enabling and Disabling Interrupts

The Interrupt Enable Flag (IF) of the Flag Register EFLAGS can disable the servicing of maskable hardware interrupts received on the INTR pin of the processor. When IF = 0, the processor disables the interrupt sent to the INTR pin; when IF = 1, the interrupt signal sent to the INTR pin is processed by the processor.

The IF flag does not affect the non-masked interrupts sent to the NMI pin, nor does it affect the exception generated by the processor. As with the other flags in EFLAGS, the processor clears the IF flag (IF=0) in response to a hardware reset operation.

The IF flag can be set or cleared using the instructions STI and CLI. These two instructions can only be executed when the program's CPL <= IOPL, otherwise a general protective exception will be raised. The IF flag is also affected by the following operations:

- The PUSHF instruction can store the EFLAGS contents on the stack, where they can be examined and modified. The POPF instruction can be used to put the contents of the modified flags back into the EFLAGS register.
- The task switch, POPF, and IRET instructions load the EFLAGS register. Therefore, they can be used to modify the setting of the IF flag.
- When an interrupt is processed through the interrupt gate, the IF flag is automatically cleared (reset), which disables the maskable hardware interrupt. However, if an interrupt is handled through the trap gate, the IF flag will not be reset.

4.6.7 Priority of exceptions and interrupts

If there are multiple exceptions or interrupt pending processing at the boundary of an instruction, the processor processes them in the specified order. Table 4-9 shows the priority of the exception and interrupt source classes. The processor first processes exceptions or interrupts in the highest priority class. Low priority exceptions are discarded, while low priority interrupts are held waiting. When the interrupt handler returns to a program or task that generated an exception and/or interrupt, the discarded exception will reoccur.

Table 4-9 Priority of exceptions and interrupts

Priority	Description
1(Highest)	Hardware reset: RESET
2	Task switching trap: T flag is set in TSS
3	External hardware intervention
4	Previous instruction trap: breakpoint, debug trap exception
5	External interrupt: NMI interrupt, maskable hardware interrupt

6	Code breakpoint error
7	Take an instruction error: violation of code segment limit, code page error
8	The next instruction decode error: instruction length >15 bytes, invalid opcode, coprocessor does not exist
9(Lowest)	Execution instruction error: overflow, boundary check, invalid TSS, segment not present, stack error, general protection, data page, alignment check, floating point exception

4.6.8 Interrupt Descriptor Table (IDT)

The Interrupt Descriptor Table (IDT) associates each exception or interrupt vector with a gate descriptor of their process or task, which is used to handle related exceptions and interrupts. Similar to the GDT and LDT tables, IDT is also an array of 8-byte long descriptors. Unlike GDT, the first item in the table can contain descriptors. To form an index into the IDT table, the processor puts the vector number of the exception or interrupt *8. Since there are at most 256 interrupt or exception vectors, the IDT does not need to contain more than 256 descriptors. IDTs can contain fewer than 256 descriptors because descriptors are only needed for exceptions or interruptions that may occur. However, all empty descriptor entries in the IDT should have their presence bit (flag) set to zero.

The IDT table can reside anywhere in the linear address space, and the processor uses the IDTR register to locate the location of the IDT table. This register contains the 32-bit base address of the IDT table and the 16-bit length (length limit) value, as shown in Figure 4-26. The IDT table base address should be aligned on an 8-byte boundary to improve processor access efficiency. The limit length value is the length of the IDT table in bytes.

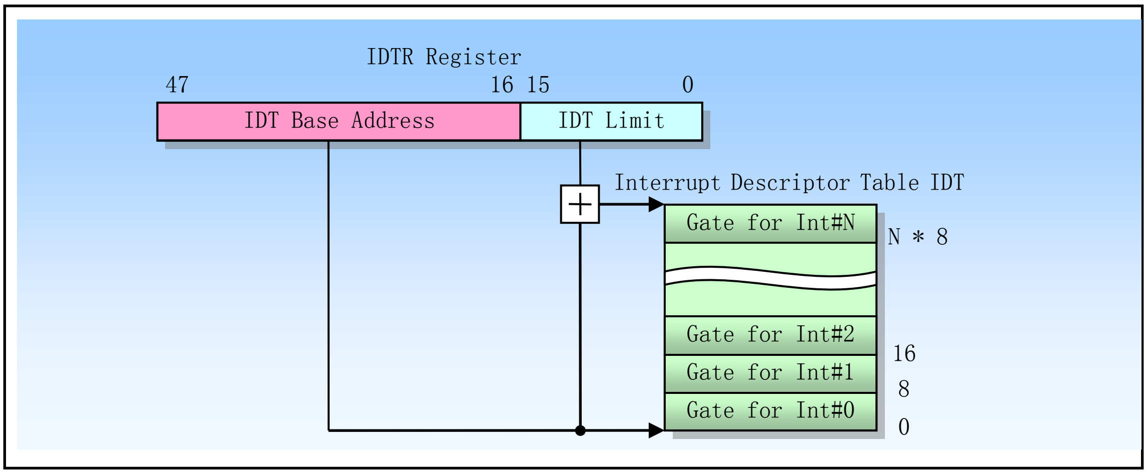


Figure 4-26 Interrupt Descriptor Table IDT and Register IDTR

The LIDT and SIDT instructions are used to load and store the contents of the IDTR register, respectively. The LIDT instruction loads the limit and the base address operand in memory into the IDTR register. This instruction can only be executed by code whose current privilege level CPL is 0, and is usually used in the operating system initialization code when IDT is created. The SIDT instruction is used to copy the base address and the limit content in the IDTR into the memory. This instruction can be executed at any privilege level. If the descriptor referenced by the interrupt or exception vector exceeds the bounds of the IDT, the processor generates a general protection exception.

4.6.9 IDT Descriptors

Three kinds of gate descriptors can be stored in the IDT table:

- Interrupt gate descriptor;
- Trap gate descriptor;
- Task gate descriptor.

The format of these three gate descriptors is shown in Figure 4-27. The interrupt gate and trap gate contain a long pointer (ie, segment selector and offset value) that the processor uses to transfer program execution rights to exceptions or interrupts in the code segment. The main difference between the two segments is that the processor operates on the IF flag of the EFLAGS register. The format of the task gate descriptor in the IDT is the same as the format of the task gate in the GDT and LDT. The task gate descriptor contains a selector for the task TSS segment that is used to handle exceptions and/or interrupts.

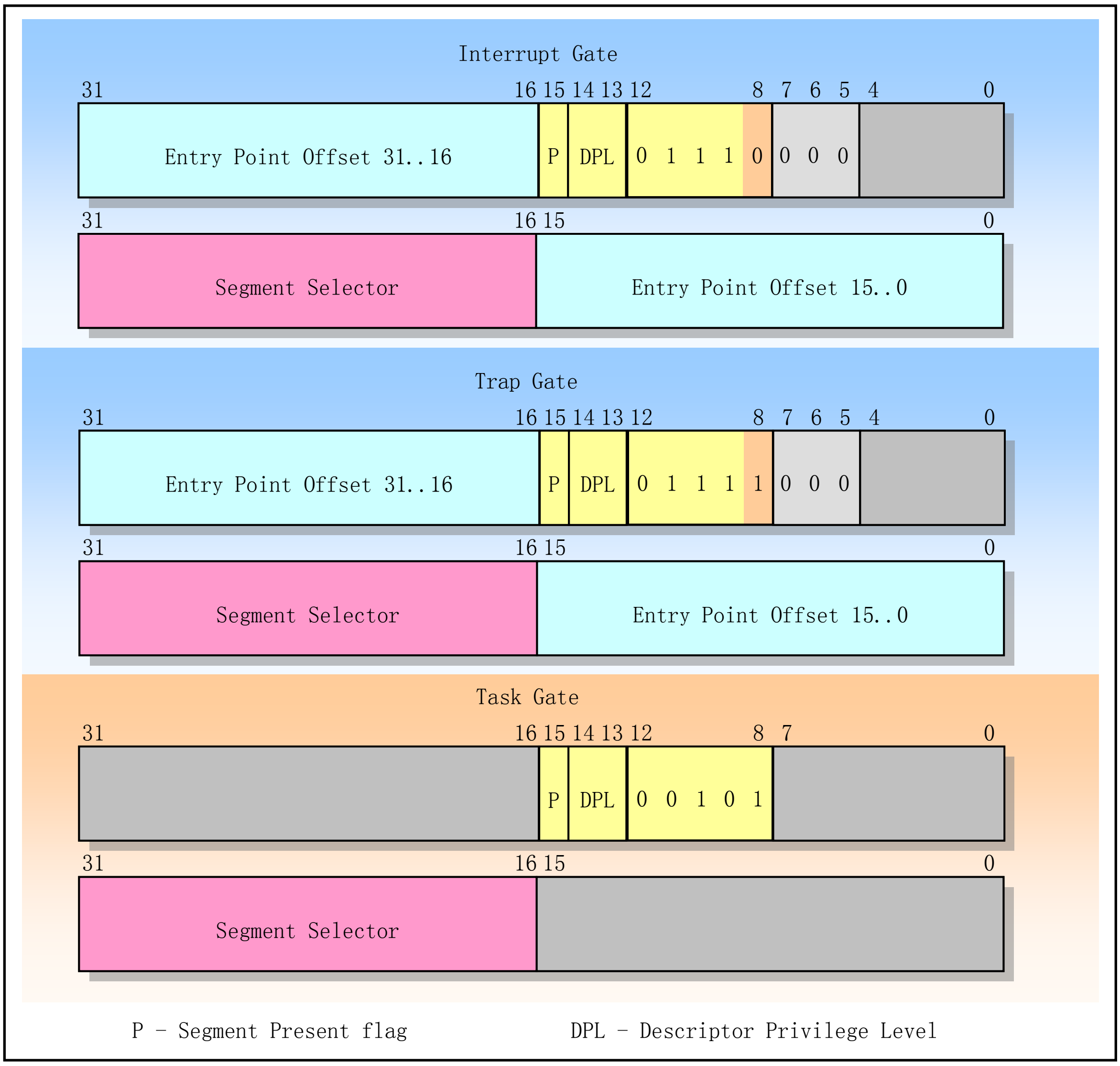


Figure 4-27 Interrupt gate, trap gate, and task gate descriptor format

4.6.10 Exception and Interrupt Handling

The processor's method of invoking exceptions and interrupt handlers is similar to calling a procedure and task using the CALL instruction. When responding to an exception or interrupt, processor uses the vector of the exception or interrupt as an index in the IDT table. If the index value points to an interrupt gate or trap gate, the processor invokes an exception or interrupt handler using a method similar to the CALL instruction operation call gate. If the index value points to the task gate, the processor performs a task switch using a method similar to the CALL instruction operation task gate, and performs an exception or interrupted processing task.

An exception or interrupt gate references an exception or interrupt handler that runs in the context of the current task, as shown in Figure 4-28. The segment selector in the gate points to the executable code segment descriptor in the GDT or the current LDT. The offset field in the gate descriptor points to the beginning of the exception or interrupt handling process.

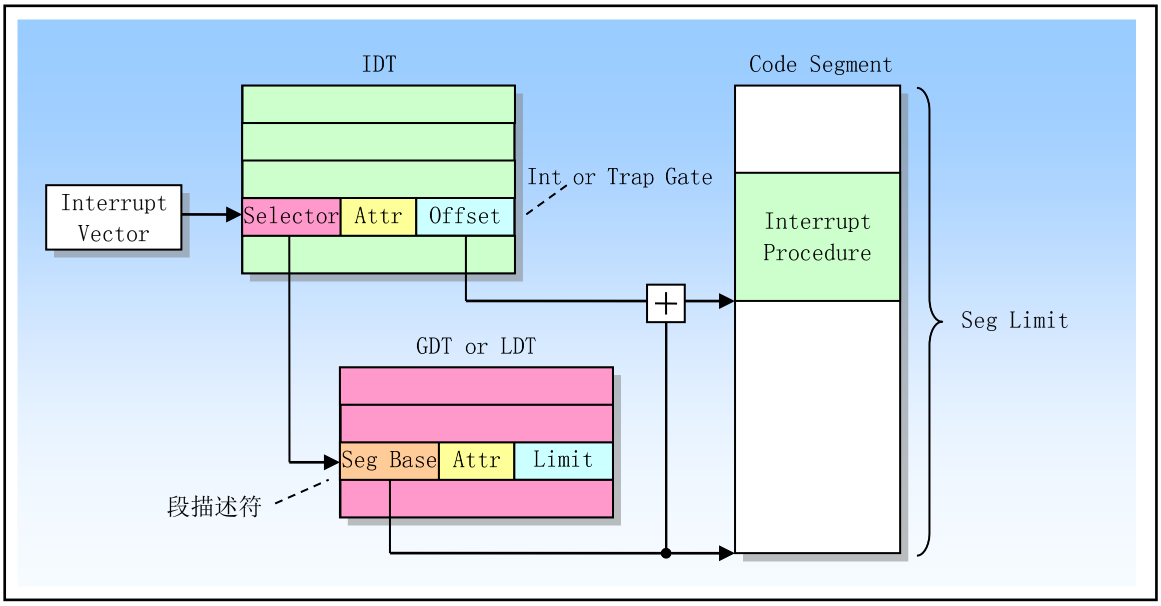


Figure 4-28 Interrupt Procedure Call

When the processor executes an exception or interrupt handler call, the following actions are taken:

- If the handler procedure will be executed at a high privilege level (such as level 0), then a stack switch operation will occur. The stack switching process is as follows:

  The processor gets the segment selector and stack pointer of the stack used by the interrupt or exception handler from the TSS segment of the currently executing task (eg tss.ss0, tss.esp0). The processor then pushes the stack selector and stack pointer of the interrupted program (or task) onto the new stack, as shown in Figure 4-29. The processor then pushes the current values of the EFLAGS, CS, and EIP registers onto the new stack. If the exception generates an error code, the error code will also be pushed to the new stack.

- If the handler procedure will run on the same privilege level as the interrupted task, then:

  The processor saves the current values of the EFLAGS, CS, and EIP registers on the current stack. If the exception generates an error code, the error code will also be pushed to the new stack.

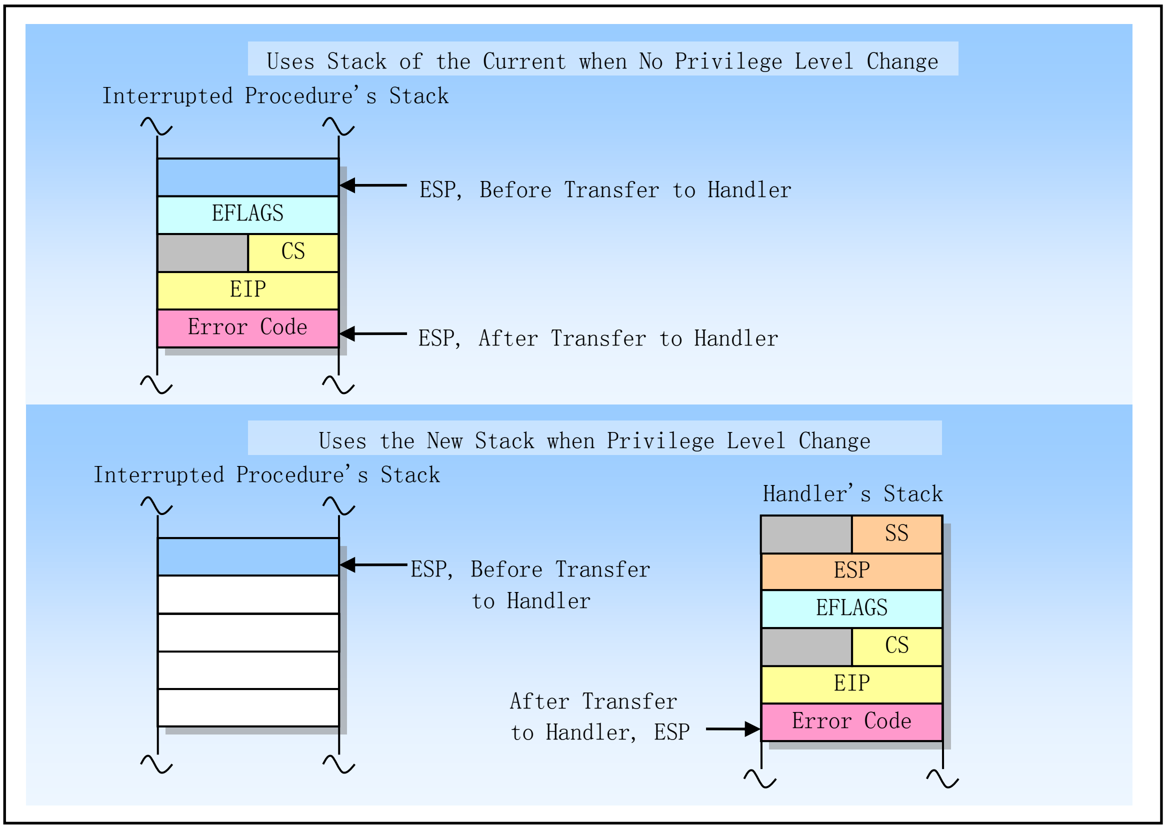

Uses Stack of the Current when No Privilege Level Change
Interrupted Procedure's Stack
EFLAGS
CS
EIP
Error Code
ESP, Before Transfer to Handler
ESP, After Transfer to Handler
Uses the New Stack when Privilege Level Change
Interrupted Procedure's Stack
ESP, Before Transfer to Handler
Handler's Stack
SS
ESP
EFLAGS
CS
EIP
Error Code
After Transfer to Handler, ESP

Figure 4-29 Stack usage method when transferring to interrupt processing

To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. The IRET instruction is similar to the RET instruction except that it restores the saved flags into the EFLAGS register. The IOPL field of the EFLAGS register is restored only if the CPL is 0. The IF flag is changed only if the CPL <= IOPL. If a stack switch occurred when calling the handler procedure, the IRET instruction switches back to the interrupted procedure's stack on the return.

4.6.10.1 Protection of exceptions and interrupt handler procedures

The privilege level protection mechanism for exception and interrupt handler procedures is similar to calling a normal procedure through a call gate. The processor does not allow control to be transferred to interrupt handler procedure that is lower than the CPL privileged code segment, otherwise a general protection exception will be generated. In addition, the protection mechanism for interrupts and exceptions is different from the general call gate procedure in the following respects:

- Because the interrupt and exception vectors do not have an RPL, the RPL is not checked when the exception and interrupt handler procedures are implicitly called.
- The processor checks the DPL in the interrupt or trap gate only when an exception or interrupt is generated using the INT n, INT 3, or INTO instruction. At this time, the CPL must be less than or equal to the DPL of the gate. This restriction prevents applications running at privilege level 3 from using software interrupts to access important exception handling procedures, such as page fault handling, assuming that these processes have been placed in a higher privilege level code segment. For hardware-generated interrupts and processor-detected exceptions, the processor ignores the DPL in the interrupt gate and trap gate.

Because exceptions and interrupts do not usually occur on a regular basis, these rules about privilege levels effectively enhance the privilege level limits that exceptions and interrupt handlers can run. We can use one of the following techniques to avoid violating privilege level protection:

- Exception or interrupt handlers can be stored in a consistent code segment. This technique can be used for handlers that only need to access data available on the stack (for example, divide error exceptions). If the handler needs data in the data segment, then privilege level 3 must be able to access this data segment. But there is no protection at all.
- The handler can be placed in a nonconforming code segment with privilege level 0. This handler can always be performed regardless of the current privilege level CPL of the interrupted program or task.

4.6.10.2 Flag usage by Exception or Interrupt Handler Procedure

When an exception or interrupt handler is accessed through an interrupt gate or trap gate, the processor clears the TF flag in EFLAGS after saving the contents of the EFLAGS register on the stack. Clearing the TF flag prevents the instruction trace from affecting the interrupt response. The subsequent IRET instruction will restore the original TF flag of EFLAGS with the contents of the stack.

The only difference between an interrupt gate and a trap gate is the way the processor operates the IF flag of the EFLAGS register. When an exception or interrupt handler is accessed through the interrupt gate, the processor resets the IF flag to prevent other interrupts from interfering with the current interrupt handler. Subsequent IRET instructions will restore the IF flag of the EFLAGS register with the contents stored on the stack. Accessing the handler procedure through the trap gate does not affect the IF flag.

4.6.11 Interrupt handler Tasks

Task switching occurs when an exception or interrupt handler is accessed through the task gate in the IDT. Using separate tasks to handle exceptions or interruptions has the following benefits:

- The complete context of the interrupted program or task is automatically saved;
- A new TSS permits the handler to use a new privilege level 0 stack when handling the exception or interrupt. If an exception or interrupt occurs when the current privilege level 0 stack is corrupted, accessing the handler through a task gate can prevent a system crash by providing the handler with a new privilege level 0 stack.
- The handler can be further isolated from other tasks by giving it a separate address space. This is done by giving it a separate LDT.

The disadvantage of using a separate task to handle exceptions or interrupts is that the amount of machine state must be saved during task switching, making it slower than using interrupt gates, resulting in increased interrupt latency.

The task gate in the IDT will reference the TSS descriptor in the GDT, as shown in Figure 4-30. The process of switching to a handler task is the same as the normal task switching process. The back link to the interrupted task will be saved in the previous task link field of the handler task TSS. If an exception generates an error code, the error code is copied to the new task stack.

When an exception or interrupt handler task is used in an operating system, there are actually two mechanisms for dispatching tasks: operating system software scheduling and hardware scheduling of the processor interrupt mechanism. The software scheduler needs to accommodate interrupt tasks that may be dispatched when interrupts are enabled.

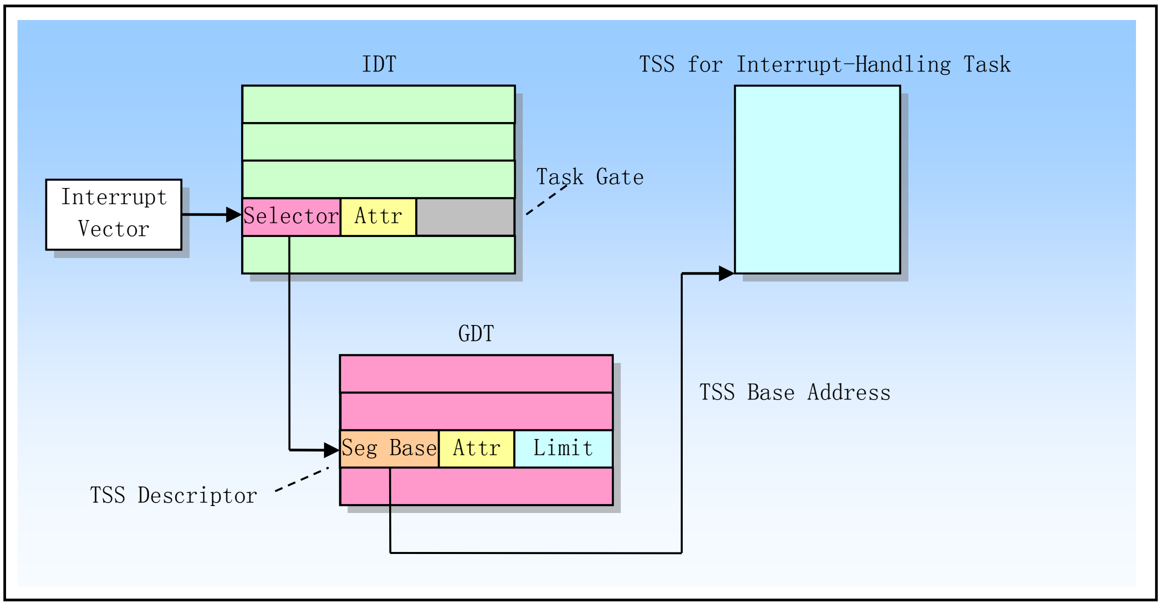


Figure 4-30 Interrupt Task Switch

4.6.12 **Error Code**

When an exception condition is associated with a particular segment, the processor pushes an error code onto the stack of exception handler. The format of the error code is shown in Figure 4-31. The error code is much like a segment selector, but the lowest 3 bits are not TI and RPL fields, but the following 3 flags:

- Bit 0 is the external event (EXT) flag. When set, indicates that an event external to the program caused an exception, such as a hardware interrupt.
- Bit 1 is a descriptor location (IDT) flag. When this bit is set, the index portion indicating the error code points to a gate descriptor in the IDT. When this bit is reset, it indicates that the index refers to a descriptor in the GDT or LDT.
- Bit 2 is the GDT/LDT table select flag TI. Only useful when IDT (bit 1) is 0. When the TI=1, the index portion indicating the error code points to a descriptor in the LDT. When TI=0, it indicates that the index part in the error code points to a descriptor in the GDT table.

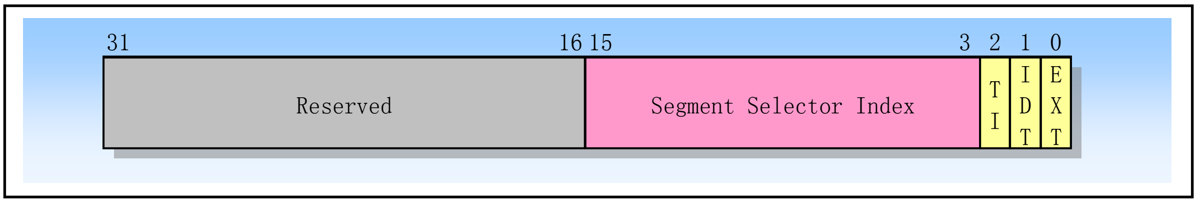


Figure 4-31 Error Code format

The Segment Selector Index field provides the index into the IDT, GDT, or current LDT to the segment or gate selector being referenced by the error code. In some cases the error code is null (ie, the lower 16 bits are all clear). A null error code indicates that the error was not caused by referencing a particular segment, or that an null segment descriptor was referenced in the operation.

The error code format of the Page-fault exception is different from the above, as shown in Figure 4-32. Only the lowest 3 bits are useful, and their names are the same as the last three bits in the page table entry (U/S, W/R, P). The meanings and effects are:

- Bit 0 (P), the exception is caused by a page not being present or violating access privileges. P=0, indicating that the page does not exist; P=1 means that the page-level protection authority is violated.
- Bit 1 (W/R), the exception is due to a memory read or write operation. W/R=0, indicating that it is caused by a read operation; W/R=1, indicating that it is caused by a write operation.
- Bit 2 (U/S), the code level at which the CPU executes when an exception occurs. U/S=0, indicating that the CPU is executing the super user code; U/S=1, indicating that the CPU is executing the general user code.

In addition, the processor also stores the linear address used to cause the page fault exception to be stored in CR2. The page fault exception handler can use this address to locate the relevant page directory and page table entry.

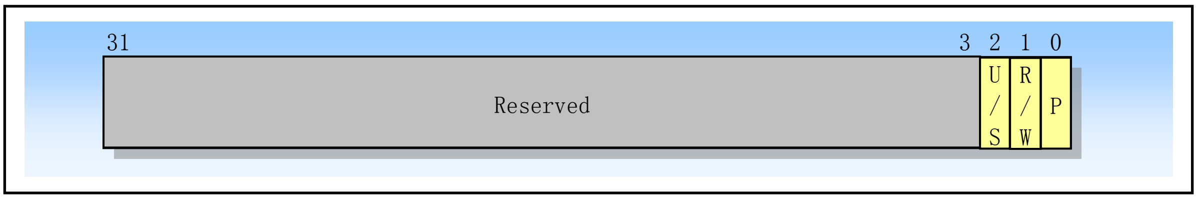


Figure 4-32 Page fault error code format

Note that the error code is not automatically popped off the stack by the IRET instruction, so the interrupt handler must clear the error code on the stack before returning. In addition, although some exceptions generated by the processor will generate an error code and will be automatically saved to the stack of the handler procedure, an external hardware interrupt or an exception generated by the program executing the INT n instruction will not push the error code onto the stack.

4.7 Task Management

A task is a unit of work that a processor can allocate to schedule, execute, and suspend. It can be used to execute programs, tasks or processes, operating system services, interrupt or exception handling procedures, and kernel code. A task is a running program or a program waiting to be run.

The 80X86 provides multi-tasking hardware support for saving the status of tasks, dispatching tasks, and switching from one task to another. When working in protected mode, all of the processor's operations are in the task. Even a simple system must define at least one task. More complex systems can use the task management capabilities of the processor to support multitasking applications.

We can perform a task by interrupt, exception, jump or call with a specified entry in a descriptor table. There are two kinds of task-related descriptors in the descriptor table: task state segment descriptors and task gates. When execution rights are passed to any of these kinds of descriptors, task switching occurs. Task switching is much like a procedure call, but task switching saves more processor state information. Task switching will completely transfer control to a new execution environment, the execution environment of the new task. This transfer operation requires saving the current contents of almost all registers in the processor, including the flag register EFLAGS and all segment registers. However, the task cannot be reentrant. Task switching does not push any information onto the stack. The processor's state information is stored in a data structure called a task state segment (TSS) in memory.

4.7.1 Task Structure and State

A task consists of two parts: the task execution space and the task state segment (TSS). The task execution space includes a code segment, a stack segment, and one or more data segments, as shown in Figure 4-33. If an operating system uses the processor's privilege level protection mechanism, then the task execution space needs to provide a separate stack space for each privilege level. The TSS specifies the segments that make up the task execution space and provides storage for the task state information. In a multitasking environment, TSS also provides a way to handle links between tasks.

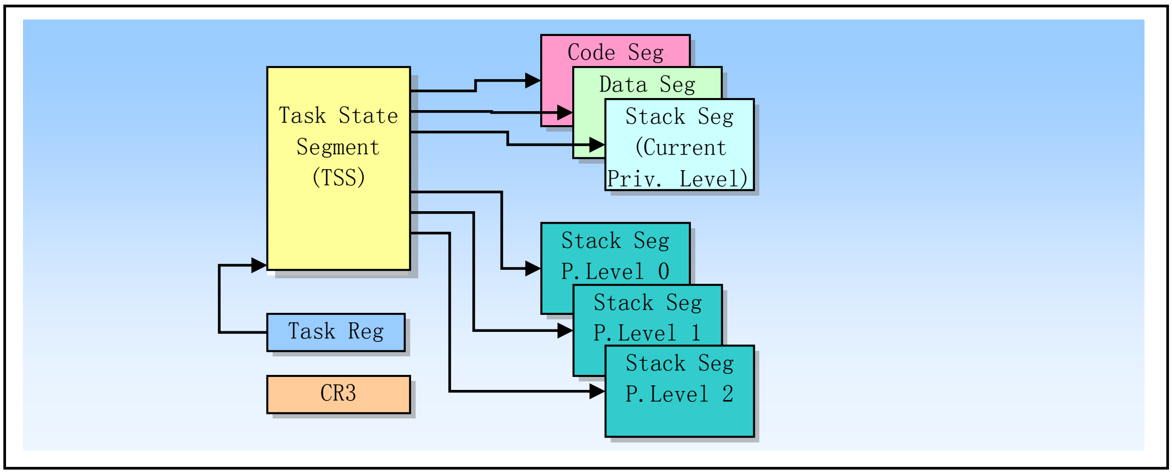


Figure 4-33 Structure of a Task

A task is specified using a segment selector that points to its TSS. When a task is loaded into the processor for execution, then the segment selector, base address, segment length, and TSS segment descriptor attributes of the task are loaded into the task register (TR). If the paging mechanism is used, the page directory base address used by the task is loaded into control register CR3. The state of the currently executing task consists of the following:

- The task’s current execution space, defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS);
- The state of the general purpose registers;
- The state of the EFLAGS, EIP, control register CR3, task register and LDTR register;
- The I/O map base address and I/O map (contained in the TSS);
- Stack pointers to the privilege 0, 1, and 2 stacks (contained in the TSS);
- Link to previously executed task (contained in the TSS).

Before distributing a task, all of these items are included in the task's TSS, except for the status of the task registers. In addition, the complete contents of the LDTR register are not included in the TSS and only contain the segment selector of the LDT.

4.7.2 Execution of Tasks

The software or processor can dispatch a task for execution in one of the following methods:

- A explicit call to a task with the CALL instruction;
- A explicit jump to a task with the JMP instruction (the way the Linux kernel uses);
- An implicit call (by the processor) to an interrupt-handler task;

- An implicit call to an exception-handler task;
- A return (initiated with an IRET instruction) when the NT flag in the EFLAGS register is set.

All of these methods of scheduling task execution use a selector that points to the task gate or task TSS segment to determine a task. When dispatching a task using a CALL or JMP instruction, the selector in the instruction may either directly select the task's TSS or the task gate that holds the selector for the TSS. When dispatching a task to handle an interrupt or exception, then the interrupt or exception entry in the IDT must contain a task gate that holds the selector for the TSS of the interrupt or exception handler task.

When a task is dispatched for execution, a task switching operation occurs automatically between the currently running task and the scheduled task. During a task switch, the execution environment (called the state or context of the task) that is currently running the task is saved to its TSS and the execution of the task is suspended. The context of the newly scheduled task is then loaded into the processor and a new task is executed starting from the instruction pointed to by the loaded EIP.

If the currently executing task (the caller) invokes the scheduled new task (the callee), then the caller's TSS segment selector is stored in the callee TSS, providing a link back to the caller. For all 80X86 processors, tasks are not recursively called, ie tasks cannot be called or jumped to themselves.

Interrupts or exceptions can be handled by switching to a handler task. In this case, the processor not only can perform a task switch to handle interrupts or exception, but also automatically switch back to the interrupted task when the interrupt or exception handler task returns. This mechanism can handle interrupts that occur during interrupt tasks.

As part of the task switching operation, the processor also switches to another LDT, allowing each task to have a different logical-to-physical address mapping for LDT-based segments. At the same time, the page directory register CR3 is also reloaded at the time of switching, so each task can have its own set of page tables. These protections can be used to isolate individual tasks and prevent them from interfering with each other.

It is optional to use the processor's task management capabilities to handle multitasking applications. We can also use software to implement multitasking so that each software-defined task is executed in the context of a single 80X86 architecture task.

4.7.3 Task Management Data Structures

The processor defines the following registers and data structures that support multitasking:

- Task-state segment (TSS);
- TSS descriptor;
- Task registor (TR);
- Task-gate descriptor;
- NT flag in the EFLAGS register.

Using these data structures, the processor can switch from one task to another while preserving the context of the original task to allow the task to re-execute. When operating in protected mode, a TSS and TSS descriptor must be created for at least one task, and the segment selector for the TSS must be loaded into the task register (using the LTR instruction).

4.7.3.1 Task-State Segment (TSS)

The processor state information for restoring a task execution is saved in a segment called a task state segment TSS (Task state segment). Figure 4-34 shows the format of the TSS used by the 32-bit CPU. The fields in the TSS segment can be divided into two broad categories: dynamic fields and static fields.

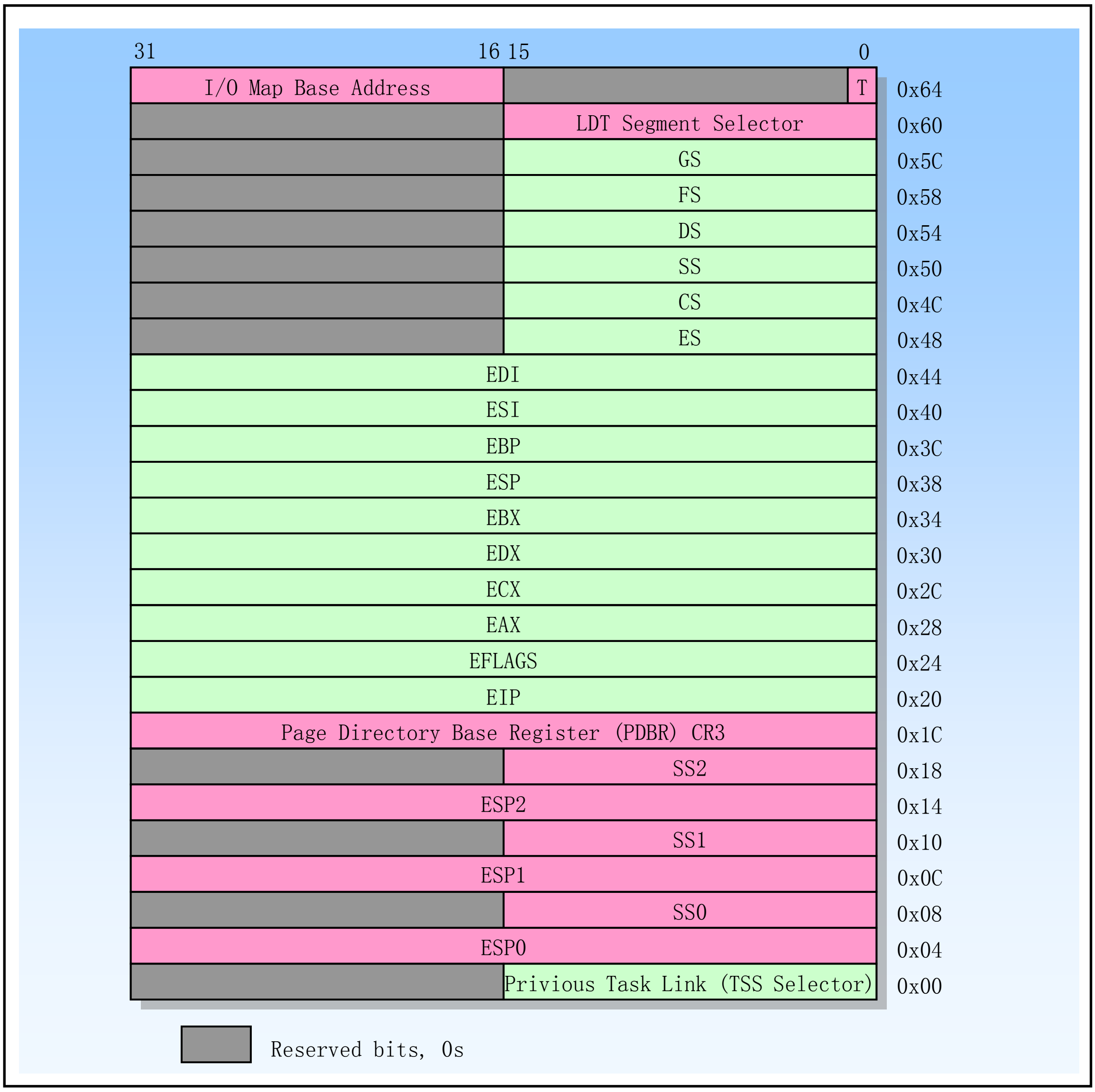


Figure 4-34 32-bit task status segment TSS format

1. Dynamic fields. When the task is switched and suspended, the processor updates the contents of the dynamic field. These fields include:
   - General purpose register field. Used to save the contents of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers.
   - Segment selector field. Used to save the contents of the ES, CS, SS, DS, FS, and GS segment registers.
   - Flag register EFLAGS field. Save EFLAGS before switching.
   - Instruction pointer EIP field. Save the contents of the EIP register before switching.
   - The previous task link field. Contains the previous task TSS segment selector (updated during a task switch of a call, interrupt, or exception fire). This field (also commonly referred to as the Back link field) allows the task to switch to the previous task using the IRET instruction.
2. Static fields. The processor reads the contents of static fields, but usually does not change them. These field contents are set when the task is created. These fields are:
   - LDT segment selector field. Containins the segment selector for the task's LDT.
   - CR3 Control Register Field. Contains the physical base address of the page directory used by the task.

The control register CR3 is also commonly referred to as a page directory base register (PDBR).

- Stack pointer field for privilege levels 0, 1, and 2. These stack pointers consist of stack segment selectors (SS0, SS1, and SS2) and offset pointers in the stack (ESP0, ESP1, and ESP2). Note that the values of these fields are constant for a given task. Therefore, if a stack switch occurs in a task, the contents of the registers SS and ESP will change.
- Debug Trap T flag field. This field is located at byte 0x64 bit 0. When this bit is set, then a debug exception will be generated when the processor switches to the task.
- I/O bitmap base address field. This field contains the 16-bit offset value from the beginning of the TSS segment to the I/O permission bitmap. When present, these maps are stored at the higher address of the TSS. The I/O mapping base address points to the beginning of the I/O permission bitmap and the end of the interrupt redirection bitmap.

If a paging mechanism is used, the memory page boundaries should be avoided in the TSS segment of the processor operation (in the first 104 bytes) during task switching. If the TSS part contains a memory page boundary, then the pages on both sides of the boundary must exist simultaneously and continuously in physical memory. In addition, if the paging mechanism is used, the pages related to the original task TSS and the new task TSS, and the corresponding descriptor table entries should be readable and writable.

4.7.3.2 TSS Descriptor

Like other segments, the task status segment TSS is also defined using a segment descriptor. Figure 4-35 shows the format of the TSS descriptor. TSS descriptors can only be stored in the GDT.

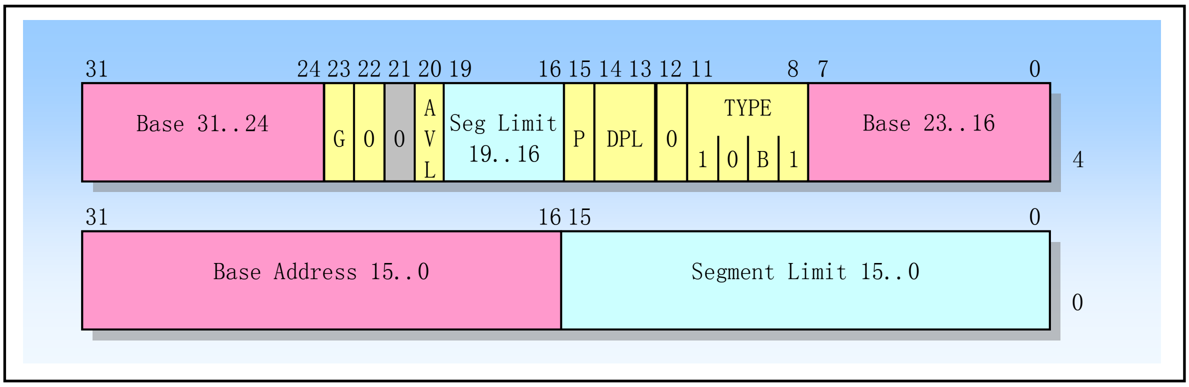


Figure 4-35 TSS segment descriptor format

The busy flag (B) in the type field TYPE is used to indicate whether the task is busy. A busy task is a task that is currently executing or a task that is waiting to be executed (suspended). A type field with a value of 0b1001 indicates that the task is inactive; a type field with a value of 0b1011 indicates that the task is busy. The processor uses the busy flag to detect an attempt to call a task whose execution has been interrupted. To insure that there is only one busy flag is associated with a task, each TSS should have only one TSS descriptor that points to it.

The base address, limit, the descriptor privilege level DPL, the granularity G, and the present flags have the same functions as the corresponding fields in the data segment descriptor. When G=0, the limit field must have a value equal to or greater than 103 (0x67), that is, the minimum length of the TSS segment must not be less than 104 bytes. If the TSS segment also contains an I/O permission bitmap, the TSS segment length needs to be larger. In addition, if the operating system still wants to store some other information in the TSS segment, the TSS segment needs to be larger.

With a call or jump instruction, any program that can access the TSS descriptor can cause a task switch.

Programs that can access the TSS descriptor must have a CPL that is numerically less than or equal to the DPL of the TSS Descriptor. In most systems, the DPL field of the TSS descriptor should be set to less than 3. In this way, only privileged software can perform task switching operations. However, in a multitasking application, the DPL for some TSSs can be set to 3 so that task switching operations can also be performed at the user privilege level.

The ability to access a TSS segment descriptor does not give the program the ability to read and write the descriptor. If you want to read or modify a TSS segment descriptor, you can use a data segment descriptor (ie, an alias descriptor) that maps to the same location in memory. Loading the TSS descriptor into any segment register will result in an exception. Attempting to access the TSS segment using the selector set by the TI flag (ie, the selector in the current LDT) will also cause an exception.

4.7.3.3 Task Register

The task register TR (Task Register) stores a 16-bit segment selector and the entire descriptor (invisible portion) of the current task TSS segment. This information is copied from the TSS descriptor of the current task in the GDT. The processor uses the invisible portion of the task register TR to buffer the contents of the TSS segment descriptor.

The instructions LTR and STR are used to load and save the visible portion of the task register, ie the selector of the TSS segment, respectively. The LTR instruction can only be executed by a privileged level 0 program. The LTR instruction is typically used to load the initial value of the TR register during system initialization (eg, the TSS segment selector for task 0), and then during system operation, the contents of the TR are automatically changed upon task switching.

4.7.3.4 Task-Gate Descriptor

The task gate descriptor provides an indirect, protected reference to a task, as shown in Figure 4-27. The task gate descriptor can be stored in a GDT, LDT, or IDT table.

The TSS Segment Selector field in the Task Gate Descriptor points to a TSS Segment Descriptor in the GDT. The RPL field in this TSS segment selector is not used. The DPL in the task gate descriptor is used to control access to the TSS segment at the time of task switching. When a program makes a call or jump to a task through a task gate, the CPL and the RPL field of the gate selector pointing to the task gate must be less than or equal to the DPL of the task-gate descriptor. Note that when using the task gate, the DPL of the target TSS segment descriptor is ignored.

The program can access a task through a task gate descriptor or a TSS segment descriptor. Figure 4-36 shows how the task gates in the LDT, GDT, and IDT tables all point to the same task.

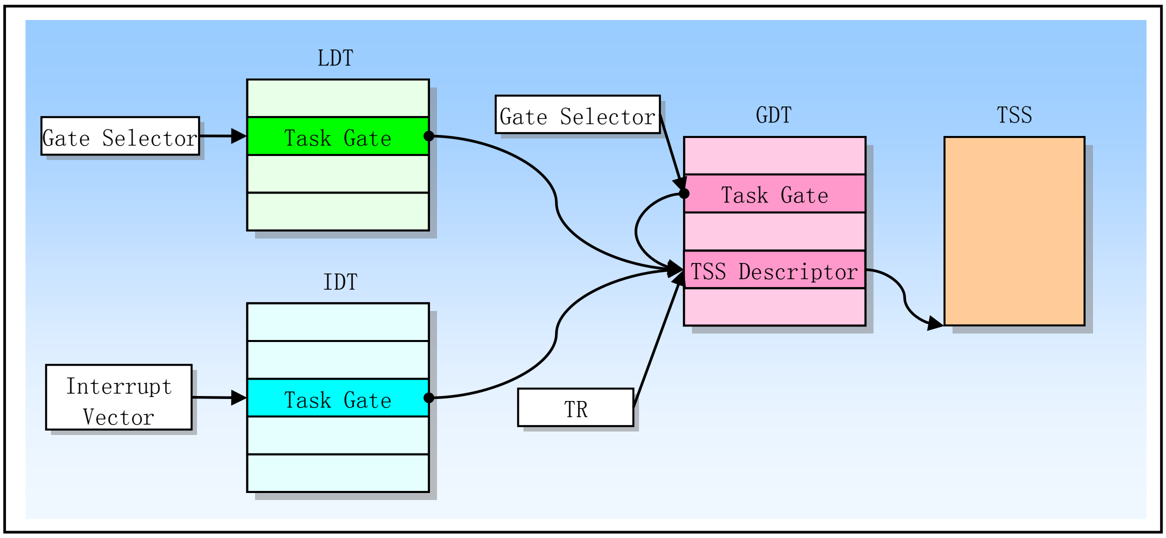


Figure 4-36 Task gates that reference the same task

4.7.4 Task Switching

The processor transfers execution to another task in any of four cases:

1. The current program or task executes a JMP or CALL instruction to a TSS descriptor in the GDT.
2. The current program or task executes a JMP or CALL instruction to a task gate descriptor in the GDT or the current LDT;
3. The interrupt or exception vector points to a task gate descriptor in the IDT table;
4. The current task executes an IRET instruction when the NT flag in the EFLAGS register is set.

The JMP, CALL, and IRET instructions, as well as interrupts and exceptions, are all generalized mechanisms for redirecting a program. The referencing of a TSS descriptor or a task gate (when calling or jumping to a task) or the state of the NT flag (when executing an IRET instruction) determines whether a task switch occurs.

For task switching, JMP or CALL instructions can transfer control to the TSS descriptor or task gate. Using the same two methods will cause the processor to transfer control to the specified task, as shown in Figure 4-37.

When an interrupt or an exception vector index is a task gate in the IDT, an interrupt or exception will cause a task switch. But if the vector index is an interrupt or trap gate in the IDT, it will not cause a task switch.

The interrupt service handler procedure always returns execution rights to the interrupted program or procedure, and the interrupted program may be in another task. If the NT flag is in the reset state, a general return operation is performed. If the NT flag is set, the return operation will result in a task switch. The new task to switch to is specified by the TSS selector (previous task link field) in the interrupt service procedure TSS.

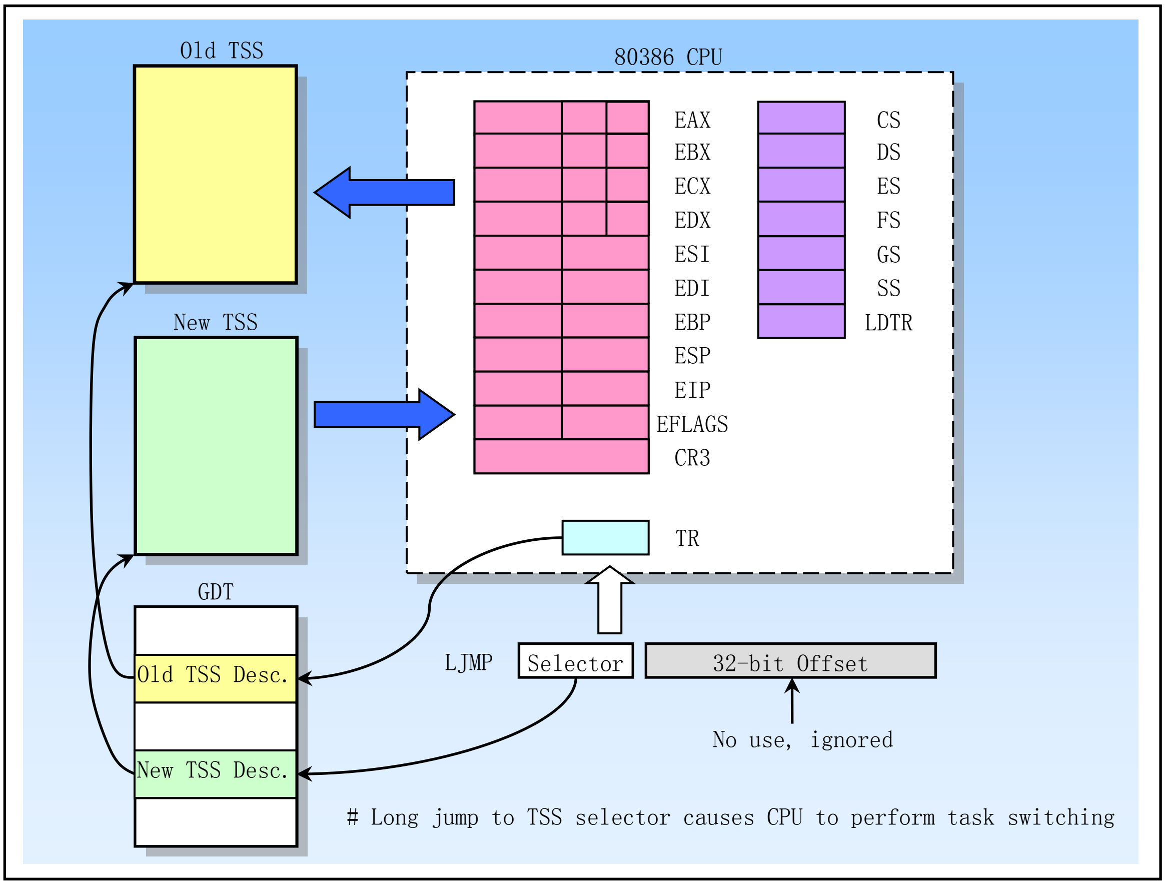


Figure 4-37 Task switching operation diagram

When switching to a new task, the processor does the following:

1. Obtain a TSS segment selector for a new task as the operand of the JMP or CALL instruction, or from the task gate, or from the previous task link field of the current TSS (for task switching caused by IRET).
2. Check if the current task is allowed to switch to the new task. Apply data access privilege rules to JMP and CALL instructions. The CPL of the current task, and the RPL of the new task segment selector must be less than or equal to the DPL of the TSS descriptor or task gate being referenced. Regardless of the DPL of the target task gate or TSS descriptor, exceptions, interrupts (except for interrupts generated using the INT n instruction), and IRET instructions allow task switching. The interrupt generated by the INT n instruction will check the DPL.
3. Check that the TSS descriptor for the new task is marked as present (P=1) and that the limit is valid (greater than 0x67). Any changes to the state of the processor are resumed when attempting to execute an instruction that will generate an error. This causes the return address of the exception handler to point to the error instruction instead of the next instruction of the error instruction. Therefore the exception handler procedure can handle the error condition and re-execute the task. The intervention of the exception handler procedure is completely transparent to the application.
4. If the task switch is generated from a JMP or IRET instruction, the processor will reset the busy flag B in the current task (old task) TSS descriptor; if the task switch is generated by a CALL instruction, exception or interrupt, the busy flag B Will not be changed.
5. 5. If the task switch is initiated with an IRET instruction, the processor resets the NT flag in a temporarily saved EFLAGS image; if the task switch is initiated with a CALL, JMP instruction, or an exception or interrupt, the NT flag is left unchanged in the saved EFLAGS image.
6. Save the state of the current (old) task to the TSS of the current task. The processor retrieves the base address of the current task TSS from the task register and copies the states of the following registers into the current TSS: all general purpose registers, the segment selector in the segment register, the flag register EFLAGS, and the instruction pointer EIP.
7. If the task switch was initiated with a CALL instruction, an exception, or an interrupt, the processor sets the NT flag in the EFLAGS image stored in the new task’s TSS; if initiated with an IRET instruction, the processor restores the NT flag from the EFLAGS image stored on the stack. If initiated with a JMP instruction, the NT flag is left unchanged.
8. If the task switch was initiated by a CALL, JMP instruction, or exception or interrupt, the processor sets the busy flag B in the new task TSS descriptor. If the task switch is generated by the IRET, the B flag is not changed.
9. Load the task register TR (including the hidden portion) using the segment selector and descriptor of the new task TSS. Set the TS flag in the control register CR0 image stored in the new task's TSS.
10. 10. Load the TSS status of the new task into the processor. This includes the LDTR register, the PDBR (CR3) register, the EFLAGS register, the EIP register, and general purpose registers and segment selectors. Any errors detected during this time will appear in the context of the new task.
11. Begins executing the new task. (To an exception handler, the first instruction of the new task appears not to have been executed.)

When the task switching operation is successfully performed, the state of the currently executing task is always saved. When the task resumes execution, the task will execute from the instruction pointed to by the saved EIP, and all registers will be restored to the value when the task was suspended.

When performing a task switch, the privilege level of the new task has nothing to do with the privilege level of the original task. The new task starts running at the privilege level specified by the CPL field of the CS register, which is loaded from the TSS. Because each task is isolated from each other by their independent address space and TSS segments, and the privilege level rules already control access to the TSS, the software does not need to perform privilege level checks at the time of task switching.

The task switching flag TS in the control register CR0 is set each time the task is switched. This flag is very useful for system software. The system software can use the TS flag to coordinate operations between the processor and the floating point coprocessor. The TS flag indicates that the context in the coprocessor may be different from that of the current task.

4.7.5 **Task Linking**

The TSS's previous task link field (Backlink) and the NT flag in EFLAGS are used to return execution to the previous task. The NT flag indicates whether the currently executing task is nested within the execution of another task, and the previous task link field of the current task holds the TSS selector for the higher level task in the nesting hierarchy, if there is one (see figure 4-38).

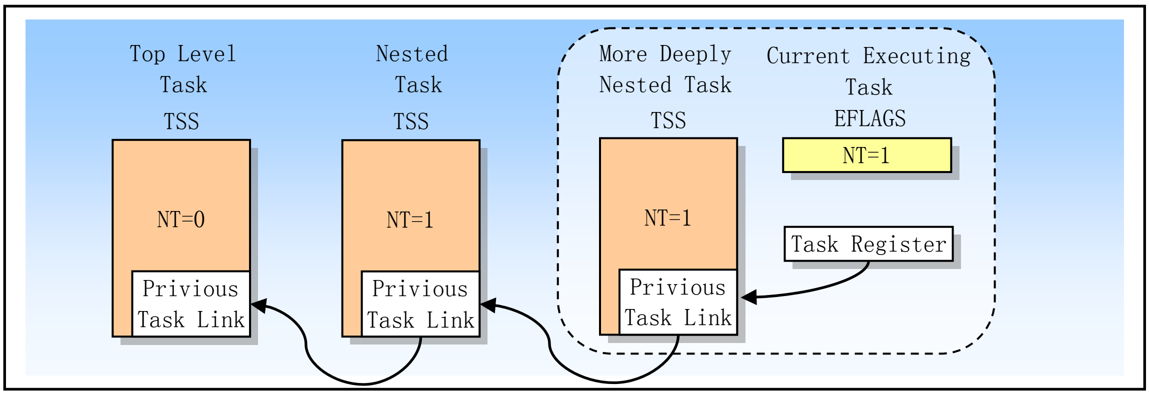


Figure 4-38 Nested Tasks

When a CALL instruction, interrupt, or exception causes a task switch, the processor copies the selector of the current TSS segment to the previous task link field of the new task TSS segment, and then sets the NT flag in EFLAGS. The NT flag indicates that the saved TSS segment selector is stored in the previous task link field of the TSS. If the software suspends a new task using the IRET instruction, the processor will return to the previous task using the value in the previous task link field and the NT flag. That is, if the NT flag is set, the processor will switch to the task specified in the previous task link field to execute.

Note that when the task switch is caused by a JMP instruction, the new task will not be nested. That is, the NT flag will be set to 0 and the previous task link field will not be used. JMP instructions are used in task switching where nesting is not desired.

Table 4-10 summarizes the usage of busy flag B (in the TSS segment descriptor), the NT flag, the previous task link field, and the TS flag (in CR0) during task switching. Note that programs running at any privilege level can modify the NT flag, so any program can set the NT flag and execute the IRET instruction. This approach will cause the processor to perform the tasks specified in the previous task link field of the current task TSS. In order to avoid successful execution of such forged task switching, the operating system should initialize this field of each TSS to zero.

Table 4-10 Effects of a task switch on Busy flag, NT flag, previous task link field, and TS flag

Flag or Field	Effect of JMP	Effect of CALL or Interrupt	Effect of IRET
Busy (B) flag of new task.	Flag is set. Must have been clear before.	Flag is set. Must have been clear before.	No change. Must have been set.
Busy flag of old task.	Flag is cleared.	No change. Flag is currently set.	Flag is cleared.
NT flag of new task.	No change.	Flag is set.	Restored to value from TSS of new task.
NT flag of old task.	No change.	No change.	Flag is cleared.
Previous task link field of new task.	No change.	Loaded with selector for old task's TSS	No change.
Previous task link filed of old task.	No change.	No change.	No change.
TS flag in CR0	Flag is set.	Flag is set.	Flag is set.

4.7.6 Task Address Space

The address space of a task consists of the segments that the task can access. These segments include the code referenced in the TSS, the data, the stack and system segments, and any other segments accessed by the task code. These segments are mapped to the processor's linear address space and then mapped to the processor's physical address space (either directly or through paging).

The LDT field in the TSS can be used to give each task its own LDT. For a given task, by putting all the segment descriptors associated with the task into the LDT, the task's address space can be isolated from other tasks. Of course, several tasks can also use the same LDT. This is a simple and effective way to allow certain tasks to communicate or control each other without having to discard the entire system's protective barrier. Because all tasks have access to the GDT, it is also possible to create shared segments that are accessed through this table.

If the paging mechanism is enabled, the CR3 register (PDBR) field in the TSS allows each task can also have its own set of page table for mapping linear addresses to physical addresses. Or, several tasks can share the same set of page tables.

4.7.6.1 Mapping Tasks to the Linear and Physical Address Spaces

There are two ways to map tasks to linear address spaces and physical address spaces:

- All tasks share a linear to physical address space mapping. This method can only be used when the paging mechanism is not enabled. When paging is not turned on, all linear addresses are mapped to the same physical address. When the paging mechanism is turned on, we can use this mapping from linear to physical address space by having all pages use a single page directory. If the demand page virtual storage technology is supported, the linear address space can exceed the size of the existing physical address space.
- Each task has its own linear address space and is mapped to the physical address space. We can use this mapping form by having each task use a different page directory. Because PDBR (Control Register CR3) is loaded every time a task is switched, each task can have a different page directory.

Linear address spaces for different tasks can be mapped to completely different physical addresses. If the entries (table entries) of different page directories point to different page tables, and the page tables also point to different pages in the physical address, then each task will not share any physical address.

For both methods of mapping task linear address space, the TSSs for all tasks must be stored in the shared physical address space area, and all tasks can access this area. In order for the processor to perform task switching and the TSS address mapping does not change when reading or updating the TSS, this mapping method is required. The linear address space mapped by the GDT should also be mapped to the shared physical address space. Otherwise, the role of the GDT is lost.

4.7.6.2 Task Logical Address Space

To share data between tasks, use one of the following methods to establish a shared logical-to-physical address space mapping for the data segment:

- By using the segment descriptor in the GDT. All tasks must be able to access the segment descriptors in the GDT. If some segment descriptors in the GDT point to segments in the linear address space and these segments are mapped into the physical address space shared by all tasks, then all tasks can share the code and data in those segments.
- Through a shared LDT. Two or more tasks can use the same LDT if their LDT fields in their TSSs point to the same LDT. If some segment descriptors in a shared LDT point to segments mapped to a common area of the physical address space, then all tasks sharing the LDT can share all of the code and data in those segments. This kind of sharing is better than sharing through GDT, because doing so can limit sharing to certain tasks. There are other tasks in the system that do not have access to these shared segments.
- Segment descriptors in different LDTs mapped to the common address area in the linear address space. If this common area in the linear address space maps each task to the same area of the physical address space, then these segment descriptors allow tasks to share the segments. Such segment descriptors are often referred to as alias segments. This sharing method is better than the one given above, because other segment descriptors in the LDT can point to separate unshared linear address regions.

4.8 The Initialization of Protected Mode

When the machine is powered on or hardware reset, the processor operates in the 8086-compatible real-address mode and executes the software initialization code starting at physical address 0xFFFFFFF0 (usually in EPROM) . The software initialization code must first set the necessary data structure information for basic system function operations, such as real-mode IDT tables (ie, interrupt vector tables) that handle interrupts and exceptions. If the processor will still work in real mode, the software must load the operating system modules and corresponding data to allow the application to run reliably in real mode. If the processor is going to work in protected mode, then the operating system software must load the data structure information necessary to protect the mode and then switch to protected mode.

4.8.1 First Instruction Executed

As described above, the first instruction acquired and executed after the hardware reset is located at the physical address 0xFFFFFFF0. This address is at the 16 bytes of the processor's highest physical address. This is usually the address range in which the EPROM firmware containing the software initialization code is located.

In real address mode, the address 0xFFFFFFF0 is outside the processor's 1 MB addressable range. The processor initializes to the starting address in the following manner. The CS register has two parts: the visible segment selector portion and the hidden base portion. In real address mode, the base address is typically formed by shifting the 16-bit segment selector value to the left by 4 bits to produce a 20-bit base address. However, during hardware reset, the segment selector in the CS register is loaded as 0xF000 and the base address is loaded as 0xFFFF0000. Therefore, the start address is formed by adding the base address to the EIP register (ie, 0xFFFF0000 + 0xFFF0 = 0xFFFFFFF0).

When the CS register first loads a new value after a hardware reset, the processor will follow the normal rules for address translation in real-address mode (ie, [CS base address = CS segment selector * 16]). To ensure that the base address in the CS register remains unchanged until the EPROM-based software initialization code is completed, the code must not contain a far-hop or remote call or allow an interrupt to occur (this will cause the CS selector value to change).

4.8.2 Initialization operation when entering protection mode

The processor is in real address mode after a hardware reset. Some basic data structures and code modules must be loaded into physical memory during initialization to support further initialization of the processor. Some of the data structures required for protected mode are determined by the processor memory management function. The processor supports a segmentation model that can be used from a single, unified address space flat model to a highly structured multi-segment model with several protected address spaces per task. The paging mechanism can be used to process large pieces of data structure information that is partially in memory and partially on disk. Both forms of address translation require the operating system to set the required data structure for the memory management hardware in memory. Therefore, before the processor can be switched to protected mode, the operating system's loading and initialization software (bootsect.s, setup.s, and head.s) must first set the basics of the data structure used in protected mode in memory. These data structures include the following:

- A protected-mode interrupt descriptor table IDT;
- A global descriptor table GDT;
- A task status segment TSS;
- A local descriptor table LDT;
- If paging is enabled, at least one page directory and one page table need to be set;
- A code segment containing execution code for the processor to switch to protected mode;
- Code modules that contain interrupts and exception handlers.

The software initialization code must also set the following system registers before being able to switch to protected mode:

- Global descriptor table base address register GDTR;
- Interrupt descriptor table base address register IDTR;
- Control register CR1--CR3;

After initializing these data structures, code modules, and system registers, the processor can be switched to protected mode by setting the protection mode flag PE (bit 0) of the CR0 register.

4.8.2.1 Protection Mode System Structure Table

The protected mode system table set in memory during software initialization relies primarily on the type of memory management that the operating system will support: flat, flat with paging, segmentation, or segmentation with paging.

In order to implement a flat memory model without paging, the software initialization code must at least set up a GDT table with one code segment and one data segment. Of course, the first item of the GDT table also needs to place a null descriptor. The stack can be placed in normal readable and writable data segments, so no special stack descriptors are needed. A flat memory model that supports the paging mechanism also requires a page directory and at least one page table. Before the GDT table can be used, the base address and limit for the GDT must be loaded into the GDTR register using the LGDT instruction.

Multi-segment models also require additional segments for the operating system, as well as segments and LDT table segments for each application. The segment descriptors of the LDT table are required to be stored in the GDT table. Some operating systems will allocate new segments and new LDT segments for the application. This approach provides maximum flexibility for dynamic programming environments, such as the Linux operating system. An embedded system like a process controller can pre-allocate a fixed number of segments and LDTs for a fixed number of applications, which is a simple and efficient way to implement the real-time system software environment structure.

4.8.2.2 Exceptions and Interrupt Initialization in Protected Mode

The software initialization code must set a protection mode IDT, which at least needs to contain the gate descriptor corresponding to each exception vector that the processor may generate. If an interrupt or trap gate is used, the gate descriptor can all point to the same code segment containing the interrupt and exception handling. If a task gate is used, each exception handling process that uses the task gate requires a TSS and associated code, data, and stack segments. If the hardware is allowed to generate an interrupt, then the gate descriptor must be set in the IDT for one or more interrupt handlers.

The IDT table base address and limit length must be loaded into the IDTR register using the LIDT instruction before the IDT can be used.

4.8.2.3 Paging initialization

The paging mechanism is set by the PG flag in the control register CR0. When this flag is cleared to 0 (ie, the state at the time of hardware reset), the paging mechanism is turned off; when the PG flag is set, the paging mechanism is turned on. The following data structures and registers must be initialized before setting the PG flag:

- Software must create at least one page directory and one page table in physical memory. If the page directory table contains a entry that points to itself, then you can eliminated the use of page table. At this point, the page directory table and the page table are stored on the same page.
- Load the physical base address of the page directory table into the CR3 register (also known as the PDBR register).
- The processor is in protected mode. If all other restrictions are met, the PG and PE flags can be set at the same time.

To maintain compatibility, the following rules must be observed when setting the PG flag (and PE flag):

- Instructions that set the PG flag should follow a JMP instruction immediately. The JMP instruction following the MOV CR0 instruction changes the execution stream, so it clears the instructions that 80X86 processor has taken or decoded. However, the Pentium and above processors use the Branch Target Buffer (BTB) for branch code orientation, thus eliminating the need to refresh the queue for branch instructions.
- The code that sets the PG flag to the jump instruction JMP must come from a page on the peer mapping (that is, the linear address before the jump is the same as the physical address after the paging is turned on).

4.8.2.4 Multitasking Initialization

If the multitasking mechanism is to be used and/or the privilege level is allowed to change, then the software initialization code must have at least one TSS and the corresponding TSS segment descriptor (because the stack segment pointers for privilege levels 0, 1, and 2 need to be taken from the TSS). Do not mark the TSS descriptor as busy (do not set the busy flag), which is only set by the processor when performing task switching. Like the LDT segment descriptor, the TSS descriptor is also stored in the GDT.

After the processor switches to protected mode, the selector of the TSS segment descriptor can be loaded into the task register TR using the LTR instruction. This command marks the TSS as busy (B = 1), but does not perform a task switch operation. The processor can then use this TSS to locate the stack of privilege levels 0, 1, and 2. In protected mode, the selector of the TSS segment must be loaded first before the software performs the first task switch, because the task switch will copy the current task state into the TSS.

Subsequent to the LTR instruction execution, subsequent operations on the task register are performed by task switching. Similar to other segments and LDTs, TSSs and TSS descriptors can be pre-allocated or allocated when needed.

4.8.3 Mode Switching

In order for the processor to operate in protected mode, mode switching must be performed from real address mode. Once in protected mode, the software usually no longer needs to go back to real address mode. In order to be able to run programs programmed for real-address mode, it is usually more convenient to run in virtual-8086 mode than to switch back to real mode.

4.8.3.1 Switching to Protected Mode

Before switching to protected mode, you must first load some minimum system data structures and code modules. Once these system tables are created, the software initialization code can be switched to protected mode. By executing the MOV CR0 instruction that sets the PE flag in the CR0 register, we can enter the protection mode. (In the same instruction, the PG flag of CR0 can be used to enable the paging mechanism.) When running in protected mode, the privilege level CPL is 0. In order to ensure the compatibility of the program, the switching operation should be carried out as follows:

1. Disable interrupts. Maskable hardware interrupts can be disabled using the CLI instruction. The NMI is disabled by hardware circuitry. At the same time, the software should ensure that no exceptions or interruptions occur during mode switching operations.
2. Execute the LGDT instruction to load the base address of the GDT table into the GDTR register.
3. Execute the MOV CR0 instruction that sets the PE flag (optional setting of the PG flag) in the control register CR0.
4. Execute a far jump JMP or a far call CALL instruction immediately after the MOV CR0 instruction. This operation is usually a far jump to or far from the next instruction in the instruction stream.
5. If a local descriptor table is to be used, execute the LLDT instruction to load the LDT segment selector into the LDTR register.
6. Execute the LTR instruction to load the task register TR with the segment selector of the initial protected mode task or the segment descriptor of the writable memory area. This writable memory area is used to store the TSS information of the task when the task is switched.
7. After entering protected mode, the segment register still contains the contents in real address mode. The JMP or CALL instruction in step 4 resets the CS register. Do one of the following to update the contents of the remaining segment registers: (1) Reload registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not used, load them with a null selector. (2) Execute a JMP or CALL instruction on a new task that automatically resets the value of the segment register and branches to the new code segment.
8. Execute the LIDT instruction to load the base address and limit of the protected mode IDT table into the IDTR register.
9. Execute the STI instruction to turn on the maskable hardware interrupt and perform the necessary hardware operations to turn on the NMI interrupt.

In addition, the JMP or CALL instruction immediately following the MOV CR0 instruction changes the execution flow. If the paging mechanism is enabled, the code between the MOV CR0 instruction and the JMP or CALL instruction must come from a page on the peer mapping (ie, the linear address before the jump is the same as the physical address after paging). The target instruction to which the JMP or CALL instruction jumps does not need to be on the peer mapping page.

4.8.3.2 Switch back to real address mode

If you want to switch back to real address mode, you can use the MOV CR0 instruction to clear the PE flag in control register CR0. The process of re-entering the real address mode should follow these steps:

1. Disable interrupts. Maskable hardware interrupts can be disabled using the CLI instruction. The NMI is disabled by hardware circuitry. At the same time, the software should ensure that no exceptions or interruptions occur during mode switching operations.
2. 2. If the paging mechanism is enabled, you need to execute:
   - Transfer the program control to linear address of the peer map (ie the linear address is equal to the physical address).
   - Make sure the GDT and IDT are on the peer mapped page.
   - Clear the PG flag in CR0.
   - Set to 0x00 in the CR3 register to refresh the TLB buffer.
3. Transfer the control of the program to a readable segment of length 64KB (0xFFFF). This step loads the CS register using the segment limit required by the real mode.
4. Load the SS, DS, ES, FS, and GS segment registers with a selector that points to a descriptor with the following set values.
   - Limit = 64KBytes (0xFFFF).
   - Byte granularity (G = 0).
   - Expand up (E=0).
   - Writable (W=1).
   - Present (P=1).
5. Execute the LIDT instruction to point to the real address mode interrupt table in the 1MB real mode address range.
6. Clear the PE flag in CR0 to switch to real address mode.
7. Execute a far jump instruction to jump to a real mode program. This step refreshes the instruction queue and loads the appropriate base address and access value for the CS register.
8. The SS, DS, ES, FS, and GS registers are loaded as needed for the real address mode code. If any of the registers are not used in real address mode, write 0 to them.
9. Execute the STI instruction to turn on the maskable hardware interrupt and perform the necessary hardware operations to turn on the NMI interrupt.

4.9 A Simple Multitask Kernel Example

As a summary of this chapter and the previous chapters, this section provides a complete description of the design and implementation of a simple multitasking kernel. This kernel example contains two privilege level 3 user tasks and a privilege level 0 system call interrupt procedure. We first explain the basic structure of this simple kernel and the basic principles of load operation, then we describe how it is loaded into the machine's RAM memory and how the two tasks are switched. Finally, we give the source code to implement this simple kernel: boot boot.s and protected mode multitasking kernel program head.s.

4.9.1 Multitasking program structure and working principle

The kernel example given in this section consists of two source files. One is the bootloader boot.s compiled with the as86 language, which is used to load kernel code into the memory from the boot disk when the computer system is powered up; the other is the kernel program head.s compiled with GNU as assembly language. It implements two tasks running on the privilege level 3 to switch between each other under the control of the clock interrupt, and also implements a system call for displaying characters on the screen.

We refer to these two tasks as task A and task B (or task 0 and task 1). They call the system call to display the characters 'A' and 'B' on the screen, respectively, and switch to another task every 10 milliseconds. Task A continuously calls the system call to display the character 'A' on the screen; task B always displays the character 'B'. To terminate this kernel instance program, you will need to restart the machine or shut down the running simulated PC runtime environment software.

The boot.s program generates a total of 512 bytes of code that will be stored in the first sector of the floppy image file, as shown in Figure 4-39. When the PC is powered on, the program in the ROM BIOS will load the first sector on the boot disk to the beginning of the physical memory 0x7c00 (31KB) position, and transfer the execution control to 0x7c00 to start running the boot code.

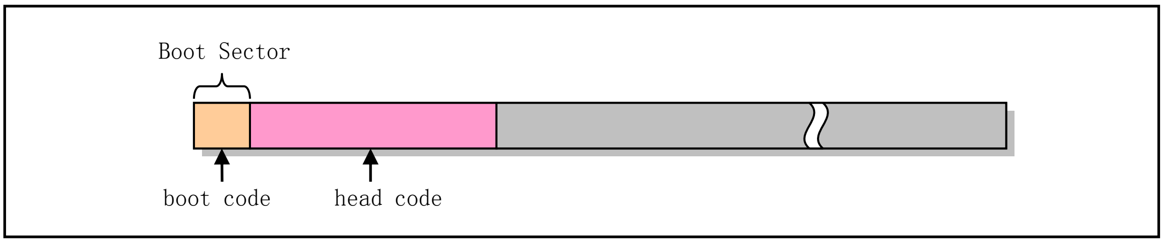


Figure 4-39 Floppy disk image file

The main function of the boot program is to load the head kernel code in the floppy disk or image file to a specified location in the memory. After setting the temporary GDT table and other information, set the processor to run in protected mode. Then jump to the head code to run the kernel code. In fact, the boot.s program will first use the ROM BIOS interrupt int 0x13 to read the head code in the floppy disk to the beginning of the memory location 0x10000 (64KB), and then move the head code to the beginning of the memory location 0. Finally, the enable protection operation mode flag in the control register CR0 is set, and jumps to the memory location 0 to start executing the head code. A schematic diagram of the boot code moving the head code in memory is shown in Figure 4-40.

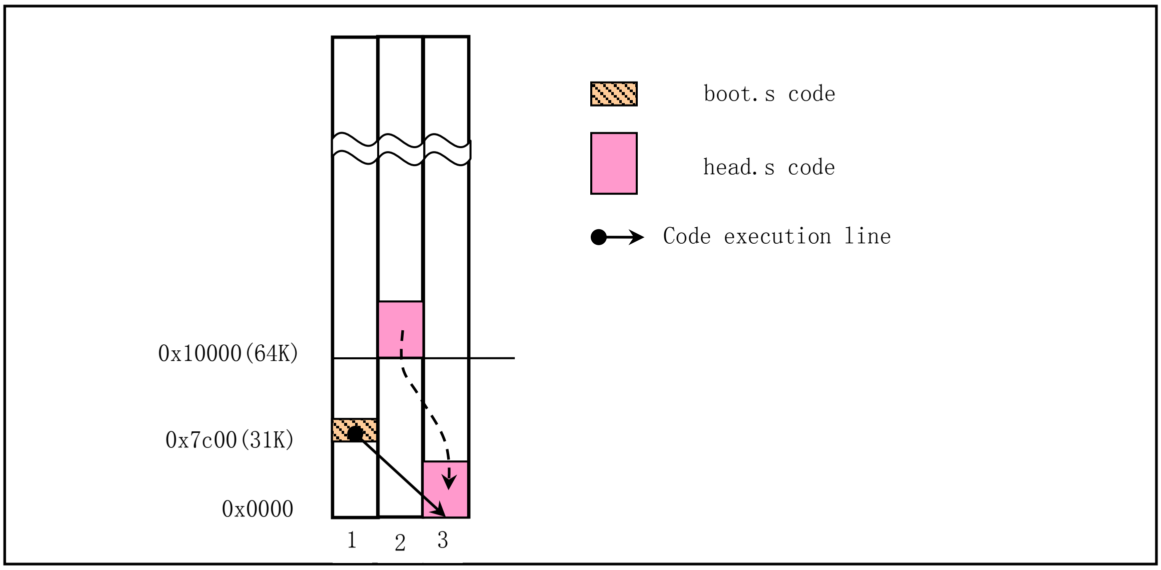


Figure 4-40 The movement and distribution of kernel code in physical memory

The main reason for moving the head kernel code to the beginning of physical memory 0 is that it is simpler to set up the GDT table, so it can also make the head.s program as short as possible. But we can't let the boot program load the head code directly from the floppy or image file into memory location 0. Because the interrupt vector table used by the BIOS is at the beginning of memory 0, and the load operation requires the use of the interrupt process provided by the ROM BIOS. So if you load the head code directly into memory 0, the BIOS interrupt vector table will be destroyed.

Of course, we can also load the head code into the memory 0x10000 and then jump directly to the location to run the head code. The source program using this method can be downloaded from the oldlinux.org website, as explained below.

The head program runs in 32-bit protected mode, which mainly includes the code of the initial setting, the process code of the clock interrupt int 0x08, the process code of the system call interrupt int 0x80, and the code and data of task A and task B. The initial setting work mainly includes: (1) resetting the GDT table; (2) setting the system timer chip; (3) resetting the IDT table and setting the clock and system call interrupt gate; (4) moving to the task A for execution.

In the virtual address space, the kernel and task code allocation diagram for the head.s program is shown in Figure 4-41. In fact, all the code and data segments in this kernel example correspond to the same area of physical memory, that is, the area starting from physical memory location 0.

The contents of the global code segment and the data segment descriptor in the GDT are set to as following:

- The base address is 0x0000;
- The segment limit value is 0x07ff. Since the granularity is 1, the actual segment length is 8MB.

The global display data segment is set as following:

- The base address is 0xb8000;
- The segment limit length is 0x0002, so the actual segment length is 8KB, corresponding to the display memory area.

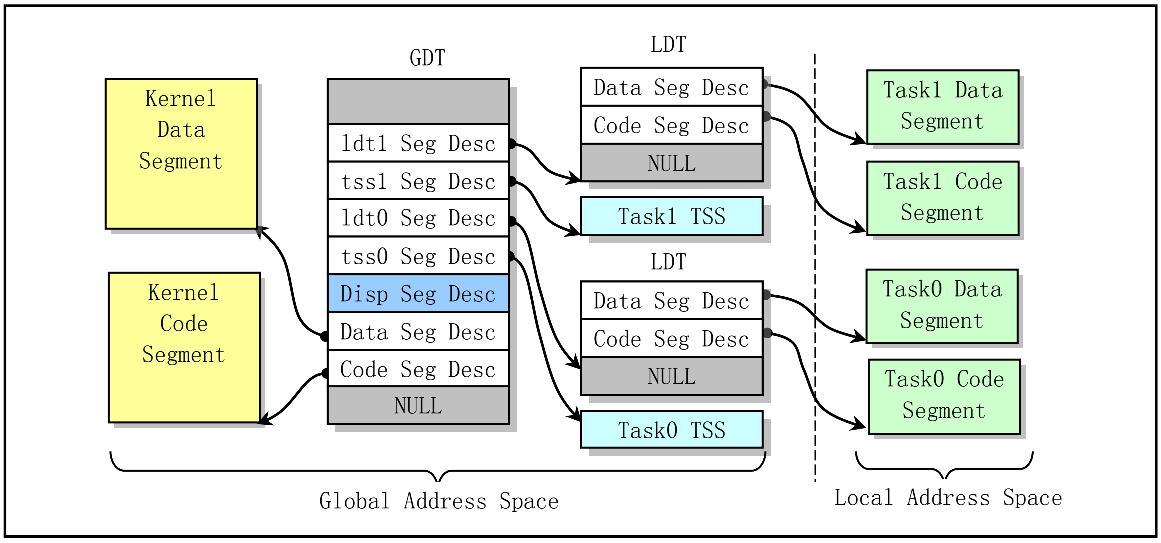


Figure 4-41 Diagram of the allocation of kernel and tasks in the virtual address space

The contents of the code segment and data segment descriptors in the LDT for both tasks are also set as follows:

- The base address is 0x0000;
- The segment length is 0x03ff and the actual segment length is 4MB.

Therefore, in the linear address space, the code and data segments of this "core" and the code and data segments of the task start from linear address 0 and since they do not use the paging mechanism, they all directly correspond to the beginning of physical address 0. In the object file compiled by the head program and the resulting floppy image file, the organization of the code and data is shown in Figure 4-42.

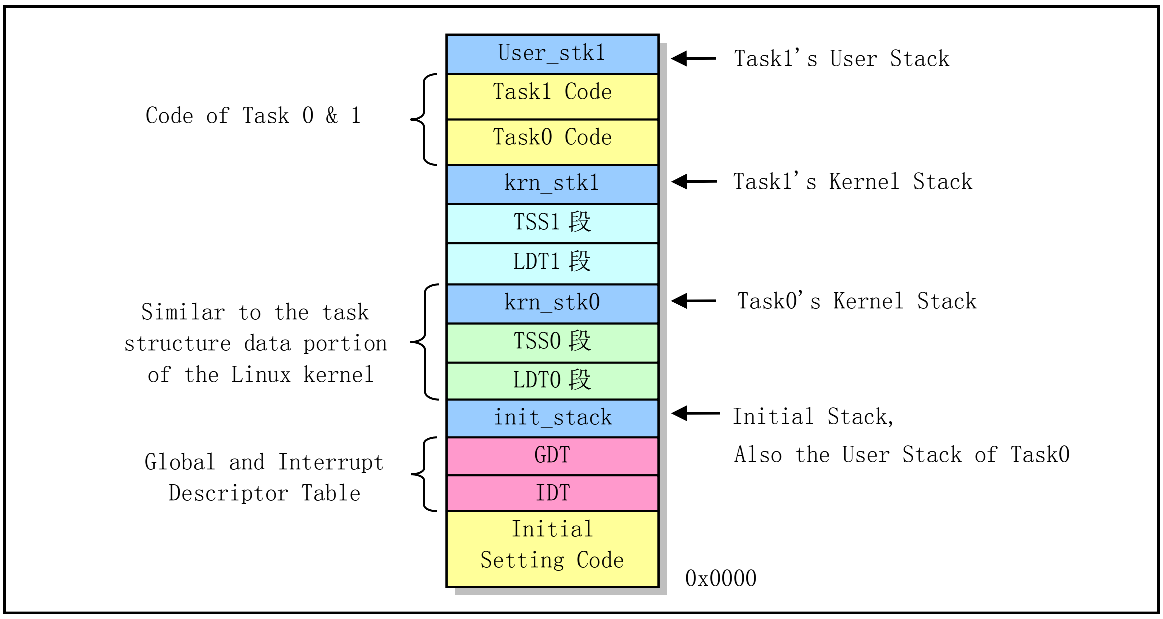


Figure 4-42 Kernel image file and in-memory head code and data diagram

Code at privilege level 0 cannot be directly transferred to the code of privilege level 3, but control transfer can be achieved by using the interrupt return operation. So when the GDT, IDT, and timing chips are initialized, we use the interrupt return instruction IRET to start the first task.

The specific implementation method is to manually set a return environment in the initial stack init_stack. That is, after loading the TSS segment selector of task 0 into the task register LTR and loading the LDT segment selector into the LDTR, the user stack pointer of task 0 (0x17: init_stack), code pointer (0x0f: task0)), and the flag register value is pushed onto the stack, and then the interrupt return instruction IRET is executed.

The IRET instruction pops the stack pointer on the stack as the task 0 user stack pointer, restores the contents of the hypothetical task 0 flag register, and pops the code pointer in the stack into the CS:EIP register, thus starting the execution of the task 0 code. This complete the control transfer from privilege level 0 to privilege level 3 code.

In order to switch the running task every 10 milliseconds, the channel 0 of the timer chip 8253 is set in the head.s program to send a clock interrupt request signal to the interrupt control chip 8259A every 10 milliseconds. When PC is powered on, the ROM BIOS program has set the clock interrupt request signal is set to the interrupt vector 8 in the 8259A, so we need to perform the task switching operation in the interrupt 8 handler procedure. The task switching method is implemented by looking at the current running task number in the current variable. If current is currently 0, the TSS selector of task 1 is used as the operand to execute the far jump instruction, thereby switching to task 1 for execution, otherwise vice versa.

Each task will first put the ASCII code of a character into the register AL, and then call the system interrupt to call int 0x80. The system call processing process will call a simple character write screen subroutine, display the characters in the register AL on the screen, and record the next position of the screen displaying the characters as the screen position for displaying the characters next time. After a character has been displayed, the task code is delayed for a period of time using the loop statement, and then jumps to the beginning of the task code to continue the loop until a timed interrupt occurs for 10 milliseconds. At this point the code will switch to another task to run. For task A, the character 'A' will always be stored in the register AL, and the character 'B' will always be stored in the AL during the task B runtime. Therefore, when the program is running, we will see a series of characters 'A' and a series of characters 'B' displayed continuously on the screen at intervals, as shown in Figure 4-43.

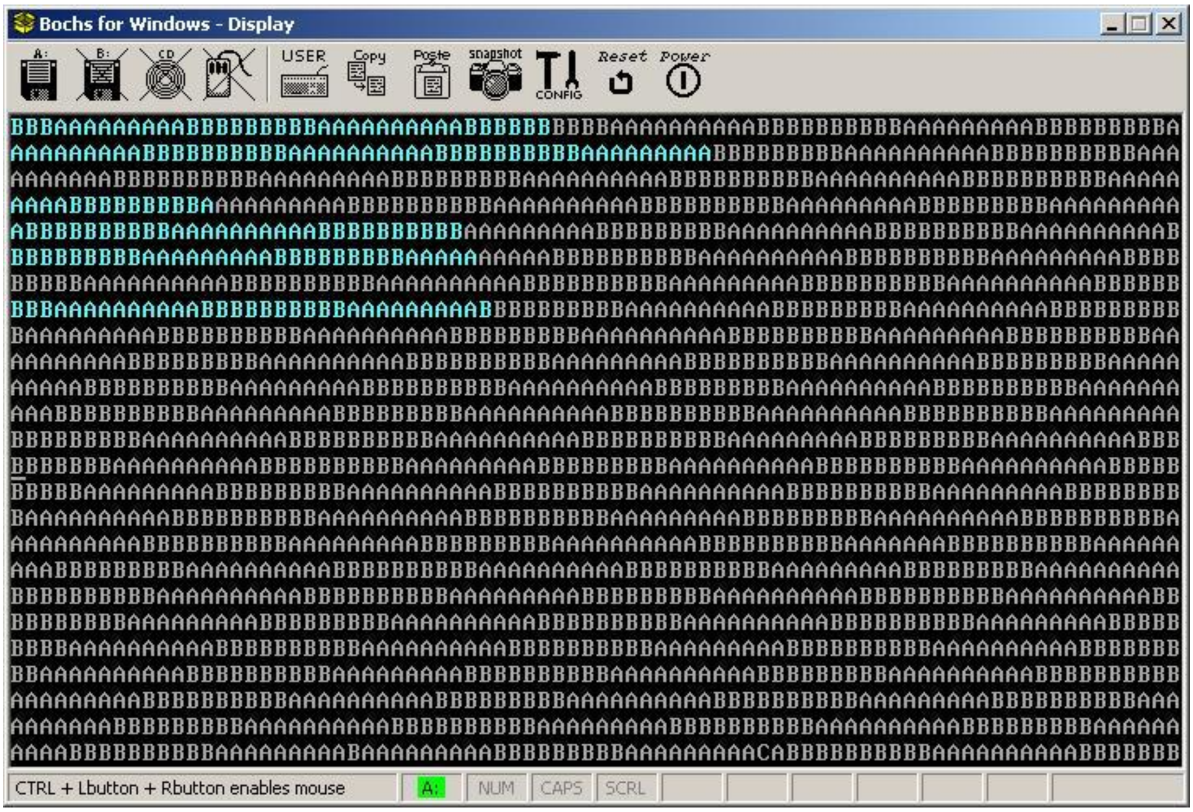

Bochs for Windows - Display
A:
B:
CD
USER
Copy
Paste
snapshot
CONFIG
Reset
Power
CTRL + Lbutton + Rbutton enables mouse
A:
NUM
CAPS
SCRL

Figure 4-43 Simple kernel running screen display

Figure 4-43 shows the screen display of this running kernel in Bochs. Careful readers will find that a character 'C' is displayed on the bottom line of the figure. This is because the PC accidentally generates another interrupt that is not a clock interrupt or a system call interrupt. Because we have installed a default interrupt handler for all other interrupts in the program. When a different interrupt occurs, the system will run the default interrupt handler with code to display character 'C', so a character 'C' will be displayed on the screen and the interrupt will be exited.

Detailed comments on the boot.s and head.s programs are given below. For the compilation and operation of this simple kernel example, please refer to the section "Compiling and Running a Simple Kernel Sample Program" in the last chapter of this book.

4.9.2 Boot Startup Program boot.s

In order to make the program as simple as possible, this bootloader can only load the head code of no more than 16 sectors, and directly use the interrupt vector number set by the ROM BIOS, that is, the interrupt number of the timer interrupt request number is still 8. This is different from what is used in Linux systems. The Linux system resets the 8259A interrupt control chip during kernel initialization and maps the clock interrupt request signal to interrupt 0x20. See the chapter "Kernel Boot Boot Program" for details.

```
! boot.s program
! First use BIOS interrupt to load head code into memory 0x10000, and then move it to memory 0.
! Finally enter protected mode, jump to the beginning of head code at 0 to continue running.
```

```
BOOTSEG = 0x07c0                     ! This program is loaded into memory at 0x7c00 by BIOS.
SYSSEG  = 0x1000                     ! The head is first loaded to 0x10000 and moved to 0x0.
SYSLEN  = 17                         ! Max num of disk sectors occupied by the kernel.
entry start
start:
        jmpi    go,#BOOTSEG          ! Jump between segments to 0x7c0:go. All segment reg
go:     mov     ax,cs                ! are 0 when runs. This jump ins load CS with 0x7c0.
        mov     ds,ax                ! Both DS and SS point to the 0x7c0 segment.
        mov     ss,ax
        mov     sp,#0x400            ! Set temp stack pointer. Its value needs to be larger
                                     ! than this program and has a certain space.
! Load the kernel code to the beginning of memory at address 0x10000.
load_system:
        mov     dx,#0x0000           ! Use BIOS int 0x13 func2 to load head code from bootdisk.
        mov     cx,#0x0002           ! DH-head no; DL-drive no; CH-10 bit track no low 8 bits
        mov     ax,#SYSSEG           ! CL-Bits7,6 track num high 2 bits ,bit 5-0 start secter
        mov     es,ax                ! ES:BX - Read in buffer location (0x1000:0x0000).
        xor     bx,bx                ! AH-read sector func num; AL- num of sectors read (17).
        mov     ax,#0x200+SYSLEN
        int     0x13
        jnc     ok_load              ! If no error occurs, then continues, otherwise dead.
die:    jmp     die

! Move kernel code to memory location 0. Total of 8KB bytes are moved (kernel code <8kb).
ok_load:
        cli                          ! Disable interrupts.
        mov     ax, #SYSSEG          ! Move from DS:SI(0x1000:0) to ES:DI(0:0).
        mov     ds, ax
        xor     ax, ax
        mov     es, ax
        mov     cx, #0x1000          ! Set the move 4K times, one word at a time.
        sub     si,si
        sub     di,di
        rep     movw                 ! Execute the repeat move instruction.
! Load IDT and GDT base address registers IDTR and GDTR.
        mov     ax, #BOOTSEG
        mov     ds, ax               ! Let DS point to 0x7c0 segment again.
        lidt    idt_48               ! Load IDTR. 2 byte table limit, 4 byte linear base addr.
        lgdt    gdt_48               ! Load GDTR. 2 byte table limit, 4 byte linear base addr.

! Set CR0 to enter protection mode. The seg selector value 8 refers to 2nd descriptor in GDT.
        mov     ax,#0x0001           ! Set the protection mode flag PE (bit 0) in CR0.
        lmsw    ax                   ! Jump to segment specified by the selector, offset 0.
        jmpi    0,8                  ! Seg value is now a selector. The linear base addr is 0.

! The following is the content of GDT. It has 3 seg descriptors. The first one is not used,
! the other two are code and data segment descriptors.
gdt:    .word   0,0,0,0              ! First descriptor not used. Occupies 8 bytes.

        .word   0x07FF               ! Descriptor 1. 8Mb - limit=2047 (2048*4096=8MB).
        .word   0x0000               ! Segment base address = 0x00000.
        .word   0x9A00               ! Code segment, readable/executable.
        .word   0x00C0               ! Segment attribute granularity = 4KB, 80386.

        .word   0x07FF              ! Descriptor 1. 8Mb - limit=2047 (2048*4096=8MB).
        .word   0x0000              ! Segment base address = 0x00000.
        .word   0x9200              ! Data segment, readable and writable.
        .word   0x00C0              ! Segment attribute granularity = 4KB, 80386.
! The following are the 6-byte operands of the LIDT and LGDT instructions, respectively.
idt_48: .word   0                   ! The IDT table length is 0.
        .word   0,0                 ! The linear base address of IDT table is also zero.
gdt_48: .word   0x7ff               ! GDT limit is 2048 bytes, can hold 256 descriptors.
        .word   0x7c00+gdt,0        ! Linear base of GDT is at offset gdt of seg 0x7c0.
.org 510
        .word   0xAA55           ! Boot sector flag. Must be at last 2 bytes of boot sector.
```

4.9.3 Multitasking Kernel program head.s

After entering the protection mode, the main reason for the head.s program to re-establish and set the IDT and GDT tables is to make the program clearer in structure, and to be consistent with the setting of the two tables in the Linux 0.12 kernel source code. . Of course, for this program, we can directly use the IDT and GDT table locations set in boot.s, and fill in the appropriate descriptor items.

```
# Head.s contains code for 32-bit protected mode init, clock & system call interrupts, and two
# tasks code. After initialization, the program moves to task 0 to start execution, and the
# switching operation between tasks 0 and 1 is performed under the clock interrupt.
LATCH           = 11930             # Timer count, interrupt is sent every 10 ms.
SCRN_SEL        = 0x18              # The segment selector for the screen display memory.
TSS0_SEL        = 0x20              # TSS segment selector for task 0.
LDT0_SEL        = 0x28              # LDT segment selector for task 0.
TSS1_SEL        = 0X30              # TSS segment selector for task 1.
LDT1_SEL        = 0x38              # LDT segment selector for task 1.
.text
startup_32:
# First load DS, SS, and ESP. The linear base address of all segments is 0.
        movl $0x10,%eax             # 0x10 is the data segment selector in the GDT.
        mov %ax,%ds
        lss init_stack,%esp
# Re-set the IDT and GDT tables at new location.
        call setup_idt              # Setup IDT.
        call setup_gdt              # Setup GDT.
        movl $0x10,%eax             # Reload all segment registers after changing GDT.
        mov %ax,%ds
        mov %ax,%es
        mov %ax,%fs
        mov %ax,%gs
        lss init_stack,%esp
# Set 8253 timing chip. Channel 0 is set to generate an interrupt request every 10 ms.
        movb $0x36, %al             # Control word: Channel 0 in mode 3, Count in binary.
        movl $0x43, %edx            # 0x43 is write port of control word register.
        outb %al, %dx
        movl $LATCH, %eax           # Init count set to LATCH (1193180/100), freq. 100HZ.
        movl $0x40, %edx             # The port of channel 0.
        outb %al, %dx                # Write initial count value to channel 0 in two steps.
        movb %ah, %al
        outb %al, %dx
# Set the timer interrupt gate descriptor at item 8 of the IDT table.
        movl $0x00080000, %eax       # EAX high word set to kernel code seg selector 0x0008.
        movw $timer_interrupt, %ax   # Set timer int gate descriptor. Get handler address.
        movw $0x8E00, %dx           # Interrupt gate type is 14, plevel is 0 or hardware used.
        movl $0x08, %ecx             # Clock interrupt vector no. set by BIOS is 8.
        lea idt(,%ecx,8), %esi       # Put IDT Descriptor 0x08 address into ESI and set it.
        movl %eax,(%esi)
        movl %edx,4(%esi)
# Set the system call trap gate descriptor at item 128 (0x80) of the IDT table.
        movw $system_interrupt, %ax  # Set system call gate descriptor. Get handler address.
        movw $0xef00, %dx            # Trap gate type is 15, code of plevel 3 is executable.
        movl $0x80, %ecx             # System call vector no. is 0x80.
        lea idt(,%ecx,8), %esi       # Put IDT Descriptor 0x80 address into ESI and set it.
        movl %eax,(%esi)
        movl %edx,4(%esi)
# Now, to use IRET to move to task 0 (A), we manually prepare to setup the stack contents.
# See Figure4-29 for the stack contents we need to setup. Refer to include/asm/system.h.
        pushfl                       # Reset NT flag in EFLAGS to disable task switch when
        andl $0xffffbfff, (%esp)     # execute IRET instruction.
        popfl
        movl $TSS0_SEL, %eax         # Load task0's TSS seg selector into task register TR.
        ltr %ax
        movl $LDT0_SEL, %eax         # Load task0's LDT seg selector into LDTR.
        lldt %ax                     # TR and LDTR need only be manually loaded once.
        movl $0, current             # Save current task num 0 into current variable.
        sti                          # Enable int, build a scene on stack for int returns.
        pushl $0x17                  # Push task 0 data (stack) seg selector onto stack.
        pushl $init_stack            # Push the stack pointer (same as push ESP).
        pushfl                       # Push the EFLAGS.
        pushl $0x0f                  # Push current local space code seg selector.
        pushl $task0                 # Push task 0 code pointer onto stack.
        iret                         # This causes execution moves to task0 in plevel 3.

# The following are the subroutines for setting descriptor items in GDT and IDT.
setup_gdt:                           # GDT table position & limit are set using
        lgdt lgdt_opcode             # 6-byte operand lgdt_opcode.
        ret
# The following code is used to temporarily set all 256 interrupt gate descriptors in the
# IDT table to the same default value. All use the default interrupt handler ignore_int.
# The specific method of setting is: first set the contents of 0-3 bytes and 4-7 bytes of
# the default interrupt gate descriptor into the eax and edx register pairs. Then, using
# this register pair, the interrupt descriptor is cyclically filled into the IDT table.
setup_idt:                       # Set all 256 int gate descriptors to use default handler.
        lea ignore_int,%edx          # The same way as setting timer int gate descriptor.
        movl $0x00080000,%eax        # The selector is 0x0008.
        movw %dx,%ax
        movw $0x8E00,%dx             # Interrupt gate type is 14, plevel is 0.
        lea idt,%edi
        mov $256,%ecx                # Loop through all 256 gate descriptor entries.
```

```
rp_idt: movl %eax,(%edi)
        movl %edx,4(%edi)
        addl $8,%edi
        dec %ecx
        jne rp_idt
        lidt lidt_opcode             # IDTR register is loaded with a 6-byte operand.
        ret

# Display characters subroutine. Get current cursor position & display char in AL.
# The entire screen can display 80 X 25 (2000) characters.
write_char:
        push %gs                     # First save the register to be used, EAX is
        pushl %ebx                   # saved by the caller.
        mov $SCRN_SEL, %ebx          # Then let GS point to display mem seg (0xb8000).
        mov %bx, %gs
        movl scr_loc, %bx            # Get current char display position from scr_loc.
        shl $1, %ebx                 # Since each char has one attribute byte, so actual
        movb %al, %gs:(%ebx)         # display memory offset should multiplied by 2.
        shr $1, %ebx                 # After putting char into display memory, divide the
        incl %ebx                    # position value by 2 plus 1 to get the next position.
        cmpl $2000, %ebx             # If position is greater than 2000, it is reset to 0.
        jb 1f
        movl $0, %ebx
1:      movl %ebx, scr_loc           # Finally save this position value (scr_loc),
        popl %ebx                    # and pop up the contents of saved register, return.
        pop %gs
        ret

# The following are 3 interrupt handlers: default, timer, and system call interrupt.
# Ignore_int is default handler. If system generates other interrupts, it display char 'C'.
.align 2
ignore_int:
        push %ds
        pushl %eax                   # Let DS point to the kernel data segment because
        movl $0x10, %eax             # the interrupt handler belongs to the kernel.
        mov %ax, %ds
        movl $67, %eax               # Put 'C' in AL, call write_char to display on screen.
        call write_char
        popl %eax
        pop %ds
        iret

# This is the timer interrupt handler. The main function is to perform task switching operations.
.align 2
timer_interrupt:
        push %ds
        pushl %eax
        movl $0x10, %eax             # First let DS point to the kernel data segment.
        mov %ax, %ds
        movb $0x20, %al              # Then send EOI to 8259A to allow other interrupts.
        outb %al, $0x20
        movl $1, %eax                # Then check current task to switch task 0 and 1.
        cmpl %eax, current
        je 1f
        movl %eax, current           # If current task is 0, save 1 in current and jump to
        ljmp $TSS1_SEL, $0           # task 1 to execute. The offset of jump is useless.
        jmp 2f
1:      movl $0, current             # If current task is 1, save 0 in current and jump to
        ljmp $TSS0_SEL, $0           # task 0 to execute.
2:      popl %eax
        pop %ds
        iret

# The system call int 0x80 handler. This example has only one display char function.
.align 2
system_interrupt:
        push %ds
        pushl %edx
        pushl %ecx
        pushl %ebx
        pushl %eax
        movl $0x10, %edx             # First let DS point to the kernel data segment.
        mov %dx, %ds
        call write_char              # Then call routine write_char to display char in AL.
        popl %eax
        popl %ebx
        popl %ecx
        popl %edx
        pop %ds
        iret

/*********************************************/
current:.long 0                      # Store current task number (0 or 1).
scr_loc:.long 0                      # Store screen current display position.

.align 2
lidt_opcode:
        .word 256*8-1                # 6-byte operand for set IDTR : table size & base.
        .long idt
lgdt_opcode:
        .word (end_gdt-gdt)-1        # 6-byte operand for set IDTR : table size & base.
        .long gdt

.align 3
idt:    .fill 256,8,0                # IDT. 256 gate descriptors, each 8 bytes, total 2KB.
# The following is GDT table contents (of descriptors).
gdt:    .quad 0x0000000000000000     # [0] The first segment descriptor is not used.
        .quad 0x00c09a00000007ff     # [1] Kernel code descriptor. Its selector is 0x08
        .quad 0x00c09200000007ff     # [2] Kernel data descriptor. Its selector is 0x10
        .quad 0x00c0920b80000002     # [3] Display mem descriptor. Its selector is 0x18
        .word 0x68, tss0, 0xe900, 0x0  # [4] TSS0 descriptor. Its selector is 0x20.
        .word 0x40, ldt0, 0xe200, 0x0  # [5] LDT0 descriptor. Its selector is 0x28
        .word 0x68, tss1, 0xe900, 0x0  # [6] TSS1 descriptor. Its selector is 0x30
        .word 0x40, ldt1, 0xe200, 0x0  # [7] LDT1 descriptor. Its selector is 0x38
end_gdt:
        .fill 128,4,0                # Initial kernel stack space.
```

```
init_stack:                          # Stack pointer when first enter protected mode.
        .long init_stack             # Stack segment offset position.
        .word 0x10                   # Stack segment, same as kernel data seg (0x10).

# Below is the local segment descriptor in the LDT table segment of task 0.
.align 3
ldt0:   .quad 0x0000000000000000     # [0] The first descriptor is not used.
        .quad 0x00c0fa00000003ff     # [1] The local code descriptor, its selector is 0x0f
        .quad 0x00c0f200000003ff     # [2] The local data descriptor, its selector is 0x17
# Content of TSS seg for task 0. Note fields such as labels do not change when task switches.
tss0:   .long 0                      /* back link */
        .long krn_stk0, 0x10         /* esp0, ss0 */
        .long 0, 0, 0, 0, 0          /* esp1, ss1, esp2, ss2, cr3 */
        .long 0, 0, 0, 0, 0          /* eip, eflags, eax, ecx, edx */
        .long 0, 0, 0, 0, 0          /* ebx esp, ebp, esi, edi */
        .long 0, 0, 0, 0, 0, 0       /* es, cs, ss, ds, fs, gs */
        .long LDT0_SEL, 0x8000000    /* ldt, trace bitmap */

        .fill 128,4,0                # This is the kernel stack space for task 0.
krn_stk0:

# Task 1 LDT table content and TSS segment content.
.align 3
ldt1:   .quad 0x0000000000000000     # [0] The first descriptor is not used.
        .quad 0x00c0fa00000003ff     # [1] The selector is 0x0f, base = 0x00000。
        .quad 0x00c0f200000003ff     # [2] The selector is 0x17, base = 0x00000。

tss1:   .long 0                      /* back link */
        .long krn_stk1, 0x10         /* esp0, ss0 */
        .long 0, 0, 0, 0, 0          /* esp1, ss1, esp2, ss2, cr3 */
        .long task1, 0x200           /* eip, eflags */
        .long 0, 0, 0, 0             /* eax, ecx, edx, ebx */
        .long usr_stk1, 0, 0, 0      /* esp, ebp, esi, edi */
        .long 0x17,0x0f,0x17,0x17,0x17,0x17    /* es, cs, ss, ds, fs, gs */
        .long LDT1_SEL, 0x8000000     /* ldt, trace bitmap */

        .fill 128,4,0                # This is the kernel stack space for task 1. Its user
krn_stk1:                            # stack uses the initial kernel stack space directly.

# The programs of tasks 0 and 1, which cyclically display chars 'A' and 'B', respectively.
task0:
        movl $0x17, %eax             # DS point to the local data segment of the task.
        movw %ax, %ds                # No local data, these 2 instructions can be omitted.
        movl $65, %al                # Put 'A' into the AL register.
        int $0x80                    # Execute system call to display it.
        movl $0xfff, %ecx            # Execute a loop, act as a delay.
1:      loop 1b
        jmp task0                    # Jump to start of task 0 to continue displaying.
task1:
        movl $66, %al                # Put 'B' into the AL register.
        int $0x80                    # Execute system call to display it.
        movl $0xfff, %ecx            # Execute a loop, act as a delay.
1:      loop 1b
```

```
        jmp task1

        .fill 128,4,0               # This is user stack space for task 1.
usr_stk1:
```

4.10 Summary

This chapter describes the protection mode memory management and programming principles of the Intel 80X86 CPU. It describes in detail the specific meanings of the global and local descriptor tables, segment descriptors, and segment selectors. It also gives the transformation relationships between program logical address, CPU linear address, and physical memory address. Finally, a simple kernel sample program is given and introduced at the end of this chapter. Based on an understanding of the sample program, we can roughly explain how well we master the protection mode programming.

Below we use a whole chapter to provide a comprehensive overview of the hardware settings, memory allocation and usage of the Linux kernel and the function of the task data structure, and then classify all the source code in the kernel source tree, let the reader first Have a general understanding of the entire kernel code file structure. Then, in the following chapters, the source code files in the kernel are described and annotated in detail.

5 Linux kernel architecture

This chapter can be seen as a general overview of the kernel source code and can be used as a reference for reading subsequent chapters. For content that is difficult to understand, you can skip it first. When you read the related content in the following chapters, return to refer to this chapter. Please review or learn about how the 80X86 protected mode mode of operation works before reading this chapter.

This chapter begins with an overview of the Linux kernel's composition modes and architecture, and then details the source file organization in the kernel source directory, as well as the main functions of the various code files in the subdirectories and the hierarchical relationships of the basic calls. Then directly cut into the topic, starting from the first file Makefile in the kernel source file Linux / directory to explain each line of code in detail. We will briefly describe the basic architecture and main components based on Linux 0.12 kernel source code. It also explains several important data structures that appear in the source code. Finally, the method of building the Linux 0.12 kernel compilation experimental environment is described.

From a layered perspective, a complete operating system can be composed of four parts: hardware, operating system kernel, operating system services, and user applications, as shown in Figure 5-1. User applications are those word processors, Internet browser programs, or various applications compiled by users themselves; Operating system services are those that provide services to users and are considered part of the operating system. In the Linux operating system, these programs include X window system, shell command interpretation system and system programs such as kernel programming interface; The operating system kernel is the part of this book that is of interest. It is mainly used to abstract hardware resources and schedule management of all system resources.

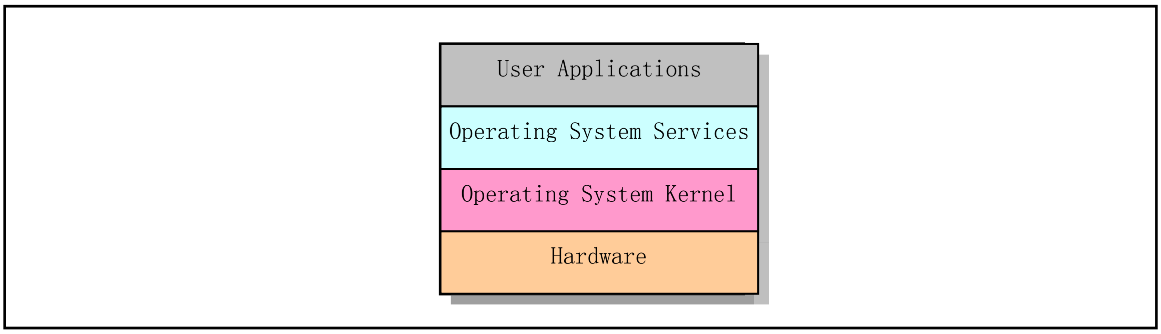


Figure 5-1 Operating system components

The main purpose of the Linux kernel is to interact with computer hardware, implement interface operations and programmatic control of components, schedule access to hardware resources, and provide an easy-to-use execution environment and a common hardware virtual interface for user programs on the computer. .

5.1 Linux kernel mode

At present, the structural mode of the operating system kernel can be mainly divided into a monolithic single-core model and a hierarchical micro-kernel model, and a mixed mode of the two. The Linux 0.12 kernel annotated in this book uses a single-core mode.

In a monolithic single-core system model, the service process provided by the operating system is: the application program executes the system call instruction (int x80) with the specified parameters, so that the CPU switches from the user mode to the core state (Kernel Model).), then the operating system calls the specific system call service procedure according to the specific parameters, and these service procedures call some of the underlying support functions to complete the specific functions as needed. After completing the services required by the application, the operating system switches the CPU back from the kernel mode to the user mode, and returns to the application to continue executing the following instructions. So in summary, the single-core mode kernel can also be roughly divided into three levels: the main program layer that calls the service, the service layer that executes the system call, and the underlying functions that support the system call. See Figure 5-2. The main advantage of the monolithic model is that the kernel code is compact and fast, and the disadvantages are mainly that the hierarchy is not strong.

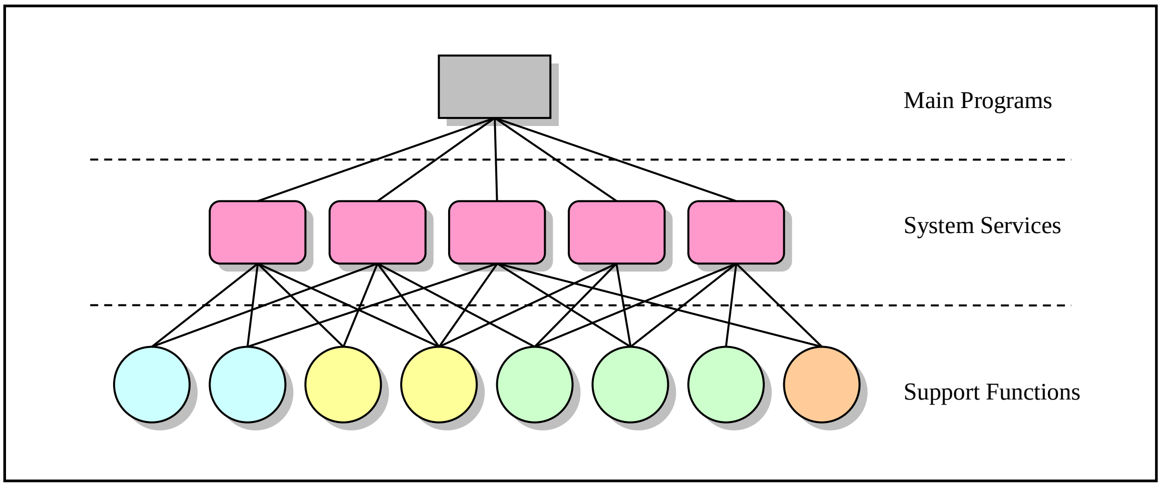


Figure 5-2 Simple structural of monolithic model

For the microkernel architecture model, its main features are function modularization and messaging between service threads or processes. The system core provides a basic hardware abstraction management layer and key system service functions. These key functions are main process/inter-thread communication services, virtual memory management, and process scheduling. The rest of the operating system functions in user space in a variety of modular forms. Therefore, the advantage of the microkernel structure is that the system service coupling is low, which facilitates system improvement, expansion, and porting. The main disadvantage is that a large amount of message passing and synchronization operations between system service modules are required during the running, and these operations cause communication resource consumption and time delay. Typical microkernel architecture systems are the MINIX operating system and the Mac OS system with the Mach kernel.

5.2 Linux kernel system architecture

The Linux kernel consists of five modules: process scheduling module, memory management module, file system module, interprocess communication module, and network interface module, as shown in Figure 5-3.

The process scheduling module is responsible for controlling the use of CPU resources by the process. The scheduling strategy adopted is that each process can access the CPU fairly and reasonably, while ensuring that the kernel can perform hardware operations in a timely manner. The memory management module is used to ensure that all processes can safely share the main memory area of the machine. At the same time, the memory management module also supports the virtual memory management mode, so that the Linux support process uses more memory capacity than the actual memory space. You can use the file system to swap unused memory blocks to an external storage device and exchange them when needed. File system modules are used to support the drive and storage of external devices. The Virtual File System module hides the different details of various hardware devices by providing a common file interface to all external storage devices. This provides and supports multiple file system formats that are compatible with other operating systems. The interprocess communication module is used to support the exchange of information between multiple processes. Network interface modules provide access to a variety of network communication standards and support many network hardware.

The dependencies between these modules are shown in Figure 5-3. The connections represent the dependencies between them, and the dashed and dashed boxes represent the unimplemented parts of Linux 0.12 (the virtual file system is gradually implemented from the Linux 0.95 version, while the network interface support is only available in version 0.96 or later) .

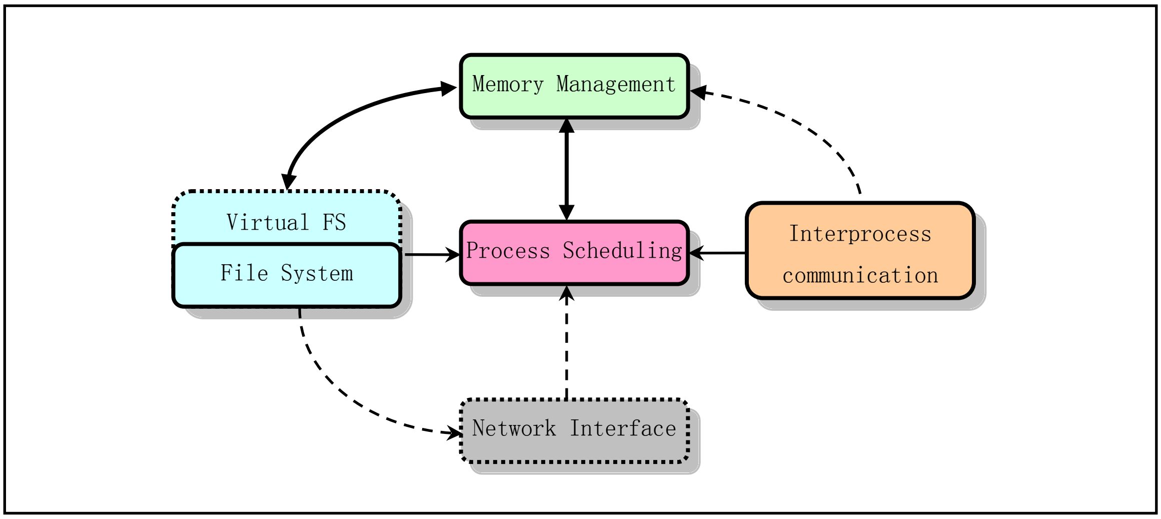


Figure 5-3 Linux kernel module structure and interdependence

As can be seen from the figure, all modules have dependencies on the process scheduling module. Because they all rely on the process scheduler to suspend (pause) or re-run their processes. Typically, a module will be suspended while waiting for hardware operations, and will continue to run until the operation is complete. For example, when a process attempts to write a block of data to a floppy disk, the floppy disk driver may place the process in a suspend wait state during the boot floppy disk rotation, and cause the process to continue after the floppy disk enters the normal rpm. run. The other three modules are also dependent on the process scheduling module for similar reasons.

The dependencies of several other modules are somewhat less obvious, but they are also important. The process scheduling subsystem needs to use memory management to adjust the physical memory space used by a particular process. The interprocess communication subsystem relies on a memory manager to support shared memory communication mechanisms. This communication mechanism allows two processes to access the same area of memory for the exchange of information between processes. The virtual file system also uses the network interface to support the Network File System (NFS), as well as the memory management subsystem to provide memory ramdisk devices. The memory management subsystem also uses the file system to support the swapping of memory blocks.

From the monolithic structure model, we can also draw the kernel main modules into the block diagram structure shown in Figure 5-4 according to the structure of the Linux 0.12 kernel source code.

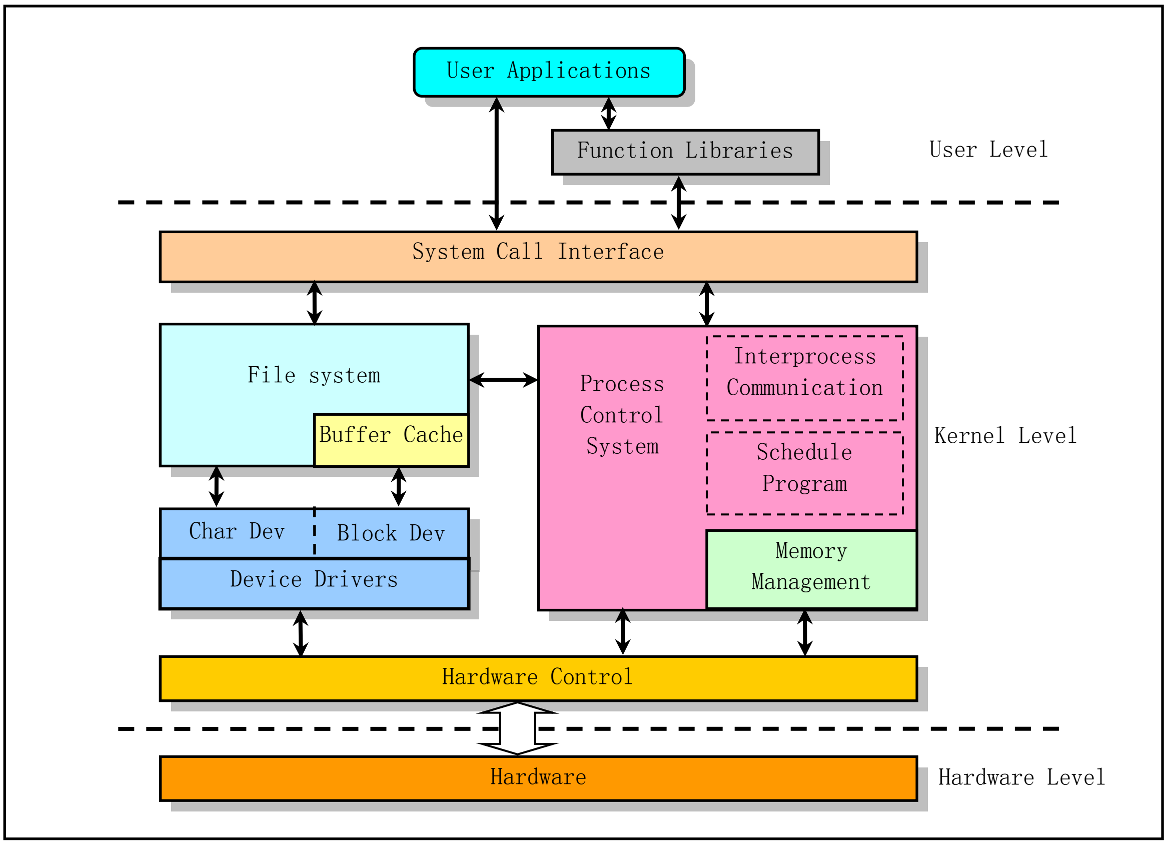


Figure 5-4 Kernel block diagram

In several boxes in the kernel level, the thick line boxes other than the hardware control block correspond to the directory organization of the kernel source code. In addition to the dependencies already given in these diagrams, all of these modules also rely on common resources in the kernel. These resources include memory allocation and reclaim functions, print warning or error message functions, and some system debugging functions.

5.3 Linux kernel memory management

This section first describes the relatively straightforward physical memory usage in Linux 0.12 systems. Then combined with the application situation in the Linux 0.12 kernel, it outlines the segmentation and paging management mechanism of the memory, as well as the CPU multitasking operation and protection mode. Finally, we will comprehensively explain the correspondence between the kernel code and data in the Linux 0.12 system and the code and data of each task in the virtual, linear and physical address space.

The description in this section can be seen as a summary or review of memory management. See Chapter 4 for a detailed description of memory management in protection mode.

5.3.1 Physical address

In the Linux 0.12 kernel, in order to effectively use the physical memory in the machine, the memory is divided into several functional areas during the system initialization phase, as shown in Figure 5-5.

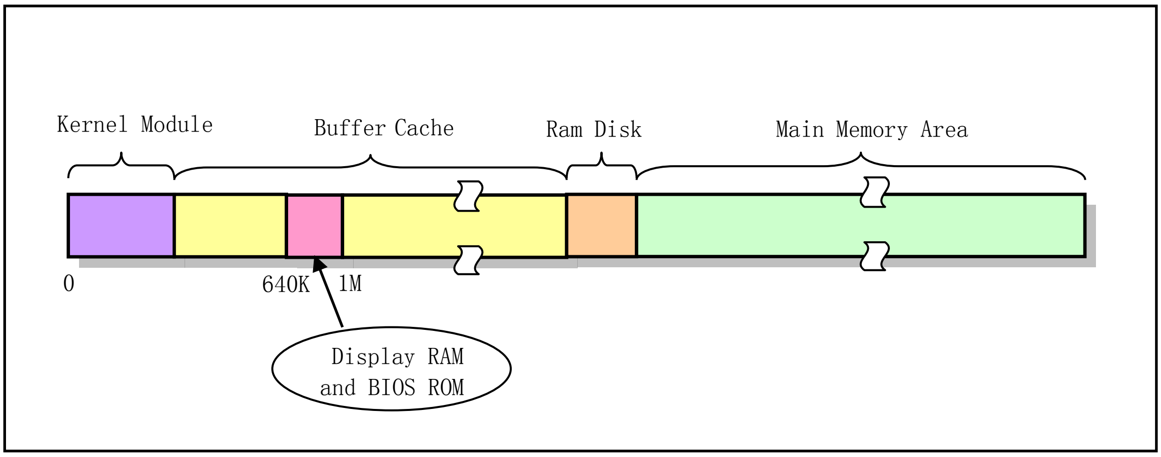


Figure 5-5 Functional distribution of physical memory usage

Among them, the Linux kernel occupies the beginning of the physical memory, followed by the high-speed buffer portion for the block device such as the hard disk or floppy disk (in which the memory address range of the display card memory and the ROM BIOS is 640K--1MB). When a process or task needs to read data from a block device, the system first reads the data into the cache. When there is data to be written to the block device, the system first puts the data into the cache, and then the block device driver writes to the corresponding device. The last part of the memory is the main memory area that all programs can request and use at any time. When the kernel program uses the main memory area, it also needs to apply to the kernel memory management module first, and can use the memory after the application is successful. For systems with RAM virtual disks, the main memory area header is also partially removed for the virtual disk to store data.

Because the actual physical memory capacity contained in the computer system is limited, the CPU usually provides a memory management mechanism to effectively manage the memory in the system. In the Intel 80386 and later CPUs, two memory management (address translation) systems are provided: the Memory Segmentation and the Paging System. The paging management system is optional and is programmed by the system programmer to determine whether to adopt. In order to use physical memory effectively, the Linux system uses both memory segmentation and paging management mechanisms.

5.3.2 Memory Address Space Concept

In the Linux 0.12 kernel, when performing address mapping, we need to first distinguish between three types of addresses and the concept of transformation between them: a). virtual and logical addresses of programs (processes); b). linear addresses of CPUs; c). actual physics Memory address.

Virtual Address refers to the address generated by the program consisting of two parts: the segment selector and the offset address within the segment. Because the two parts of the address are not directly used to access the physical memory, but need to be processed or mapped by the segmentation address translation mechanism to correspond to the physical memory address, such an address is called a virtual address. The virtual address space consists of the global address space mapped by the GDT and the local address space mapped by the LDT. The index portion of the selector is represented by 13 bits, plus one bit that distinguishes between GDT and LDT, so the Intel 80X86 CPU can index a total of 16384 selectors. If the length of each segment takes a maximum of 4G, the maximum virtual address space is 16384 * 4G = 64T.

Logical Address is the portion of the offset address associated with the sections generated by the program. In Intel protected mode, it refers to the offset address within the program execution code section limit length (assuming the code section and data section are identical). Application programmers only need to deal with logical addresses, and the segmentation and paging mechanism is completely transparent to him, only by system programmers. However, some materials do not distinguish between the concept of logical addresses and virtual addresses, but collectively refer to them as logical addresses.

The Linear Address is the middle layer between the virtual address and physical address translation and is the address in the processor's addressable memory space (called the linear address space). The program code will generate a logical address, or an offset address in the segment, plus a base address for the corresponding segment to generate a linear address. If the paging mechanism is enabled, the linear address can be transformed to produce a physical address. If the paging mechanism is not enabled, the linear address is directly the physical address. The Intel 80386 has a linear address space of 4G.

The Physical Address is the address signal indicating the addressed physical memory on the CPU external address bus, which is the final result address of the address translation. If the paging mechanism is enabled, the linear address is transformed into a physical address using the items in the page directory and page table. If the paging mechanism is not enabled, the linear address becomes the physical address directly.

Virtual storage (or virtual memory) is the amount of memory that a computer presents to be much larger than the actual amount of memory it has. It therefore allows the programmer to program and run programs that are much larger than the actual system has. This allows many large projects to be implemented on systems with limited memory resources. A very appropriate analogy is that you don't need a long track to get a train from Shanghai to Beijing. You only need a long enough rail (say 10 km) to complete this task. The method is to immediately lay the rear rails in front of the train. As long as your operation is fast enough and can meet the requirements, the train can run like a complete track. This is the task that virtual memory management needs to accomplish. In the Linux 0.12 kernel, each program (process) is divided into a virtual memory space with a total capacity of 64MB. Therefore, the program's logical address range is 0x0000000 to 0x4000000.

As mentioned above, sometimes we also refer to logical addresses as virtual addresses. Because the logical address is similar to the concept of virtual memory space, and is also independent of the actual physical memory capacity.

5.3.3 Memory Segmentation Mechanism

In a memory segmentation system, the logical address of a program is automatically mapped (transformed) into the 4GB (2^32) linear address space of the middle layer by a segmentation mechanism. Each reference to memory by the program is a reference to memory in the memory segment. When a program references a memory address, a corresponding linear address is formed by adding the corresponding segment base address to the logical address visible to the programmer. If the paging mechanism is not enabled at this time, the linear address is sent to the CPU's external address bus for direct addressing of the corresponding physical memory. See Figure 5-6.

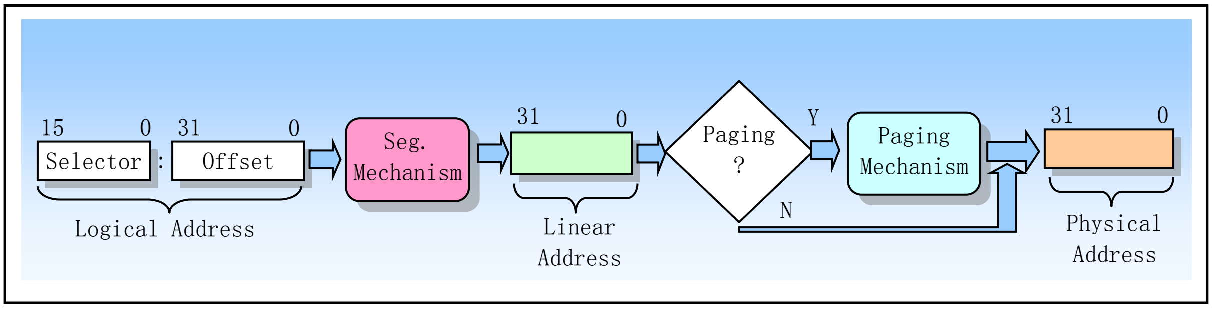


Figure 5-6 Logical address to physical address translation

The main purpose of the CPU to perform address translation (mapping) is to solve the mapping problem from virtual memory space to physical memory space. The meaning of virtual memory space refers to a method that uses secondary or external storage space to enable programs to use memory without being limited by the actual amount of physical memory. Usually the virtual memory space is much larger than the actual physical memory.

So how is virtual storage management implemented? The principle is similar to the analogy of the above train operation. First, when a program needs to use a memory that does not exist (that is, the corresponding memory page is not in memory in the memory page table entry), the CPU needs a way to know the situation. This is achieved by the 80386 page fault exception interrupt. When a process references a memory address in a non-existing page, it triggers the CPU to generate a page fault exception interrupt and places the linear address that caused the interrupt into the CR2 control register. Therefore, the process of processing the interrupt can know the exact address of the page exception, so that the page requested by the process can be loaded into the physical memory from the secondary storage space (such as the hard disk). If the physical memory is already occupied at this time, you can use a part of the secondary storage space as a swap buffer (Swapper) to swap the temporarily unused pages in the secondary buffer, and then transfer the requested page to the memory. in. This is the page fault loading mechanism of memory management. It is implemented in the program mm/memory.c in the Linux 0.12 kernel.

The Intel CPU uses the concept of segment to address the program. Each segment defines information such as an area in memory and the priority of access. Assuming that everyone knows the memory addressing principle in real mode, we now use the comparison method to briefly explain the main features of memory addressing under the 32-bit protected mode operating mechanism according to the different addressing modes of the CPU in real mode and protected mode.

In real mode, addressing a memory address primarily uses segment and offset values, segment values are stored in segment registers (such as DS), and the length of the segment is fixed at 64 KB. The intra-segment offset address is stored in any register that can be used for addressing (eg, SI). Therefore, based on the values in the segment register and the offset register, the actual pointed memory address can be calculated, as shown in Figure 5-7 (a).

In the protected mode mode, the segment register is no longer the base address of the addressed segment, but the index value of a descriptor entry in the segment descriptor table. The segment descriptor item specified by the index value contains information such as the base address of the memory segment to be addressed, the limit length of the segment, and the access privilege level of the segment. The addressed memory location is a combination of the segment base address specified in the segment descriptor entry and an intra-segment offset. The limit length of the segment is variable and is specified by the content in the descriptor. It can be seen that, compared with the addressing in the real mode, the segment register value is replaced with the index value of the corresponding segment descriptor in the segment descriptor table, and the segment table selection bit and the privilege level, which is called a Segment Selector, but The offset value still uses the concept in the original mode. Thus, addressing a memory address in protected mode requires one more procedure than in real mode, which requires the use of a segment descriptor table. This is because there is more information to access a memory segment in protected mode, and a 16-bit segment register can't hold much of this content. The schematic is shown in Figure 5-7 (b). Note that if you do not define a memory linear address space area in a segment descriptor, the address area will not be addressed at all and the CPU will deny access to the address area.

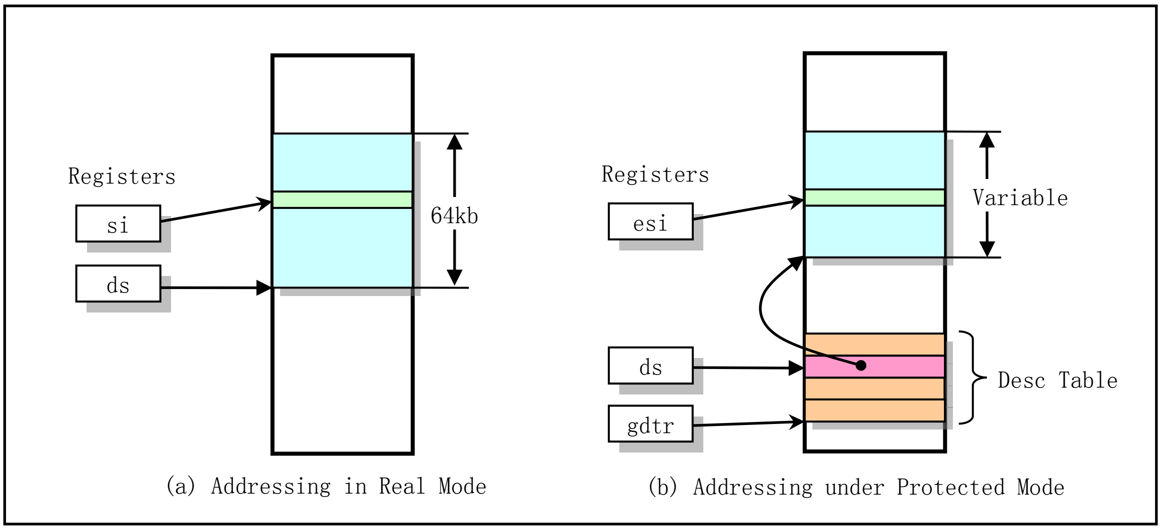

Registers
si
ds
64kb
(a) Addressing in Real Mode
Registers
esi
Variable
ds
gdtr
Desc Table
(b) Addressing under Protected Mode

Figure 5-7 Comparison of addressing in real mode and protected mode

Each descriptor occupies 8 bytes, containing the starting address (base address) of the described segment in the linear address space, the limit length of the segment, the type of the segment (such as code segments and data segments), the privilege level of the segment, and Some other information. The maximum length a segment can define is 4GB.

There are three types of descriptor tables for saving descriptor items, each for a different purpose. The Global Descriptor Table (GDT) is the main base descriptor table that can be used by all programs to reference a memory segment. The Interrupt Descriptor Table (IDT) holds a segment descriptor that defines the interrupt or exception handling process. The IDT table directly replaces the interrupt vector table in the 8086 system. In order to operate normally in 80X86 protected mode, we must define a GDT table and an IDT table for the CPU. The last type of table is the Local Descriptor Table (LDT). This table is used in multitasking systems, usually using one LDT table per task. As an extension to the GDT table, each LDT table provides more available descriptor entries for the corresponding task, thus providing a range of addressable memory spaces for each task.

These tables can be saved anywhere in the linear address space. In order to allow the CPU to locate the GDT table, the IDT table, and the current LDT table, three special registers of GDTR, IDTR, and LDTR need to be set for the CPU. The 32-bit linear base address of the corresponding table and the limit-length byte value of the table are stored in these registers. The table length value is the length value of the table -1.

When the CPU is to address a segment, the selector in the 16-bit segment register is used to locate a segment descriptor. In an 80X86 CPU, the value in the segment register is shifted to the right by 3 bits, which is the index value of a descriptor in the descriptor table. A 13-bit index value can locate up to 8192 (0--8191) descriptor entries. The selector bit 2 (TI) is used to specify which table to use. If the bit is 0 then the selector specifies the descriptor in the GDT table, otherwise it is the descriptor in the LDT table.

Each program can consist of several memory segments. The logical address (or virtual address) of the program is used to address the specific address locations in these segments. In Linux 0.12, the translation process of program logical address to linear address uses the global segment descriptor table GDT and the local segment descriptor table LDT. The address space mapped by the GDT is called the global address space, and the address space mapped by the LDT is called the local address space, and the two constitute the space of the virtual address. The specific usage is shown in Figure 5-8.

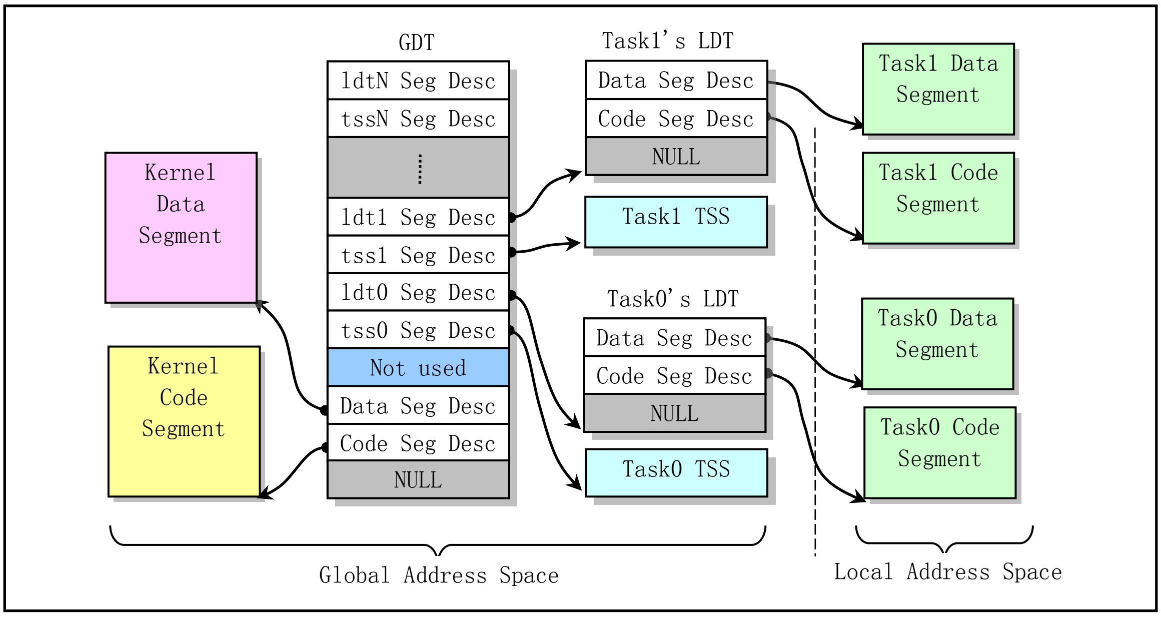


Figure 5-8 Virtual address space allocation map in Linux

The figure shows the situation when there are two tasks. It can be seen that the local descriptor table LDT of each task is also a memory segment defined by the descriptor in the GDT, in which the code segment and the data segment descriptor of the corresponding task are stored, so the LDT segment is very short. The segment length is usually as long as it is larger than 24 bytes. Similarly, the task status segment TSS for each task is also a memory segment defined by the descriptors in the GDT, and the segment length limit is sufficient as long as it satisfies the ability to store a TSS data structure.

对 In the interrupt descriptor table IDT, it is stored in the kernel code segment. In the Linux 0.12 kernel, since the code segment and data segment of the kernel and each task are respectively mapped to the same base address in the linear address space, and the segment length is the same, the code segment and the data segment of the kernel are overlapped. The code segment and data segment of each task are also overlapped respectively, as shown in Figure 5-10 or Figure 5-11. The Task State Segment (TSS) is used to automatically save or restore the current execution context (CPU current state) of the related task when the task is switched. For example, for a task that is switched out, the CPU saves its register and other information in the TSS segment of the task, and the CPU uses the information in the TSS segment newly switched into the task to set each register to restore the execution environment of the new task.

In Linux 0.12, the TSS segment content of each task is saved in the task data structure of the task. In addition, the fourth descriptor in the GDT table (the syscall descriptor entry in the figure) is not used in the Linux 0.12 kernel. From the original English comment on line 201 in the include/linux/sched.h file shown below, it can be guessed that Linus had designed the kernel to place the code for the system call in this specialized section when designing the kernel.

```
/*
 * Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall
 * 4-TSS0, 5-LDT0, 6-TSS1 etc ...
*/
```

5.3.4 Memory paging management

If paging is used, then the linear address is only an intermediate result, and it needs to be transformed by using the paging mechanism, and finally mapped to the actual physical memory address. Similar to the segmentation, paging mechanism allows us to redirect (transform) each memory reference to accommodate our particular requirements. The most common use of the paging is when the system memory is actually divided into a lot of messy blocks, it can create a large and continuous memory space image, so that the program does not have to worry about and manage these scattered memory blocks. The paging mechanism enhances the performance of the segmentation mechanism. In addition, the page address transformation is based on the segment transformation. The protection measures of any paging mechanism will not replace the protection measures of the segment transformation but only carry out further checking operations.

The basic principle of the memory paging is to divide the entire linear memory area of the CPU into 4096 bytes of a page of memory pages. When the program requests to use memory, the system allocates it in units of memory pages. The memory paging is implemented in a similar way to the segmentation mechanism, but not as sophisticated as the segmentation. Because the paging is implemented on top of the segmentation, the result is very flexible control over system memory, and the paging protection mechanism is added to the memory protection of the segmentation mechanism. In order to use the paging in the 80X86 protected mode, the highest bit (bit 31) of the control register CR0 needs to be set.

When using this memory paging method, each executing process (task) can use a contiguous address space that is much larger than the actual memory capacity. In order to map linear addresses to a relatively small physical memory space using the paging, the 80386 uses page directory tables and page tables. Page directory entries are basically the same format as page table entries, occupying 4 bytes, and each page directory table or page table contains only 1024 page table entries. Therefore, a page directory table or a page table occupies a total of 1 page of memory. The small difference between a page directory entry and a page table entry is that the page table entry has a written bit D (Dirty), while the page directory entry does not.

The transfer process from linear address to physical address is shown in Figure 5-9. The control register CR3 in the figure holds the base address of the current page directory table in physical memory (hence CR3 is also referred to as the page directory base address register PDBR). The 32-bit linear address is divided into three parts for locating the corresponding entries in the page directory table and the page table, and specifying the offset position within the page in the corresponding physical memory page. Because a page table can have 1024 entries, a page table can map up to 1024 * 4KB = 4MB memory; and because a page directory table has up to 1024 entries, corresponding to 1024 secondary page tables, a page directory table can be up to Maps 1024 * 4MB = 4GB of memory. That is, a page directory table can map the entire linear address space range.

Since the kernel and all tasks in the Linux 0.1x system share the same page directory table, the mapping function of the processor linear address space to the physical address space is the same at any time. Therefore, in order for the kernel and all tasks to not overlap and interfere with each other, they must be mapped from the virtual address space to different locations of the linear address space, that is, occupying different linear address space ranges.

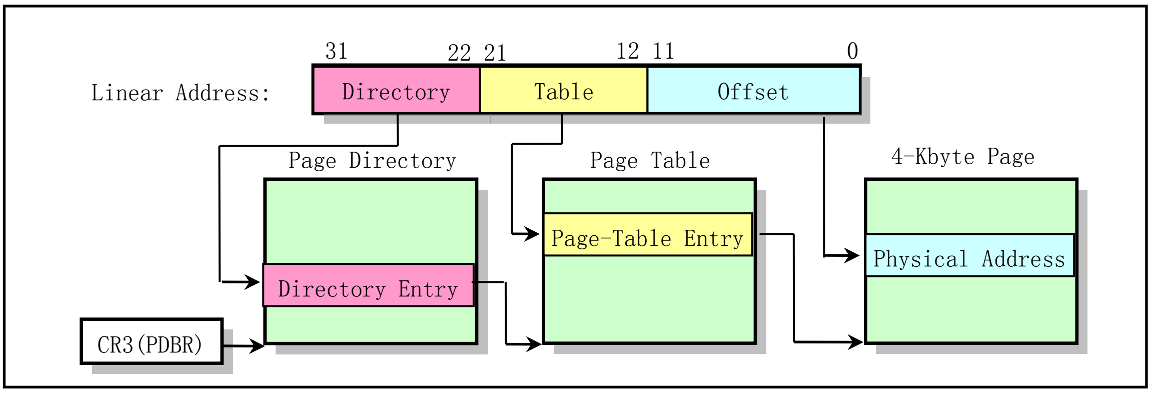


Figure 5-9 Diagram of linear address to physical address transformation

For Intel 80386 systems, the CPU can provide up to 4G of linear address space. The virtual address of a task needs to be first transformed into the address in the entire linear address space of the CPU by its local segment descriptor, and then mapped to the actual using the page directory table PDT (primary page table) and page table PT (secondary page table). On the physical address page. To use actual physical memory, the linear address of each process is dynamically mapped to different physical memory pages in the main memory area by the secondary memory page table.

Since Linux 0.12 defines the maximum available virtual memory space per process as 64MB, the logical address of each process can be converted to an address in linear space by adding (task number) *64MB. However, in the code comments in this book, we sometimes simply refer to such addresses in a process as logical addresses or linear addresses without confusing them.

For Linux 0.12 kernel, the maximum number of segment descriptor entries set in the GDT is 256. Two of them are not used and two are system enrties, and each process or task uses two. Therefore, at this point the system can accommodate up to (256-4)/2 = 126 tasks, and the virtual address range is ((256-4)/2)* 64MB is approximately equal to 8G. However, the maximum number of tasks manually defined in the 0.12 kernel is NR_TASKS = 64, the logical address range of each task is 64M, and the starting position of each task in the linear address space is (task number) *64MB. So the linear address space used by all tasks is 64MB*64 = 4G, as shown in Figure 5-10.

The figure shows the situation when the system has 4 tasks. The kernel code segment and data segment are mapped to the beginning 16MB portion of the linear address space, and both the code and data segments are mapped to the same region, completely overlapping each other. The first task (task 0) is started by the kernel "manually". The code and data are contained in the kernel code and data, so the linear address space used by this task is quite special. The length of the code segment and the data segment of task 0 is a range of 640 KB from the linear address 0, and the code and the data segment also completely overlap, and overlap with the kernel code segment and the data segment. In fact, the instruction space I (Instruction) and the data space D (Data) of all tasks in Linux 0.12 use one piece of memory. That is, all the code, data, and stack parts of a process are in the same memory segment, which is a way of using I&D without separation.

Task 1 has a linear address space starting at address 64MB and only 640KB in length. The detailed correspondence between them will be described later. Task 2 and Task 3 are mapped to the linear addresses 128MB and 192MB, respectively, and their logical address ranges are 64MB. Since the 4G address space range is exactly the linear address space range of the 32-bit CPU, it is also the addressable maximum physical address space range. Moreover, when the logical address range of task 0 and task 1 is regarded as 64 MB, the system may have The logical address range of the task is also 4GB, so it is easier to confuse the three address concepts in the 0.12 kernel.

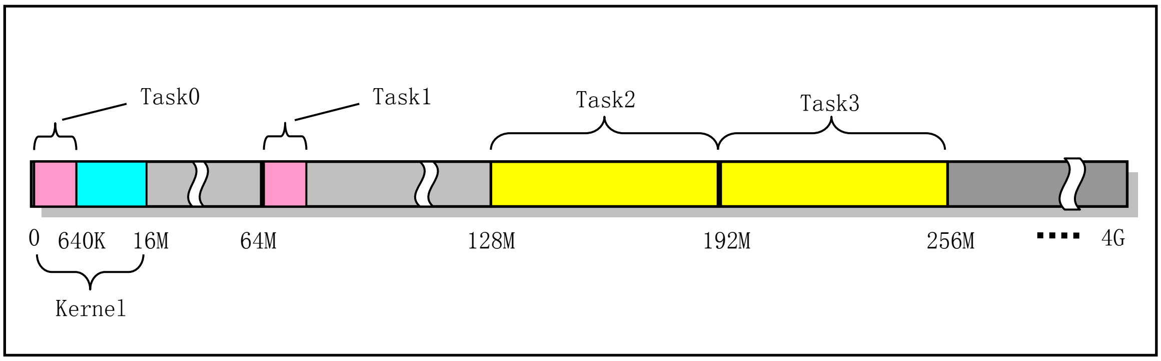


Figure 5-10 Diagram of the use of Linux 0.12 linear address space

If the tasks in the virtual space are also arranged according to the order of the tasks in the linear space, then we can have the system shown in Figure 5-11 and have a schematic diagram of all tasks in the virtual address space, and the virtual space range is also 4 GB. It does not consider the scope of kernel code and data in the virtual space. In addition, in the figure, the positions of the code segments and data segments (including data and stack contents) in the respective logical spaces are also given for task 2 and task 3.

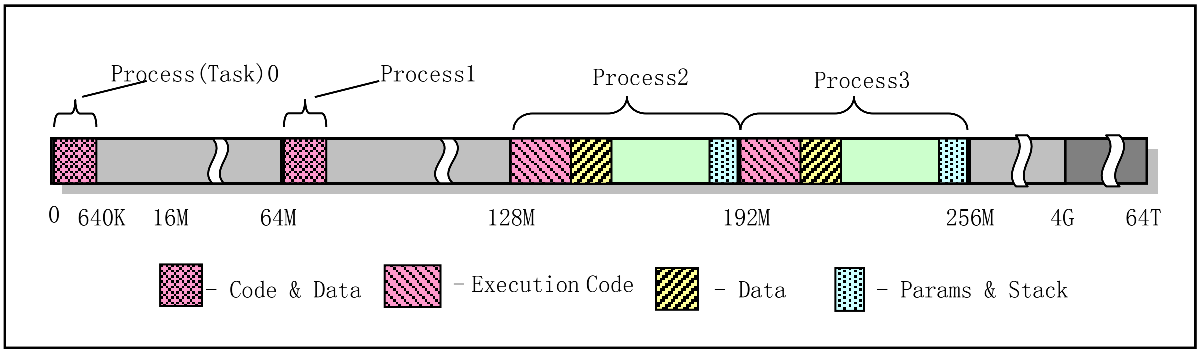


Figure 5-11 The spatial extent of tasks in virtual space in Linux 0.12

It should also be noted that the concept of Code Section and Data Section in the task logical address space is not the same concept as the code segment and data segment in the CPU segmentation mechanism. In the segmentation of the CPU, the concept of a segment determines the purpose of a segment in a linear address space and the constraints and restrictions that are enforced or accessed. Each segment can be placed anywhere in the 4GB linear address space, and they can be independent of each other. They can also overlap completely or partially. The code section and data section of a task refer to the code area, initialization & uninitialization data area, and stack area specified in the process logic space specified by the compiler when compiling the program and when the operating system loads the program. The structure of the code segment and data segment in the task logical address space is shown in Figure 5-12. The nr in the figure is the process or task number, and start_code is the starting location of the process in the linear address space. All other variables contain values in the process logic space.

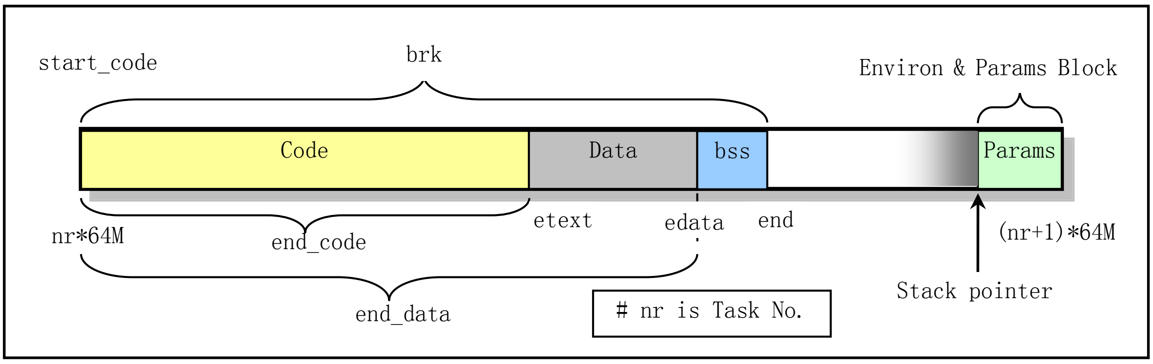


Figure 5-12 The distribution of task code and data in its logical address space

5.3.5 CPU multitasking and protection

The protection mechanism of the 80X86 CPU has four protection levels, with level 0 having the highest priority and level 3 having the lowest priority. The Linux 0.12 operating system uses two protection levels of CPU, 0 and 3. Each task has its own code and data area. These two areas are stored in the local address space, so other tasks in the system are invisible (not accessible). The kernel code and data are shared by all tasks, so it is stored in the global address space. A schematic of this structure is shown in Figure 5-13. The concentric circles in the figure represent the protection level of the CPU (protection layer), and only the 0 and 3 levels of the CPU are used here. Radial rays are used to distinguish between tasks in the system. Each radial ray indicates the boundary of each task. Except for the global address area of each task virtual address space, the address in task 1 is independent of the same address in task 2.

When a task (process) executes a system call and is executed in kernel code, we call the process in kernel operation (or simply kernel mode). At this point the processor is executed in the highest privileged level (level 0). When the process is in kernel mode, the executed kernel code uses the kernel stack of the current process and each process has its own kernel stack. When the process is executing the user's own code, it is said to be in the user's running state (user mode). That is, the processor is now running in the lowest privileged level (level 3) user code.

When the user program is being executed and suddenly interrupted to execute the interrupt handler procedure, the user program can also be symbolically referred to as the kernel state of the process. Because the interrupt handler will use the kernel stack of the current process. This is somewhat similar to the state of a process in kernel mode. The kernel state and user mode of the process are described in more detail later in the section on process running states.

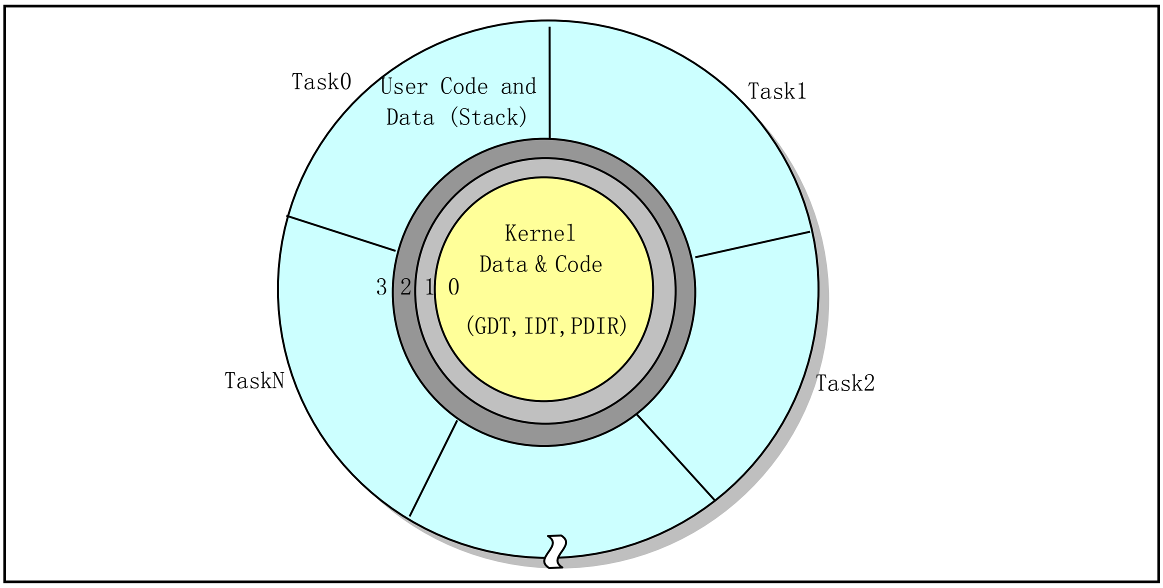


Figure 5-13 Multitasking protection system

5.3.6 Virtual Addresses, Linear Addresses, and Physical Addresses

Earlier we explained the CPU memory management method according to the memory segmentation and paging mechanism. Now let's take the Linux 0.12 system as an example to explain the correspondence between the kernel code and data and the code and data of each task in the virtual address space, linear address space and physical address space. Since the generation or creation process of Task 0 and Task 1 is special, we will describe them separately.

Kernel code and data address

For Linux 0.12, both the kernel code segment and the data segment have been set to a segment of length 16MB in the initialization operation of the head.s program. The range of the two segments overlaps in the linear address space, starting from linear address 0 to address 0xFFFFFF for a total of 16MB address range. This range contains all the kernel code, kernel segment tables (GDT, IDT, TSS), page directory tables and secondary page tables, kernel local data, and kernel temporary stack (which will be used as the user stack for task 0). Its page directory table and secondary page table have been set to map the linear address space of 0--16MB to the physical address one by one, occupying 4 directory entries, that is, 4 secondary page tables. So for kernel code or data addresses, we can think of them directly as addresses in physical memory. At this time, the relationship between the virtual address space, the linear address space, and the physical address space of the kernel can be represented by figure 5-14.

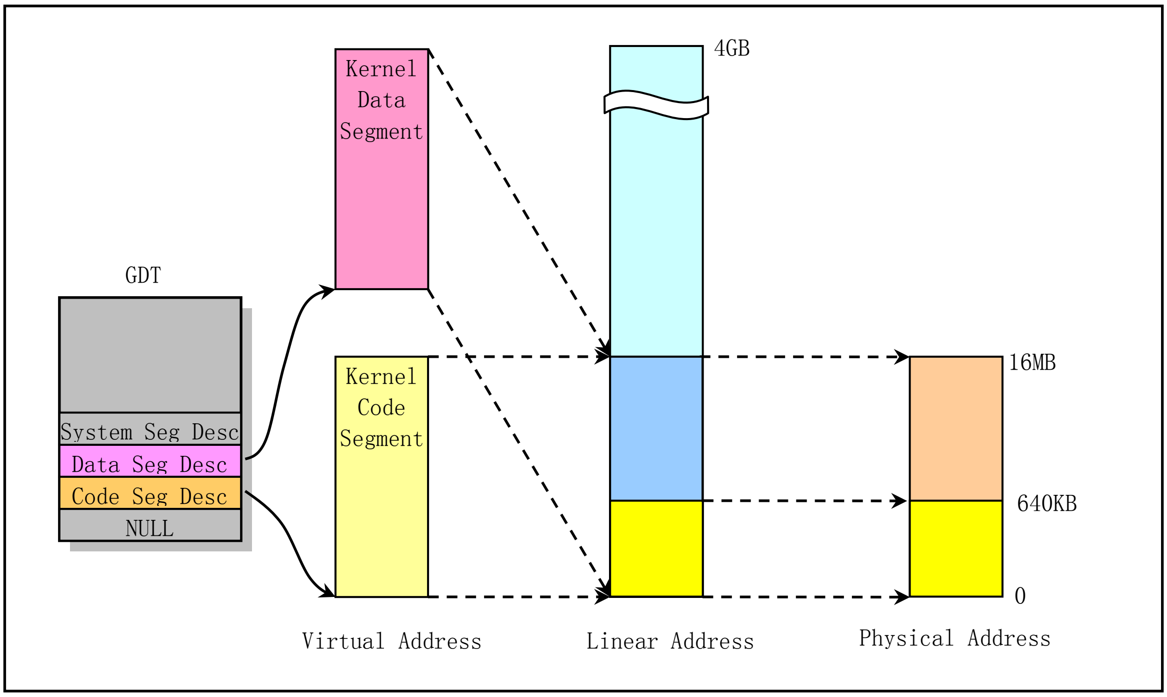


Figure 5-14 Kernel code and data segments in three address spaces

Therefore, by default, the Linux 0.12 kernel can manage up to 16MB of physical memory, with a total of 4096 physical pages (page frames), 4KB per page. Through the above analysis, it can be seen that:

- The kernel code and the data segments region are the same in the linear and physical address space. This setting can greatly simplify the initialization of the kernel.
- GDT and IDT are in the kernel data segment, so their linear addresses are also equal to their physical addresses. In the setup.s program initialization operation in real mode, we have set the temporary GDT and IDT, which must be set before entering the protection mode. Since the two tables were in physical memory at about 0x90200, the kernel system module was in the physical memory 0 start position after entering protected mode, and the space at 0x90200 would be used for other purposes (for caching). So after entering protected mode, we need to reset the two tables in the first program head.s that are running. That is, setting GDTR and IDTR to point to the new GDT and IDT, the descriptor also needs to be reloaded. However, since the position of the two tables does not change when the paging mechanism is turned on, there is no need to re-establish or move the table position.
- Except for task 0, the physical memory pages used by all other tasks are at least partially different from the pages in the linear address, so the kernel needs to dynamically map them in the main memory area to dynamically create page directory entries and page table entries. . Although the code and data of task 1 are also in the kernel, since it needs to be allocated separately to obtain memory, it also needs its own mapping table entry.

Although Linux 0.12 can manage 16MB of physical memory by default, it is not necessary to have such physical memory in the system. As long as there are 4MB (or even 2MB) of physical memory in the machine, you can run Linux 0.12 system. If the machine has only 4MB of physical memory, then the kernel 4MB--16MB address range will be mapped to the non-existing address. But this does not hinder the operation of the system. Because the kernel memory manager knows the exact amount of physical memory in the machine at initialization time, it does not let the CPU paging mechanism map the linear address page to the 4MB--16MB that does not exist. The default setting in the kernel is mainly to facilitate the expansion of the system's physical memory, and actually does not use the physical memory area that does not exist. For the system has more than 16MB of physical memory, because the initialization of the init/main.c program limits the use of more than 16MB of memory, and here the kernel only maps the memory range of 0--16MB. Therefore, physical memory above 16MB will not be used.

Of course, we can extend this limitation by adding some page tables to the kernel here and making minor changes to the init/main.c program. For example, if there are 32MB of physical memory in the system, we need to create 8 secondary page table entries for the kernel code and data segments to map the 32MB linear address range to physical memory.

The address correspondence of task 0

Task 0 is the first task manually initiated in the system. Its code and data segment length are set to 640KB. The code and data for this task are included directly in the kernel code and data, and are 640KB of content starting from linear address 0. Therefore, it can directly use the page directory and page table that the kernel has set to perform paging address translation. Similarly, its code and data segments overlap in the linear address space. The corresponding task status segment TSS0 is also manually pre-set and located in the task 0 data structure information. See the data starting with line 156 in the include/linux/sched.h. The TSS0 segment is located in the code of the kernel sched.c file and has a length of 104 bytes. For details, see the "Task 0 Structure Information" item in Figure 5-24. The mapping correspondence in the three address spaces is shown in Figure 5-15.

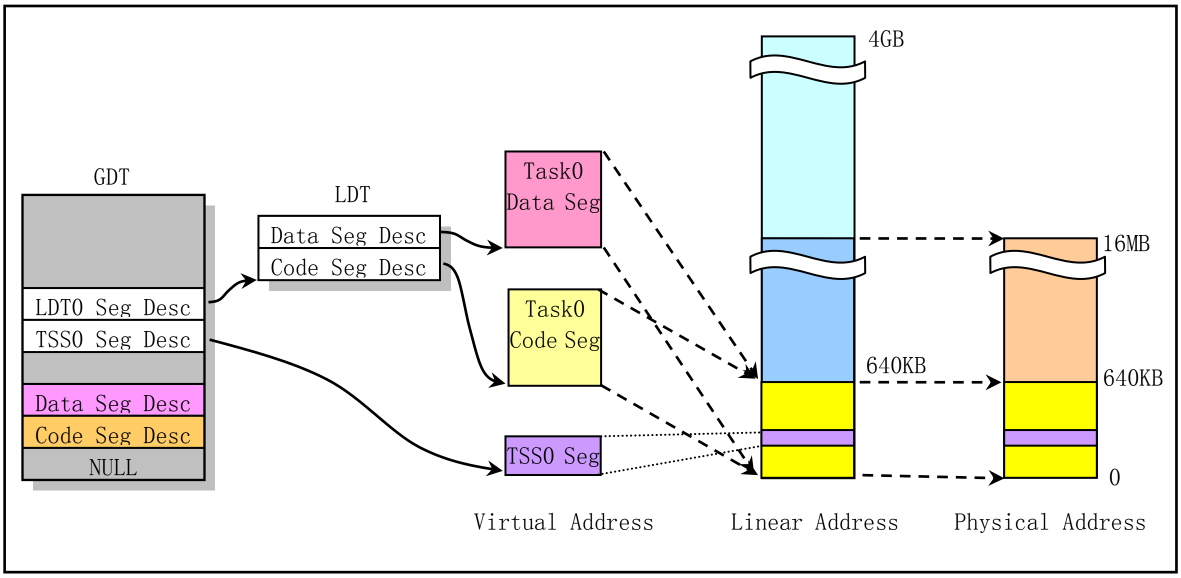


Figure 5-15 The relationship of task 0 in three address spaces

Since task 0 is directly included in the kernel code, there is no need to allocate additional memory pages for it. The kernel-mode stack and user-mode stack space required for its operation are also in the kernel code area, and since the kernel page initialization (head.s), the properties of these kernel pages in the page table entry have been set to 0b111. That is, the corresponding page user can read and write and exist. Therefore, although the user stack user_stack[] space is in kernel space, task 0 can still read and write to it.

Task 1 address correspondence

Similar to task 0, task 1 is also a special task. Its code is also in the kernel code area. Different from task 0, in the linear address space, when the task (init process) is created using fork(), the system stores a page of memory in the main memory area for storing the secondary page table of task 1. The page directory and the secondary page table entry of the parent process (task 0) are copied. Therefore, task 1 has its own page directory and page table entry, which maps the linear space range of task 1 from 64MB to 128MB (actually 64MB to 64MB + 640KB) to the physical address 0--640KB. At this time, the length of task 1 is also 640KB, and its code segment and data segment overlap, occupying only one page directory entry and one secondary page table. In addition, the system will also request a page of memory for task 1 to store its task data structure and kernel stack space. The task data structure (also called process control block PCB) information includes the TSS segment structure information of task 1. See Figure 5-16.

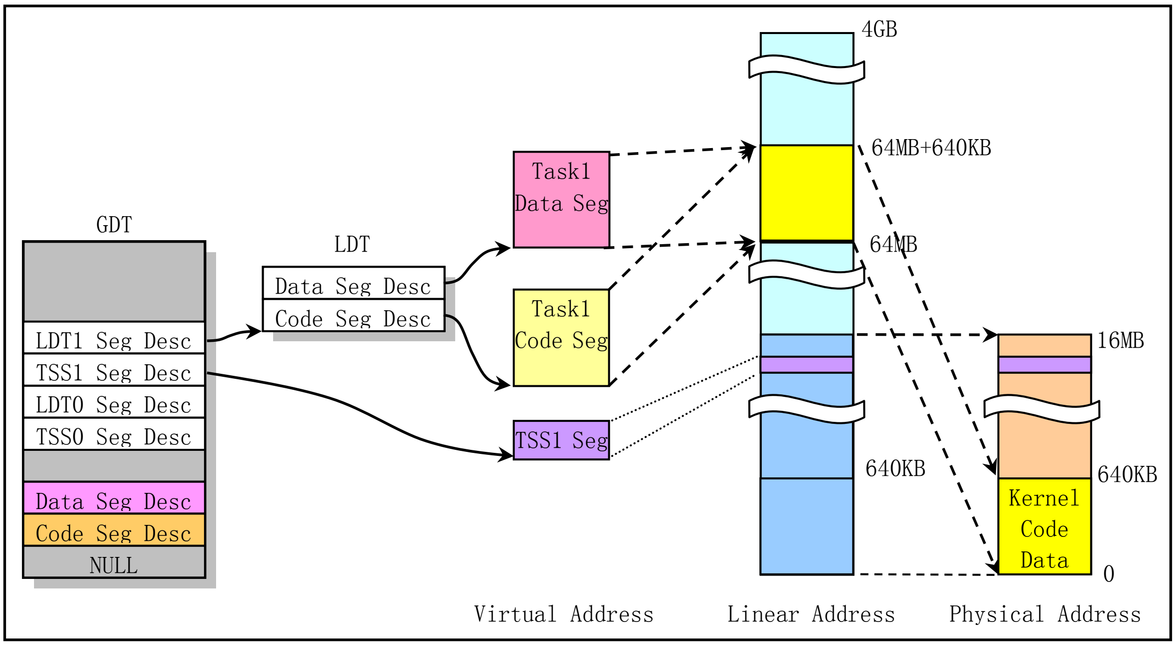


Figure 5-16 Task 1 relationship in three address spaces

The user-mode stack space of task 1 will directly share the user-mode stack space user_stack[] of task 0 in kernel code and data region (linear address 0--640KB) (see kernel/sched.c, lines 82--87)). Therefore this stack needs to be "clean" until Task 1 is actually used to ensure that the stack being copied for Task 1 does not contain useless data. At the beginning of the creation of Task 1, the user-mode stack user_stack[] of Task 0 is shared with Task 1, but when Task 1 starts running, the page table entry mapped to user_stack[] is set to read-only. This causes Task 1 to cause a write page exception when performing a stack operation, so that the kernel allocates the main memory area page as the user stack space.

Address correspondence of other tasks

For the other tasks that were created starting from Task 2, their final parent processes are all init (task 1) processes. We already know that there are 64 processes in the Linux 0.12 system. Below we use task 2 as an example to illustrate the use of address space by any other task.

Starting from task 2, if the task number is represented by nr, the starting position of task nr in the linear address space will be set at nr*64MB. For example, the starting position of task 2 = nr * 64MB = 2 * 64MB =

128MB. The maximum length of the task code segment and data segment is set to 64MB, so Task 2 occupies a linear address space ranging from 128MB to 192MB, occupying a total of 64MB/4MB = 16 page directory entries. The task code segments and data segments in the virtual space are mapped to the same range of linear address spaces, so they also completely overlap. Figure 5-17 shows the correspondence between the code segment of the task 2 and the data segment in the three address spaces.

After task 2 is created, the execve() function will be run in it to execute the shell program. When the kernel just created task 2 through replication of task 1, in addition to occupying a linear address space range (128MB--128MB+640KB), the relationship between the code and data of task 2 in the three address spaces is similar to that of task 1. When the code of task 2 (init()) calls the execve() system call to start loading and executing the shell program, the system call releases the page directory and page table entries and corresponding memory pages copied from task 1. Then re-create the relevant page directory and page table entries for the new executor shell. Figure 5-17 shows the situation when task 2 starts executing the shell program, that is, the case where the code and data of task 2 were originally copied by the code segment and data segment of the shell program. The figure shows a situation where one page of physical memory pages has been mapped. Note here that when executing the execve() function, although the system allocates a 64MB space range for Task 2 in the linear address space, the kernel does not immediately allocate and map physical memory pages for it. The memory manager allocates and maps a page of physical memory into its linear address space in the main memory area only when an exception occurs due to a lack page fault when task 2 begins execution. This method of allocating and mapping physical memory pages is called load on demand. See the related description in the Memory Management chapter.

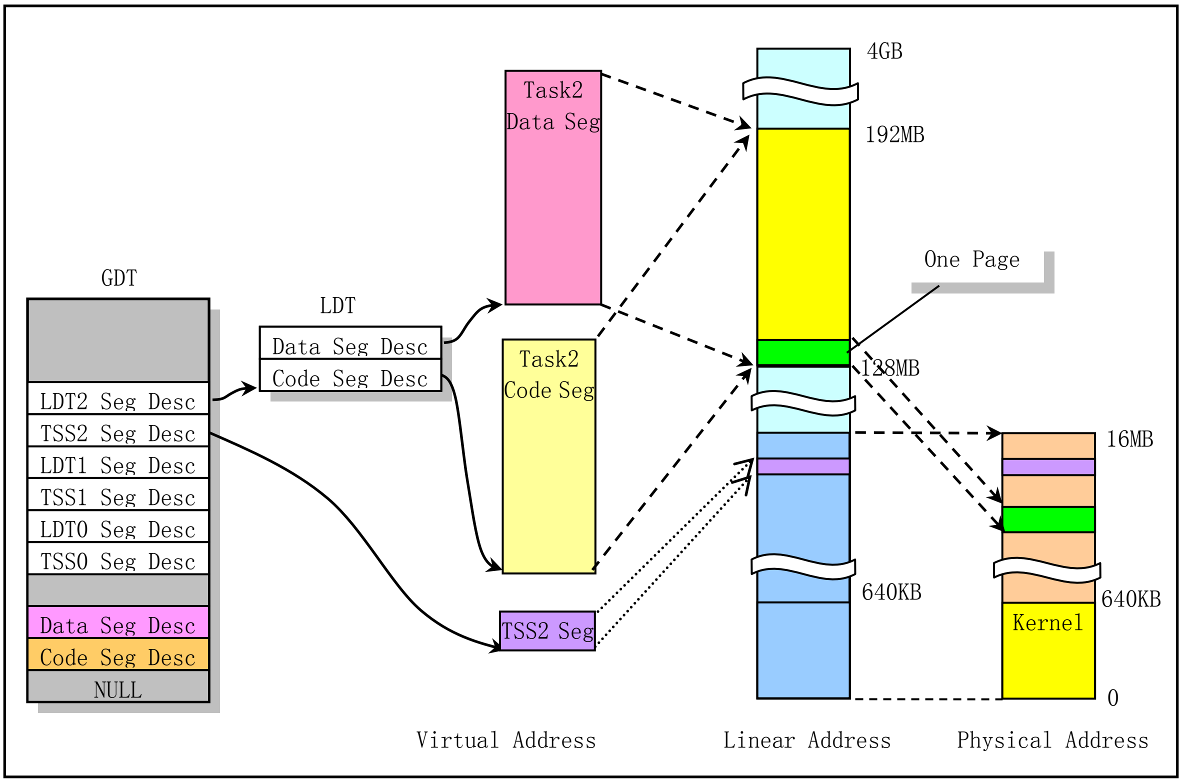


Figure 5-17 Correspondence in other task address spaces

Since the Linux kernel version 0.99, the use of memory space has changed. By using the page directory table independently, each process can enjoy the entire 4G address space range. If we can understand the memory management concept described in this section, then the memory management principles used in the Linux 2.x kernel that is currently used can be immediately understood. Due to space limitations, this will not be explained here.

5.3.7 User application for memory dynamic allocation

When the user program uses the memory allocation function malloc() in the C library to apply for memory, the memory capacity or size of these dynamic applications is managed by the high-level C library function malloc(), and the kernel itself does not intervene. Because the kernel has allocated 64MB of space in the 4G linear address space of the CPU for each process (except tasks 0 and 1, which are resident in memory with the kernel code). Therefore, as long as the range of the task or process execution is within the 64MB range, the kernel will also automatically allocate the physical memory page and map the operation for the corresponding page through the memory page fault management mechanism.

But the kernel maintains a current position variable brk for the code and data space used by the process. This variable value is stored in the data structure of each process. It indicates the end position of the process code and data (including the dynamically allocated data space) in the process address space. When the malloc() function allocates memory for the program, it notifies the kernel of the length of the space requested by the program by the system call brk(). The kernel code can update the value of brk based on the information provided by malloc().However, the physical memory page is not mapped for the newly requested space at this time. Only when the program addresses an address that does not have a corresponding physical page, the kernel performs mapping operations on the relevant physical memory page.

If the page where a certain data is addressed by the process code does not exist, and the location of the page belongs to the process heap scope, that is, it does not belong to the memory range corresponding to the executable file image file, the CPU generates a page fault exception. And allocate and map a page of physical memory pages for the specified page in the exception handler. As for the memory size of the application and the specific location in the corresponding physical page, the memory allocation function malloc() in the C library is responsible for management. The kernel allocates and maps physical memory in units of pages. This function specifically records how many bytes of memory is used by the user program. The remaining capacity will be reserved for use when the program re-applies for memory.

When the user program uses the function free() to dynamically release the requested memory block, the memory management function in the C library marks the released memory block as free, in case the program requests the memory again. The physical page allocated by the kernel for this process will not be released during this process. Only when the process ends up will the kernel fully reclaim all physical memory pages that have been allocated and mapped to the process's address space.

The specific code implementation for the library functions malloc() and free() can be found in the lib/malloc.c program in the kernel library.

5.4 Interrupt mechanism

This section describes the basic principles of the interrupt mechanism and related programmable controller hardware logic, as well as methods for using interrupts in Linux systems. For the specific programming method of the programmable controller, please refer to the description after the setup.s program in the next chapter.

5.4.1 Principle of Interrupt Operation

Microcomputer systems typically include input and output devices. One way the processor provides services to these devices is to use polling. In this method the processor sequentially queries each device in the system and "queries" if they need service. The advantage of this method is that software programming is simple, but the disadvantage is that it consumes processor resources and affects system performance.

Another way to provide services to a device is to make a request to the processor itself when the device needs service. The processor also serves the device only when requested by the device. When the device makes a service request to the processor, the processor will respond to the device's request as soon as the current instruction is executed, and then execute the relevant service program of the device. When the service program is executed, the processor will continue to do the program that was just interrupted. This type of processing is called an interrupt method, and the service request that the device sends to the processor is called an IRQ - Interrupt Request. The device-related program that the processor executes in response to the request is called an interrupt service routine or an ISR.

The Programmable Interrupt Controller (PIC) is the administrator that manages device interrupt requests in a microcomputer system. It accepts the Terminal Service Request signal from the device through an interrupt request pin connected to the device. When the device activates its interrupt request IRQ signal, the PIC will detect it immediately. In the case of receiving interrupt service requests from several devices at the same time, the PIC will prioritize them and select the highest priority interrupt request for processing. If the processor is currently executing an interrupt service routine for a device, the PIC also needs to compare the selected interrupt request with the priority of the interrupt request being processed and determine whether to issue an interrupt to the processor based on the comparison. When the PIC issues an interrupt to the processor's INT pin (INTR pin in Figure 5-18), the processor immediately stops what was done at that time and asks the PIC which interrupt service request to execute. The PIC informs the processor which interrupt service process to execute by sending an interrupt number corresponding to the interrupt request to the data bus. The processor obtains the interrupt vector of the relevant device (ie, the address of the interrupt service routine) by querying the interrupt vector table (or the interrupt descriptor table IDT in the 32-bit protected mode) according to the read interrupt number and starts executing the interrupt service routine. When the execution of the interrupt service routine ends, the processor continues to execute the program interrupted by the interrupt signal.

What has been described above is the interrupt service processing of the input/output device. But the interrupt method is not necessarily hardware-dependent, it can also be used in software. By using the INT instruction and using its operand to indicate the interrupt number, the processor can be executed to perform the corresponding interrupt processing. The PC/AT series of microcomputers provide support for 256 interrupts, most of which are used for software interrupts or exceptions. The exceptions are interrupts generated by the processor detecting errors during processing. Only some of the interrupts mentioned below are used on the device.

5.4.2 Interrupt subsystem of 80X86 PC

The 8259A programmable interrupt controller chip is used in a microcomputer system composed of 80X86. Each 8259A chip can manage eight interrupt sources. Through multi-chip cascading, the 8259A can form a system that manages up to 64 interrupt vectors. In the PC/AT series compatible PC, two 8259A chips are used to manage 15 levels of interrupt vectors. A schematic diagram of the cascade is shown in Figure 5-18. The INT pin of the slave chip is connected to the IR2 pin of the master chip, that is, the interrupt signal sent by the 8259A slave chip will be the IRQ2 input signal of the 8259A master chip. The port base address of the master 8259A chip is 0x20, and the slave chip is 0xA0. The IRQ9 pin functions the same as the IRQ2 of the PC/XT. That is, the PC/AT machine uses the hardware circuit to redirect the IRQ2 pin of the device using IRQ2 to the IRQ9 pin of the PIC, and uses the software in the BIOS to interrupt the IRQ9. Int 71 redirects to IRQ2 interrupt int 0x0A interrupt handler procedure. This allows any 8-bit card with PC/XT using IRQ2 to function properly under the PC/AT machine. The backward compatibility of the PC series has been achieved.

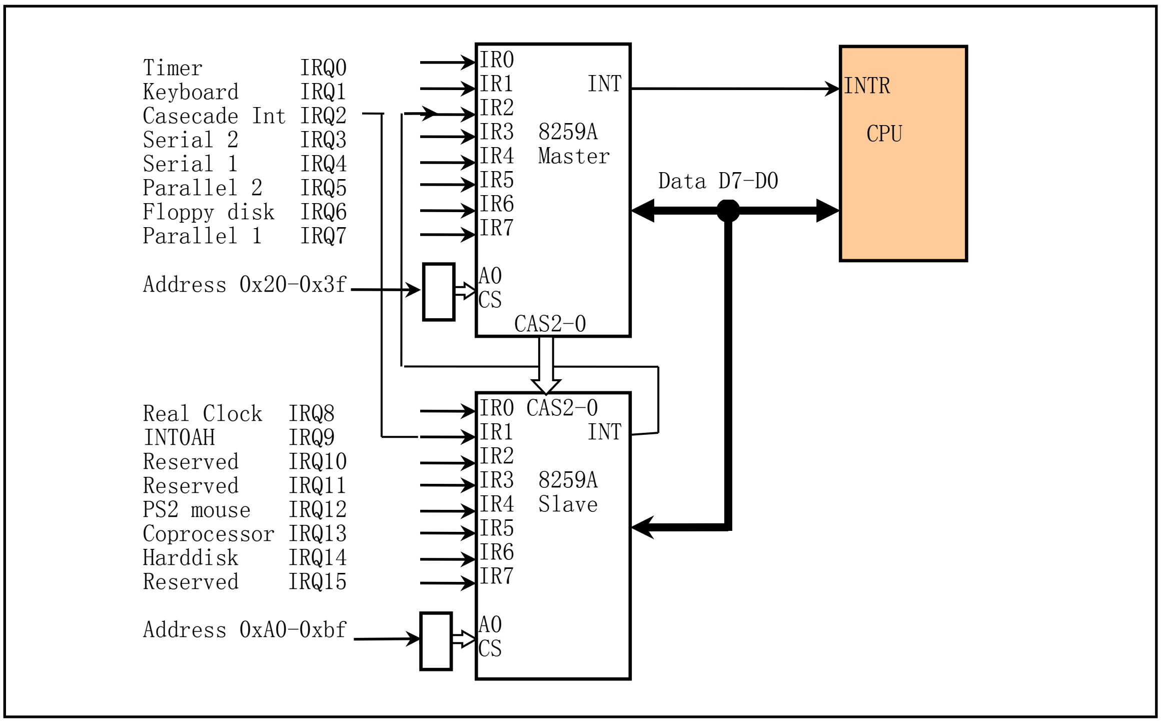


Figure 5-18 PC/AT microcomputer cascaded 8259 control system

Under the control of the bus controller, the 8259A chip can be in the programming state and operating state. The programming state is the state in which the CPU initializes the 8259A chip using the IN or OUT instruction. Once the initialization programming is completed, the chip enters the operating state. At this time, the chip can respond to the interrupt request (IRQ0 – IRQ15) proposed by the external device at any time, and the system can also modify the interrupt processing mode at any time by using the operation command word. Through the interrupt arbitration selection mechanism, the chip will select the current highest priority interrupt request as the interrupt service object, and notify the CPU of the interrupt request by the CPU pin INT. After the CPU responds, the chip sends the programmed interrupt number of the current service object from the data bus D7-D0, and the CPU acquires the corresponding interrupt vector value and executes the interrupt service routine.

5.4.3 Interrupt Vector Table

The previous section has indicated that the CPU fetches the interrupt vector value based on the interrupt number, which corresponds to the entry address value of the interrupt service routine. Therefore, in order for the CPU to find the corresponding interrupt vector from the interrupt number, it is necessary to create a lookup table in the memory, that is, the interrupt vector table (in the 32-bit protection mode, the table is called the interrupt descriptor table IDT, see below).

The 80X86 microcomputer supports 256 interrupts, and an interrupt service routine is required for each interrupt. In the 80X86 real mode mode, each interrupt vector consists of 4 bytes. These 4 bytes indicate the segment value and the intra-segment offset value of an interrupt service routine. Therefore the length of the entire vector table is 4 * 256 = 1024 bytes. When the 80X86 microcomputer starts up, the program in the ROM

BIOS initializes and sets the interrupt vector table at the physical memory start address 0x0000:0x0000, and the default interrupt service routine for each interrupt is given in the BIOS. Since the vectors in the interrupt vector table are arranged in the order of the interrupt numbers, given an interrupt number N, the position of the corresponding interrupt vector in memory is 0x0000: N * 4, that is, the corresponding interrupt service program entry address is stored in Physical memory 0x0000: N * 4 position.

When the BIOS performs the initialization operation, it sets the 16 hardware interrupt vectors supported by the two 8259A chips and the interrupt calling function provided by the BIOS with the interrupt number 0x10-0x1F. For interrupts that are not actually used, the vector is filled with the address of the temporary dummy interrupt service routine. Later, when the system boots the operating system, some interrupt vector values will be modified according to actual needs. For example, for the DOS operating system, it will reset and modify the interrupt vector value of interrupt 0x20-0x2F. For Linux systems, in addition to the display and disk read interrupts provided by the BIOS when you first load the kernel, a new interrupt vector table is created. That is, the 8259A chip is reinitialized in the setup.s program, and an interrupt vector table (interrupt descriptor table) is re-established in the head.s program. Therefore, the Linux completely abandons the BIOS interrupt vector table after the kernel is running normally.

When the Intel CPU is running in 32-bit protected mode, you need to use the Interrupt Descriptor Table (IDT) to manage interrupts or exceptions. IDT is a direct replacement for the interrupt vector table used in the Intel 8086 - 80186 CPU. Its role is similar to the interrupt vector table, except that each interrupt descriptor entry contains information about the privilege level and descriptor class in addition to the address of the interrupt service routine. The Linux operating system works in 80X86 protected mode, so it uses the interrupt descriptor table to set and save the "vector" information for each interrupt.

5.4.4 Linux kernel interrupt handling

For the Linux kernel, interrupt signals are usually divided into two categories: hardware interrupts and software interrupts (or exceptions). Each interrupt is identified by a number between 0-255, called the interrupt number. For interrupts INT0--INT31 (0x00--0x1f), the function of each interrupt is fixed or reserved by Intel Corporation, as shown in Table 5-1. As can be seen from the above section, the range of interrupt numbers set by the BIOS conflicts with it.

Table 5-1 Exceptions and Interrupts reserved by Intel Co.

Vecter No	Name	Type	Error Code	Signal	Source
0	Devide error	Fault (Error)	No	SIGFPE	DIV and IDIV instructions.
1	Debug	Fault/ Trap	No	SIGTRAP	Any code or data reference or the INT instruction.
2	nmi	Interrupt	No		Non maskable external interrupt.
3	Breakpoint	Trap	No	SIGTRAP	INT 3 instruction.
4	Overflow	Trap	No	SIGSEGV	INTO instruction.
5	Bounds check	Fault	No	SIGSEGV	BOUND instruction.
6	Invalid Opcode	Fault	No	SIGILL	UD2 instruction or reserved opcode.
7	Device not available	Fault	No	SIGSEGV	Floating-point or WAIT/FWAIT instruction.
8	Double fault	Abort	Yes(0)	SIGSEGV	Any instruction that can generate an exception, NMI, or an INTR.

9	Coprocessor seg overrun	Abort	No	SIGFPE	Floating-point instruction.
10	Invalid TSS	Fault	Yes	SIGSEGV	Task switch or TSS access.
11	Segment not present	Fault	Yes	SIGBUS	Loading segment registers or accessing system segments.
12	Stack segment	Fault	Yes	SIGBUS	Stack operations and SS register loads.
13	General protection	Fault	Yes	SIGSEGV	Any memory reference and other protection checks.
14	Page fault	Fault	Yes	SIGSEGV	Any memory reference.
15	Intel reserved		No		
16	Coprocessor error	Fault	No	SIGFPE	Floating-point or WAIT/FWAIT
17	Alignment check	Fault	Yes(0)		Any data reference in memory.
20-31	Intel reserved.				
32-255	User Defined interrupts	Interrupt			External interrupt or INT n instruction.

These interrupts are soft interrupts, but Intel calls them exceptions. Because these interrupts are caused by an abnormal condition detected when the CPU executes the instruction. It can usually be divided into two categories: faults and traps. The interrupt INT32--INT255 (0x20--0xff) can be set and defined by the user. The classification of all interrupts and the way the CPU operates after execution are shown in Table 5–2.

Table 5-2 Interrupt classification and how the CPU handles it

Interrupt	Name	CPU Check Method	Processing Method
Hardware	Maskable	CPU pin INTR	Clear the IF maskable interrupt flag of EFLAGS.
	Nonmaskable	CPU pin NMI	Non-Maskable Interrupts.
Software	Fault	Detected before error occurred	CPU re-executes the instruction that caused the error.
	Trap	Detected after error occurred	CPU continues to execute the following instruction.
	Abort	Detected after error occurred	Programs that caused this error should be terminated.

In Linux systems, INT32--INT47 (0x20--0x2f) corresponds to the hardware interrupt request signal IRQ0--IRQ15 issued by the 8259A interrupt control chip (see Table 5-3), and set the software interrupt issued by the user program to INT128 (0x80), called the system call (System Call) interrupt. System call interrupt is the only interface for user programs that use operating system resources.

Table 5-3 List of interrupt numbers for Linux system interrupt requests

Interrup Request No.	Interrupt No.	Purpose
IRQ0	0x20 (32)	100HZ clock interrupt signal from 8253 chip
IRQ1	0x21 (33)	Keyboard interrupt
IRQ2	0x22 (34)	Cascade to slave chip
IRQ3	0x23 (35)	Serial port 2
IRQ4	0x24 (36)	Serial port 1
IRQ5	0x25 (37)	Parallel port 2
IRQ6	0x26 (38)	Floppy disk drive
IRQ7	0x27 (39)	Parallel port 1

IRQ8	0x28 (40)	Real clock
IRQ9	0x29 (41)	Reserved
IRQ10	0x2a (42)	Reserved
IRQ11	0x2b (43)	Reserved (Network interface)
IRQ12	0x2c (44)	PS/2 mouse
IRQ13	0x2d (45)	Coprocessor
IRQ14	0x2e (46)	Harddisk
IRQ15	0x2f (47)	Reserved

At system initialization, the kernel first uses a dummy interrupt vector (interrupt descriptor) to default settings for all 256 descriptors in the interrupt descriptor table (IDT). This dummy interrupt vector points to a default "no interrupt" handler procedure. The message "Unknown interrupt" is displayed when an interrupt occurs and the interrupt vector has not been reset. For some interrupts that need to be used in the system, the kernel will re-edit the interrupt descriptor items of these interrupts during the process of continuing initialization, so that they point to the corresponding actual hander procedure. Usually, the exception interrupt processing (INT0 -- INT 31) is reset in the initialization function of traps.c, and the system call interrupt int128 is reset in the scheduler initialization function.

In addition, the Linux kernel uses both the interrupt gate and the trap gate descriptors when setting the interrupt descriptor table IDT. The difference between them is the effect on the interrupt enable flag IF in the flag register EFLAGS. The interrupt executed by the interrupt gate descriptor resets the IF flag, so other interrupts can be prevented from interfering with the current interrupt processing. The subsequent interrupt end instruction IRET will restore the original value of the IF flag from the stack. Interrupts that are executed through the trap gate do not affect the IF flag.

5.4.5 Interrupt Flag of Flag Register

In order to avoid contention and disruption of the critical code area, the CLI and STI instructions are used in many places in the Linux 0.12 kernel code. The CLI instruction is used to reset the interrupt flag IF in the CPU flag register so that the system does not respond to external interrupts after executing the CLI instruction. The STI instruction is used to set the interrupt flag in the flag register to allow the CPU to recognize and respond to interrupts from external devices.

When entering a code area that may cause race conditions, the kernel will use the CLI instruction to turn off the response to the external interrupt, and the kernel will execute the STI instruction to re-allow the CPU to respond to the external interrupt when the content code area is executed. For example, when modifying the lock flag of the file super block and the task entry/exit wait queue operation, it is necessary to first use the CLI instruction to disable the CPU from responding to the external interrupt, and then use the STI instruction to enable the response to the external interrupt after the operation is completed. If you do not use the CLI, STI instruction pair, that is, when you need to modify a file super block without using the CLI to disable the response to the external interrupt, then before the modification, it is judged that the super block lock flag is not set and you want to set this flag. If the system clock interrupt occurs just at this time and switches to another task to run, and it happens that other tasks also need to modify the super block, then this other task will first set the lock flag of the super block and modify the super block. When the system switches back to the original task, at this time, the task will not judge the lock flag and will continue to execute the lock flag of the set super block, thereby causing two tasks to simultaneously perform multiple operations on the critical code area, causing the super block data. Inconsistency can lead to kernel system crashes in severe cases.

5.5 Linux system calls

5.5.1 System Call Interface

System calls (commonly referred to as syscalls) are the only interfaces that the Linux kernel can communicate with upstream applications, as shown in Figure 5-4. From the description of the interrupt mechanism, the user program can use kernel resources, including system hardware resources, by calling interrupt int 0x80 directly or indirectly (through the library function) and specifying the system call function number in the EAX register. However, usually the application uses the kernel's system calls indirectly using functions in the C library with standard interface definitions, as shown in Figure 5-19.

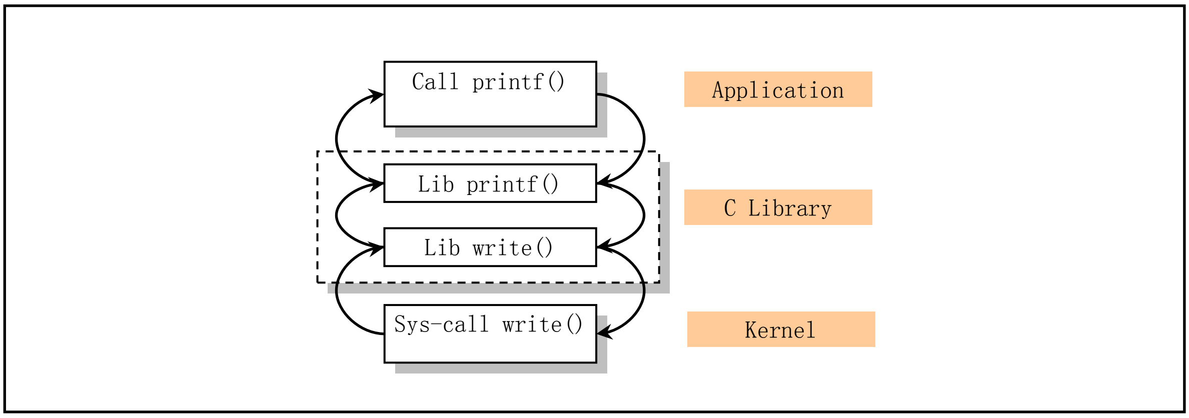


Figure 5-19 Relationship between user programs, library functions, and kernel system calls

Usually system calls are made using a functional form, so they can take one or more parameters. For the result of the system call execution, it will be represented in the return value. Usually a negative value indicates an error and a 0 indicates success. In the case of an error, the wrong type code is stored in the global variable errno. By calling the library function perror(), we can print out the error string information corresponding to the error code.

In the Linux kernel, each system call has a unique system call function number. Kernel 0.12 has a total of 87 system call functions (0-86). These feature numbers are defined at the beginning of line 62 in the file include/unistd.h. For example, the write system call has a function number of 4, defined as the symbol __NR_write. These system call function numbers actually correspond to the index of the items in the system call handler pointer array table sys_call_table[] defined in include/linux/sys.h. So the handler pointer for the write() system call is at item 4 of the array.

When we want to use these system call symbols directly in our own program, we need to define the symbol "__LIBRARY__" before including the file "<unistd.h>" as shown below.

```
#define __LIBRARY__
#include <unistd.h>
```

In addition, we can see from sys_call_table[] that the names of all system call handlers in the kernel basically start with the symbol 'sys_'. For example, the implementation function of the system call read() in the kernel source code is sys_read().

5.5.2 System Call Processing

When the application issues an interrupt INT 0x80 to the kernel via a library function, a system call is initiated. The system call number is stored in the register EAX, and the carried parameters can be stored in the registers EBX, ECX and EDX in turn. Therefore, the user program in the Linux 0.12 kernel can pass up to three parameters directly to the kernel. Of course, you can also take no parameters. The process of handling the system call interrupt INT 0x80 is the system_call in the program kernel/system_call.s.

In order to facilitate the execution of system calls, the kernel source code defines the macro function _syscalln() in the include/unistd.h file (150-200 lines), where n represents the number of parameters carried, which can be 0 to 3 respectively. Therefore, up to 3 parameters can be passed directly. If you need to pass large chunks of data to the kernel, you can pass the pointer of the chunk data. For example, for the read() system call, its definition is:

```
int read(int fd, char *buf, int n);
```

If we execute the corresponding system call directly in the user program, the macro of the system call is:

```
#define __LIBRARY__
#include <unistd.h>

_syscall3(int, read, int, fd, char *, buf, int, n)
```

So we can use the above _syscall3() directly in the user program to execute a system call read() instead of mediating through the C library. In fact, the form of the function call in the C function library is the same as that given here.

For each system call macro given in include/unistd.h, there are 2+2*n parameters. The first parameter corresponds to the type of the system call return value; the second parameter is the name of the system call; followed by the type and name of the parameter carried by the system call. This macro will be extended to a C function that contains inline assembly statements, as shown below.

```
int read(int fd, char *buf, int n)
{
    long __res;
    __asm__ volatile (
        "int $0x80"
        : "=a" (__res)
        : "0" (__NR_read), "b" ((long)(fd)), "c" ((long)(buf)), "d" ((long)(n)));
    if (__res>=0)
        return int __res;
    errno=-__res;
    return -1;
}
```

It can be seen that this macro is expanded to be a concrete implementation of a read system call. It uses the embedded assembly statement to execute the Linux system interrupt call 0x80 with the function number __NR_read(3). This interrupt call returns the actual number of bytes read in the EAX (__res) register. If the returned value is less than 0, it means that the read operation error occurs, so the error number is inverted and stored in the global variable errno, and the value of -1 is returned to the calling program.

If a system call requires more than 3 parameters, the kernel usually uses the parameter as a parameter buffer block and passes the pointer of the buffer block as a parameter to the kernel. So for system calls with more than 3 parameters, we only need to use the macro _syscall1() with one argument to pass the pointer of the first argument to the kernel. For example, the select() function system call has 5 arguments, but we only need to pass a pointer to its first argument, see the description of the fs/select.c program.

When entering the kernel call handler kernel/sys_call.s in the kernel, the system_call code will first check if the system call function number in EAX is within the valid system call number range. Then execute the corresponding system call handler according to the sys_call_table[] function pointer table call.

```
    call _sys_call_table(,%eax, 4)                    // kernel/sys_call.s  第 99 行。
```

The meaning of this assembly statement operand is: indirectly call the function at _sys_call_table + %eax * 4. Since the sys_call_table[] pointer is 4 bytes each, it is necessary to multiply the system call function number by 4. Then use the resulting value to get the address of the called handler from the table.

5.5.3 Parameter Passing Method of Linux System Call

Regarding Linux user processes passing parameters to the system interrupt call procedure, Linux systems use general-purpose register transfer methods such as registers EBX, ECX, and EDX. A significant advantage of this method of using register-passing parameters is that when the system interrupt service routine is entered and the register values are saved, the registers that pass the parameters are also automatically placed on the kernel-state stack. Therefore, there is no need to specialize the special processing of the registers that pass the parameters. This is the simplest and fastest method of parameter transfer that Mr. Linus knew at the time. There is also a parameter transfer method using the system call gate provided by the Intel CPU, which automatically copies the passed parameters in the process user state stack and the kernel state stack. But the method used in this method is more complicated.

In addition, the parameters passed should be verified in each system call handler to ensure that all parameters are legal and valid. In particular, user-supplied pointers should be rigorously reviewed to ensure that the range of memory regions pointed to by the pointer is valid and has appropriate read and write permissions.

5.6 System time and timing

5.6.1 System time

In order to allow the operating system to automatically and accurately provide current time and date information, battery-powered real-time RT (Real Time) circuit support is provided in the PC/AT microcomputer system. Usually, this part of the circuit is integrated on a single chip with a small amount of CMOS RAM that holds system information, so this part of the circuit is called an RT/CMOS RAM circuit. Motorola's MC146818 chip is used in the PC/AT microcomputer or its compatible machine.

At initialization, the Linux 0.12 kernel reads the current time and date information stored in the chip and converts it to the current time in seconds from 0:00 on January 1, 1970. We call this the UNIX calendar time. This time determines the calendar time at which the system starts running and is saved in the global variable startup_time for all kernel code to use. The user program can use the system call time() to read the value of startup_time, while the superuser can modify the system time value by calling stime().

In addition, the program can uniquely determine the current running time by the following system tick value jiffies counted from the system start. Each tick is generated by a timer as described below.Since each tick timing value is 10 milliseconds, a macro is defined in the kernel code to facilitate access to the code at the current time. This macro is defined on line 192 of the include/linux/sched.h file and has the following form:

```
    #define CURRENT_TIME (startup_time + jiffies/HZ)
```

Among them, HZ = 100, is the core system clock frequency. The current time macro CURRENT_TIME is defined as the system boot time startup_time plus the time jiffies/100 of the system running after booting. This macro is used when modifying the time when a file is accessed or when its i-node is modified.

5.6.2 System Timing

The basic timing beat of the system is generated by the timing chip. During the initialization of the Linux 0.12 kernel, the counter channel 0 of the PC's programmable timing chip Intel 8253 (8254) is set to operate in mode 3, and the initial count value LATCH is set to emit a square wave rising edge at output OUT every 10 milliseconds. Since the clock input frequency of the 8254 chip is 1.193180 MHz, the initial count value LATCH=1193180/100 is approximately 11931. Since the OUT pin is connected to level 0 of the programmable interrupt control chip, the system issues a clock interrupt request (IRQ0) every 10 milliseconds. This time beat is the pulse of the operating system, we call it a system tick or a system clock cycle. Therefore, every a tick time elapses, the system will call the clock interrupt handler (timer_interrupt).

The clock interrupt handler timer_interrupt is mainly used to accumulate the number of clock ticks that have passed since the system was started by the jiffies variable. The jiffies value is incremented by one each time a clock interrupt occurs. Then call the C language function do_timer() for further processing. The parameter CPL with the call is obtained from the segment selector of the interrupted program (the CS segment register value stored in the stack) to obtain the current code privilege level CPL.

The do_timer() function accumulates the current process runtime based on the privilege level. If CPL=0, it means that the process is interrupted when it runs in kernel mode, so the kernel will increase the kernel state running time stime of the process by one, otherwise it will increase the running value of the process user state by one. If the floppy disk program floppy.c adds a timer during the operation, the timer list is processed. If a timer expires (equal to 0 after decrement), the handler of the timer is called. Then the current process running time is processed, and the current process running time slice is decremented by one. A time slice is the CPU time that a process can continue to run before being switched out. The unit is the number of ticks defined above.

If the process time slice value is decremented and is still greater than 0, it means that its time slice has not been used up, so it exits do_timer() and continues to run the current process. If the process time slice has been decremented to 0 at this time, it means that the process has used up the time slice of the CPU, and the program will determine the further processing method according to the level of the interrupted program. If the interrupted current process is working in user mode (privilege level is greater than 0), then do_timer() will call the scheduler schedule() to switch to another process to run. If the interrupted current process is working in kernel mode, that is, it is interrupted while running in the kernel program, do_timer() will exit immediately. Therefore, this way of processing determines that the Linux system process will not be switched by the scheduler when running in kernel mode. That is, the process is nonpreemptive when running in a kernel mode program.

Note that the above timer code is dedicated to floppy motor turn-on and turn-off timing operations. This kind of timer is similar to the dynamic timer in modern Linux systems and is only used by the kernel. Such timers can be dynamically created when needed and dynamically revoked when the timing expires. In the Linux 0.12 kernel, there are up to 64 timers at the same time. The processing code for the timer is in the sched.c program 283--368 lines.

5.7 Linux Process Control

A program is an executable file, and a process is an instance of a program that is executing. With time-sharing technology, multiple processes can be run simultaneously on the Linux operating system. The basic principle of time-sharing technology is to divide the running time of the CPU into time slices of a specified length, so that each process runs in one time slice. When the time slice of the process runs out, the system uses the scheduler to switch to another process to run. So in fact, for a machine with a single CPU, only one process can be run at a time. But since each process runs a short time slice (for example, 15 system tick = 150 milliseconds), it appears as if all processes are running at the same time.

For the Linux 0.12 kernel, the system can have up to 64 processes at the same time. Except for the first process which created "manually", the rest are new processes created by existing processes using the system call fork. The created process is called the child process, and the creator is called the parent process.

The kernel program uses the process ID (process ID, pid) to identify each process. A process consists of executable instruction code, data, and a stack sections. The code and data parts in the process correspond to the code segments and data sections in one execution file respectively. Each process can only execute its own code and access its own data and stack area. Communication between processes needs to be done through system calls. For systems with only one CPU, only one process can be running at a time. The kernel schedules each process to run in a time-sharing manner through the scheduler.

We already know that a process in a Linux system can be executed in kernel mode or user mode, and each uses its own separate kernel state stack and user state stack. The user stack is used by the process to temporarily save the parameters of the calling function, local variables, etc. in the user state; the kernel stack contains the information when the kernel program executes the function call.

Also in the Linux kernel, processes are often referred to as tasks, and programs running in user space are called processes. This book will mix these two terms while trying to follow this default rule.

5.7.1 Task Data Structure

The kernel program manages the process through the process table, and each process occupies one item in the process table. In a Linux system, a process table entry is a task_struct task structure pointer. Some books refer to it as process control block (PCB) or process descriptor (PD). It holds all the information used to control and manage the process. It mainly includes the status information of the current running of the process, the signal, the process number, the parent process number, the running time accumulated value, the file being used, the local descriptor of the task, and the task status segment information. The task data structure is defined in the header file include/linux/sched.h. The specific meaning of each field of the structure is as follows.

```
struct task_struct {
   long state;                 // -1 unrunnable, 0 runnable (ready), > 0 stopped.
   long counter;               // Task run time tick (decrement), run time slice.
   long priority;              // Priority. When task starts running, counter=priority.
   long signal;                // Signal bitmap, each bit is a signal( = bit offset + 1).
```

```
    struct sigaction sigaction[32]; // Signal attribute struct. Signal operation and flags.
    long blocked;                 // Process signal mask (Bitmap of masked signal).
    int exit_code;                // Exit code after task stops, its parent will get it.
    unsigned long start_code;     // Code start location in linear address space.
    unsigned long end_code;       // Code length or size (bytes).
    unsigned long end_data;       // Code size + data siez (bytes).
    unsigned long brk;            // Total size (number of bytes).
    unsigned long start_stack;    // Stack bottom location.
    long pid;                     // Process identifier.
    long pgrp;                    // Process group number.
    long session;                 // Process session number.
    long leader;                  // Leader session number.
    int groups[NGROUPS];          // Group numbers. A process can belong to more groups.
    task_struct *p_pptr;          // Pointer to parent process.
    task_struct *p_cptr;          // Pointer to youngest child process.
    task_struct *p_ysptr;         // Pointer to younger sibling process created afterwards.
    task_struct *p_osptr;         // Pointer to older sibling process created earlier.
    unsigned short uid;           // User id.
    unsigned short euid;          // Effective user id.
    unsigned short suid;          // Saved user id.
    unsigned short gid;           // Group id.
    unsigned short egid;          // Effective group id.
    unsigned short sgid;          // Saved group id.
    long timeout;                 // Kernel timing timeout value.
    long alarm;                   // Alarm timing value (ticks).
    long utime;                   // User state running time (ticks).
    long stime;                   // System state runtime (ticks).
    long cutime;                  // Child process user state runtime.
    long cstime;                  // Child process system state runtime.
    long start_time;              // Time the process started running.
    struct rlimit rlim[RLIM_NLIMITS];  // Resource usage statistics array.
    unsigned int flags;           // per process flags.
    unsigned short used_math;     // Flag: Whether a coprocessor is used.
    int tty;                      // The tty subdevice number used. -1 means no use.
    unsigned short umask;         // The mask bit of the file creation attribute.
    struct m_inode * pwd;         // Current working directory i node structure pointer.
    struct m_inode * root;        // Root i-node structure pointer.
    struct m_inode * executable;  // The pointer to i-node structure of the executable file.
    struct m_inode * library;     // The loaded library i-node structure pointer.
    unsigned long close_on_exec; // A bitmap flags that close file handles on execution.
    struct file * filp[NR_OPEN]; // File structure pointer table, up to 32 items.
                                  // The index is the value of file descriptor.
    struct desc_struct ldt[3];    // LDT. 0-empty, 1-code seg cs, 2-data & stack seg ds & ss.
    struct tss_struct tss;        // The task status segment structure TSS of the process.
};
```

◆long state -- The state field contains current state of the process. At some point, a Linux process can be in one of five states and can transition between these states under the operation of the kernel scheduler. The five states are: running state (TASK_RUNNING), interruptible sleep state (TASK_INTERRUPTIBLE), uninterruptible sleep state (TASK_UNINTERRUPTBLE), zombie state (TASK_ZOMBIE), and stopped state (TASK_STOPPED). The way the kernel changes the state of the process is described in the next section.

◆long counter -- The counter field holds the number of time ticks that the process can execute before it is temporarily stopped. That is, it usually takes several system clock cycles to switch to another process. The scheduler uses the counter value of the process to select the next process to execute, so the counter can be thought of as a dynamic feature of a process. The initial value of the counter is equal to the priority when a process has just been created.

◆long priority -- Priority is used to assign the initial value to the counter. In Linux 0.12 this initial value is 15 system clock cycle times (15 ticks). When needed, the scheduler will use the value of priority to assign an initial value to the counter, see the sched.c and the fork.c programs. Of course, the unit of priority is also the number of time ticks.

◆long signal -- The signal field is a bitmap of the signal currently received by the process. The bitmap has 32 bits, each bit represents a signal, and the signal value = bit offset value +1. So the Linux kernel has up to 32 signals. At the end of each system call process, the signal is preprocessed using the signal bitmap.

◆struct sigaction sigaction[32] -- The sigaction structure array is used to store the operations and attributes used to process each signal. Each item of the array corresponds to one signal.

◆long blocked -- The blocked field is a blocking bitmap of the signal that the process does not currently want to process. Similar to the signal field, each bit represents a blocked signal.

◆int exit -- The exit field is used to save the exit code when the program terminates. After the child process ends, the parent process can query its exit code.

◆unsigned long start_code -- The start_code field is the starting address of the process code in the CPU linear address space. In the Linux 0.1x kernel, its value is an integer multiple of 64MB.

◆unsigned long end_code -- The end_code field holds the byte length value of the process code.

◆unsigned long end_data -- The end_data field holds the code length of the process + the total byte length value of the data length.

◆unsigned long brk -- The brk field is also the total byte length value (pointer value) of the process code and data, but also includes the uninitialized data area bss, as shown in Figure 5-12. This is the initial value of brk when a process starts executing. By modifying this pointer, the kernel can add and release dynamically allocated memory for the process. This is usually done by the kernel by calling the malloc() function and by calling the brk system call.

◆unsigned long start_stack -- The start_stack field value points to the beginning of the stack in the process's logical address space. See also the stack pointer location in Figure 5-12.

◆long pid -- Pid is the process identification number. It is used to uniquely identify the process.

◆long pgrp -- Pgrp refers to the process group number to which the process belongs.

◆long session -- Session is the session number of the process, which is the process ID of the session.

◆long leader -- The leader is the first process number of the session. For the concept of process groups and sessions, see the instructions following Chapter 7, Program Listing.

◆int groups[NGROUPS] -- Groups is an array of group numbers for each group to which the process belongs. A process can belong to more than one group.

◆task_struct *p_pptr -- p_pptr is a pointer to the parent process's task structure.

◆task_struct *p_cptr -- p_cptr is a pointer to the most recent subprocess's task structure. That is the youngest child's task structure. Refer to figure 5-20.

◆task_struct *p_ysptr -- p_ysptr is a pointer to an adjacent process created later than itself. That is pointer to the younger sibling process.

◆task_struct *p_osptr -- *p_osptr is a pointer to an adjacent process created earlier than itself. That is point to the older sibling process. See Figure 5-20 for the relationship between the above four pointers. In the task data structure of the Linux 0.11 kernel, there is a parent process number field father, but it is not used in the

0.12 kernel. At this point we can use the process's pptr->pid to get the process number of the parent process.

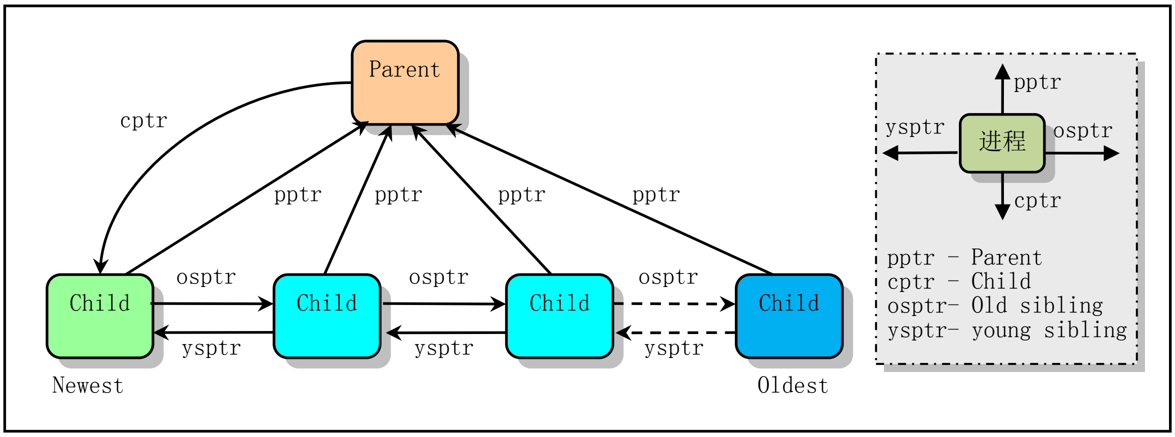


Figure 5-20 Relationship between process pointers

◆unsigned short uid -- Uid is the user identification number (user id) that owns the process.

◆unsigned short euid -- Euid is the effective user identification number that indicates the permissions to access the file.

◆unsigned short suid -- Suid is the saved user identification number. When the set-user-ID flag of the execution file is set, the suid of the execution file is saved in the suid. Otherwise suid is equal to the euid of the process.

◆unsigned short gid -- Gid is the group identification number (group id) to which the user belongs. It identifies the user group that owns the process.

◆unsigned short egid -- Egid is an effective group identification number that indicates the permissions of the group of users to access the file.

◆unsigned short sgid -- Sgid is the saved user group identification number. When the set-group-ID flag of the execution file is set, the gid of the execution file is stored in the sgid. Otherwise sgid is equal to the process's egid. For a description of these user id and group id, see the overview of the sys.c program in Chapter 5.

◆long timeout -- Kernel timing timeout value.

◆long alarm -- Alarm is the alarm timing value (number of ticks) for the process. If the process sets the field value using the system call alarm(), then when the system time ticking value exceeds the alarm field value, the kernel sends a SIGALRM signal to the process. By default this signal will terminate the execution of the program. Of course, we can also use the signal capture function (signal () or sigaction ()) to capture the signal for the specified operation. The function alarm() starts at line 370 of kernel/sched.c. The kernel converts the function value of the function in seconds into a tick value, which is stored in the field after the current time tick value of the system.

◆long utime -- Utime is the cumulative time (ticks) that the process runs in user state.

◆long stime -- Stime is the cumulative time (ticks) that the process runs in the system state.

◆long cutime -- Cutime is the cumulative time (ticks) that child processes runs in user state.

◆long cstime -- Cstime is the cumulative time (ticks) that child processes runs in system state.

◆long start_time -- Start_time is the time when the process is generated and starts running.

◆struct rlimit rlim[RLIM_NLIMITS] -- The resource usage statistics array for the process.

◆unsigned int flags -- Its the flag for each process, and 0.12 kernel is not yet in use.

◆unsigned short used_math -- It is a flag indicating whether the process uses a coprocessor.

◆int tty -- It is the subdevice number of the process using the tty terminal. -1 means no use.

◆unsigned short umask -- It is the 16-bit attribute mask word used by the process to create a new file (each bit represents a file), that is, the access attribute set by the new file. If a bit of the mask word is set, it means that the corresponding attribute is disabled (masked). This attribute mask word is used with the attribute value given when the file was created (mode &~umask) as the actual access attribute of the newly created file. See the include/fcntl.h and include/sys/state.h files for the specific meaning of the masked word and file attributes.

◆struct m_inode * pwd -- Pwd is a pointer to the i-node structure of the current working directory of the process. Each process has a current working directory that resolves relative path names and can be changed using the system call chdir.

◆struct m_inode * root -- Root is the process's own root i-node structure. Each process can have its own specified root directory for parsing absolute path names. Only the superuser can modify this root directory by calling chroot.

◆struct m_inode * executable -- Executable is the pointer to the i-node structure in memory for the execution file of the process. The system can use this field to determine if there is another process running the same executable file in the system. If so, then the in-memory i-node reference count value of executable->i_count will be greater than 1. When the process is created, the field is given the same value as the same field of the parent process, which means that the same program is being run with the parent process. When the kind of exec() function is called to execute a specified new executable file, the field value is replaced with the memory i-node pointer of the new program executed by the exec() function. When the process calls the exit() function and performs exit processing, the reference count of the memory i node pointed to by this field is decremented by 1, and the field will be blanked. The main role of this field is reflected in the share_page() function of the memory.c program. This function code can determine whether there are multiple copies of the currently running program in the system (at least 2) according to the reference count of the node pointed to by the execution of the process. If so, try a page sharing operation between them.

At system initialization, the execution of all tasks created by the system is 0 before the first call to execute the execve() function. These tasks include Task 0, Task 1, and all tasks created directly by Task 1 that have not yet executed execve().That is, the executable field of all tasks directly included in the kernel code are 0. Because the code for task 0 is included in the kernel code, it is not loaded by the system from the file system. Therefore, the executable code has a fixed value of 0 in the kernel code. In addition, when creating a new process, fork() will copy the task data structure of the parent process, so task 1's executable is also 0. But after runing execve(), the executable is given a pointer to the memory i-node of the file being executed. After that, this value of all tasks will not be 0.

◆unsigned m_inode * library -- Library is the i-node structure pointer of the library file that is loaded when the program is executed.

◆unsigned long close_on_exec -- It is a file descriptor (file handle) bitmap flag for a process. Each bit represents one file descriptor that is used to determine the file descriptor that needs to be closed when the system call execve() is called (see include/fcntl.h). When a program creates a child process using the fork(), it usually calls the execve() function in the child process to load another new program. At this point the child process will be completely replaced by the new program and the new program will start executing in the child process. If the corresponding bit of a file descriptor in close_on_exec is set, then the file descriptor corresponding to the open file will be closed when the child process executes the execve().That is, the file descriptor is closed in the new program, otherwise the file descriptor will always be open.

◆struct file * filp[NR_OPEN] -- It is a table of file structure pointers for all open files used by the process, up to a maximum of 32 entries. The value of the file descriptor is the index value in the structure table. Each of these is used for file descriptors to locate file pointers and access files.

◆struct desc_struct ldt[3] -- It is the process local descriptor table structure LDT. It defines the code segment and data segment of the task in the virtual address space. Where array item 0 is a null item, item 1 is a code segment descriptor, and item 2 is a data segment (including data and stack) descriptor.

◆struct tss_struct tss -- It is the task state segment TSS information structure of the process. The tss_struct structure holds all register values of the current processor when the task is switched out from execution. When the task is re-executed, the CPU uses these values to restore to the state when the task was switched out and starts execution.

When a process is executing, the values in all registers of the CPU, the state of the process, and the contents of the stack are called the context of the process. When the kernel needs to switch to another process, it needs to save all the state of the current process, that is, save the context of the current process, so that when the process is executed again, it can be restored to the state before the switch. In Linux, the current process context is stored in the task's task data structure. When an interrupt occurs, the kernel executes the interrupt service routine in the kernel state in the context of the interrupted process. At the same time, all the resources that need to be used are retained, so that the execution of the interrupted process can be resumed when the interrupt service ends.

5.7.2 Process Running States

A process can be in a different set of states during its lifetime, called the process state, as shown in Figure 5-21. Each circle in the figure with a different number represents a different state. As mentioned earlier, the process state is saved in the state field of the process task structure.

If a process is waiting for using CPU or is running, it is said to be in a ready or running state. At this point, the process state field value is TASK_RUNNING. If a process is asleep while waiting for system resources or event to occur, it is said to be in an interruptible sleep state, or an uninterruptible sleep state. At this time, the state field of the process may be TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE, respectively. If a process has been terminated, but it has not completely released the kernel resources, it is said to be in a dead state. At this point, the state field value of the process is TASK_ZOMBIE. If a process is temporarily stopped, it is said to be in a suspended state. At this point, the state field value of the process is TASK_STOPPED.

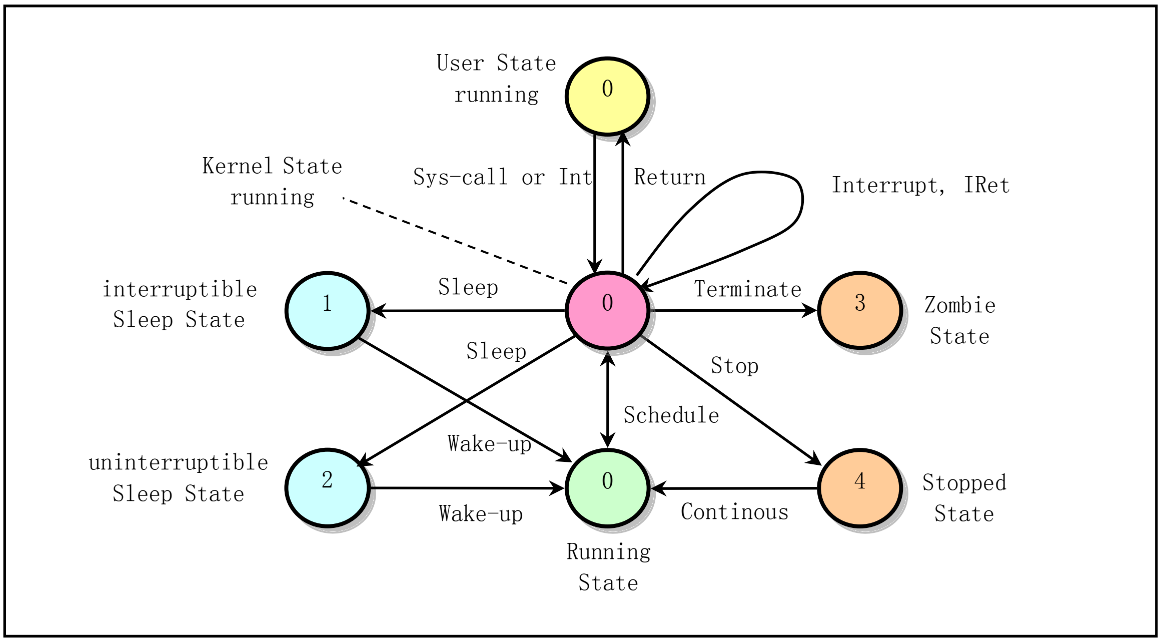


Figure 5-21 Process status and conversion relationship

The constant symbol names used for the five states of the Linux process are shown below. They are defined in line 46-50 in the file include/linux/sched.h.

```
   // This defines the state of a process.
46 #define TASK_RUNNING             0   // is running or ready to run
47 #define TASK_INTERRUPTIBLE       1   // is in an interruptible wait state.
48 #define TASK_UNINTERRUPTIBLE     2   // uninterruptible wait state for wait I/O operation.
49 #define TASK_ZOMBIE              3   // is in a zombie state and has been terminated.
50 #define TASK_STOPPED             4   // The process has stopped.
```

◆Running status (0, TASK_RUNNING)

When a process is being executed by the CPU, or is ready to be executed by the scheduler at any time, the process is said to be running state. If the process is not executed by the CPU at this time, it is said to be in the ready-to-run state, as shown in Figure 5-21. In the figure, the middle three circles from top to bottom contains the same value 0, which means that they are all ready or running states. Processes can run in kernel mode or in user mode. When a process is running in kernel code, we call it kernel running state, or simply kernel mode; when a process is executing its own code, we call it user running state (user mode). When system resources are available, the process wakes up and enters the ready-to-run state, which is called the ready state. These states (the middle column in the figure) represent the same method in the kernel and are said to be in the TASK_RUNNING state. When a new process has just been created, it is in this state (the bottom 0).

◆Interruptible sleep state (1, TASK_INTERRUPTIBLE)

When a process is in an interruptible wait (sleep) state, the system does not schedule the process to execute. When the system generates an interrupt or releases the resource that the process is waiting for, or the process receives a signal, it can wake up the process to change to the ready state (that is, the running state).

◆Uninterruptible sleep state (2, TASK_UNINTERRUPTIBLE)

This state is similar to the interruptible sleep state except that it is not woken up by the receipt of a signal.

However, a process in this state can only be converted to a runnable ready state when it is explicitly awake using the wake_up() function. This state is typically used when a process needs to wait undisturbed or when a waiting event occurs quickly.

◆Zombie state (3, TASK_ZOMBIE)

When a process has stopped running, but its parent has not called wait() to ask for its status, the process is said to be in a dead state. In order for the parent process to get the information that it stopped running, the task data structure information of the child process needs to be retained. Once the parent process calls wait() to get the information of the child process, the task data structure of the process in that state is released.

◆Stop status (4, TASK_STOPPED)

When the process receives the signal SIGSTOP, SIGTSTP, SIGTTIN or SIGTTOU, it will enter the stop state. The SIGCONT signal can be sent to the process to transition to a runnable state. Any signal received by the process during debugging will enter this state. In Linux 0.12, the conversion processing for this state has not been implemented. Processes in this state will be processed as process termination.

When a process runs out of time, the system uses the scheduler to force a switch to another process to execute. In addition, if the process needs to wait for a certain resource of the system when it executes in kernel mode, the process will call sleep_on() or interruptible_sleep_on() to voluntarily give up the usage rights of the CPU, and let the scheduler execute other processes. The process goes to sleep (TASK_UNINTERRUPTIBLE or TASK_INTERRUPTIBLE).

The kernel will only perform process switching operations when the process moves from "kernel running state" to "sleep state". Processes running in kernel mode cannot be preempted by other processes, and one process cannot change the state of another process. In order to avoid kernel data errors during process switching, the kernel will disable all interrupts when executing critical area code.

5.7.3 Process initialization

In the boot/ directory, the bootloader loads the kernel from disk into memory and puts the system into protected mode, then starts the system initialization program init/main.c. The main program first determines how to allocate and use the system's physical memory, and then calls the initialization functions of the various parts of the kernel to initialize the memory management, interrupt handling, block devices, character devices, process management, and hard disk and floppy disk hardware. After these operations are completed, the various parts of the system are operational. Program control is then "manually" moved to task 0 (process 0), and process 1 is created for the first time using the fork() call. In process 1, the program will continue to initialize the application environment and execute the shell login program. The original process 0 will be scheduled to execute when the system is idle. At this point task 0 only executes the pause() system call, which in turn executes the scheduler function.

The process "Move to task 0 execution" is done by the macro move_to_user_mode (include/asm/system.h). It runs the main.c program execution flow from kernel mode (privilege level 0) to task 0 of user mode (privilege level 3). Before the move, the system first sets the running environment of task 0 in the initialization process (sched_init()) of the scheduler. This includes manually pre-setting the values of the fields of the task 0 data structure (include/linux/sched.h), adding the task state segment (TSS) descriptor of task 0 and the local descriptor table (LDT) in the global descriptor table. The segment descriptors are loaded into the task register tr and the local descriptor table register ldtr, respectively.

It should be emphasized here that kernel initialization is a special process, and the kernel initialization code is also the code of task 0. From the initial data set in the task 0 data structure, the base address of the code segment and the data segment of task 0 is 0, and the segment length is 640 KB. The base address of the kernel code segment and the data segment is 0, and the segment length is 16 MB. Therefore, the code segment and data segment of task 0 are included in the kernel code segment and the data segment, respectively. The kernel initialization program main.c is the code for task 0, except that the system is running the main code with kernel mode privilege level 0 before moving to task 0. The function of the macro move_to_user_mode is used to move the running privilege level from level 0 of the kernel state to level 3 of the user mode, but still continue to execute the original code instruction stream.

During the move to task 0, the macro move_to_user_mode uses the interrupt return instruction to cause a privilege level change. The use of this method for control transfer is caused by the CPU protection mechanism. The CPU allows low-level (such as privilege level 3) code to be called or transferred to high-level code by calling a gate or interrupt, a trap gate, but not vice versa. So the kernel uses a method that simulates IRET to return low-level code. The main idea of this method is to construct the contents of the interrupt return instruction in the stack, and set the segment selector of the return address to the task 0 code segment selector with a privilege level of 3. Thereafter, executing the interrupt return instruction IRET will cause the system CPU to jump from privilege level 0 to the privilege level 3 of the outer layer. See Figure 5-22 for a diagram of the interrupt return stack structure when the privilege level changes.

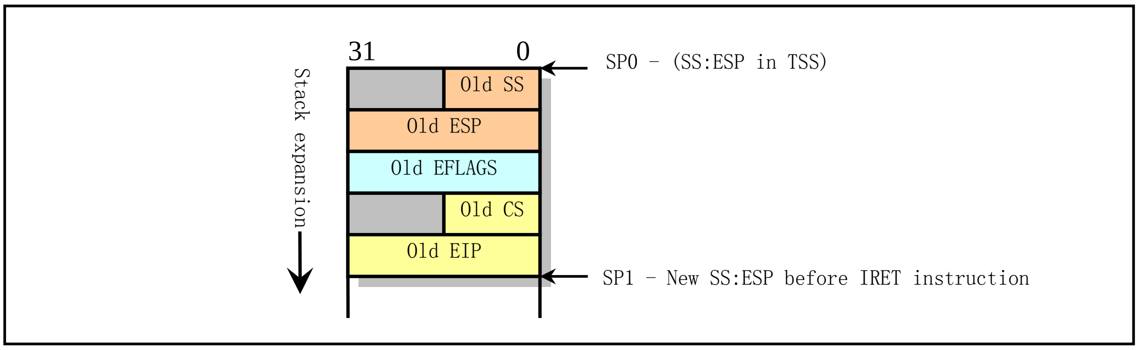


Figure 5-22 Interrupt return stack structure when privilege level changes

The macro move_to_user_mode first pushes the task 0 stack segment (ie, data segment) selector and kernel stack pointer into the kernel stack. Then push in the contents of the flag register. Finally, the task 0 code segment selector and the offset of the next instruction to be executed after the "interrupt return" is pushed in.

When the IRET instruction is executed, the CPU sends the return address to the CS:EIP and pops up the contents of the flag register in the stack. Since the CPU judges that the privilege level of the destination code segment is 3, it is different from the level 0 of the current kernel state. The CPU then pops the stack segment selector and stack pointer in the stack into SS:ESP. As the privilege level changes, the values of the segment registers DS, ES, FS, and GS become invalid, and the CPU clears these segment registers. Therefore, these segment registers need to be reloaded after the IRET instruction is executed. Thereafter, the system begins running on privilege level 3 on the code of task 0. At this point, the original stack used before the move is used as the subscriber station stack for task 0. The kernel state stack is specified by the contents of the TSS of task 0. It is specified begining at the top of the page where the task data structure is located (PAGE_SIZE + (long) & init_task). Since the task data structure of task 0, including its user stack pointer, needs to be copied later when creating a new process, the user-mode stack of task 0 is required to remain "clean" until task 1 (process 1) is created.

5.7.4 Creating new processes

Creating a new process in a Linux system requires the use of the fork() system call. All processes created after initialization originates from process 0, which are child processes of process 0.

In the process of creating a new process, the system first finds an empty item (empty slot) in the task data structure array that has not been used by any process. If the system already has 64 processes running, the fork() system call will return an error because there are no empty items available in the task array table. Otherwise, the system will apply for a page of memory in the main memory area for the new process to store its task data structure information, and copy all the contents of the current process task data structure as a template for the new process task data structure. In order to prevent this new process that has not been processed from being executed by the scheduler, the new process state should be immediately set to the uninterruptible wait state (TASK_UNINTERRUPTIBLE).

The copied task data structure is then modified. Set the current process to be the parent of the new process, clear the signal bitmap and reset the statistics of the new process, and set the initial runtime slice value to 15 system ticks (150 milliseconds). Next, the values of the registers in the task status segment (TSS) are set according to the current process. Since the new process return value should be 0 when the process is created, you need to set tss.eax = 0. The new process kernel state stack pointer tss.esp0 is set to the top of the memory page where the new process task data structure is located, and the stack segment tss.ss0 is set to the kernel data segment selector. Tss.ldt is set to the index value of the local table descriptor in the GDT. If the current process uses a coprocessor, you also need to save the full state of the coprocessor to the tss.i387 structure of the new process.

After that, the system sets the code and data segment base address and limit length of the new task, and copies the current process memory page table. Note that at this point the system does not allocate the actual physical memory page for the new process, but instead lets it share the memory page of its parent process. Only when any of the parent process or the new process has a write memory operation will the system allocate the relevant memory page for the write operation. This type of processing is called Copy On Write technology. For a detailed description of this technique, see the Write-Only Replication mechanism in the Memory Management chapter.

Later, if there are files in the parent process that are open, you should increase the number of open times of the corresponding file by one. The TSS and LDT descriptor entries for the new task are then set in the GDT, where the base address information points to tss and ldt in the new process task structure. Finally, set the new task to a runnable state and return a new process number.

Also note that creating a new child process and loading an executable file is two different concepts. When a child process is created, it completely copies the code and data area of the parent process and executes the code for the child process part there. When executing a program on a block device, it is generally run in the child process by running the exec() system call. After entering exec(), the original code and data area of the child process will be cleared (released). When the child process starts running a new program, since the kernel has not loaded the code of the program from the block device at this time, the CPU immediately generates an exception of page does not exist. At this point, the memory manager loads the corresponding code page from the block device, and the CPU re-executes the instruction that caused the exception. Until now, the code of the new program will actually start to be executed.

5.7.5 Process Scheduling

The scheduler in the kernel is used to select the next process to run in the system. This selective operating mechanism is the basis of a multitasking operating system. The scheduler can be thought of as management code that allocates CPU runtime between all running processes. As can be seen from the foregoing description, the Linux process is preemptive, but the preempted process is still in the TASK_RUNNING state, but it is not temporarily run by the CPU. The preemption of the process occurs when the process is in the user state execution phase and cannot be preempted when executed in kernel state.

In order to enable the process to effectively use the system resources and allow the process to have a faster response time, it is necessary to adopt a certain scheduling strategy for the process switching scheduling. A scheduling strategy based on priority queuing is adopted in Linux 0.12.

The scheduler

The schedule() function first scans the task array. For each ready state (TASK_RUNNING) task, determine which process currently runs the least by comparing their run time count counter values. Which one has a large value means that the running time is not long, so the process is selected and the task switching macro function is used to switch to the process.

If all the time slices of the process in the TASK_RUNNING state have been used up at this time, the system will recalculate the time slice (counter) required for each process (including the sleeping process) in the system according to the priority value of each process. The calculation formula is:

$$counter = \frac{counter}{2} + priority$$

Thus, for a sleeping process, they have a higher time slice counter value when they are woken up. Then, the schedule() function rescans all processes in the TASK_RUNNING state in the task array and repeats the process until a process is selected. Finally, switch_to() is called to perform the actual process switch operation.

If no other processes can run at this time, the system will select process 0 to run. For Linux 0.12, process 0 calls pause() to put itself into an interruptible sleep state and call schedule() again. However, when scheduling a process, schedule() does not care what state of process 0 is. As long as the system is idle, the process 0 is scheduled to run.

Process switching

Whenever a new runnable process is selected, the schedule() function calls the switch_to() macro to perform the actual process switch operation. Switch_to() is defined in include/asm/system.h. This macro replaces the current process state (context) of the CPU with the state of the new process. Before switching, switch_to() first checks if the process to be switched to is the current process, and if so, does nothing and exits directly. Otherwise, the kernel global variable current is first set to the pointer of the new task, and then jumps to the address of the task state segment TSS of the new task, causing the CPU to perform the task switching operation. At this point, the CPU saves the state of all its registers to the tss structure of the current process task data structure pointed to by the TSS segment selector in the current task register TR. The register information in the tss structure in the new task data structure pointed to by the new task status segment selector is then restored to the CPU. After that, the system officially started the task of running the new switch. This processing can be seen in Figure 5-23.

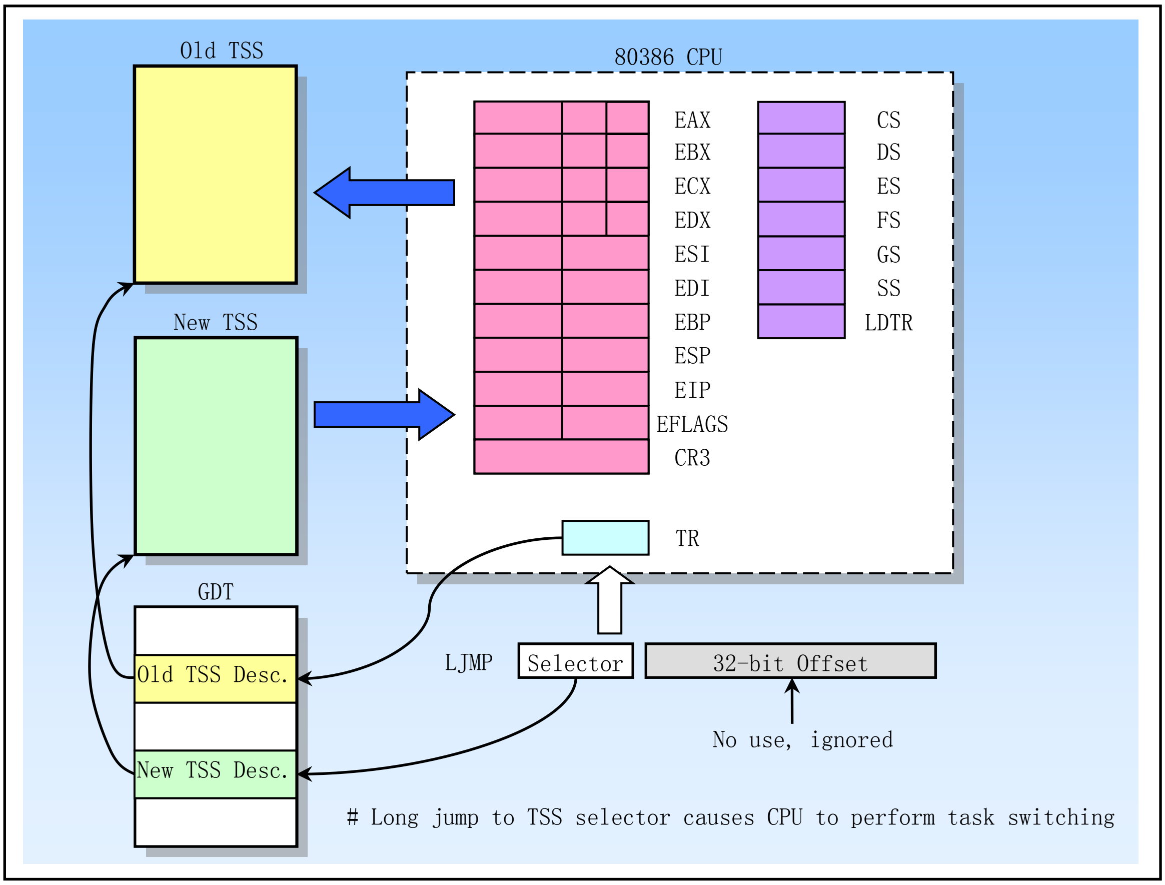


Figure 5-23 Task switching operation diagram

5.7.6 Terminating a process

When a process finishes running or terminates running halfway, the kernel needs to free up system resources occupied by the process. This includes files opened while the process is running, memory used, and so on.

When a user program calls the exit() system call, the kernel function do_exit() is executed. The function first releases the memory page occupied by the process code segment and the data segment, closes all files opened by the process, and synchronizes the current working directory, the root directory, and the i-node running the program used by the process. If the process has child processes, let the init process be the parent of all its child processes. If the process is a session header process and has a control terminal, the control terminal is released and the hang up signal SIGHUP is sent to all processes belonging to the session. This usually terminates all processes in the session. Then set the process state to TASK_ZOMBIE. And send a SIGCHLD signal to its original parent process to inform that some of its child processes have terminated. Finally do_exit() calls the scheduler to execute other processes. It can be seen that when the process is terminated, its task data structure remains, because its parent process also needs to use the information.

During the execution of a child process, the parent process usually waits for a child process to terminate using the wait() or waitpid() function. When the waiting child process is terminated and is in a zombie state, the parent process accumulates the time spent by the child process into its own process. Eventually, the memory page occupied by the child process task data structure is freed, and the pointer items occupied by the child process in the task array are blanked.

5.8 How to use the stack in Linux

This section provides an overview of how the Linux kernel uses the stack from boot-up to system uptime. The description of this part is closely related to the kernel code and can be skipped first. You can come back and study it when reading the relating code.

Four types of stacks are used in the Linux 0.12 system. One is the stack that is temporarily used during system boot initialization; the other is the stack that is used to initialize the kernel after entering protected mode, located at a fixed location in the kernel code address space. This stack is also the user-mode stack used by Task 0 later; the next is the stack used by each task to execute the kernel code through system calls, which we call the kernel-level stack of the task. Each task has its own independent kernel-mode stack; the last is the stack that the task executes in user mode, at the near end of the task (process) logical address space.

There are two main reasons for using multiple stacks or using different stacks in different situations. The first is that since the real mode enters the protection mode, the CPU has changed the memory access mode, so it is necessary to re-adjust the setup stack area. In addition, in order to solve the protection problem caused by the use of the stack by different privilege levels of the CPU, it is necessary to use a different stack for executing the level 0 kernel code and the level 3 user code. When a task enters kernel mode, it uses the stack pointer tss.ss0, tss.esp0, which is the privilege level 0 given in its TSS section, which is the kernel stack. The original user stack pointer will be saved in the kernel stack. When returning from user mode to kernel mode, the user-mode stack is restored. They are explained separately below.

5.8.1 Initialization phase

Boot initialization (bootsect.s, setup.s)

When the bootsect code is loaded by the ROM BIOS to physical memory 0x7c00, the stack segment is not set. Of course, the program does not use the stack. After the bootsect is moved to 0x9000:0, the stack segment register SS is set to 0x9000, and the stack pointer esp register is set to 0xff00. That is, the top of the stack is at 0x9000:0xff00, see line 61 of boot/bootsect.s. The stack section set in bootsect is also used in the setup.s program. This is the stack that is temporarily used during system initialization.

When entering protected mode (head.s)

From the head.s program, the system is officially running in protected mode. At this point the stack segment is set to the kernel data segment (0x10), the stack pointer esp is set to point to the top of the user_stack array (see head.s, line 31), and one page of memory (4K) is reserved for use as a stack. The user_stack array is defined in lines 67--72 of sched.c and contains a total of 1024 long words. Its location in physical memory can be seen in Figure 5-24 below. At this point this is the stack used by the kernel itself. The addresses given in the figure are approximate values, which are related to the actual setup parameters at compile time. These address locations can be found in the system.map file generated when the kernel is compiled.

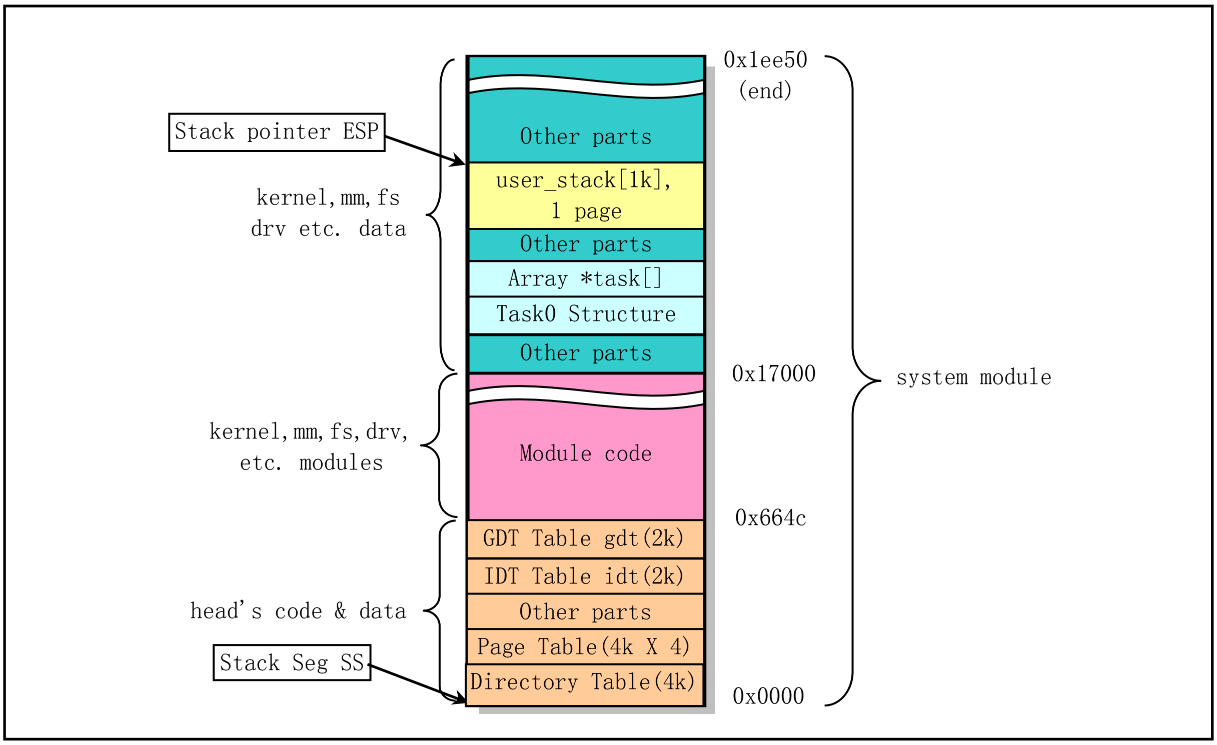


Figure 5-24 Stack diagram used by the kernel when entering protection mode

Initialization (main.c)

In the init/main.c program, the system always uses the above stack before executing the move_to_user_mode() code to transfer control to task 0. After the move_to_user_mode() is executed, the code of main.c is "switched" to task 0. By executing the fork() system call, init() in main.c will be executed in task 1 and use the stack of task 1. Main(), after being "switched" to task 0, continues to use the kernel's own stack as the user-mode stack for task 0. A detailed description of the stack used by task 0 is described later.

5.8.2 Stack of tasks

Each task has two stacks for the execution of user-mode and kernel-mode programs, and is called the user-mode stack and the kernel-state stack, respectively. In addition to being in different CPU privilege levels, the main difference between the two stacks is that the kernel mode stack of the task is small, and the amount of data saved cannot exceed (4096 – task data structure block) bytes, which is about 3K bytes. The user-mode stack of the task can extend within the user's 64MB space.

When running in user mode

Each task (except task 0 and 1) has its own 64MB address space. When a task (process) has just been created, its user state stack pointer is set nearly at the end of its address space (64MB top). In fact, the end part also includes the parameters of the execution program and environment variables, and then the user stack space, as shown in Figure 5-25. The application always uses this stack when it is running in user mode. The physical memory actually used by the stack is determined by the CPU paging mechanism. Since Linux implements copy-on-write, if the process and its parent process do not use the stack after the process is created, the two share the physical memory page corresponding to the same stack. The kernel memory manager allocates new memory pages for the write process only when one of the processes performs a stack write operation (such as a push operation). The user stacks for tasks 0 and 1 are special, as explained below.

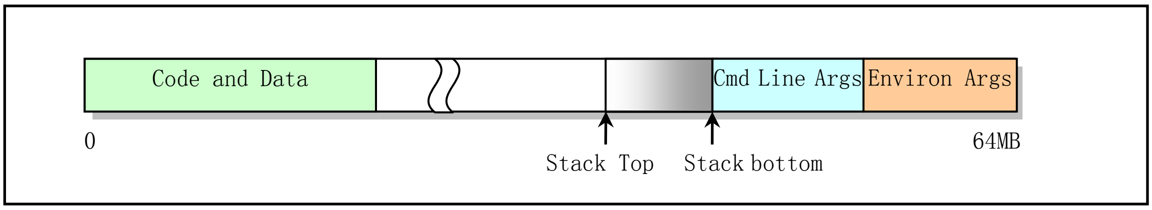


Figure 5-25 User state stack in logical space

When running in kernel mode

Each task has its own kernel-mode stack for the task to execute during execution in kernel code. The location in its linear address is specified by the ss0 and esp0 fields in the task TSS segment. ss0 is the segment selector for the task kernel state stack, and esp0 is the stack stack low pointer. Therefore, whenever a task is transferred from user code into kernel code, the kernel state stack of the task is always empty. The task kernel state stack is placed at the end of the page where its task data structure is located, that is, on the same page as the task's task data structure (task_struct). This is when the new task is created, the fork() program is set in the kernel-level stack fields (tss.esp0 and tss.ss0) of the task tss segment, see kernel/fork.c, line 92:

```
    p->tss.esp0 = PAGE_SIZE + (long)p;
    p->tss.ss0 = 0x10;
```

Where p is the task data structure pointer for the new task and tss is the task state segment structure. The kernel requests memory for a new task to hold its task_struct structure data, and the tss structure (segment) is a field in the task_struct. The kernel stack segment value tss.ss0 for this task is also set to 0x10 (ie, the kernel data segment selector), while tss.esp0 points to the end of the page where the task_struct structure is saved, as shown in Figure 5-26. In fact, tss.esp0 is set to point to the top byte of the page (outside) (at the bottom of the stack in the figure). This is because the Intel CPU performs the stack operation by first decrementing the stack pointer esp value and then saving the contents of the stack at the esp pointer.

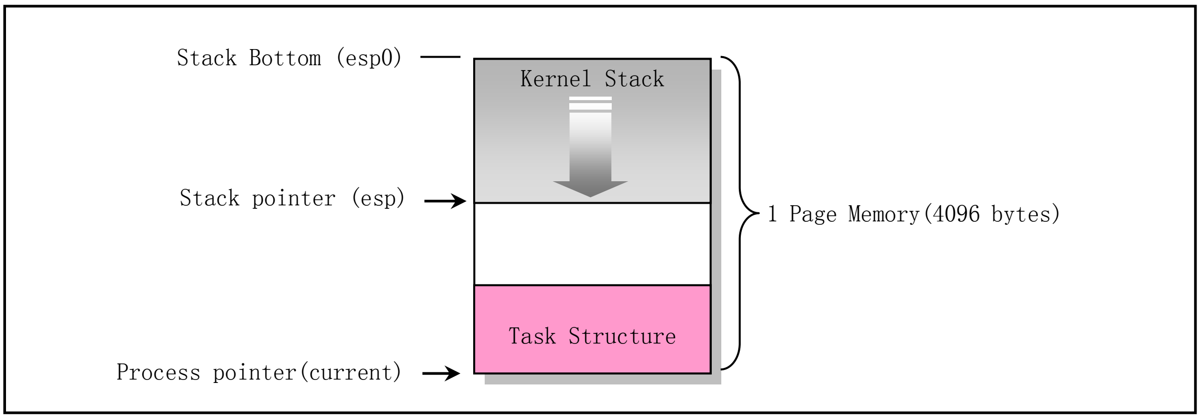


Figure 5-26 Kernel state stack diagram of a process

Why can a page of memory used to save the task data structure from the main memory area be set to be the data in the kernel data segment, that is, why can tss.ss0 be set to 0x10? This is because the user's kernel state stack still belongs to the kernel data space. We can illustrate this from the length of the kernel code segment. At the end of the head.s program, the kernel code and data segment descriptors are set respectively, and the segment length are all set to 16MB. This segment length is the maximum physical memory length that the

Linux 0.12 kernel can support (see head.s, notes beginning with line 110). Therefore, the kernel code can address anywhere in the entire physical memory range, including the main memory area. Whenever a task executes a kernel code and needs to use its kernel stack, the CPU uses the TSS structure to set its kernel state stack to consist of two values, tss.ss0 and tss.esp0. When the task is switched, the kernel stack pointer esp0 of the old task is not saved. For the CPU, these two values are read-only. So every time a task enters kernel mode execution, its kernel state stack is always empty.

Stacks of task 0 and task 1

The stack of task 0 (idle process) and task 1 (init process) is special and needs special explanation. The code segments and task segments of task 0 and 1 are the same, and the limit length is also 640 KB, but they are mapped to different linear address ranges. The segment base address of task 0 starts from linear address 0, while the segment base address of task 1 starts from 64 MB, but they all map to the physical address 0--640KB range. This address range is where the kernel code and basic data are stored. After the move_to_user_mode() is executed, the kernel state stacks of task 0 and task 1 are respectively located at the end of the page where the respective task data structures are located, and the user state stack of task 0 is the stack used before entering the protected mode, at the location of the user_stack[] array in sched.c. Since Task 1 replicated the user stack for Task 0 at the time of creation, Task 0 and Task 1 share the same user stack space at the beginning. However, when task 1 starts running, since the page table entry mapped by task 1 to user_stack[] is set to read-only, task 1 will cause a write page exception when performing a stack operation. Thus, the kernel will use the copy-on-write mechanism to allocate the main memory area page for task 1 as the stack space. Only then will Task 1 start using its own separate user stack memory page. Therefore, the stack of task 0 needs to be kept "clean" until task 1 actually starts to use, that is, task 0 cannot use the stack at this time to ensure that the copied stack page does not contain the data of task 0.

The kernel state stack for task 0 is specified in its manually set initialization task data structure. Its user-mode stack is set in the stack before the simulated iret return when executing move_to_user_mode(), as shown in Figure 5-22. We know that when a privilege level change occurs, the destination code uses the new privilege level stack, and the original privilege level code stack pointer is saved in the new stack. Therefore, the task 0 user stack pointer is first pushed into the stack currently at privilege level 0, and the code pointer is also pushed onto the stack. The execution of the IRET instruction then implements the transfer of control from the privilege level 0 code to the privilege level 3 code of task 0. In this manually set stack, the original esp value is set to still be the original position in user_stack, and the original ss segment selector is set to 0x17, that is set to the data segment selector in the user local table LDT. Then the task 0 code segment selector 0x0f is pushed onto the stack as the selector of the original CS segment in the stack, and the pointer of the next instruction is pushed onto the stack as the original EIP. In this way, by executing the IRET instruction, it can be "returned" to the code of task 0 to continue execution.

5.8.3 Switching between the kernel stack and user stack

In Linux 0.12, all interrupt service routines belong to the kernel code. If an interrupt is generated while the task is executing in user code, then the interrupt will cause a change in the CPU privilege level from level 3 to level 0, at which point the CPU will switch between the user state stack and the kernel state stack. The CPU will get the segment selector and offset value of the new stack from the TSS of the current task. Because the interrupt service routine is in the kernel and belongs to level 0 privilege level code, the 48-bit kernel state stack pointer is obtained from the ss0 and esp0 fields of the TSS. After locating the new stack (the kernel state stack), the CPU first pushes the original user state stack pointers ss and esp into the kernel state stack, and then pushes the contents of the flag register eflags and the return positions cs and eip into the kernel state stack.

The system call is a software interrupt, so when a task calls the system call, it enters the kernel and executes the interrupt service code in the kernel. At this point the kernel code will operate using the kernel-mode stack of the task. Similarly, when entering the kernel, as the privilege level changes (from user mode to kernel mode), the stack segment and stack pointers of the user-mode stack and eflags are stored in the kernel-state stack of the task. When the IRET instruction is executed to exit the kernel and return to the user program, the user state stack and eflags are restored. This process is shown in Figure 5-27.

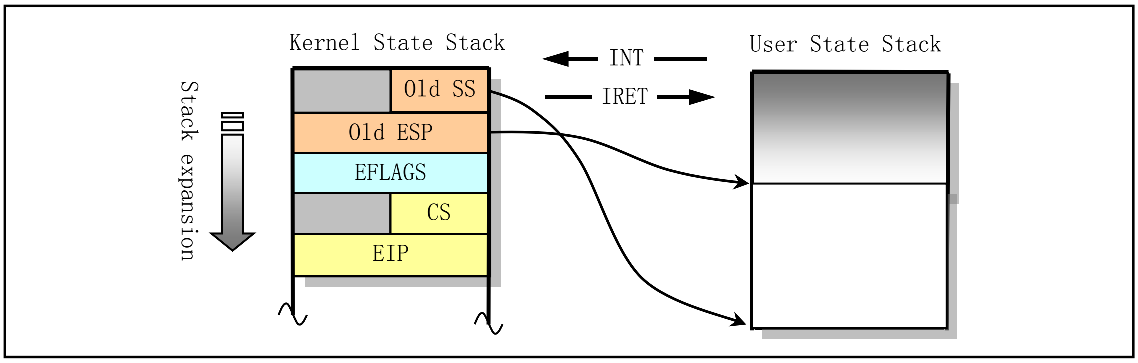


Figure 5-27 Kernel state and user state stack switching

If a task is running in kernel mode, then the stack switch operation is no longer needed if the CPU responds to the interrupt. Because the kernel code that the task is running at this time is already using the kernel state stack, and does not involve changes in priority levels. Therefore, the CPU only pushes the eflags and interrupt return pointers cs and eip into the current kernel state stack, and then executes the interrupt service process.

5.9 File System for Linux 0.12

File system support is required for the kernel to function properly. The root file system is used to provide the most basic information and support to the kernel. That is, when the Linux system boots and starts, the default file system is the root file system. This includes some of the operating system's minimum configuration files and command execution programs. For UNIX-like file systems used in Linux systems, it mainly includes some specified directories, configuration files, device drivers, executable programs, user applications, data or text files. These generally include the following subdirectories and files:

etc/	The directory mainly contains some system configuration files;
dev/	Contains device special files for file operation statement on devices;
bin/	Store system execution programs. Such as sh, mkfs, fdisk, etc.;
usr/	This directory stores library functions, manuals, and other files;
usr/bin	These directories store commands commonly used by users;
var/	This directory is used to store system runtime data or log information.

The device that holds the file system is the file system device. For example, for the commonly used Windows 10 operating system, the hard disk C is the file system device, and the files stored on the hard disk according to certain rules constitute the file system. Windows 10 has NTFS, FAT32 and other file systems of different formats. Linux also has EXT2, EXT3 and other file systems of different formats to choose from. The file system supported by the Linux 0.12 kernel is the MINIX 1.0, which is created by Andrew Tanenbaum, the author of MINIX operation system. But the most widely used on current Linux systems is the ext3 or ext4 file system.

For the Linux 0.12 system running on a floppy disk described in Chapter 1, it consists of two simple floppy disks: the bootimage disk and the rootimage disk. Bootimage is the boot image file, which mainly includes the disk boot sector code, operating system loader and kernel binary code. Rootimage is the root file system used to provide the most basic support to the kernel. These two disks are equivalent to a bootable DOS operating system disk.

When the Linux boot disk loads the root file system, the root file system is loaded from the specified device according to the device number in a word (ROOT_DEV) at the 509th and 510th bytes of the boot sector on the disk. If the device number is 0, it means that the root file system needs to be loaded from the current drive where the boot disk is located. If the device number is a hard disk partition device number, the root file system is loaded from the specified hard disk partition.

Currently, the Filesystem Hierarchy Standard, which is maintained by the Linux Foundation (LF), is the specification file in this regard. Based on the original UNIX file system organization structure and content, the FHS specification has been studying the standardization of the structure and content of the Linux distribution system file system since 1994. FHS is now part of the official ISO LSB standard proposed by the Linux Standard Library (LSB).

5.10 Directories of kernel source code

Since the Linux kernel is a single-core mode system, almost all of the code in the kernel is closely related. The data dependencies and invocation relationships between them are very close. So when reading a source code file, you often need to refer to other files. Therefore, it is necessary to familiarize yourself with the directory structure and arrangement of the source code files before starting to read the kernel source code.

Here we first list the complete source directory of the kernel, including its subdirectories. Then introduce the main functions of the programs included in each directory one by one, so that the entire kernel source code arrangement can establish a general framework in our minds, so as to facilitate the source code reading work in the next chapter.

When we use the tar command to unpack the linux-0.12.tar.gz file, the kernel source files are placed in the linux/ directory. The directory structure is shown in Figure 5-28:

```
linux
├── boot          Sytem boot assembly programs
├── fs            File system files
├── include       Header files (*.h)
│    ├── asm          Files related to the CPU architecture
│    ├── linux        Linux kernel specific header files
│    └── sys          System data structures
├── init          Kernel initialization program
├── kernel        Kernel process scheduling, signal, system-calls, etc.
│    ├── blk_drv      Block device driver
│    ├── chr_drv      Character device driver
│    └── math         Math coprocessor simulation programs
├── lib           Kernel specific library
├── mm            Memory management programs
└── tools         Tool program for generating kernel image file
```

Figure 5-28 Linux kernel source code directory structure

The source code directory for this version of the kernel contains 14 subdirectories, including a total of 102 code files. The contents of these subdirectories are described one by one below.

5.10.1Kernel home directory linux

The linux directory is the main directory of the source code. In addition to including all 14 subdirectories, the home directory contains a unique Makefile. This file is the parameter configuration file for the compile aid software make. The main purpose of the make tool is to automatically determine which files need to be recompiled in a system with multiple source files by identifying which files have been modified. Therefore, the make tool is a management software for a program project.

The Makefile in the linux directory also nests the Makefiles contained in all subdirectories. This way, make will recompile any files in the linux directory (including subdirectories) that have been modified. So in order to compile all the source code files of the entire kernel, just run the make software once in the linux directory.

5.10.2 The boot program directory

The boot directory contains three assembly language files, which are the first programs compiled in the kernel source code file. The main function of these three programs is to boot the kernel when the computer is powered up, load the kernel code into memory, and do some system initialization before entering the 32-bit protection mode.

- The bootsect.s program is a disk boot block program that resides in the first sector of the disk after compilation (boot sector, 0 track (cylinder), 0 head, 1st sector). After the PC is powered up and the ROM BIOS self-test, it will be loaded into the memory 0x7C00 for execution.
- The setup.s program is primarily used to read the machine's hardware configuration parameters and move the kernel module system to the appropriate memory location.
- The head.s program will be compiled and connected to the foremost part of the kernel system module, mainly for the hardware device's probe settings and the initial setup of the memory management page.

The bootsect.s and setup.s programs need to be compiled using the as86 software, using the as86 assembly language format (similar to Microsoft's). Head.s needs to be compiled with GNU as, using the assembly language of AT&T format. These two assembly languages are briefly described in the code comments in the next chapter and in the descriptions that follow the code listing.

5.10.3 File System Directory fs

The file system of the Linux 0.12 kernel uses the version 1.0 MINIX file system, which is because Linux was developed on the MINIX system. The MINIX file system is easy to cross-compile, and Linux partitions can be loaded from MINIX. Although the MINIX file system is used, Linux handles it differently than the MINIX system. The main difference is that MINIX uses a single-threaded approach to the file system, while Linux uses a multi-threaded approach. Due to the multi-threaded processing method, Linux programs must deal with the race conditions and deadlocks caused by multi-threading. Therefore, the Linux file system code is much more complicated than the MINIX system. In order to avoid the occurrence of competitive conditions, the Linux system has strictly checked the resource allocation. And when running in kernel mode, if the task does not actively sleep (call sleep()), the kernel will not be allowed to switch tasks.

The fs/ directory is a directory of file system implementations and contains a total of 18 C programs. The main references and dependencies between these programs are shown in Figure 5-29. Each box in the figure represents a file, placed from top to bottom in a basic reference relationship. The file name is omitted from the suffix .c. The program files in the virtual box are not part of the file system. Lines with arrows indicate reference relationships, and thick lines indicate mutual reference relationships.

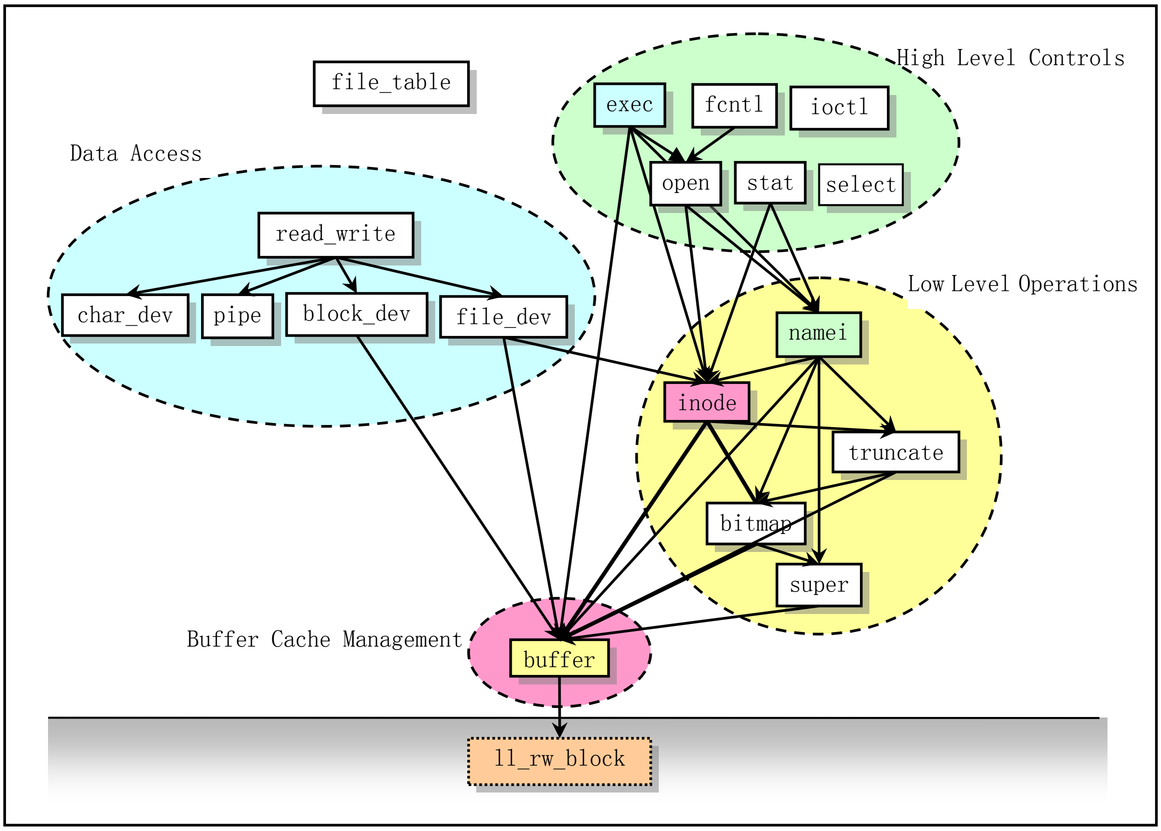


Figure 5-29 The relationship of each program in the fs directory.

As can be seen from the figure, the program in this directory can be divided into four parts: cache management, low-level file operations, file data access, and file high-level functions. When annotating the files in this directory, we will also describe them in four parts.

For the file system, we can think of it as an extension of the memory cache. All access to data in the file system needs to be first read into the cache. The programs in this directory are mainly used to manage the allocation of buffer blocks in the cache and the file system on the block device. The program that manages the cache is buffer.c, while other programs are mostly used for file system management.

- The file_table.c file currently has only one file handle (descriptor) structure array defined.
- The ioctl.c file will reference the functions in kernel/chr_drv/tty.c to implement the io control function of the character device.
- The exec.c file mainly contains an executive function do_execve(), which is the main function in all exec() function clusters.
- The fcntl.c program is used to implement the system call function for file i/o control.
- The read_write.c program is used to implement file read/write and locate three system call functions.
- The stat.c program implements two system calls that get file state information.
- The open.c program mainly contains system call functions that implement modification of file attributes and creation and closing of files.
- The char_dev.c mainly contains the character device read and write function rw_char().
- The pipe.c program contains pipe read and write functions and system calls to create pipes.
- The file_dev.c program contains file read and write functions based on the i-node and descriptor structure.
- The namei.c program mainly includes operation functions and system calls for directory names and file names in the file system.
- The block_dev.c contains block data read and write functions.
- The inode.c program contains functions for file system i-node operations.
- The truncate.c program is used to release the device data space occupied by files when deleting files.
- The bitmap.c file is used to process bitmaps of i-nodes and logical blocks in the file system.
- The super.c program contains handlers for file system superblocks.
- The buffer.c program is mainly used to process the memory cache.
- The select.c program is mainly used to effectively handle the problem of simultaneous I/O operations on multiple files.

The ll_rw_block in the virtual box is the underlying read function of the block device. It is not in the fs directory, but the block device read and write driver function in kernel/blk_drv/ll_rw_block.c. Putting it here just makes us clearly see that the file system needs to read and write the data in the block device through the high-speed buffer cache and the block device driver (ll_rw_block()). The file system's program itself does not directly interact with the driver of the block device.

In the process of annotating the program, we will additionally give the call hierarchy between the main functions in these files.

5.10.4 Header file directory include

There are a total of 36 .h header files in the header file directory. There are 13 in the main directory, 4 in the asm subdirectory, 11 in the Linux subdirectory, and 8 in the sys subdirectory. The respective functions of these header files are briefly described below. See the Notes on Header for specific actions and information.

- <a.out.h> The a.out header file defines the a.out executable file format and some macros.
- <const.h> The constant symbol file currently defines only the flags of i_mode field in the i-node.
- <ctype.h> The character type header file. Defines some macros for character type conversion.
- <errno.h> Error number header file. Contains various error numbers in the system. (Linus was introduced from minix).
- <fcntl.h> File control header file. The definition of the operation control constant symbol used for the

file and its descriptors.

- <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal manipulation function prototypes.
- <stdarg.h> Standard parameter header file. Define a list of variable parameters in the form of macros. It mainly describes one type (va_list) and three macros (va_start, va_arg and va_end) for the vsprintf, vprintf, and vfprintf functions.
- <stddef.h> The standard definition header file. NULL, offsetof(TYPE, MEMBER) is defined.
- <string.h> String header file. Mainly defines some embedded functions about string operations.
- <termios.h> Terminal input and output function header file. It mainly defines the terminal interface that controls the asynchronous communication port.
- <time.h> Time type header file. The most important of these is the definition of the tm structure and some function prototypes related to time.
- <unistd.h> Linux standard header file. Various symbol constants and types are defined and various functions are declared. If __LIBRARY__ is defined, it also includes the system call number and the inline assembly _syscall0().
- <utime.h> User time header file. The access and modification time structures and the utime() prototype are defined.

include/asm -- Architecture related header file subdirectory

These header files primarily define data structures, macro functions, and variables that are closely related to the CPU architecture.

- <asm/io.h> Io header file. Defines the function that operates on the io port in the form of a macro's embedded assembler.
- <asm/memory.h> Memory copy header file. Contains memcpy() embedded assembly macro functions.
- <asm/segment.h> Segment operation header file. An embedded assembly function is defined for segment register operations.
- <asm/system.h> System header file. An embedded assembly macro that defines or modifies descriptors/interrupt gates, etc. is defined.

include/linux -- Linux kernel dedicated header file subdirectory

- <linux/config.h> Kernel configuration header file. Define keyboard language and hard disk type (HD_TYPE) options.
- <linux/fdreg.h> Floppy disk file. Contains some definitions of floppy disk controller parameters.
- <linux/fs.h> File system header file. Define the file table structure (file, buffer_head, m_inode, etc.).
- <linux/hdreg.h> Hard disk parameter header file. Define access to the hard disk register port, status code, partition table and other information.
- <linux/head.h> Head header file. A simple structure for the segment descriptor is defined, along with several selector constants.
- <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly used functions of the kernel.
- <linux/math_emu.h> Coprocessor emulation header file. A coprocessor data structure and a floating point representation structure are defined.
- <linux/mm.h> Memory management header file. Contains page size definitions and some page release function prototypes.
- <linux/sched.h> The scheduler header file defines the task structure task_struct, the data of the initial

task 0, and some embedded assembly function macro statements about the descriptor parameter settings and acquisition.

- <linux/sys.h> The system calls the header file. Contains 72 system call C function handlers, starting with 'sys_'.
- <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial communication.

include/sys -- System-specific data structure subdirectory

- <sys/param.h> Parameter file. Some hardware-related parameter values are given.
- <sys/resource.h> Resource file. Contains information on the limits and utilization of system resources used by processes.
- <sys/stat.h> File status header file. Contains file or file system state structures stat{} and constants.
- <sys/time.h> The timeval structure and the itimerval structure are defined.
- <sys/times.h> Defines the running time structure tms and the times() function prototype in the process.
- <sys/types.h> Type header file. The basic system data types are defined.
- <sys/utsname.h> System name structure header file.
- <sys/wait.h> Wait header file. Define the system call wait() core waitpid() and related constant symbols.

5.10.5 init -- kernel initialization program directory

This directory contains only one file, main.c, for performing all initialization work on the kernel. Then control transfers to user mode to create a new process and run the shell program on the console device.

The program first allocates the buffer memory capacity based on the amount of physical memory in the machine. If the system also sets up a virtual disk to use, it also leaves space behind the buffer memory. All hardware initialization is then performed, including manually creating the first task (task 0) and setting the interrupt enable flag. After the control moves from the kernel state to the user state, the system first calls the create process function fork() to create a process for running init(). In this child process, the system will set up the console environment and generate a child process to run the shell program.

5.10.6 kernel -- kernel program main directory

The linux/kernel directory contains a total of 32 files and 3 subdirectories (blk_drv, chr_drv, and math). There are 12 code files and a Makefile in the kernel directory, 5 files in the subdirectory kernel/blk_drv, 6 files in kernel/chr_drv, and 9 files in kernel/math.

All programs that handle tasks are stored in the kernel/ directory, which including forks, exits, schedulers, and some system callers. It also includes the procedure services of handling interrupt exceptions and traps. Subdirectories include low-level device drivers such as get_hd_block and tty_write. Because of the complex calling relationships between the code in these files, the reference relationship diagram between the files is not detailed here. However, it is still possible to perform an approximate classification, as shown in Figure 5-30.

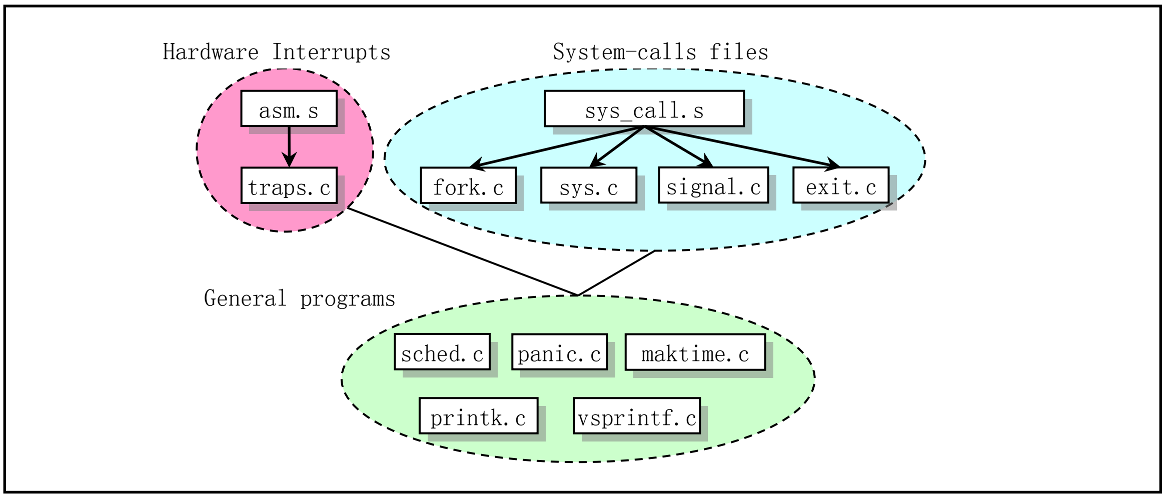


Figure 5-30 Call hierarchy of each file

- The asm.s program is used to handle interrupts caused by system hardware exceptions.
- The traps.c file is used for the actual handling of hardware exceptions. In each interrupt processing, the corresponding C language processing function in traps.c will be called separately.
- The exit.c program mainly includes system calls for processing process termination, including process release, session (process group) termination, and program exit processing functions, as well as system call functions such as killing processes, terminating processes, and suspending processes.
- The fork.c program gives two C language functions used in the sys_fork() system call: find_empty_process() and copy_process().
- The mktime.c program contains a time function mktime() used by the kernel to calculate the number of seconds from 0:00 on January 1, 1970 to the day of booting. It is only called once in init/main.c.
- The panic.c file contains a function panic() that displays kernel error messages and stops.
- The printk.c program contains a kernel-specific information display function printk().
- The sched.c program includes basic functions for scheduling (sleep_on, wakeup, schedule, etc.), as well as some simple system call functions. There are also several floppy disk operation functions related to timing.
- The signal.c program includes four system calls for signal processing and a function do_signal() that processes the signal in the corresponding interrupt handler.
- The sys.c program includes many system call functions, some of which are not yet implemented.
- The system_call.s program implements the interface processing of the Linux system call (int 0x80). The actual processing is contained in the corresponding C language functions of each system call. These processing functions are distributed throughout the Linux kernel code.
- The vsprintf.c program implements a string formatting function that is now included in the standard library functions.

kernel/blk_drv -- Block device driver subdirectory

Typically, a user accesses a device through a file system, and a device driver implements an access interface for the file system. When a block device is used, a cache mechanism is used between the user process and the block device in order to efficiently use the data on the block device due to its large data throughput. When accessing data on a block device, the system first reads the data on the block device into the cache in the form of a data block, and then copies the data into user's buffer space.

The blk_drv subdirectory contains a total of 4 c files and 1 header file. The header file blk.h is dedicated to the block device program and is therefore placed with the C file. The approximate relationship between these documents is shown in Figure 5-31.

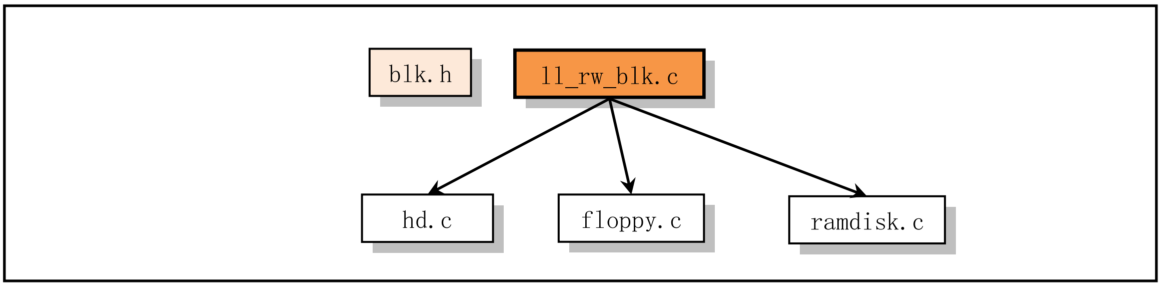


Figure 5-31 Hierarchical relationship of files in the blk_drv directory.

- blk.h block device special header file. The block device structure and data block request structure shared by several C programs are defined in it.
- The hd.c program mainly implements the underlying driver function for reading/writing hard disk data blocks, mainly the do_hd__request() function.
- The floppy.c program mainly implements the read/write drive function for floppy data blocks, mainly the do_fd_request() function.
- The ramdisk.c program is a memory virtual disk driver. The main function is do_rd_request(). A virtual disk device is a technology that uses physical memory to simulate an actual disk storage medium. Its main advantage is that it can greatly improve the speed of data access operations.
- The program in ll_rw_blk.c implements the low-level block device data read/write function ll_rw_block(). All other programs in the kernel use this function to access data from block devices. You will see that the function is called in many places where the block device data is accessed, especially in the cache file fs/buffer.c.

kernel/chr_drv -- Character device driver subdirectory

The character device subdirectory contains a total of 4 C language programs and 2 assembler files. These files enable drivers for serial port rs-232, serial terminals, keyboards, and console terminals. Figure 5-32 is the approximate call hierarchy between these files.

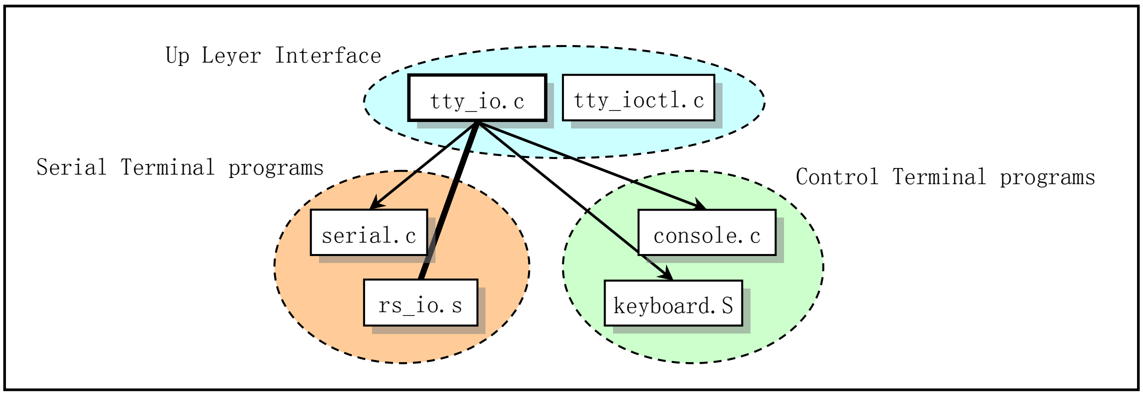


Figure 5-32 Schematic diagram of the relationship between character device programs

- The tty_io.c program contains the tty character device read function tty_read() and the write function tty_write(), which provides the upper layer access interface for the file system. It also includes the C function do_tty_interrupt() called during serial interrupt processing. This function will be called when the interrupt type is a read character.

- The console.c file mainly contains the console initialization program and the console write function con_write(), which is used by the tty device. It also includes an initialization setup program con_init() for screen display and keyboard interrupts.
- The rs_io.s program is used to implement interrupt handlers for two serial interfaces. The interrupt handler processes each of the four interrupt types retrieved from the interrupt identifier register (port 0x3fa or 0x2fa) and calls do_tty_interrupt() in the code that handles the interrupt type as a read character.
- Serial.c is used to initialize the asynchronous serial communication chip UART and set the interrupt vector of the two communication ports. Also included is the rs_write() function that tty uses to output to the serial port.
- The tty_ioctl.c program implements tty's io control interface function tty_ioctl(), and reads and writes to the termio(s) terminal io structure, and is called in the fs/ioctl.c program that implements the system call sys_ioctl().
- The keyboard.S program mainly implements the keyboard interrupt handler procedure keyboard_interrupt.

kernel/math -- Coprocessor emulator subdirectory

This subdirectory contains the math coprocessor emulation handler file, which has a total of 9 C files. When there is no math coprocessor in the machine, and the CPU executes the coprocessor instruction, it will cause the device not exist interrupt INT 7. Therefore, using this interrupt, software can be used to simulate the function of the coprocessor. These programs are closely related to the CPU hardware, but have little to do with the rest of the kernel implementation. The relevant knowledge in the program is very helpful for implementing system-level programs such as multiprocessor programming and assembly and disassembly.

- The math_emulate.c program is the main program of coprocessor emulation, which implements the device not exist exception handler, floating point instruction emulation main function do_emu() and other auxiliary functions.
- The error.c program is used to process the error signal sent by the coprocessor. Its main function is math_error().
- The ea.c program is used to calculate the effective address used by the operands in the instruction when simulating floating-point instructions.
- The convert.c program is used to implement the data type conversion operation between the user data format and the temporary real number format in the simulation calculation process.
- The add.c program implements the addition in the simulation process and performs the conversion between the mantissa symbolization and the non-symbolization.
- The compare.c program is used to simulate the size of two temporary real numbers in the coprocessor compare accumulator.
- The get_put.c program implements access to data in the user's memory.
- The mul.c program is used to simulate multiply instructions in the coprocessor.
- The div.c program is used to simulate the division of the coprocessor.

5.10.7 lib -- Kernel library directory

Unlike normal user programs, kernel code cannot use standard C libraries and other library code. The main reason is that the complete C function library is large and complex to implement. Therefore, there is a special lib/ directory in the kernel source that provides some functions that the kernel needs to use. The kernel function library is used to provide call support for the kernel initialization program init/main.c running in user mode (process 0, 1). It is exactly the same as the implementation of a normal static library. Readers can learn the basic structure of the general libc library.

There are 12 C language files in the lib/ directory. Except for a malloc.c programmed by Mr. tytso, other files are very short, and some have only one or two lines of code. They implement interface functions for some system calls. These files mainly include the exit function _exit(), the close file function close(fd), the copy file descriptor function dup(), the file open function open(), the write file function write(), the execute program function execve(). The memory allocation function malloc(), wait for the child process state function wait(), create the session system call setsid(), and all string manipulation functions implemented in include/string.h.

5.10.8 **mm -- Memory Management Directory**

This directory contains 3 code files. It is mainly used to manage the application's use of the main memory area, and implements the mapping operation from the logical to linear address and linear to physical memory address. And through the memory paging mechanism, a correspondence is established between the virtual memory page and physical memory page of the main memory area. At the same time, virtual storage technology has also been realized.

The Linux kernel handles memory in both fragmented and paged ways. The first is to divide the 80X86 4G virtual address space into 64 segments (64MB per segment). The kernel program occupies the first segment and the physical address is the same as the segment linear address. Then each task is assigned a segment to use. The paging mechanism is used to map the specified physical memory page into the segment, detect any duplicate copies created by the fork, and perform a copy-on-write mechanism.

The page.s file includes the memory page abort (int 14) handler, which is mainly used to handle page exceptions caused by page faults and page protection caused by accessing illegal addresses.

The memory.c program includes the function mem_init() that initializes the memory, and the do_no_page() and do_wp_page() functions called by the memory interrupt procedure in page.s. When you create a new process and perform a copy process operation, you use a memory handler to allocate the management memory space.

The swap.c program is used to manage page swapping between physical pages in main memory and high-speed secondary storage (hard disk) space. When the main memory space is not enough, you can save the temporarily unused memory page to the hard disk. When a page fault exception occurs, first check whether the requested page is in the hard disk swap space. If it exists, the page is directly read into the memory from the swap space.

5.10.9 **tools -- Kernel Tools Directory**

The build.c program in this directory is used to build a complete kernel module. It combines the objects compiled in the Linux directory into a runnable kernel image file image. The specific functions can be found in the next chapter.

5.11 The Kernel Code and User Programs

In Linux systems, the kernel can provide service support for user programs in two ways. One is through the system call interface, that is, the interrupt calls int 0x80; on the other hand, through the development environment library functions or kernel library functions. However, the kernel library functions are only used by task 0 and task 1 created by the kernel. They eventually still call the system call. Therefore, the kernel actually only provides a unified service interface of system calls to all user programs or processes. The implementation method of the kernel library function code in the lib/ directory is basically the same as in the C function library libc. In order to use kernel resources, the kernel system call is eventually invoked through inline assembly code, as shown in Figure 5-4.

System calls are primarily used for system software programming or for implementation of library functions. The program developed by the average user accesses kernel resources by calling functions in libraries such as libc. The functions or resources in these libraries are often referred to as application programming interfaces (APIs). It defines a set of standard programming interfaces used by the application. By calling functions in these libraries, the application code can perform a variety of common tasks, such as opening and closing access to files or devices, performing scientific calculations, error handling, and accessing system information such as group and user ID numbers.

In the UNIX-like operating system, the most commonly used API interface based on the POSIX standard, and Linux is no exception. The difference between API and system call is that in order to implement an application interface standard, such as POSIX, the API may correspond to a system call, or may be implemented by several system call functions. Of course, some API functions may not need to use system calls at all, that is, do not use the services provided by the kernel. Therefore, the function library can be regarded as a main interface that implements the POSIX standard. The application does not care what is the relationship between it and system call. No matter how much difference between the system calls provided by the two operating systems, as long as it provides compliance with the same API standard, the application can be portable between these operating systems.

The system call is the highest level of the interface between the kernel and the outside world. In the kernel, each system call is often implemented as a macro and has a serial number (defined in the include/unistd.h header file). Applications should not use system calls directly, otherwise the portability of the program will be worse. Therefore, the current Linux standard base (LSB) and many other standards do not recommend that applications directly access system call macros. Documentation for system calls can be found in Part 2 of the online manual for the Linux operating system.

Library files typically include user-level functions such as input/output and string manipulation functions that are not provided by the C language to perform advanced functions. Some library functions are just enhancements to system calls. For example, the standard I/O library functions fopen and fclose provide similar functionality to the system calls open and close, but at a higher level. In this case, system calls usually provide slightly better performance than library functions, but library functions provide more functionality and are more error-detecting. The library functions provided by the system can be found in the online manual section 3 of the operating system.

5.12 linux/Makefile

From this section, we begin to annotate the kernel source code files. First let's comment on the first file Makefile encountered in the Linux directory. Subsequent sections will be annotated in a similar description structure.

5.12.1 Function Description

The Makefile is equivalent to a batch file during program compilation. It is the default compilation settings input file for the utility program make at runtime. Just type the make command in the directory containing the Makefile, it will call the compiler and linker to compile, link, or install the source code or the target code file according to the settings in the Makefile.

The make utility automatically determines which files in a program package containing multiple source files need to be recompiled and issues commands to compile those program files. So before using make, we need to write a Makefile text file. This file describes the relationship between the programs in the entire package and gives specific control commands for each file that needs to be updated. Typically, executable target are updated based on their object files, which are created by the compiler. Once you have written a suitable Makefile, you can perform all necessary recompilation tasks after each modification of some source code files in the program package. The make tool uses the Makefile file and the last-modification time of code files to determine which files need to be updated. For each file that needs to be updated, it will issue the appropriate command based on the information in the Makefile. In the Makefile, lines starting with '#' are comment lines. The '=' assignment statement at the beginning of the file defines abbreviations for some parameters or commands.

The main function of this Makefile in the kernel directory is to instruct the make program to eventually connect and merge all kernel compiled code into a runnable kernel image file using the build executable in the tools/directories that are independently compiled. The specific process is: (1) use the 8086 assembler to compile the bootsect.s and setup.s in boot/, and generate their respective objects. (2) Compile all other programs in the source code using GNU compiler gcc/gas, and link to generate module system. (3) Finally, use the build tool to combine the three parts into a kernel image file image.

The build tool is a stand-alone executable compiled from the tools/build.c source file. It is not compiled and linked into the kernel code. It is only used as a tool in the process of building a kernel image file. The basic compilation link/combination structure is shown in Figure 5-33.

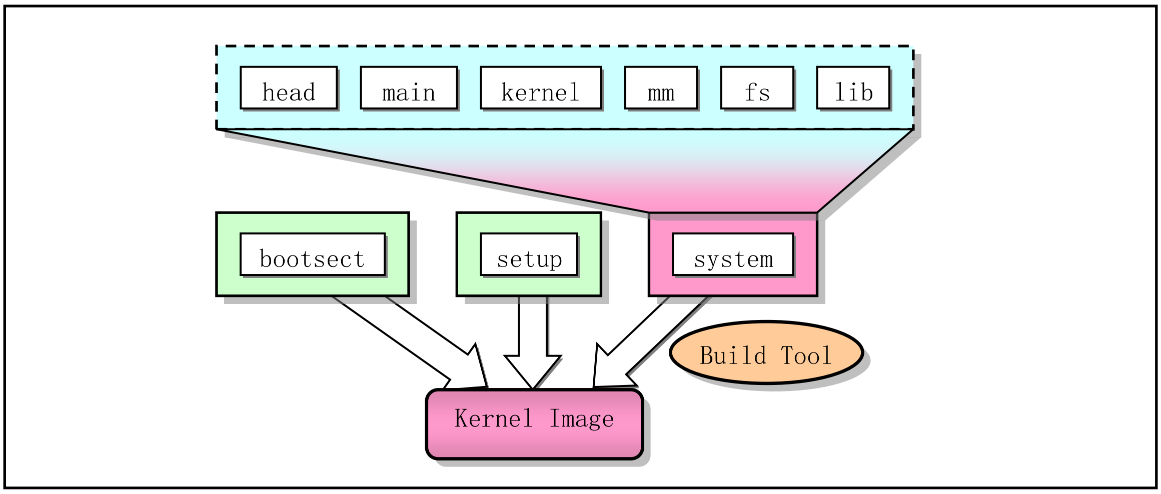


Figure 5-33 Kernel compilation link/combination structure

In the Linux kernel source code, except for the tools/, init/, and boot/ directories, each of the subdirectories contains a corresponding Makefile, which is identical in structure. Due to space limitations, only one annotation of the Makefile is given in the book. Program 5-1 is a detailed comment of the file.

Note that the statement with line number in the source file is the original statement, and statement without line number is the comment of the author of this book.

5.12.2 Program Annotation

Program 5-1 linux/Makefile

```
1 #
2 # if you want the ram-disk device, define this to be the
3 # size in blocks.
4 #
  # Define the size of the block if you want to use a RAM disk device. The default RAMDISK
  # is not defined here (commented out), otherwise gcc will be compiled with the option
  # '-DRAMDISK=512', see line 13 below.
5 RAMDISK = #-DRAMDISK=512
6
7 AS86    =as86 -0 -a       # 8086 assembly and linker. The meaning of parameter is: -0
8 LD86    =ld86 -0          # generates 8086 object; -a code compatible with gas and gld.
9
10 AS      =gas              # GNU assembler and linker.see chapter 3 for more information.
11 LD      =gld
  # gld options: -s all symbolic info omitted in output file; -x removes all local symbols;
  # -M indicates that a link map is required, which is a memory address map generated by the
  # linker, which lists the location information that the code block is loaded into memory.
12 LDFLAGS =-s -x -M

  # gcc is the GNU compiler. For UNIX-like scripts, when you reference an identifier, you
  $ need to precede it with a sign and enclose the identifier in parentheses.
13 CC      =gcc $(RAMDISK)

  # gcc options: -Wall prints all warnings; -O optimizes code. '-f flag' specifies flag.
  # Where, -fstrength-reduce is used to optimize loop statements; -fomit-frame-pointer
  # indicates that do not leave frame pointer in register for functions that do not require
  # frame pointer. This avoids operation and maintenance of frame pointer in the function.
  # -fcombine-regs is used to indicate that the compiler combines instructions that copy
  # one register to another. -mstring-insns is an option that Linus adds to gcc. It is used
  # by gcc-1.40 to copy string structures using 386 CPU string instructions and can be removed.
14 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer \
15 -fcombine-regs -mstring-insns

  # cpp is gcc's preprocessor,used for macro substitution, conditional compilation, and
  # inclusion files specified with '#include'. All lines starting with '#' need to be
  # processed by preprocessor. All macros defined by '#define' will be replaced with their
  # definitions. All conditional lines such as '#if', '#ifdef', '#ifndef', and '#endif' are
  # used to determine whether to include statements in their specified range.
  # The option '-nostdinc -Iinclude' means not searching standard header file directory,
  # ie not using files in /usr/include/, but using directory specified by the '-I' option
  # or searching in the current directory.
16 CPP     =cpp -nostdinc -Iinclude
17
18 #
19 # ROOT_DEV specifies the default root-device when making the image.
20 # This can be either FLOPPY, /dev/xxxx or empty, in which case the
21 # default of /dev/hd6 is used by 'build'.
22 #
  # Here /dev/hd6 corresponds to first partition of the sceond hard disk. This is the
```

```
   # location where the root file system is located when Linus develops the Linux kernel.
   # /dev/hd2 represents 2nd partition of 1st hard disk and is used as a swap partition.
23 ROOT_DEV=/dev/hd6
24 SWAP_DEV=/dev/hd2
25
   # Below are the object files generated in the kernel, mm, and fs directories. For ease of
   # reference, they are represented here by the ARCHIVES identifier.
26 ARCHIVES=kernel/kernel.o mm/mm.o fs/fs.o

   # Block and character device library files. '.a' indicates that the file is an archive
   # file, that is, a library file containing a collection of executable binary code
   # subroutines, usually generated by GNU's ar program. Ar is a GNU binary file tool for
   # creating, modifying, and extracting files from archive files.
27 DRIVERS =kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a
28 MATH    =kernel/math/math.a
29 LIBS    =lib/lib.a                 # A generic library compiled from files in lib/ dir.
30
   # Here is the old-fashioned implicit suffix rule for make. This line instructs make to
   # compile all '.c' files into '.s' assembly using the commands of rule after ':'. The
   # whole sentence means that gcc uses the option specified by the CFLAGS and only uses the
   # header file in include/ dir, and stops (-S) compiling, thereby generating an assembly
   # file corresponding to each input C file. By default, the resulting assembly file is the
   # original C file with suffix '.c' replaced with '.s'. '-o' indicates the output file.
   # Where '$*.s' (or '$@') is automatic object variable, and '$<' represents the first
   # prerequisite, here is the file that meets the condition '*.c'.
   # The following three rules are used for different requirements. If the target is a .s
   # file and the source is a .c file, the first rule will be used; if target is .o and the
   # original is .s, the second rule will be used; if the target is a .o file, the original
   # is a c file, then the third rule is used.
31 .c.s:
32         $(CC) $(CFLAGS) \
33         -nostdinc -Iinclude -S -o $*.s $<
34 .s.o:
35         $(AS) -c -o $*.o $<
36 .c.o:
37         $(CC) $(CFLAGS) \
38         -nostdinc -Iinclude -c -o $*.o $<
39

   # The following 'all' means to create the topmost target of the Makefile, here is the
   # Image file. It is the boot disk image file. If you write it to a floppy disk, you can
   # use the floppy disk to boot the Linux system. See the line 46 for commands to write an
   # Image to a floppy disk under Linux. The software rawrite.exe can be used under DOS.
40 all:    Image
41
   # The target (Image) is generated by 4 elements behind the colon, which are the bootsect
   # and setup files in boot/, the system and build files in tools/. Lines 43--44 are commands
   # executed. Line 43 indicates that the bootsect, setup, and system files are assembled
   # to be kernel image file, with $(ROOT_DEV) device using build utility in tools directory.
   # The sync cmd on line 45 forces buffer to immediately write disk and update super block.
42 Image: boot/bootsect boot/setup tools/system tools/build
43         tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) \
44                 $(SWAP_DEV) > Image
```

```
45          sync
46
   # Indicates that the disk target is created from Image. dd is a standard cmd: copy a file,
   # convert and format it according to the options. bs= the number of bytes read/written at
   # a time. If= the input file, and of= the file to be output. Here /dev/PS0 refers to first
   # floppy disk drive (device file). Use /dev/fd0 under current linux system.
47 disk: Image
48          dd bs=8192 if=Image of=/dev/PS0
49
50 tools/build: tools/build.c          # Create executable build tool.
51          $(CC) $(CFLAGS) \
52          -o tools/build tools/build.c
53
54 boot/head.o: boot/head.s            # Generate head.o object using the .s.o rule.
55
   # Indicates that the tools/system is to be generated by the elements to the right of the
   # colon. Line 57--62 is cmds to generate system object. The last '> System.map' means that
   # gld needs to redirect the link info into the System.map file.
56 tools/system:   boot/head.o init/main.o \
57                 $(ARCHIVES) $(DRIVERS) $(MATH) $(LIBS)
58          $(LD) $(LDFLAGS) boot/head.o init/main.o \
59          $(ARCHIVES) \
60          $(DRIVERS) \
61          $(MATH) \
62          $(LIBS) \
63          -o tools/system > System.map
64
   # The archive file math.a is built by cmds on line 64: cd into kernel/math/; run make.
65 kernel/math/math.a:
66          (cd kernel/math; make)
67
68 kernel/blk_drv/blk_drv.a:          # Create block driver archive file.
69          (cd kernel/blk_drv; make)
70
71 kernel/chr_drv/chr_drv.a:          # Character driver archive file.
72          (cd kernel/chr_drv; make)
73
74 kernel/kernel.o:                   # kernel object file.
75          (cd kernel; make)
76
77 mm/mm.o:                           # Memory management object file.
78          (cd mm; make)
79
80 fs/fs.o:                           # File system object file.
81          (cd fs; make)
82
83 lib/lib.a:                         # Internal lib.a
84          (cd lib; make)
85
   # Compile setup.s file to generate setup.o using the 8086 assembler and linker. The option
   # -s indicates that the symbol information in the target file need to be removed.
86 boot/setup: boot/setup.s
87          $(AS86) -o boot/setup.o boot/setup.s
```

```
88          $(LD86) -s -o boot/setup boot/setup.o
89
    # Execute preprocessor, replace macro in *.S file to generate the corresponding *.s file.
90  boot/setup.s:   boot/setup.S include/linux/config.h
91          $(CPP) -traditional boot/setup.S -o boot/setup.s
92
93  boot/bootsect.s:        boot/bootsect.S include/linux/config.h
94          $(CPP) -traditional boot/bootsect.S -o boot/bootsect.s
95
96  boot/bootsect:  boot/bootsect.s
97          $(AS86) -o boot/bootsect.o boot/bootsect.s
98          $(LD86) -s -o boot/bootsect boot/bootsect.o
99
    # When 'make clean' is executed, the commands on lines 101--107 are executed, and all
    # files generated are removed. 'rm' is a file deletion cmd, and the option -f means to
    # ignore files that do not exist and does not display deletion messages.
100 clean:
101         rm -f Image System.map tmp_make core boot/bootsect boot/setup \
102                 boot/bootsect.s boot/setup.s
103         rm -f init/*.o tools/system tools/build boot/*.o
104         (cd mm;make clean)
105         (cd fs;make clean)
106         (cd kernel;make clean)
107         (cd lib;make clean)
108
    # The rule will first execute the above clean rule, then compress the linux/ directory to
    # generate a 'backup.Z' compressed file. 'tar cf - linux' means to execute the tar archiver
    # on the linux/ directory. '-cf' indicates creating a archive file. '|compress -' means that
    # the execution of the tar program is passed to the compressor compress by the pipeline
    # operation, and the output of the compressor is saved as a backup.Z file.
109 backup: clean
110         (cd .. ; tar cf - linux | compress - > backup.Z)
111         sync
112
113 dep:
    # This goal or rule is used to generate dependencies between files. These dependencies are
    # created to let the make command use them to determine if a target object needs to be rebuilt.
    # For example, when a header file has been changed, make can recompile all *.c files related
    # to it through the generated dependencies. The specific method is as follows:
    # Use the string editor sed to process the Makefile (here, this file), the output is to
    # delete all the lines after the '### Dependencies' line in the Makefile, that is, delete
    # all lines from 122 to the end of the file, and generate a temporary file tmp_make (also
    # known as 114 lines). # Then perform a gcc preprocessing operation on each C file (in fact,
    # only one file main.c) in the specified directory (init/). The flag '-M' tells the preprocessor
    # cpp to output rules describing the relevance of each object file, and these rules conform
    # to the make syntax. For each source file, the preprocessor outputs a rule whose result
    # is the target file name of the corresponding source file plus its dependencies, that is,
    # a list of all the header files contained in the source file. Then add the pre-processing
    # results to the temporary file tmp_make, and finally copy the temporary file into a new Makefile.
    # The '$$i' on line 115 is actually '$($i)'. Here '$i' is the value of the shell variable
    # 'i' in front of this sentence.
114         sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
115         (for i in init/*.c;do echo -n "init/";$(CPP) -M $$i;done) >> tmp_make
```

```
        cp tmp_make Makefile
        (cd fs; make dep)
        (cd kernel; make dep)
        (cd mm; make dep)

### Dependencies:
init/main.o : init/main.c include/unistd.h include/sys/stat.h \
  include/sys/types.h include/sys/time.h include/time.h include/sys/times.h \
  include/sys/utsname.h include/sys/param.h include/sys/resource.h \
  include/utime.h include/linux/tty.h include/termios.h include/linux/sched.h \
  include/linux/head.h include/linux/fs.h include/linux/mm.h \
  include/linux/kernel.h include/signal.h include/asm/system.h \
  include/asm/io.h include/stddef.h include/stdarg.h include/fcntl.h \
  include/string.h
```

5.13 Summary

This chapter provides an overview of the kernel modes and architecture of the early Linux operating systems. First, the Linux 0.12 kernel usage and management memory methods, kernel state stack and user state stack settings and usage methods, interrupt mechanism, system clock timing, and process creation, scheduling, and termination methods are given. Then according to the directory structure of the source code, the basic functions and hierarchical relationships of the code files in each subdirectory are introduced in detail. It also explains the target file format used by Linux 0.12. Finally, we started with the makefile in the Linux kernel home directory and began to comment on the kernel source code.

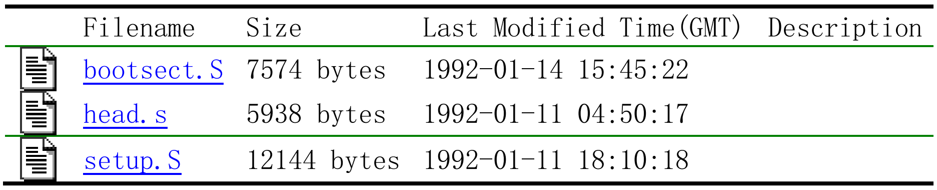
This chapter can be seen as a summary of the important information about the Linux 0.12 kernel, so it can be used as a reference for reading subsequent chapters. From the bootloader in the next chapter, we formally began to annotate the source code of the kernel.

6 Booting System

This chapter describes three assembly language files in the boot/ directory, as shown in Listing 6-1. As mentioned in the previous chapter, these three files are all assembly language programs, but use two different syntax formats. bootsect.S and setup.S are real-mode 16-bit code programs that use Intel's assembly language syntax and require the 8086 assembly compiler and linker as86 and ld86. However, head.s uses an AT&T assembly syntax format and runs in protected mode, which needs to be compiled with GNU's as (gas) assembler.

The main reason why Linus Torvalds used two assemblers at the time was that for Intel x86 processors, the GNU assembly compiler in 1991 only supported i386 and later 32-bit CPU code instructions. It is not supported to generate 16-bit code programs that run in real mode. Until 1994, the GNU as assembler began to support the .code16 directive for compiling 16-bit code (See the "Writing 16-Bit Codes" of the "80386 Related Features" section in the GNU Assembler manual.). Starting with kernel 2.4.X, the bootsect.S and setup.S programs began to be uniformly written using GNU as.

List 6-1 linux/boot/ directory



Filename	Size	Last Modified Time(GMT)	Description
bootsect.S	7574 bytes	1992-01-14 15:45:22	
head.s	5938 bytes	1992-01-11 04:50:17	
setup.S	12144 bytes	1992-01-11 18:10:18	

In addition to knowing some of the 8086 assembly language knowledge, reading these codes requires an understanding of some PC architectures with Intel 80X86 microprocessors and the basic principles of programming in 32-bit protected mode. So you should have a basic understanding of the previous chapters before start reading the source code. When we read the code, we will explain the specific problems encountered in detail.

6.1 Main Functions

Let us first describe the main execution process of the Linux operating system booting part. When the PC power is turned on, the 80x86 CPU will automatically enter the real mode. The program code is automatically executed starting at address 0xFFFF0. This address is usually the address in the ROM-BIOS. The PC's BIOS will perform hardware detection and diagnostic operations of the system and begin to initialize the interrupt vector at physical address 0. Thereafter, it reads the first sector of the bootable device (disk boot sector, 512 bytes) into the absolute address 0x7C00 of the memory and jumps to this location to start the boot process. The boot device is usually a floppy drive or a hard drive. The description here is simple, but it is enough to understand the beginning of the kernel initialization work.

The first part of Linux is written in 8086 assembly language (boot/bootsect.S) and stored in the first sector of the boot device. It will be loaded by the BIOS to the absolute address of the memory 0x7C00 (31KB). When executed, it will move itself to the absolute address 0x90000 (576KB) in memory, and read the 2KB byte code (boot/setup.S) in the boot device to memory 0x90200. The rest of the kernel (system module) is read in and loaded to the beginning of the memory address 0x10000 (64KB). Therefore, the process of sequential execution from the power-on of the machine is shown in Figure 6-1.

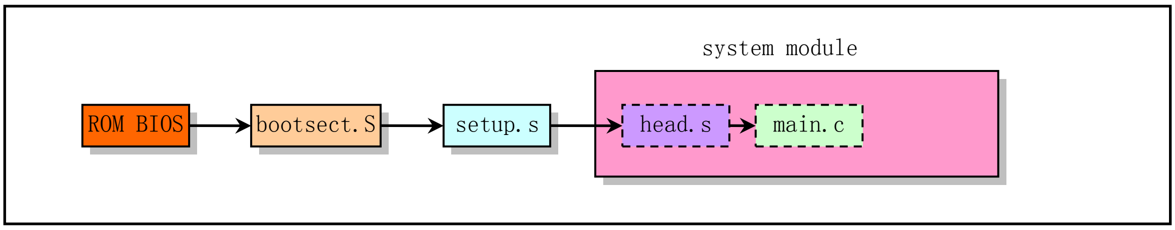


Figure 6-1 The order of execution from system power up

Because the length of the system module at that time will not exceed 0x80000 bytes (ie 512KB), therefore, when the bootsect program reads the system module into the physical address 0x10000 start position, it will not overwrite the bootsect and setup modules starting at 0x90000 (576KB). The subsequent setup program will also move the system module to the physical memory start location, so that the address of the code in the system module is equal to the actual physical address, which is convenient for operating the kernel code and data. Figure 6-2 clearly shows the dynamic location of these programs or modules in memory when the Linux system starts. In the figure, each vertical bar represents the image location map of each program in memory at a certain moment. The message "Loading..." will be displayed during system loading. Control is then passed to the code in boot/setup.S, which is another real-mode assembly language program.

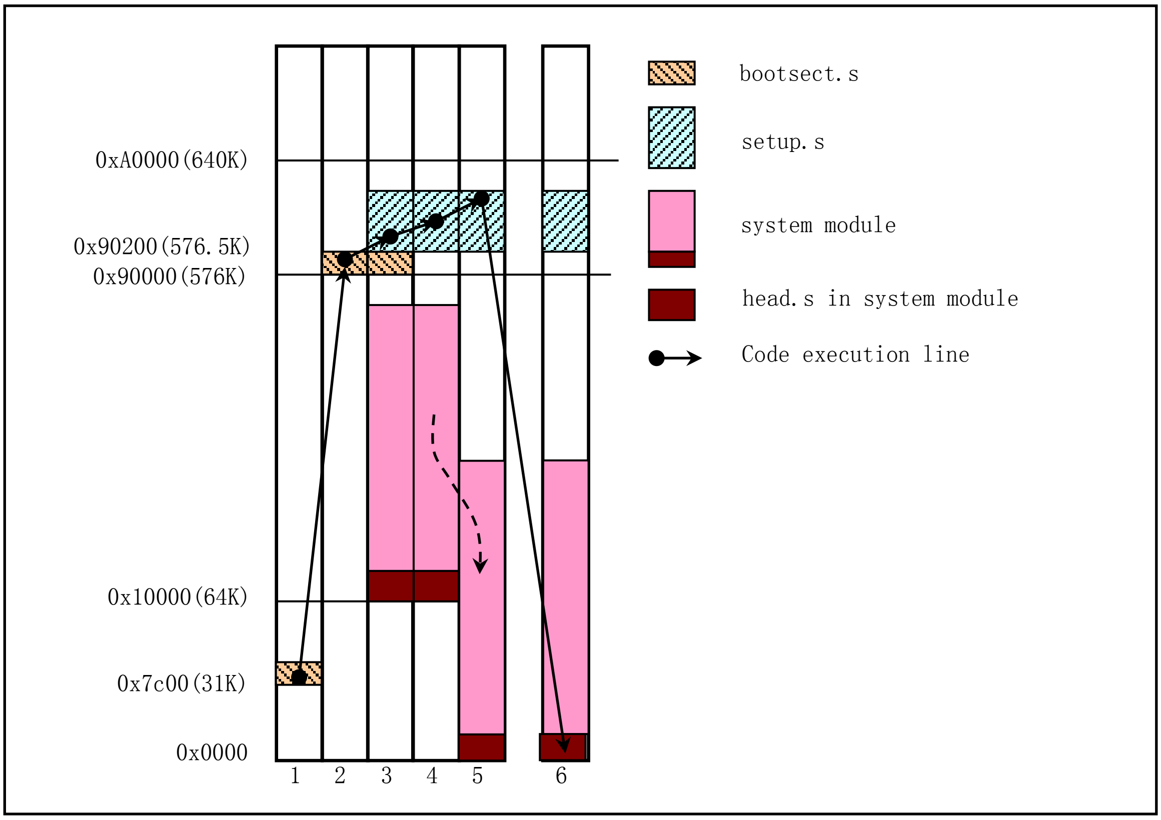


Figure 6-2 Dynamic location of the kernel in memory

The boot part identifies certain features of the host and the type of VGA card, and if required, it asks the user to select a display mode for the console. Then move the entire system from address 0x10000 to 0x0000, enter protected mode and jump to the rest of the system (at 0x0000). At this point, the boot settings for all 32-bit modes have been completed: IDT, GDT, and LDT are loaded, the processor and coprocessor are also confirmed, and paging is also set. Finally, the main() code in init/main.c is called. The source code for the above operation is in boot/head.s, which is probably the most tricky code in the entire kernel. Note that if something goes wrong in any of the previous steps, the computer will deadlock. It can't be handled before the operating system is fully operational.

Someone may ask, why does the bootsect code not load the system module directly to the beginning of the physical address 0x0000, but to move it again in the setup program? This is because the subsequent setup code also needs to use the interrupt call function provided by the ROM BIOS to get some parameters about the machine configuration (such as display card mode, hard disk parameter table, etc.). However, when the BIOS is initialized, an interrupt vector table of size 0x400 bytes (1KB) is placed at the beginning of the physical memory. Directly placing the system module at this location will cause the BIOS interrupt vector table to be overwritten. Therefore, the bootloader needs to move the system module to this area after using the BIOS interrupt call.

In addition, loading the above kernel modules only in memory is not enough for the Linux system to run. As a fully operational Linux system, you also need a basic file system support, the root file system. The Linux 0.12 kernel only supports the MINIX 1.0 file system. The root file system usually exists on another floppy disk or in a hard disk partition. In order to inform the kernel where the root file system is stored, the default block device number ROOT_DEV where the root file system is located is given on line 44 of the bootsect.S program. See the comments in the program for the meaning of the block device number. The specified device number in the 509, 510 (0x1fc--0x1fd) byte of the boot sector is used when the kernel is initialized. The swap device number SWAP_DEV is given on line 45 of the bootsect.S, which indicates the external device number used as the virtual storage swap space.

6.2 bootsect.S

6.2.1 Functional Description

The bootsect.S code is a disk boot block program that resides in the first sector of the disk (boot sector, 0 track (cylinder), 0 head, 1st sector). After the PC is powered on and the ROM BIOS self-tests, the ROM BIOS loads the boot sector code bootsect to the memory address 0x7C00 and executes it. During the execution of the bootsect code, it moves itself to the beginning of the memory absolute address 0x90000 and continues execution. The main function of this program is to first load the four-sector setup module (compiled from setup.s) starting from the second sector of the disk into memory immediately after the bootsect (0x90200). Then use BIOS interrupt 0x13 to take the parameters of the current boot disk in the disk parameter table, and then display the "Loading system..." string on the screen. Then load the system module behind the setup module on the disk to the beginning of the memory 0x10000. The device number of the root file system is then determined. If not specified, the type of disk (is 1.44M A disk?) is discriminated according to the saved number of sectors per track of the boot disk, and the device number is stored in root_dev (at position 508 of the boot block). Finally, long jumps to the beginning of the setup program (0x90200) to execute setup. On the disk, the location and size of the boot block, setup module, and system module are shown in Figure 6-3.

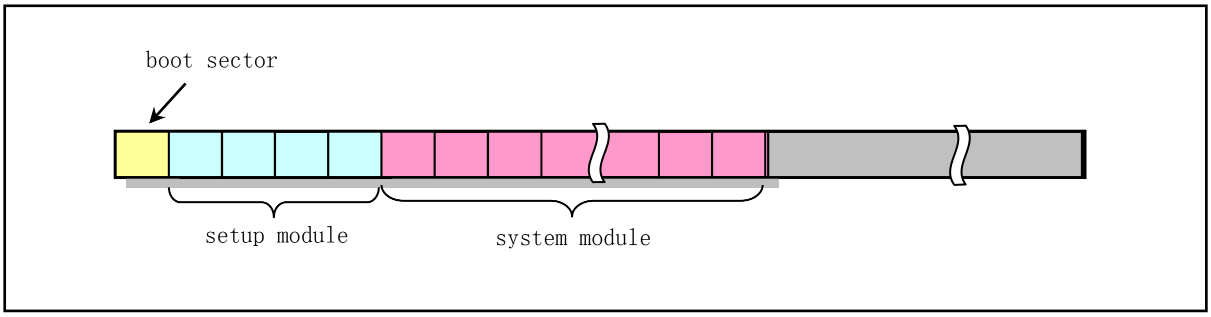


Figure 6-3 Distribution of Linux 0.12 kernel on 1.44MB disk

The figure shows the distribution of sectors occupied by the Linux 0.12 kernel on a 1.44MB disk. There are 80 tracks (cylinders) on each side of the 1.44MB disk platter, each with 18 sectors and a total of 2880 sectors. The boot program code occupies the first sector, the setup module occupies the next four sectors, and the 0.12 kernel system module occupies approximately the next 260 sectors. There are still more than 2610 sectors left unused. These remaining unused space can be used to store a basic root file system, creating an integrated disk that can be used to run the system using a single disk. This will be covered in more detail in the chapter on block device drivers.

In addition, the file name of this program is different from other gas assembly language programs. Its suffix is the uppercase '.S'. Using such a suffix allows gas to use the preprocessing functions of the GNU compiler, so you can include statements such as "#include" and "#if" in assembly language programs. The program uses the uppercase suffix mainly to use the "#include" statement in the program to include the constants defined in the linux/config.h header file. See line 6 of the program.

Note again that the text or statements with line number in the source file are original, and statements without line number are the comments of the author.

6.2.2 **Code Comments**

Program 6-1 linux/boot/bootsect.S

```
  ! Here the '!' or ';' at the beginning of a line indicates a comment statement.
1 !
2 ! SYS_SIZE is the number of clicks (16 bytes) to be loaded.
3 ! 0x3000 is 0x30000 bytes = 196kB, more than enough for current
4 ! versions of linux
  ! SYS_SIZE is the size of the system module to be loaded. The unit of size is paragraph,
  ! 16-byte per paragraph in X86 memory segmentation scope. 0x3000 is 0x30000 bytes=196KB.
  ! If 1KB=1024 bytes, it is 192 KB. This space size is sufficient for the current kernel
  ! version. When the it is 0x8000, it means that the kernel is at most 512KB. Since the
  ! memory of the bootsect and setup code is stored at 0x90000, the value must not exceed
  ! 0x9000 (indicating 584KB).
5 !
  ! File linux/config.h defines some constant symbols used by the kernel and the default
  ! hard disk parameter block used by Linus himself. For example, the following constants
  ! are defined:
  ! DEF_SYSSIZE  = 0x3000 - The default system module size (in paragraph);
  ! DEF_INITSEG  = 0x9000 - The default destination position to move to;
  ! DEF_SETUPSEG = 0x9020 - The default location for setup code;
  ! DEF_SYSSEG   = 0x1000 - The default location for system module from disk.
```

```
#include <linux/config.h>
SYSSIZE = DEF_SYSSIZE
!
!       bootsect.s              (C) 1991 Linus Torvalds
!       modified by Drew Eckhardt
!
! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves
! iself out of the way to address 0x90000, and jumps there.
!
! It then loads 'setup' directly after itself (0x90200), and the system
! at 0x10000, using BIOS interrupts.
!
! NOTE! currently system is at most 8*65536 bytes long. This should be no
! problem, even in the future. I want to keep it simple. This 512 kB
! kernel size should be enough, especially as this doesn't contain the
! buffer cache as in minix
!
! The loader has been made as simple as possible, and continuos
! read errors will result in a unbreakable loop. Reboot by hand. It
! loads pretty fast by getting whole sectors at a time whenever possible.

! The directive (pseudo-operator) .globl or .global is used to define whether the
! subsequent identifier is external or global and is forced to be introduced even if
! it is not used. .text, .data, and .bss are used to define the current code section,
! data section, and uninitialized data section, respectively.
! When link multiple objects, the ld86 combines (merges) the corresponding sections
! in each object module according to their categories. Here, all three sections are
! defined in the same address range, so the program is not actually segmented.
! In addition, string followed by a colon is a label, such as the 'begtext:'.
.globl begtext, begdata, begbss, endtext, enddata, endbss
.text                              ! text section.
begtext:
.data                              ! data section
begdata:
.bss                               ! uninitialized data section
begbss:
.text

! The equal sign '=' or symbol 'EQU' is used to define the value of identifier or label.
SETUPLEN = 4                       ! nr of setup-sectors
                                   ! sectors occupied by setup code.
BOOTSEG  = 0x07c0                  ! original address of boot-sector
INITSEG  = DEF_INITSEG             ! we move boot here - out of the way
                                   ! Move to 0x90000 - avoid location used by system;
SETUPSEG = DEF_SETUPSEG            ! setup starts here
                                   ! setup code start from 0x90200
SYSSEG   = DEF_SYSSEG              ! system loaded at 0x10000 (65536).
ENDSEG   = SYSSEG + SYSSIZE        ! where to stop loading

! ROOT_DEV & SWAP_DEV are now written by "build".
! The root fs device number ROOT_DEV and swap device SWAP_DEV are now written by build.
! Device number 0x306 specifies that the root fs device is the first partition of the
! second hard disk. Linus installed the Linux 0.11 system on the second hard disk, so
```

```
   ! ROOT_DEV is set to 0x306. When compiling this kernel, you can modify the device number
   ! according to the location of the device where your root fs is located. For example, if
   ! your root file system is on the first partition of the first hard disk, then the value
   ! should be 0x0301, that is (0x01, 0x03).This device number is the old-fashioned drive
   ! naming method of Linux, until kernel 0.95. The specific values of the hard disk device
   ! number are as follows:
   ! Device nr = major device nr * 256 + minor device nr, (or dev_no = (major<<8) + minor)
   ! (Major nr: 1-memory, 2-disk, 3-drive, 4-ttyx, 5-tty, 6-parallel port, 7-unnamed pipe)
   ! 0x300 - /dev/hd0 - Represents the entire 1st hard drive;
   ! 0x301 - /dev/hd1 - The 1st partition of the 1st disk;
   !  …
   ! 0x304 - /dev/hd4 - The 4th partition of the 1st disk;
   ! 0x305 - /dev/hd5 - Represents the entire 2nd hard drive;
   ! 0x306 - /dev/hd6 - The 1st partition of the 2nd disk;
   !  …
   ! 0x309 - /dev/hd9 - The 4th partition of the 2nd disk;
   !
44 ROOT_DEV = 0                     ! The root fs device uses the same boot device;
45 SWAP_DEV = 0                     ! The swap device uses the same boot device;
46
   ! The directive entry forces the linker to include the specified identifier or label in
   ! the generated executable (a.out). Here is the starting point for program execution.
   ! The function of 49 -- 58 lines is to move itself (bootsect) from the current segment
   ! position 0x07c0 (31KB) to 0x9000 (576KB), a total of 256 words (512 bytes), and then
   ! jump to the label go of the moved code, that is, the next statement of the program.
47 entry start                      ! Tell the linker, program starts at the label start.
48 start:
49         mov     ax,#BOOTSEG      ! Set the ds segment register to 0x7C0;
50         mov     ds,ax
51         mov     ax,#INITSEG      ! Set the es segment register to 0x9000;
52         mov     es,ax
53         mov     cx,#256          ! Set move count = 256 words (512 bytes);
54         sub     si,si            ! Source address ds:si = 0x07C0:0x0000
55         sub     di,di            ! Destination address es:di = 0x9000:0x0000
56         rep                      ! counting down, until cx = 0.
57         movw                     ! Move cx words from memory [si] to [di].
   ! Jump Intersegment. INITSEG - the segment to jump to, go - the offset within the segment.
58         jmpi    go,INITSEG
59
   ! Starting from below, the CPU executes in code that has moved to 0x90000 position.
   ! This code sets several segment registers, including the stack registers ss and sp.
   ! The stack pointer sp can be pointed as long as it points farther than the 512-byte
   ! offset (ie address 0x90200). Because the setup program needs to be placed from the
   ! 0x90200 address ( 4 sectors in size), so the sp should point to a position greater
   ! than (0x200 + 0x200 * 4 + stack size). Here sp is set to 0x9ff00 - 12 (parameter table
   ! size), ie sp = 0xfef4. A self-built drive parameter list will be stored above this
   ! location, as explained below. In fact, when the BIOS loads the boot sector to 0x7c00
   ! and gives execution to the bootloader, ss = 0x00, sp = 0xfffe.
   ! In addition, the desired function of the push instruction on line 65 is to temporarily
   ! save the segment on the stack, and then wait for the following to determine the number
   ! of track sectors before popping the stack, and assign to the segment registers fs and
   ! gs (line 109). However, since the two statements on lines 67 and 68 modify the position
   ! of the stack. Therefore, this design is wrong unless the stack is restored to its
   ! original location before the stack pop operation is performed. So there is a bug here.
   ! One of the corrections is to remove line 65 and change line 109 to "mov ax, cs".
60 go:     mov     ax,cs              ! Set ds, es, and ss to segment after moved (0x9000).
61         mov     dx,#0xfef4         ! arbitrary value >>512 - disk parm size
62
63         mov     ds,ax
64         mov     es,ax
65         push    ax                 ! Temp save segment (0x9000) for 109 lines. (slipper!)
66
67         mov     ss,ax              ! put stack at 0x9ff00 - 12.
68         mov     sp,dx
69 /*
70  *      Many BIOS's default disk parameter tables will not
71  *      recognize multi-sector reads beyond the maximum sector number
72  *      specified in the default diskette parameter tables - this may
73  *      mean 7 sectors in some cases.
74  *
75  *      Since single sector reads are slow and out of the question,
76  *      we must take care of this by creating new parameter tables
77  *      (for the first disk) in RAM.  We will set the maximum sector
78  *      count to 18 - the most we will encounter on an HD 1.44.
79  *
80  *      High doesn't hurt.  Low does.
81  *
82  *      Segments are as follows: ds=es=ss=cs - INITSEG,
83  *              fs = 0, gs = parameter table segment
84  */
85 ! The interrupt 0x1E set by BIOS is not actually an interrupt. The location corresponding
   ! to the interrupt vector is the address of floppy drive parameter table.This interrupt
   ! vector is located at memory 0x1E * 4 = 0x78. This code first copies the original floppy
   ! disk parameter table from memory 0x0000:0x0078 to 0x9000:0xfef4, and then modifies the
   ! maximum number of sectors per track at offset 4 in the table to 18. Table size is 12 bytes.
86
87         push    #0                 ! set segment reg fs = 0.
88         pop     fs
89         mov     bx,#0x78           ! fs:bx is parameter table address
   ! The following instruction indicates that the operand of the next statement is in fs
   ! segment, and it only affects its next statement. Here, the table address pointed to by
   ! fs:bx is placed in the register pair gs:si as the original address, and register pair
   ! es:di = 0x9000:0xfef4 is used as the destination address.
90         seg fs
91         lgs     si,(bx)            ! gs:si is source
92
93         mov     di,dx              ! es:di is destination ! dx = 0xfef4, set on line 61.
94         mov     cx,#6              ! copy 12 bytes
95         cld                        ! Clear direction flag. Incremente pointer when copying.
96
97         rep                        ! Copy the 12-byte table to 0x9000:0xfef4.
98         seg gs
99         movw
100
101        mov     di,dx              ! Es:di points to new table, then modifies the table.
102        movb    4(di),*18          ! patch sector count
```

```
103
104         seg fs                      ! Let interrupt vector 0x1E point to the new table.
105         mov     (bx),di
106         seg fs
107         mov     2(bx),es
108
    ! ax is segment value (0x9000) saved on line 65. Set fs=gs=0x9000 to restore original segment.
109         pop     ax
110         mov     fs,ax
111         mov     gs,ax
112
    ! BIOS INT 0x13 function 0 is used to reset floppy disk controller, that is to forces
    ! controller to recalibrate drive heads (seek to track 0).
113         xor     ah,ah               ! reset FDC
114         xor     dl,dl               ! dl = drive, here set to first disk drive.
115         int     0x13
116
117 ! load the setup-sectors directly after the bootblock.
118 ! Note that 'es' is already set up.
119
    ! The purpose of line 121--137 is to use the BIOS INT 0x13 function 2 (read disk sectors)
    ! to read setup module from the beginning of the second sector on disk to the memory at
    ! 0x90200, total of 4 sectors. If an error occurs during a read operation, the location of
    ! the error sector is displayed, then the drive is reset and retried without a retreat.
    ! The INT 0x13 (read sector) parameters are set as follows:
    ! ah = 0x02  - read disk sector;  al = nr of sectors to read;
    ! ch = low 8 bits of cylinder nr; cl = sector nr(bit 0-5) high 2 bits cylinder(bit 6-7);
    ! dh = head number;               dl = drive number(bit 7 set for hard disk);
    ! return:
    ! If error, flag CF is set, and ah contains error code.
    ! es:bx ->point to data buffer;
120 load_setup:
121         xor     dx, dx                  ! drive 0, head 0
122         mov     cx,#0x0002              ! sector 2, track 0
123         mov     bx,#0x0200              ! address = 512, in INITSEG
124         mov     ax,#0x0200+SETUPLEN     ! service 2, nr of sectors
125         int     0x13                    ! read it
126         jnc     ok_load_setup           ! ok - continue
127
128         push    ax                      ! dump error code
129         call    print_nl                ! print next line.
130         mov     bp, sp                  ! ss:bp point to chars (word)
131         call    print_hex               ! display hex value.
132         pop     ax
133
134         xor     dl, dl                  ! reset FDC !
135         xor     ah, ah
136         int     0x13
137         j       load_setup              ! j = jmp
138
139 ok_load_setup:
140
141 ! Get disk drive parameters, specifically nr of sectors/track
```

```
    ! The following code uses INT 0x13 function 8 to take the parameters of the disk drive.
    ! In fact,the number of sectors per track is taken and stored at the location sectors.
    ! The INT 0x13 (get disk drive parameters) parameters are set as follows:
    ! ah = 0x08     dl = drive number(bit 7 set for hard disk);
    ! Return:
    ! CF is set if an error occurs, and ah = status code.
    ! ah = 0,  al = 0,              bl = drive type (AT/PS2);
    ! ch = low 8 bits of max cylinder nr;
    ! cl = max sector number (bit 5-0), high 2 bits of max cylinder number (bit 7-6);
    ! dh = max head number;         dl = number of drives;
    ! es:di -> drive parameter table ( floppy only).
142
143         xor     dl,dl
144         mov     ah,#0x08            ! AH=8 is get drive parameters
145         int     0x13
146         xor     ch,ch
    ! The following instruction indicates that the operand of the next statement is in cs
    ! segment. It only affects its next statement. In fact, since the code and data are all
    ! set in the same segment, the values of the segment registers cs and ds, es are the same.
    ! Therefore, the instruction may not be used here.
147         seg cs
    ! The next sentence saves the number of sectors per track. For a floppy disk (dl=0), its
    ! maximum track number will not exceed 256, and ch is enough to represent it, so bits 6-7
    ! of cl must be zero. Also, since 146 lines have been set to ch=0, at this time, cx is the
    ! number of sectors per track.
148         mov     sectors,cx
149         mov     ax,#INITSEG
150         mov     es,ax               ! Because the interrupt changed es, here restore it.
151
152 ! Print some inane message
    ! Using BIOS INT 0x10 function 0x03 and 0x13 to display the message: "'Loading' + cr + lf",
    ! which displays a total of 9 chars, including carriage return and line feed control chars.
    ! The BIOS INT 0x10 (read cursor location) parameters are set as follows:
    ! ah = 0x03, read cursor position and size;
    ! bh = page number;
    ! Return:
    ! ch = start scan line;cl = end scan line;
    ! dh = row (0x00 is top); dl = colum(0x00 is left);
    !
    ! The BIOS INT 0x10 (write string) parameters are set as follows:
    ! ah = 0x13, write string;
    ! al = write mode. 0x01-use attributes in bl, cursor stop at end of string.
    ! bh = page number; bl = attributes;   dh,dl = row, colum at which to start writing.
    ! cx = number of characters in string.
    ! es:bp -> string to write.
    ! Return: Nothing.

153
154         mov     ah,#0x03            ! read cursor pos
155         xor     bh,bh
156         int     0x10                ! position: dh - row(0--24), dl - colum(0--79).
157
158         mov     cx,#9               ! Total 9 charachers.
```

```
159         mov     bx,#0x0007          ! page 0, attribute 7 (normal)
160         mov     bp,#msg1            ! es:bp point to message.
161         mov     ax,#0x1301          ! write string, move cursor
162         int     0x10
163
164 ! ok, we've written the message, now
165 ! we want to load the system (at 0x10000)
166
167         mov     ax,#SYSSEG
168         mov     es,ax               ! segment of 0x010000
169         call    read_it             ! load system module, es is parameter.
170         call    kill_motor          ! stop motor to know the drive status.
171         call    print_nl
172
173 ! After that we check which root-device to use. If the device is
174 ! defined (!= 0), nothing is done and the given device is used.
175 ! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
176 ! on the number of sectors that the BIOS reports currently.
    !
    ! The meanings of the above two device files are as follows:
    ! In Linux, the floppy drive's major number is 2 (see comment on line 43), the
    ! minor device number = type*4 + nr, where nr is 0-3 for floppy drive A, B, C or D
    ! respectively; type is floppy drive Type (2--1.2MB or 7--1.44MB, etc.).
    ! Since 7*4 + 0 = 28, /dev/PS0(2,28) refers to 1.44MB A drive with device number 0x021c.
    ! Similarly, /dev/at0 (2,8) refers to 1.2MB A drive with device number 0x0208.

    ! The root_dev is defined at the location of the boot sector 508, 509 bytes, referring
    ! to the device number of the root file system.
    ! This value needs to be modified based on the drive where your root fs is located.
    ! For example, if your root fs is on the 1st partition of the 1st hard disk, then the
    ! value should be 0x0301, that is (0x01, 0x03). If the root fs is on the second 1.44MB
    ! floppy disk, then the value should be 0x021D, which is (0x1D, 0x02).
    ! When compiling the kernel, you can specify your own values in the Makefile. The kernel
    ! image file creation program tools/build will also use the value you specify to set the
    ! device number of your root file system.
177
178         seg cs
179         mov     ax,root_dev
180         or      ax,ax               ! root_dev is defined (not 0) ?
181         jne     root_defined
    ! The following statements use the number of sectors per track saved in 'sectors' in
    ! line 148 above to determine the disk type. If sectors = 15, it means 1.2MB drive; if
    ! sectors = 18, it means 1.44MB floppy drive. Because it is a bootable drive, it's
    ! definitely an A drive.
182         seg cs
183         mov     bx,sectors
184         mov     ax,#0x0208          ! /dev/ps0 - 1.2Mb
185         cmp     bx,#15              ! sectors = 15 ?
186         je      root_defined
187         mov     ax,#0x021c          ! /dev/PS0 - 1.44Mb
188         cmp     bx,#18
189         je      root_defined
190 undef_root:                         ! If not, then an infinite loop (dead).
```

```
191         jmp undef_root
192 root_defined:
193         seg cs
194         mov     root_dev,ax        ! Save the checked device number to root_dev.
195
196 ! after that (everyting loaded), we jump to
197 ! the setup-routine loaded directly after
198 ! the bootblock:
    ! The inter segment jump instruction, jump to 0x9020:0000 to execute setup code.
199
200         jmpi    0,SETUPSEG         ! At this point, the bootsect code ends!!!

    ! Here are a few subroutines. read_it is used to read the system module on the disk.
    ! Kill_moter is used to close the floppy drive motor. There are also some screen display
    ! subroutines.
201
202 ! This routine loads the system at address 0x10000, making sure
203 ! no 64kB boundaries are crossed. We try to load it as fast as
204 ! possible, loading whole tracks whenever we can.
205 !
206 ! in:   es - starting address segment (normally 0x1000)
207 !
    ! The following directive .word defines a 2-byte target, which is equivalent to the
    ! variables defined in the C language program and the amount of memory space occupied.
    ! The value '1+SETUPLEN' indicates that one boot sector has been read at the beginning,
    ! plus the number of sectors occupied by the setup code (SETUPLEN = 4).
208 sread:  .word 1+SETUPLEN        ! sectors read of current track
209 head:   .word 0                 ! current head
210 track:  .word 0                 ! current track
211
212 read_it:
    ! First check the input segment. The data read from disk must be stored at the beginning
    ! of the boundary of memory address 64KB, otherwise it will enter an infinite loop.
    ! Register bx is the starting position for storing data in the current segment.
    ! The test instruction on line 214 is bitwise logical with two operands. If the bits
    ! corresponding to both operands are 1, the resulting value is 1, otherwise 0. The result
    ! of this operation only affects flags (zero flag ZF, etc.). For example, if AX=0x1000,
    ! the test result is (0x1000 & 0x0fff) = 0x0000, and the ZF flag is set. At this point,
    ! the next instruction jne condition does not hold.
213         mov ax,es
214         test ax,#0x0fff
215 die:    jne die                 ! es must be at 64kB boundary
216         xor bx,bx               ! bx is starting address within segment
217 rp_read:
    ! Then check if all the data has been read. Compare whether the currently read segment is
    ! the segment where the end of the system data is located (#ENDSEG). If not, jump to the
    ! label ok1_read below to continue reading data. Otherwise return.
218         mov ax,es
219         cmp ax,#ENDSEG          ! have we loaded all yet?
220         jb ok1_read
221         ret
222 ok1_read:
    ! Next, calculate and verify the number of sectors that the current track needs to read,
    ! and put it in ax register.The method is as follows:
    ! According to the number of sectors that have not been read on the current track and the
    ! offset position of the data bytes in the segment, it is calculated whether the total
    ! number of bytes read will exceed the limit of 64 KB segment length if all unread sectors
    ! are read. If it is exceeded, the number of sectors that need to be read this time is
    ! calculated based on the maximum number of bytes that can be read (64KB - offset ).
223         seg cs
224         mov ax,sectors              ! get nr of sectors per track.
225         sub ax,sread                ! Subtract the nr of sectors the track has been read.
226         mov cx,ax                   ! cx = ax = the nr of unread sectors on the track.
227         shl cx,#9                   ! cx = cx * 512 + current offset (bx).
228         add cx,bx                   !    = total nr of bytes read after the operation.
229         jnc ok2_read                ! If it does not exceed 64KB, jump to ok2_read.
230         je ok2_read
    ! Add the data of the unread sectors on the track. If the result exceeds 64KB, then
    ! calculate the maximum number of bytes that can be read at this time:(64KB - offset).
    ! Then convert to the number of sectors to be read. Where 0 minus a certain number is the
    ! complement number of 64KB.
231         xor ax,ax
232         sub ax,bx
233         shr ax,#9
234 ok2_read:
    ! Read the data on the track to es:bx based on the specified start sector (cl) and number
    ! of sectors (al). The number of sectors that have been read on the current track is then
    ! counted and compared to the maximum number of sectors in the track.If it is less than
    ! 'sectors', there is still sectors unread on the track, so jump to ok3_read to continue.
235         call read_track             ! Reads data for given number of sectors on the track.
236         mov cx,ax                   ! cx = the nr of sectors read by this time.
237         add ax,sread                ! plus the nr of sectors that have been read.
238         seg cs
239         cmp ax,sectors        ! If there are unread sectors on the track, jump to ok3_read
240         jne ok3_read
    ! If all sectors of the current head of the track have been read, the data on the next
    ! head of the track (head 1) is read. If it is already done, go to the next track.
241         mov ax,#1
242         sub ax,head                 ! check current head no.
243         jne ok4_read                ! If it is head 0, then go get sectors on head 1.
244         inc track
245 ok4_read:
246         mov head,ax                 ! store current head no to head.
247         xor ax,ax                   ! Clear the number of sectors read.
248 ok3_read:
    ! If there are still unread sectors on the current track, first save the number of sectors
    ! read and then adjust the starting position where the data is stored. If it is less than
    ! the 64KB boundary value, jump to rp_read (line 217) and continue reading data.
249         mov sread,ax                ! save the nr of sectors read on the track.
250         shl cx,#9                   ! nr of sectors read * 512 bytes.
251         add bx,cx                   ! Adjust the starting position of the data.
252         jnc rp_read
    ! Otherwise it indicates that 64KB of data has been read. At this point, adjust the
    ! current segment to prepare for reading the next segment.
253         mov ax,es
254         add ah,#0x10                ! Adjust segment base address to point to next 64KB.
```

```
255         mov es,ax
256         xor bx,bx                   ! clear offse value.
257         jmp rp_read
258
    ! Read_track subroutine. Read the data of the specified start sector and the number of
    ! sectors on the track to the beginning of es:bx. See the description of the BIOS disk
    ! read interrupt int 0x13, ah=2 under line 119.
    ! al - sectors to be read;  es:bx - data buffer.
259 read_track:
    ! First call BIOS INT 0x10, function 0x0e (write characters by telex), the cursor moves
    ! forward one position. Show a dot '.'.
260         pusha                       ! push all registers.
261         pusha
262         mov ax, #0xe2e              ! loading... message 2e = .
263         mov bx, #7                  ! character foreground color attribute.
264         int 0x10
265         popa
266
    ! Then the track sector read operation is formally performed.
267         mov dx,track                ! current track.
268         mov cx,sread                ! sectors already read on the current track.
269         inc cx                      ! cl = start reading sector nr.
270         mov ch,dl                   ! ch = current head nr.
271         mov dx,head                 ! get current head nr.
272         mov dh,dl                   ! dh = head nr, dl = drive (0 for A drive)
273         and dx,#0x0100              ! head nr is no more than 1.
274         mov ah,#2                   ! ah = 2, read sectors.
275
276         push dx                     ! save for error dump
277         push cx
278         push bx
279         push ax
280
281         int 0x13
282         jc bad_rt                   ! if error, jump to bad_rt
283         add sp,#8                   ! if ok, discard status info.
284         popa
285         ret
286
    ! Error reading disk. The error message is displayed first, then the drive reset operation
    ! (disk interrupt function number 0) is executed, and then jump to the read_track to try again.
287 bad_rt: push ax                     ! save error code
288         call print_all              ! ah = error, al = read
289
290
291         xor ah,ah
292         xor dl,dl
293         int 0x13
294
295
296         add sp, #10                 ! Discard the info saved for the error condition.
297         popa
298         jmp read_track
```

```

/*
 *      print_all is for debugging purposes.
 *      It will print out all of the registers.  The assumption is that this is
 *      called from a routine, with a stack frame like
 *      dx
 *      cx
 *      bx
 *      ax
 *      error
 *      ret <- sp
 *
*/

print_all:
        mov cx, #5                 ! error code + 4 registers
        mov bp, sp                 ! Save current stack pointer sp.

print_loop:
        push cx                    ! save count left
        call print_nl              ! nl for readability
        jae no_reg                 ! see if register name is needed
                                   ! if CF=0, registers are not displayed and jump.
Corresponding to the stack register order, display their names: "AX : " etc.
        mov ax, #0xe05 + 0x41 - 1  ! ah = function 0x0e; al = char (0x05 + 0x41 -1)
        sub al, cl
        int 0x10

        mov al, #0x58              ! X
        int 0x10

        mov al, #0x3a              ! :
        int 0x10

! Display the contents of stack pointed by bp. originally bp points to return address.
no_reg:
        add bp, #2                 ! next register
        call print_hex             ! print it
        pop cx
        loop print_loop
        ret

! Call BIOS INT 0x10 to display carriage return and line feed control chars.
print_nl:
        mov ax, #0xe0d             ! CR
        int 0x10
        mov al, #0xa               ! LF
        int 0x10
        ret

/*
 *      print_hex is for debugging purposes, and prints the word
 *      pointed to by ss:bp in hexadecmial.
```

```
349 */
350
    ! Call BIOS INT 0x10 to display the word pointed to by ss:bp in 4 hexadecimals.
351 print_hex:
352         mov     cx, #4              ! 4 hex digits
353         mov     dx, (bp)            ! load word into dx
354 print_digit:
    ! The high byte is displayed first, so rotate dx by 4 to move high 4 bits to dx lower 4 bits.
355         rol     dx, #4              ! rotate so that lowest 4 bits are used
356         mov     ah, #0xe
357         mov     al, dl              ! mask off so we have only next nibble
358         and     al, #0xf            ! put in al, and get lower 4 bits only.
    ! Add '0' ASCII code 0x30 to convert the value to a char. If value in al exceeds 0x39, it
    ! means that the value displayed exceeds number 9, so it needs to be represented by 'A'--'F'.
359         add     al, #0x30           ! convert to 0 based digit, '0'
360         cmp     al, #0x39           ! check for overflow
361         jbe     good_digit
362         add     al, #0x41 - 0x30 - 0xa        ! 'A' - '0' - 0xa
363
364 good_digit:
365         int     0x10
366         loop    print_digit         ! cx--. If cx>0, the next value is displayed.
367         ret
368
369
370 /*
371  * This procedure turns off the floppy drive motor, so
372  * that we enter the kernel in a known state, and
373  * don't have to worry about it later.
374  */
    ! The value 0x3f2 on line 377 below is a port address of the floppy disk controller and
    ! is referred to as a digital output register (DOR) port. It is an 8-bit register with
    ! bits 7 - 4 for controlling the start and stop of four floppy drives (D--A). Bits 3 - 2
    ! are used to enable/disable DMA and interrupt requests and to start/reset the floppy
    ! disk controller FDC. Bit 1 - Bit 0 is used to select the floppy drive for the selected
    ! operation. The value of 0 set in al on line 378 is used to select the A drive, turn
    ! off the FDC, disable the DMA and interrupt requests, and turn off the motor. See the
    ! instructions behind the kernel/blk_drv/floppy.c program for more information on floppy
    ! control card programming.
375 kill_motor:
376         push dx
377         mov dx,#0x3f2               ! floppy controller port DOR.
378         xor al, al                  ! A drive, close FDC, disable DMA & int, close moter
379         outb                        ! output al to port dx.
380         pop dx
381         ret
382
383 sectors:
384         .word 0                     ! store nr of sectors per track.
385
386 msg1:                               ! message to display, total 9 chars.
387         .byte 13,10                 ! cr, lf.
388         .ascii "Loading"
```

```

! Start at address 506 (0x1FA), so root_dev is in 2 bytes starting at 508 of boot sector.
.org 506
swap_dev:
        .word SWAP_DEV
root_dev:
        .word ROOT_DEV

! 0xAA55 is a flag for the boot disk to have a valid boot sector for use by the BIOS
! program to load the boot sector. It must be the last two bytes of the boot sector.
boot_flag:
        .word 0xAA55

.text
endtext:
.data
enddata:
.bss
endbss:

```

6.2.3 Reference information

For the description of the bootsect.S program, a large amount of documents can be found online. Among them, Alessandro Rubini's article "Tour of the Linux kernel source" describes the kernel boot process more comprehensively. Since this program runs in CPU real mode, it will be easier to understand. If you still have difficulty reading at this time, then I suggest you review the 80x86 assembly and its hardware first, and then continue reading this book. For the newly developed Linux kernel, this program has not changed much, and basically maintains the appearance of the 0.12 version of the bootsect file.

6.2.3.1 Linux 0.12 Hard Disk Device Number

In Linux systems, various devices are accessed by device number, or refered to as logical device number. The device number consists of the major device number and the minor device number. The major device number specifies the type of device, while the minor device number specifies the specific device object. The major and minor device numbers are represented by 1 byte, that is, each device number has 2 bytes. The hard disk involved in the bootsect program is a block device whose major device number is 3. The major device numbers of basic devices in Linux systems are:

- 1 - memory;
- 2 - floppy disk;
- 3 - hard disk;
- 4 - ttyx;
- 5 - tty;
- 6 - parallel port;
- 7 - Unnamed pipe.

Since there can be 1--4 partitions in a traditional hard disk, the hard disk also uses the minor device number to specify the partition. The logical device number of the hard disk is composed of the following:

```
    device number = major device number <<8 + minor device number.
--------------------------------------------------------------------------------
```

All logical device numbers for both hard disks are shown in Table 6–1. 0x0300 and 0x0305 do not correspond to any one partition, but represent the entire hard disk. Also note that since the Linux kernel version 0.95 has not used this cumbersome naming method, it uses the same naming method as it is now.

Table 6-1 Hard disk logical device number

Device nr	Device file	Description
0x0300	/dev/hd0	Represents the entire first hard driv
0x0301	/dev/hd1	The first partition of the first disk
0x0302	/dev/hd2	The second partition of the first disk
0x0303	/dev/hd3	The third partition of the first disk
0x0304	/dev/hd4	The fourth partition of the first disk
0x0305	/dev/hd5	Represents the entire second hard driv
0x0306	/dev/hd6	The first partition of the second disk
0x0307	/dev/hd7	The second partition of the second disk
0x0308	/dev/hd8	The third partition of the second disk
0x0309	/dev/hd9	The fourth partition of the second disk

6.2.3.2 Booting from hard disk

The bootsect program gives the default method and process for booting a Linux system from a floppy disk. If you want to boot your system from a hard drive, you usually need to use a different multi-OS bootloader to boot the system, such as multiple operating system bootloaders such as Shoelace, LILO, or Grub. At this point, the operations that bootsect.S needs to perform will be completed by these programs, and the bootsect program will not be executed. Because if you boot from the hard disk, usually the kernel image file will be stored in the root file system of an active partition of the hard disk. So you need to know where the kernel image file is in the file system and what file system it is, that is, your boot sector program needs to be able to recognize and access the file system and read the kernel image file from it.

The basic process of booting from a hard disk is: after the system is powered on, the first sector of the bootable hard disk (MBR - Master Boot Record) will be loaded into the memory 0x7c00 by the BIOS and execution will begin. The program will first move itself down to memory 0x600, then load into memory 0x7c00 according to the first sector (boot sector) in the active partition specified by the partition table in the MBR, and then start execution.

For the Linux 0.12 system, the kernel image file is independent of the root file system. If you use this method directly to boot the system from the hard disk, you will encounter the problem that the root file system cannot coexist with the kernel image file. There are two possible solutions. One way is to set up a small-capacity active partition to hold the kernel image file, and the corresponding root file system is placed in another partition. Although this will use one more primary partitions on the hard disk, it should be able to boot the system from the hard disk with minimal modifications to the bootsect.S program. Another approach is to combine the kernel image file with the root file system in a partition, that is, the kernel image file is placed in some sectors at the beginning of the partition, and the root file system is stored from a subsequent specified sector. Both methods require some modifications to the code. Readers can refer to the last chapter to use the bochs simulation software to do some experiments.

6.3 setup.S

6.3.1 Function Descriptions

setup.S is an operating system loader. Its main function is to use the ROM BIOS interrupt to read the machine configuration data, and save the data to the beginning of 0x90000 (covering the place where the bootsect program is located). The parameters obtained and the stored memory locations are shown in Table 6–2. These parameters will be used by the relevant programs in the kernel. For example, the console.c and tty_io.c programs in the character device driver set.

Table 6-2 Parameters read and stored by the setup program

Address	Size (bytes)	Name	Description
0x90000	2	Cursor Location	Colum (0x00-left), Row (0x00-top most)
0x90002	2	Extended Memory	Size of extended memory begin from address 1MB (in KB)
0x90004	2	Display page	Current display page
0x90006	1	Display mode	
0x90007	1	Char Colums	
0x90008	2	Char Rows ??	
0x9000A	1	Display memory	0x00-64k, 0x01-128k, 0x02-192k, 0x03=256k
0x9000B	1	Display status	0x00-Color, I/O=0x3dX; 0x01-Mono, I/O=0x3bX
0x9000C	2	Property Paras	Property parameters of display adapter.
0x9000E	1	Screen rows	Screen current display rows.
0x9000F	1	Screen colums	Screen current display colums.
...			
0x90080	16	Hd Paras Table	Hard disk parameter table for the first one.
0x90090	16	Hd Paras Table	Hd parameter table for the second one (zero if none).
0x901FC	2	Root devie no	Root file system device number (set in bootsec.s)

Then the setup program moves the system module from 0x10000-0x8ffff to the absolute address 0x00000 (At the time, it was considered that the length of the kernel system module system would not exceed this value: 512 KB). Next, load the interrupt descriptor table register (IDTR) and the global descriptor table register (GDTR), turn on the A20 address line, reconfigure the two interrupt control chips 8259A, and reconfigure the hardware interrupt number to 0x20 - 0x2f. Finally, the CPU's control register CR0 (also called the machine status word) is set, enters the 32-bit protected mode, and jumps to the head.s program at the forefront of the system module to continue running.

In order to enable head.s to run in 32-bit protected mode, the interrupt descriptor table (IDT) and the global descriptor table (GDT) are temporarily set in the program, and the descriptor of the current kernel code and data segments are set in the GDT. In the head.s program below, these descriptor tables are also reconfigured according to the needs of the kernel.

First, let's review the format of the segment descriptor, the structure of the descriptor table, and the format of the segment selector. The format of the code segment and data segment descriptor used in the Linux kernel is shown in Figure 6-4. For the detailed meaning of each field, please refer to the description in Chapter 4.

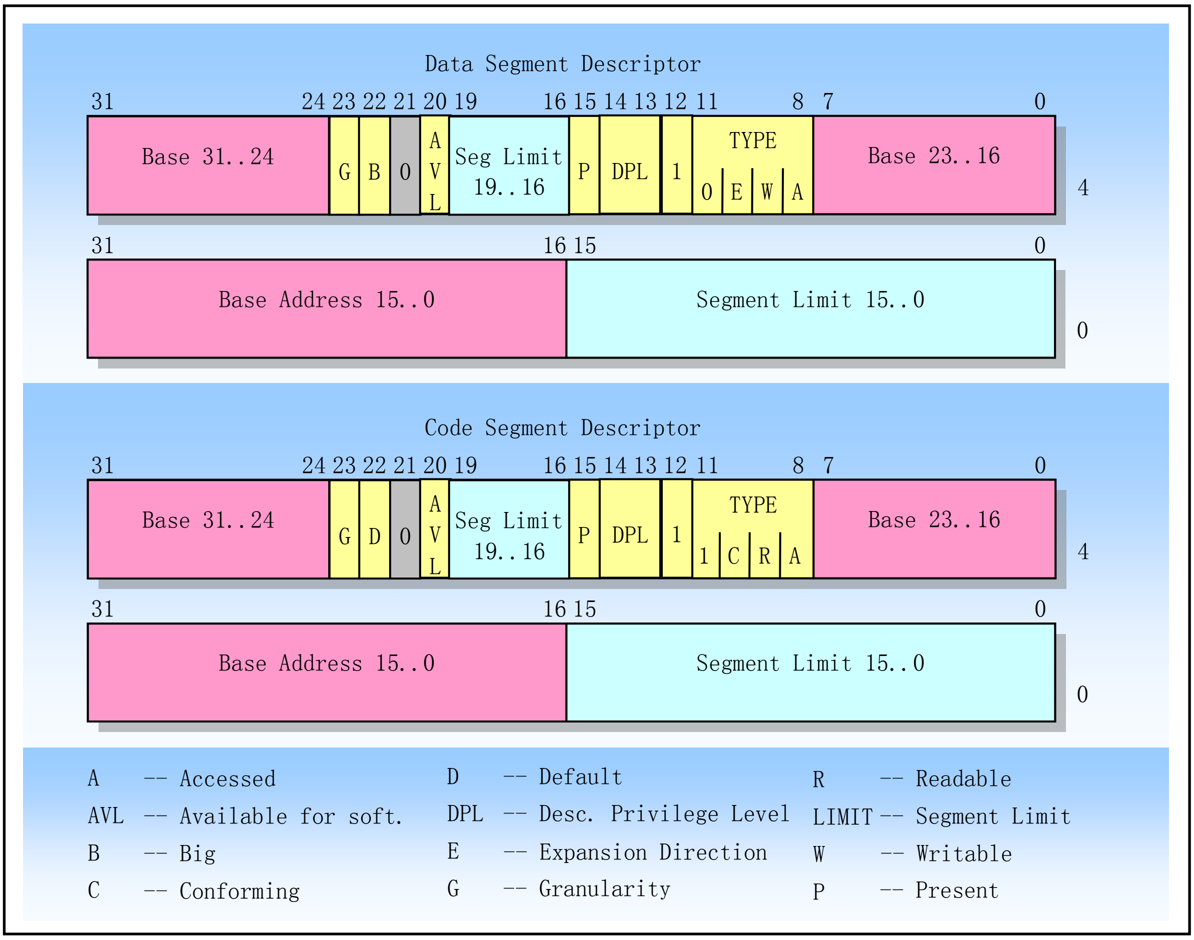


Figure 6-4 Descriptor format for code and data segments

The segment descriptor is stored in the descriptor table. The descriptor table is actually an array of descriptor items in memory. There are two types of descriptor tables: Global descriptor table (GDT) and Local descriptor table (LDT). The processor locates the GDT table and the current LDT table by using the GDTR and LDTR registers. These two registers hold the base address of the descriptor table and the length of the table in a linear address. The instructions LGDT and SGDT are used to access the GDTR register; the instructions LLDT and SLDT are used to access the LDTR register. The LGDT uses a 6-byte operand in memory to load the GDTR register. The first two bytes represent the length of the descriptor table, and the last four bytes are the base address of the descriptor table. Note, however, that the operand used by the LLDT instruction to access the LDTR register is a 2-byte operand representing the selector of a descriptor entry in the global descriptor table GDT. The descriptor item in the GDT table corresponding to the selector should correspond to a local descriptor table.

For example, the GDT descriptor item set by the setup.S program (see lines 567-578). The value of the code segment descriptor is 0x00C09A00000007FF (ie: 0x07FF, 0x0000, 0x9A00, 0x00C0). Indicates that the limit length of the code segment is 8MB (=(0x7FF + 1) * 4KB, where 1 is added because the limit length value is counted from 0), the base address of the segment in the linear address space is 0, and the segment type value 0x9A indicates that the segment exists in memory, the privilege level of the segment is 0, the segment type is a readable executable code segment, the segment code is 32 bits, and the granularity of the segment is 4 KB. The value of the data segment descriptor is 0x00C09200000007FF (ie: 0x07FF, 0x0000, 0x9200, 0x00C0), which means that the limit length of the data segment is 8MB, the base address of the segment in the linear address space is 0, the segment type value 0x92 indicates that the segment exists in the memory, and the privilege level of the segment is 0. The type is a readable and writable data segment, the segment code is 32 bits and the segment granularity is 4 KB.

Here are some more explanations for the selector. The selector part is used to specify a segment descriptor, which is done by specifying a descriptor table and indexing one of the descriptor items. Figure 6-5 shows the format of the selector.

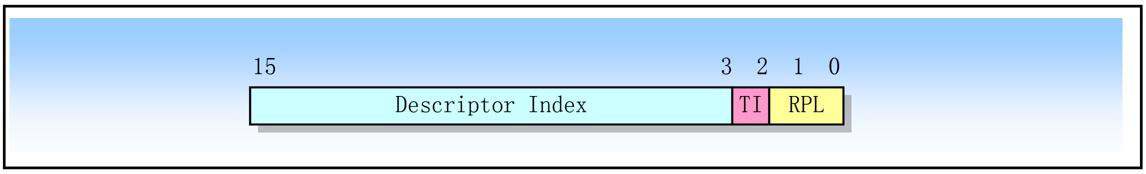


Figure 6-5 Segment selector format

The index is used to select one of the 8192 (2^13) descriptors in the specified descriptor table. The processor multiplies the index by 8, and adds the base address of the descriptor table to access the segment descriptor specified in the table. The Table Indicator (TI) is used to specify the descriptor table referenced by the selector. A value of 0 indicates that the GDT table is specified, and a value of 1 indicates that the current LDT table is specified. The Requestor's Privalege Level (RPL) is used to protect the mechanism.

Since the first item of the GDT table (index value 0) is not used, a selector with an index value of 0 and a table indicator value of 0 (that is, a selector pointing to the first item of the GDT) can be used as a null selector. When a segment register (cannot be CS or SS) loads an null selector, the processor does not generate an exception. However, an exception is generated if the segment register is used to access memory. This feature is useful for applications that initialize unused segment registers so that their unexpected references can produce a specified exception.

Before entering protected mode, we must first set up the segment descriptor table that will be used, such as the global descriptor table GDT. The instruction LGDT is then used to inform the CPU of the base address of the descriptor table (the base address of the GDT table is stored in the GDTR register). Then set the protection mode flag of the machine status word to enter the 32-bit protection mode.

In addition, the line 215-566 of the setup.S is used to identify the type of display card used in the machine. If the system uses a VGA display card, then we check to see if the display card supports an extended display mode (or display mode) of more than 25 lines x 80 columns. The so-called display mode refers to the method in which the ROM BIOS interrupts the function 0 (ah=0x00) of the INT 0x10 setting screen display information. The input parameter value in al register is the display mode or display mode number we want to set. Usually we refer to several display modes that can be set when IBM PC was first released as the standard display mode, and some modes added later are called extended display mode. For example, in addition to supporting the standard display mode, the ATI display card also supports extended display mode 0x23 and 0x33, that is, it can also display information on the screen using two display modes of 132 columns x 25 rows and 132 columns x 44 rows. When VGA and SVGA are just present, these extended display modes are supported by the BIOS on the display card. If a known type of display card is identified, the program will provide the user with an opportunity to select a resolution.

Since this part of the program involves port information unique to each of the many display cards, this fragment program is complicated. Fortunately, this part of the code has little to do with the kernel's operating principle, so you can skip it. If you want to understand this code thoroughly, then you should refer to Richard F. Ferraro's book "Programmer's Guide to the EGA, VGA, and Super VGA Cards", or refer to the classic VGA programming material that can be downloaded online:"VGADOC4". This part of the program was programmed by Mats Andersson (d88-man@nada.kth.se), and now Linus has forgotten who is d88-man :-).

6.3.2 **Code Comments**

Program 6-2 linux/boot/setup.S

```
!
!       setup.s         (C) 1991 Linus Torvalds
!
! setup.s is responsible for getting the system data from the BIOS,
! and putting them into the appropriate places in system memory.
! both setup.s and system has been loaded by the bootblock.
!
! This code asks the bios for memory/disk/other parameters, and
! puts them in a "safe" place: 0x90000-0x901FF, ie where the
! boot-block used to be. It is then up to the protected mode
! system to read them from there before the area is overwritten
! for buffer-blocks.
!

! NOTE! These had better be the same as in bootsect.s!
! config.h defines: DEF_INITSEG = 0x9000; DEF_SYSSEG = 0x1000; DEF_SETUPSEG = 0x9020
#include <linux/config.h>

INITSEG  = DEF_INITSEG            ! we move boot here - out of the way
SYSSEG   = DEF_SYSSEG             ! system loaded at 0x10000 (65536).
SETUPSEG = DEF_SETUPSEG           ! this is the current segment

.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

entry start
start:

! ok, the read went well so we get current cursor position and save it for
! posterity.

        mov     ax,#INITSEG       ! Set ds to INITSEG (0x9000)
        mov     ds,ax

! Get memory size (extended mem, kB)
! Use BIOS INT 0x15 function 0x88 to get extended memory size and store it at 0x90002.
! Return:
   ! ax = the extended memory from 0x100000(1MB) in KB. if error CF is set, ax = error code.
41
42         mov     ah,#0x88
43         int     0x15
44         mov     [2],ax            !  store at 0x90002
45
46 ! check for EGA/VGA and some config parameters
   ! Use BIOS INT 0x10 function 0x12 (video subsystem configuration) to get EGA configuration info.
   ! ah = 0x12;    bl = 0x10  -  return video configuration information.
   ! Return:
   ! bh = video state (0x00 - color, I/O port =0x3dX;   0x01 - mono, I/O port =0x3bX).
   ! bl = installed memory (0x00 - 64k; 0x01 - 128k; 0x02 - 192k; 0x03 = 256k).
   ! cx = adapter features and settings (see the description of INT 0x10 after this list).
47
48         mov     ah,#0x12
49         mov     bl,#0x10
50         int     0x10
51         mov     [8],ax            ! 0x90008 = ??
52         mov     [10],bx           ! 0x9000A = installed mem, 0x9000B = display state.
53         mov     [12],cx           ! 0x9000C = adapter features and settings.

   ! Detect screen rows and columns. If adapter is a VGA card, the user is requested
   ! to select the row and save it to 0x9000E.
54         mov     ax,#0x5019        ! set default row and colums in ax(ah = 80, al = 25).
55         cmp     bl,#0x10          ! If bl is 0x10, it means not a VGA card, jump.
56         je      novga
57         call    chsvga            ! get card manufacturer & type, modify row col(line 215)
58 novga:  mov     [14],ax           ! Save screen row & column values (0x9000E, 0x9000F).

   ! Use BIOS INT 0x10 function 0x03 to get cursor position, and save it in 0x90000 (2 bytes).
   ! ah = 0x03 get cursor position; bh = page number.
   ! Return:
   ! ch = start scan line;cl = end scan line;
   ! dh = row (0x00 is top); dl = colum(0x00 is left);
59         mov     ah,#0x03          ! read cursor pos
60         xor     bh,bh
61         int     0x10              ! save it in known place, con_init fetches
62         mov     [0],dx            ! it from 0x90000.
63
64 ! Get video-card data:
   ! Use BIOS INT 0x10, function 0x0f to get the current display mode and status.
   ! ah = 0x0f  -  get current video mode and state
   ! Return:
   ! ah = nr of screen colums; al = display mode; bh = current display page.
   ! 0x90004(word) - store current page; 0x90006 - display mode; 0x90007 - screen colums
65
66         mov     ah,#0x0f
67         int     0x10
68         mov     [4],bx           ! bh = display page
69         mov     [6],ax           ! al = video mode, ah = window width
70
71 ! Get hd0 data
   ! The address of the first hard disk parameter table is actually the vector value of
```

```
    ! interrupt 0x41! And the second hard disk parameter table is next to the first one. The
    ! vector value of the interrupt 0x46 also points to the second hard disk parameter table.
    ! The table size is 16 bytes. The following part copies the BIOS's two parameter tables to
    ! the new location: the first table is stored at 0x90080, the second is stored at 0x90090.
    ! For the description of the hard disk parameter table, see Table 6-4 in section 6.3.3.
 72
    ! The 75th line reads a long pointer value from memory and places it in the ds and si
    ! registers. Here, the 4 bytes stored at the memory 4 * 0x41 (= 0x104) are read. These 4
    ! bytes are the start address of the hard disk parameter table.
 73         mov     ax,#0x0000
 74         mov     ds,ax
 75         lds     si,[4*0x41]         ! Get INT 0x41 vector, addr of hd0 para table -> ds:si
 76         mov     ax,#INITSEG
 77         mov     es,ax
 78         mov     di,#0x0080          ! Destination of replication: 0x9000:0x0080 -> es:di
 79         mov     cx,#0x10            ! move total 16 bytes.
 80         rep
 81         movsb
 82
 83 ! Get hd1 data
 84
 85         mov     ax,#0x0000
 86         mov     ds,ax
 87         lds     si,[4*0x46]         ! INT 0x46 vector value -> ds:si
 88         mov     ax,#INITSEG
 89         mov     es,ax
 90         mov     di,#0x0090          ! 0x9000:0x0090 -> es:di
 91         mov     cx,#0x10
 92         rep
 93         movsb
 94
 95 ! Check that there IS a hd1 :-)
    ! Check if the machine has a second hard drive. If not, clear the second table.
    ! Use the ROM BIOS INT 0x13 function 0x15 to retrieve the disk type.
    ! ah = 0x15  -  get disk type.
    ! dl = drive number (0x8X for hard dirve, 0x80 - drive 0, 0x81 - drive 1)
    ! Return:
    ! ah = type code (00 - no drive; 01 - floppy, no change detection;
    !                 02  -  floppy (or other removable), change detection; 03  -  hard disk).
    ! cx:dx - number of 512-byte sectors.
    ! CF set on error, ah = status.
 96
 97         mov     ax,#0x01500
 98         mov     dl,#0x81
 99         int     0x13
100         jc      no_disk1
101         cmp     ah,#3               ! hard drive? (type == 3)?
102         je      is_disk1
103 no_disk1:
104         mov     ax,#INITSEG         ! no 2nd hard drive, clean 2nd parameter table.
105         mov     es,ax
106         mov     di,#0x0090
107         mov     cx,#0x10
```

```
108         mov     ax,#0x00
109         rep
110         stosb
111 is_disk1:
112
113 ! now we want to move to protected mode ...
114
115         cli                     ! no interrupts allowed !
116
117 ! first we move the system to it's rightful place
    ! The purpose of the following program is to move the entire system module to the 0x00000
    ! position, that is, move memory block (512KB) (0x10000 - 0x8ffff) to low end of memory.
118
119         mov     ax,#0x0000
120         cld                     ! 'direction'=0, movs moves forward
121 do_move:
122         mov     es,ax           ! destination segment ! es:di (0x0:0x0 initially)
123         add     ax,#0x1000
124         cmp     ax,#0x9000      ! Has the last seg (64KB from 0x8000 seg) code moved?
125         jz      end_move        ! yes, jump.
126         mov     ds,ax           ! source segment  ! ds:si (0x1000:0x0 initially)
127         sub     di,di
128         sub     si,si
129         mov     cx,#0x8000      ! move 0x8000 words (64K bytes)
130         rep
131         movsw
132         jmp     do_move
133
134 ! then we load the segment descriptors
    ! From here on, you will encounter 32-bit protected mode operation. See Chapter 4 for
    ! information on this. Before running into protected mode, we need to first set up the
    ! segment descriptor table to be used. Here you need to set the global descriptor table
    ! GDT and the interrupt descriptor table IDT.
    !
    ! The instruction, LIDT, is used to load the interrupt descriptor table register. Its
    ! operand (idt_48) has 6 bytes. The first 2 bytes (bytes 0-1) are the size of descriptor
    ! table; the last 4 bytes (bytes 2-5) are the 32-bit linear base of the descriptor table.
    ! See the following 580--486 lines. Each 8-byte entry in the IDT table indicates the code
    ! information that needs to be called when an interrupt occurs. It is somewhat similar to
    ! the interrupt vector, but contains more information.
    !
    ! The LGDT instruction is used to load the global descriptor table register with the same
    ! operand format as LIDT instruction. Each descriptor item (8 bytes) in the GDT describes
    ! the information of the data segment and code segment (block) in the protected mode. This
    ! includes the segment's maximum limit (16 bits), linear base address (32 bits), privilege
    ! level, in memory flag, read and write permissions, and other flags. See line 567--578.
135
136 end_move:
137         mov     ax,#SETUPSEG    ! right, forgot this at first. didn't work :-)
138         mov     ds,ax           ! ds point to this code segment (setup)
139         lidt    idt_48          ! load idt with 0,0
140         lgdt    gdt_48          ! load gdt with whatever appropriate
141
```

```
142 ! that was painless, now we enable A20
    ! In order to access and use more than 1MB of physical memory, we need to first enable the
    ! A20 address line. See the description of the A20 line after this programs. As for whether
    ! the machine actually enable the A20 line, we also need to test it after entering the
    ! protection mode (after more than 1MB memory can be accessed). This work is placed in the
    ! head.S program (32-36 lines).
143
144         call    empty_8042          ! Test 8042 status reg, wait for input buffer be empty.
                                        ! A write cmd can be run only if input buffer is empty.
145         mov     al,#0xD1            ! command write ! 0xD1 cmd code, write to P2 of 8042.
146         out     #0x64,al            ! Bit 1 of P2 is used for strobing of A20 line.
147         call    empty_8042          ! Waiting buffer to be empty, to see if cmd is accepted.
148         mov     al,#0xDF            ! A20 on        ! parameters for the A20 line.
149         out     #0x60,al            ! write to port 0x60.
150         call    empty_8042          ! if input buffer is empty, then A20 line enabled.
151
152 ! well, that went ok, I hope. Now we have to reprogram the interrupts :-(
153 ! we put them right after the intel-reserved hardware interrupts, at
154 ! int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
155 ! messed this up with the original PC, and they haven't been able to
156 ! rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
157 ! which is used for the internal hardware interrupts as well. We just
158 ! have to reprogram the 8259's, and it isn't fun.
159
    ! The PC uses two programmable interrupt controller chips 8259A. For the programming
    ! method of 8259A, please refer to the introduction after this program. The two words
    ! (0x00eb) defined on line 162 are two relative jump instructions that are directly
    ! represented by machine code, acting as a delay.
    !
    ! 0xeb is the opcode of the direct near jump instruction with a relative offset value of
    ! 1 byte. The CPU creates a new effective address by adding this relative offset to the
    ! EIP register. The number of CPU clock cycles spent on execution is 7 to 10. 0x00eb
    ! indicates an instruction whose jump offset is 0, so the next instruction is executed
    ! directly. These two instructions provide a total delay time of 14-20 CPU clock cycles.
    ! Because there is no mnemonic for the corresponding instruction in as86, Linus uses
    ! machine code directly to represent this instruction in some assembly files. In addition,
    ! the number of clock cycles per NOP instruction is 3, so 6 to 7 NOP instructions are
    ! required to achieve the same delay effect.
    !
    ! The 8259A chip master port is 0x20-0x21, and the slave port is 0xA0-0xA1. The output
    ! value 0x11 indicates the start of the initialization command, which is the ICW1 command
    ! word, indicating the edge trigger, multiple 8259 cascades, and requires the ICW4 command
    ! word to be sent last.
160         mov     al,#0x11                ! initialization sequence
161         out     #0x20,al                ! send it to 8259A-1
162         .word   0x00eb,0x00eb           ! jmp $+2, jmp $+2      ! '$' is current address.
163         out     #0xA0,al                ! and to 8259A-2
164         .word   0x00eb,0x00eb
    ! The Linux system hardware interrupt number is set to start at 0x20.
165         mov     al,#0x20                ! start of hardware int's (0x20)
166         out     #0x21,al                ! sned ICW2 cmd to master chip.
167         .word   0x00eb,0x00eb
168         mov     al,#0x28                ! start of hardware int's 2 (0x28)
```

```
        out     #0xA1,al                ! sned ICW2 cmd to slave chip.
        .word   0x00eb,0x00eb
        mov     al,#0x04                ! 8259-1 is master
        out     #0x21,al                ! ICW3 cmd, chain pin IR2 to pin INT of slave chip.
        .word   0x00eb,0x00eb
        mov     al,#0x02                ! 8259-2 is slave
        out     #0xA1,al                ! ICW3 cmd, chain pin INT to pin IR2 on master chip.
        .word   0x00eb,0x00eb
! 8086 mode. It means normal EOI, unbuffered mode, need to send instructions to reset.
! Initialization is over, chip ready.
        mov     al,#0x01                ! 8086 mode for both
        out     #0x21,al                ! ICW4 cmd (8086 mode).
        .word   0x00eb,0x00eb
        out     #0xA1,al                ! send ICW4 to slave chip.
        .word   0x00eb,0x00eb
        mov     al,#0xFF                ! mask off all interrupts for now
        out     #0x21,al
        .word   0x00eb,0x00eb
        out     #0xA1,al

! well, that certainly wasn't fun :-(. Hopefully it works, and we don't
! need no steenking BIOS anyway (except for the initial loading :-).
! The BIOS-routine wants lots of unnecessary data, and it's less
! "interesting" anyway. This is how REAL programmers do it.
!
! Well, now's the time to actually move into protected mode. To make
! things as simple as possible, we do no register set-up or anything,
! we let the gnu-compiled 32-bit programs do that. We just jump to
! absolute address 0x00000, in 32-bit protected mode.

! Below we set and enter the 32-bit protected mode to run. First load the machine status
! word (LMSW, also known as Control Register CR0), whose bit 0 is set to cause the CPU to
! switch to protected mode and run in privilege level 0, ie the current privilege level
! CPL=0. The segment register still points to the same linear address as in real-address
! mode (the linear address is the same as the physical memory address in real-address mode).
! After setting this bit, the next instruction must be an inter-segment jump instruction
! to flush the current instruction queue of the CPU.
!
! Because the CPU reads the instruction from memory and decodes it before executing an
! instruction. Therefore, those pre-fetched instructions that belong to the real mode
! after entering the protection mode become no longer valid. An inter-segment jump
! instruction will flush the current instruction queue of the CPU, that is, discard these
! invalid information. In addition, in the Intel company's manual, it is recommended that
! the CPU of 80386 or above should use the instruction "mov cr0, ax" to switch to the
! protection mode. The lmsw instruction is only for compatibility with previous 286 CPUs.

        mov     ax,#0x0001      ! protected mode (PE) bit
        lmsw    ax              ! This is it!
        jmpi    0,8             ! jmp offset 0 of segment 8 (cs)

! We have moved the system module to the beginning of 0x00000, so the offset address in
! the previous sentence is 0. The value 8 is already a segment selector in protected mode,
! which is used to select the descriptor table and its entry, and the required privilege
```

```
    ! level. The segment selector 8 (0b0000, 0000, 0000, 1000) indicates that the privilege
    ! level 0 is requested, and the second descriptor item in the GDT table is used. This entry
    ! indicates that the base address is 0 (see line 571), so the jump instruction here will
    ! execute the code in system module.
200
201 ! This routine checks that the keyboard command queue is empty
202 ! No timeout is used - if this hangs there is something wrong with
203 ! the machine, and we probably couldn't proceed anyway.
    !
    ! A write cmd can be executed only when input buffer is empty (status register bit 1 = 0).
204 empty_8042:
205         .word   0x00eb,0x00eb
206         in      al,#0x64        ! 8042 status port
207         test    al,#2           ! is input buffer full?
208         jnz     empty_8042      ! yes - loop
209         ret
210
    ! Note that the following 215--566 lines of code involve a lot of graphics card hardware
    ! information, so it is more complicated. However, since this code has little to do with
    ! the kernel, you can skip it first.

211 ! Routine trying to recognize type of SVGA-board present (if any)
212 ! and if it recognize one gives the choices of resolution it offers.
213 ! If one is found the resolution chosen is given by al,ah (rows,cols).
    !
    ! The following code first displays msg1 on line 588-589, then loops through the keyboard
    ! controller output buffer, waiting for user to press the button. If user presses Enter
    ! key, it checks the SVGA mode of the system and returns the maximum row and column values
    ! in AL and AH. Otherwise, the default of AL=25 rows, AH=80 columns is set and returned.
214
215 chsvga: cld
216         push    ds               ! Save ds, will be poped out on line 231 (or 490 or 492).
217         push    cs               ! ds = cs
218         pop     ds
219         mov     ax,#0xc000
220         mov     es,ax            ! es points to 0xc000 seg. it's BIOS area on VGA card.
221         lea     si,msg1          ! ds:si points to null terminated message msg1.
222         call    prtstr           ! displays msg1.
    ! First of all, please note that the scan code generated when a button is pressed is called
    ! make code. The scan code generated by releasing a pressed button is called break code.
    ! The following code read keyboard controller output buffer to get scan code or command.
    ! If the received scan code is smaller than 0x82, then it's a make code, because 0x82 is
    ! the minimum value of break code, and less than 0x82 means that the button has not been
    ! released. If the scan code is greater than 0xe0, it indicates that the extended scan
    ! code prefix is received. If the break code is 0x9c, it means user pressed/released the
    ! Enter key. The program then jumps to check if the system has SVGA mode. Otherwise, the
    ! return row and column are set to AL=25 rows and AH=80 columns by default.
223 nokey:  in      al,#0x60         ! read in scan code from the controller buffer.
224         cmp     al,#0x82         ! compare with minimum break code 0x82
225         jb      nokey            ! if less than it, no key is released yet.
226         cmp     al,#0xe0
227         ja      nokey            ! if great than 0xe0, it' s a prefix of code.
228         cmp     al,#0x9c
```

```
229         je      svga            ! if break code is 0x9c, enter key is pressed/released.
230         mov     ax,#0x5019      ! otherwise set al = 25, ah = 80
231         pop     ds
232         ret

    ! The following is based on the feature data string at the specified location in the ROM
    ! BIOS on the VGA card or the features supported to determine what brand of display card
    ! is installed on machine. The program supports a total of 10 display card extensions.
    ! Note that on line 220 the program has pointed es to the BIOS segment 0xc000 on the VGA
    ! card (see Chapter 2).

    ! First let's check if the display card is an ATI adapter.
    ! We point ds:si to the ATI card feature data string on line 595, and let es:si point to
    ! the specified location (offset 0x31) in the VGA BIOS. The feature string has a total of
    ! 9 characters ("761295520"), and we loop through the feature string. If they are the same,
    ! the VGA card in the machine is ATI brand. So let ds:si point to the row and column mode
    ! value dscati (line 615) that the display card can set, let di point to the number modes
    ! that can be set, and jump to the label selmod (438 lines) for further settings.
233 svga:   lea     si,idati        ! Check ATI 'clues'
234         mov     di,#0x31        ! the feature data is at 0xc000:0x0031
235         mov     cx,#0x09        ! 9 bytes
236         repe
237         cmpsb                   ! If 9 bytes are the same, means we have an ATI card.
238         jne     noati
    ! Ok, we know the adapter's barnd. Now let si points to the ATI display card optional
    ! row value table (dscati), di points to the extended mode number and extended mode number
    ! list (moati), then jumps to selmod (438 lines) to continue processing.
239         lea     si,dscati
240         lea     di,moati
241         lea     cx,selmod
242         jmp     cx

    ! Now let's test to see if it is the Ahead brand's display card.
    ! First, write the main enable register index 0x0f to be accessed to the EGA/VGA pattern
    ! index register 0x3ce, and write the open extension register flag value 0x20 to the 0x3cf
    ! port (in this case, corresponding to the main enable register). The main enable register
    ! value is then read through the 0x3cf port to check if the enable extended register flag
    ! can be set. If it can, it is a Ahead brand card. Note that when the word is output,
    ! al -> port n, ah -> port n+1.
243 noati:  mov     ax,#0x200f      ! Check Ahead 'clues'
244         mov     dx,#0x3ce       ! data port 0x0f -> 0x3ce port
245         out     dx,ax           ! set extend reg flag: 0x20 -> 0x3cf port
246         inc     dx              ! then check the flag to see if it has been set.
247         in      al,dx
248         cmp     al,#0x20        ! if it's 0x20, an Ahead A adapter found.
249         je      isahed          ! if it's 0x20, its an Ahead B adapter.
250         cmp     al,#0x21        ! if not a Ahead adapter, jump.
251         jne     noahed
    ! Ok, we know the adapter 's brand. Now let si points to the Ahead display card optional
    ! row value table (dscahead) , di points to the extended mode number and extended mode
    ! number list (moahead), then jumps to selmod (438 lines) to continue processing.
252 isahed: lea     si,dscahead
253         lea     di,moahead
```

```
254         lea     cx,selmod
255         jmp     cx

    ! Now let's check if it is a graphics card produced by Chips & Tech.
    ! The VGA enable register entry mode flag (bit 4) is set via port 0x3c3 (0x94 or 0x46e8),
    ! and the display card chipset identification value is then read from port 0x104. If the
    ! id is 0xA5, it means that it is a display card produced by Chips & Tech.
256 noahed: mov     dx,#0x3c3       ! Check Chips & Tech. 'clues'
257         in      al,dx           ! read enable reg from port 0x3c3, add setup flag(bit 4).
258         or      al,#0x10
259         out     dx,al
260         mov     dx,#0x104       ! read chip id from global id port 0x104, stored in bl.
261         in      al,dx
262         mov     bl,al
263         mov     dx,#0x3c3       ! reset setup flag to port 0x3c3.
264         in      al,dx
265         and     al,#0xef
266         out     dx,al
267         cmp     bl,[idcandt]    ! compare bl and id(0xA5) in idcandt( line 596).
268         jne     nocant
    ! Ok, we know the adapter brand is Chips & Tech. Now let si points to the card optional
    ! row value table (dsccandt), di points to the extended mode number and extended mode
    ! number list (mocandt), then jumps to selmod (438 lines) to continue processing.
269         lea     si,dsccandt
270         lea     di,mocandt
271         lea     cx,selmod
272         jmp     cx

    ! Now let us check if the card is a Cirrus display card.
    ! The detection method is to use the contents of the CRT controller index number 0x1f
    ! register to try to disable the extended function. This register is called the Eagle ID
    ! register. The value of the high and low nibbles is exchanged and written to the 6th
    ! index register of port 0x3c4. This operation should disable the extended function of the
    ! Cirrus display card. If it is not prohibited, it means that it is not a Cirrus display
    ! card. Because the content read from the 0x1f eagle register indexed by port 0x3d4 is the
    ! value after the XOR operation of the memory start address high byte register content
    ! corresponding to the eagle value and the 0x0c index number. Therefore, before reading
    ! the contents of 0x1f, we need to save the contents of the memory start high byte register
    ! and then clear it, and restore it after checking. In addition, writing the escaped Eagle
    ! ID value to the No. 6 sequence/extension register of the 0x3c4 port index will re-enable
    ! the extension.
273 nocant: mov     dx,#0x3d4       ! Check Cirrus 'clues'
274         mov     al,#0x0c        ! write reg index 0x0c to port 0x3d4 to get mem addr.
275         out     dx,al           !
276         inc     dx              ! read high byte of mem addr from port 0x3d5 to bl.
277         in      al,dx
278         mov     bl,al
279         xor     al,al
280         out     dx,al
281         dec     dx              ! write reg index 0x1f to port 0x3d4 to get Eagle ID.
282         mov     al,#0x1f
283         out     dx,al
284         inc     dx
```

```
285         in      al,dx               ! get Eagle ID from port 0x3d5, and store to bh.
286         mov     bh,al               ! swap nibbles and store to cl. left shift to ch.
287         xor     ah,ah               ! then put number 6 to cl.
288         shl     al,#4
289         mov     cx,ax
290         mov     al,bh
291         shr     al,#4
292         add     cx,ax
293         shl     cx,#8
294         add     cx,#6
    ! Finally, the cx value is stored in ax. At this time, ah is the "Eagle ID" value after
    ! transposition, and al is index number 6, which corresponds to the sequencing/extension
    ! register. Writing the ah to the 0x3c4 port indexing sequence/extension register should
    ! cause the Cirrus graphics card to disable extensions.
295         mov     ax,cx
296         mov     dx,#0x3c4
297         out     dx,ax
298         inc     dx
    ! If the extension is really disabled, then the value read in should be 0. If not, it
    ! means that it is not a Cirrus display card.
299         in      al,dx
300         and     al,al
301         jnz     nocirr
    ! Execution to this point indicates that the card in the machine may be a Cirrus display
    ! card. Then use the original value of "Eagle ID" saved in bh (line 286) to re-enable the
    ! Cirrus card extension function. The return value read should be 1. If not, it is still
    ! not a Cirrus display card.
302         mov     al,bh               !
303         out     dx,al               !
304         in      al,dx               !
305         cmp     al,#0x01
306         jne     nocirr
    ! Ok, now we know that the graphics card is a Cirrus brand. So first call the rst3d4
    ! subroutine to restore the CRT controller's display start address high byte register
    ! contents, then let si point to the brand display card's optional row value table
    ! (dsccurrus), di points to the extended mode number and the extended mode number list
    ! (mocirrus), then jump to selmod (line 438) to continue setting operation.
307         call    rst3d4
308         lea     si,dsccirrus
309         lea     di,mocirrus
310         lea     cx,selmod
311         jmp     cx
    ! This subroutine restores the display start address high byte register contents of the
    ! CRT controller using the value stored in bl (line 278).
312 rst3d4: mov     dx,#0x3d4
313         mov     al,bl
314         xor     ah,ah
315         shl     ax,#8
316         add     ax,#0x0c
317         out     dx,ax               ! note, word output, al -> 0x3d4, ah -> 0x3d5.
318         ret

    ! Now check if the Everex graphics card is in the system. The method is to call Everex's
    ! extended video BIOS function with interrupt 0x10 function 0x70 (ax =0x7000, bx=0x0000).
    ! For an Everes type display card, the interrupt call should return to the simulation
    ! state, ie the following return information:
    ! al = 0x70, if it is a Trident-based Everex display card;
    ! cl = type: 00-mono; 01-CGA; 02-EGA; 03-digital multi-freq; 04-PS/2; 05-IBM 8514; 06-SVGA.
    ! ch = attr: Bit7-6 :00-256K,01-512K,10-1MB,11-2MB;Bit4-Enable VGA protect; Bit0-6845Simu.
    ! dx = board model: Bit 15-4: board type id; bit 3-0: board correction id.
    !      0x2360-Ultragraphics II；0x6200-Vision VGA；0x6730-EVGA；0x6780-Viewpoint。
    ! di = The video BIOS version number represented in BCD code.
319 nocirr: call    rst3d4          ! Check Everex 'clues'
320         mov     ax,#0x7000      ! int 0x10 with ax = 0x7000, bx=0x0000.
321         xor     bx,bx
322         int     0x10
323         cmp     al,#0x70        ! al should contain 0x70 for Everex card.
324         jne     noevrx
325         shr     dx,#4           ! ignore board fix number(bit3-0).
326         cmp     dx,#0x678       ! if board type is 0x678, its a Trident card.
327         je      istrid
328         cmp     dx,#0x236       ! if board type is 0x236, also a Trident card.
329         je      istrid
    ! Ok, now we know that the card is a Everex brand. So first we let si point to the card's
    ! optional row value table (dsceverex), di points to the extended mode number and the
    ! extended mode number list (moeverex), then jump to selmod (line 438) to continue setting .
330         lea     si,dsceverex
331         lea     di,moeverex
332         lea     cx,selmod
333         jmp     cx
334 istrid: lea     cx,ev2tri       ! Everex card with a Trident type, jump to ev2tri
335         jmp     cx

    ! Now check if it is a Genoa display card. The way is to check the feature number string
    ! (0x77, 0x00, 0x66, 0x99) in its video BIOS. Note that at this time es has been set to
    ! the segment 0xc000 where the ROM BIOS is located on the VGA card.
336 noevrx: lea     si,idgenoa      ! Check Genoa 'clues'
337         xor     ax,ax           ! ds:si points to feature data.
338         seg es
339         mov     al,[0x37]       ! get feature data from VGA card at 0x37.
340         mov     di,ax           !  es:di point to 0x37.
341         mov     cx,#0x04
342         dec     si
343         dec     di
344 l1:     inc     si              !  compare the 4 feature bytes.
345         inc     di
346         mov     al,(si)
347         seg es
348         and     al,(di)
349         cmp     al,(si)
350         loope   l1
351         cmp     cx,#0x00
352         jne     nogen
    ! Ok, now we know that the card is a Genoa card. So we let si point to the card's
    ! optional row value table (dscgenoa), di points to the extended mode number and the
    ! extended modes list (mogenoa), then jump to selmod (line 438) to continue setting .
```

```
353         lea     si,dscgenoa
354         lea     di,mogenoa
355         lea     cx,selmod
356         jmp     cx

    ! Now check if it is a Paradise display card. The same is true for comparing the feature
    ! strings ("VGA=") in the BIOS on the display card.
357 nogen:  lea     si,idparadise    ! Check Paradise 'clues'
358         mov     di,#0x7d         ! es:di point to 0xc000:0x007d.
359         mov     cx,#0x04         ! there should be 4 bytes: “VGA=”
360         repe
361         cmpsb
362         jne     nopara
    ! Ok, we know the card is a Paradise card. So we let si point to the card's optional
    ! row value table (dscparadise), di points to the extended mode number and the extended
    ! modes list (moparadise), then jump to selmod (line 438) to continue setting.
363         lea     si,dscparadise
364         lea     di,moparadise
365         lea     cx,selmod
366         jmp     cx

    ! Now check if it is a Trident (TVGA) card. Bits 3--0 of the TVGA display card expansion
    ! mode control register 1 (0x0e of the 0x3c4 port index) are 64K memory page values. This
    ! field value has a property: when writing, we need to first XOR the value with 0x02 and
    ! then write; when reading the value, no XOR operation is required. That is, the value
    ! before XOR should be the same as the value read after writing. The following code uses
    ! this feature to check if it is a Trident display card.
367 nopara: mov     dx,#0x3c4        ! Check Trident 'clues'
368         mov     al,#0x0e         ! output index 0x0e (mode ctrl reg 1) to port 0x3c4
369         out     dx,al            ! read original value from port 0x3c5 and store to al
370         inc     dx
371         in      al,dx
372         xchg    ah,al
    ! Then we write 0x00 to this register and read its value ->al. Writing 0x00 is equivalent
    ! to the value written after "original value" 0x02 or 0x02, so if it is a Trident card,
    ! the value read after this should be 0x02. After swapping, a = the value of the original
    ! mode control register 1, ah = the last read value.
373         mov     al,#0x00
374         out     dx,al
375         in      al,dx
376         xchg    al,ah
377         mov     bl,al            ! Strange thing ... in the book this wasn't
378         and     bl,#0x02         ! necessary but it worked on my card which
379         jz      setb2            ! is a trident. Without it the screen goes
380         and     al,#0xfd         ! blurred ...
381         jmp     clrb2            !
382 setb2:  or      al,#0x02         !
383 clrb2:  out     dx,al
384         and     ah,#0x0f         ! get page number field (bit 3-0) (line 375)
385         cmp     ah,#0x02         ! if equal 0x02, it's a Trident card.
386         jne     notrid
    ! Ok, we know the card is a Trident card. So we let si point to the card's optional
    ! row value table (dsctrident), di points to the extended mode number and the extended
! modes list (motrident), then jump to selmod (line 438) to continue setting.
387 ev2tri: lea     si,dsctrident
388         lea     di,motrident
389         lea     cx,selmod
390         jmp     cx

    ! Now check if it is a Tseng card (ET4000AX or ET4000/W32). The method is to perform read
    ! and write operations on the Segment Select register corresponding to the 0x3cd port.  The
    ! upper 4 bits (bits 7--4) of the register are the 64KB segment number (Bank number) to be
    ! read, and the lower 4 bits (bits 3-0) are the segment numbers specified for writing.  If
    ! the value of the specified segment select register is 0x55 (indicating reading and
    ! writing the sixth 64KB segment), then for the Tseng display card, writing the value to
    ! the register should be still 0x55.
391 notrid: mov     dx,#0x3cd        ! Check Tseng 'clues'
392         in      al,dx            ! Could things be this simple ! :-)
393         mov     bl,al            ! read original seg selector data from 0x3cd to bl.
394         mov     al,#0x55         ! write to it with value 0x55, and read again to ah.
395         out     dx,al
396         in      al,dx
397         mov     ah,al
398         mov     al,bl            ! restore original data.
399         out     dx,al
400         cmp     ah,#0x55         ! if read value equal to write, it's a Tseng card.
401         jne     notsen
    ! Ok, we know the card is a Tseng card. So we let si point to the card's optional
    ! row value table (dsctseng), di points to the extended mode number and the extended
    ! modes list (motseng), then jump to selmod (line 438) to continue setting.
402         lea     si,dsctseng
403         lea     di,motseng
404         lea     cx,selmod
405         jmp     cx

    ! Check if it is a Video7 display card. Port 0x3c2 is the mixed output register write port
    ! and 0x3cc is the mixed output register read port. Bit 0 of this register is a mono/color
    ! flag. If it is 0, it means mono, otherwise it is color. The way to determine whether the
    ! Video7 card is used is to use the CRT control extension identification register (index
    ! number is 0x1f). The value of this register is actually the result of the XOR operation
    ! of the memory address high byte register (index number 0x0c) and value 0xea. Therefore,
    ! we only need to write a specific value to the memory start address high byte register,
    ! and then read the identification value from the identification register to check.
406 notsen: mov     dx,#0x3cc        ! Check Video7 'clues'
407         in      al,dx
408         mov     dx,#0x3b4        ! set dx to mono control index register port 0x3b4.
409         and     al,#0x01         ! If bit0 of mixed output reg is 0(mono), jump directly.
410         jz      even7            ! Otherwise set dx to color control index reg port 0x3d4.
411         mov     dx,#0x3d4
412 even7:  mov     al,#0x0c         ! Set index to 0x0c for the mem address high byte reg.
413         out     dx,al
414         inc     dx
415         in      al,dx            ! read high byte reg of vmem address , save to bl.
416         mov     bl,al
417         mov     al,#0x55         ! then write 0x55 to high byte reg and read it out.
```

```
        out     dx,al
        in      al,dx
! Then select the Video7 display card identification register whose index number is 0x1f
! through the CRT index register port 0x3b4 or 0x3d4. The contents of this register are
! actually the result of the XOR of the memory start address and the value 0xea.
        dec     dx
        mov     al,#0x1f
        out     dx,al
        inc     dx
        in      al,dx            ! read Video7 card id register value and save it in bh.
        mov     bh,al
        dec     dx               ! select addr high byte reg to restore its original value.
        mov     al,#0x0c
        out     dx,al
        inc     dx
        mov     al,bl
        out     dx,al
! Then we will verify that the "Video7 display card identification register value is the
! result value of the memory memory start address high byte and 0xea after XOR operation".
! Therefore, the result of the XOR operation of 0x55 and 0xea should be equal to the test
! value of the identification register. If it is not a Video7 card, set the default display
! row and column value (492 lines). Otherwise it is the Video7 card. So let si point to
! the display card row value table (dscvideo7), let di point to the number of extended
! mode and mode number list (movideo7).
        mov     al,#0x55
        xor     al,#0xea
        cmp     al,bh
        jne     novid7
        lea     si,dscvideo7
        lea     di,movideo7

! Through the inspection and analysis of the above display card and the Video7 display
! card here, we can see that the inspection process is usually divided into three basic
! steps. The first is to read and save the original value of the register that is needed
! for the test, then use the specific test value for the write and read operations, and
! finally restore the original register value and make a judgment on the check result.
!
! The following is based on the above-mentioned code to determine the type of the display
! card and the related extended mode information (list of row and column values pointed to
! by si; di points to the number of extended modes and the list of mode numbers), prompting
! the user to select an available display mode and setting it to display mode accordingly.
! Finally, the subroutine returns the screen row and column values currently set by the
! system (ah = columns; al = rows). For example, if the system is an ATI graphics card,
! the following message will appear on the screen:
! Mode:  COLSxROWS:
! 0.     132 x 25
! 1.     132 x 44
! Choose mode by pressing the corresponding number.
!
! The following code will display the null-terminated string "Mode: COLSxROWS:" on the screen.
selmod: push    si
        lea     si,msg2
        call    prtstr
```

```
441         xor     cx,cx
442         mov     cl,(di)             ! cl is the extended modes of the checked card.
443         pop     si
444         push    si
445         push    cx
    ! Then, the extended mode rows and columns selectable by the current display card are
    ! displayed for user to select.
446 tbl:    pop     bx                  ! bx = total extend modes number.
447         push    bx
448         mov     al,bl
449         sub     al,cl
450         call    dprnt               ! display the value in decimal format.
451         call    spcing              ! a dot, and tehn 4 spaces.
452         lodsw                       ! load row & column pointed to by si in ax, then si++.
453         xchg    al,ah               ! swap, al = columns.
454         call    dprnt               ! display column number.
455         xchg    ah,al               ! al = rows.
456         push    ax
457         mov     al,#0x78            ! show "x"
458         call    prntl
459         pop     ax                  ! al= row number
460         call    dprnt               ! display row number.
461         call    docr                ! cr,lf
462         loop    tbl                 ! display next row colums, mode number decreased by 1.
    ! Then display prompt string "Choose mode by pressing the corresponding number."
463         pop     cx                  ! cl = total extend modes number.
464         call    docr
465         lea     si,msg3             ! "Choose mode by pressing the corresponding number."
466         call    prtstr

    ! Then, the scan code of the user button is read from the keyboard port, the row and
    ! column mode number selected by the user is determined according to the scan code, and
    ! the corresponding display mode is set by using the ROM BIOS INT 0x10 function 0x00.
    ! The "mode number + 0x80" on line 468 is the break scan codes of the number key -1
    ! pressed. For the 0--9 number keys, their break codes are:
    !       0 - 0x8B; 1 - 0x82; 2 - 0x83; 3 - 0x84; 4 - 0x85;
    !       5 - 0x86; 6 - 0x87; 7 - 0x88; 8 - 0x89; 9 - 0x8A
    ! Therefore, if the read break code is less than 0x82, it means that it's not a numeric
    ! key; if the scan code is equal to 0x8B, it means that the user pressed number 0 key.
467         pop     si                  ! pop up original row & column table pointer.
468         add     cl,#0x80            ! cl + 0x80 = the break code for the "number key -1".
469 nonum:  in      al,#0x60            ! Quick and dirty...
470         cmp     al,#0x82            ! less than 0x82 ? ignore it.
471         jb      nonum
472         cmp     al,#0x8b            ! scan code = 0x8b? it's number key 0.
473         je      zero
474         cmp     al,cl               ! great than the number of modes?
475         ja      nonum               ! non number key pressed.
476         jmp     nozero

    ! Next, the break scan code is converted into a corresponding digital key value, and then
    ! the corresponding mode number is selected from the mode number and the mode number list
    ! by using the value. Then call the ROM BIOS interrupt INT 0x10 function 0 to set the
    ! screen to the mode specified by the mode number. Finally, use the mode number to select
    ! from the display card row and column table and return the corresponding row and column
    ! values in ax.
477 zero:   sub     al,#0x0a         ! al = 0x8b - 0x0a = 0x81
478 nozero: sub     al,#0x80         ! subtract 0x80 to obtain the mode selected by user.
479         dec     al               ! count from 0
480         xor     ah,ah            ! set display mode
481         add     di,ax
482         inc     di               ! di points to the mode number (skip the first).
483         push    ax
484         mov     al,(di)          ! mode number -> al, call int to set mode.
485         int     0x10
486         pop     ax
487         shl     ax,#1            ! mode nr x 2: pointer into the row & column table.
488         add     si,ax
489         lodsw                    ! get row and colum to ax (ah = colum, al = row).
490         pop     ds               ! restore ds saved on line 216. return value in ax.
491         ret

    ! If none of the graphics cards tested above, then we have to use the default 80 x 25
    ! standard row and column values.
492 novid7: pop     ds               ! Here could be code to support standard 80x50,80x30
493         mov     ax,#0x5019
494         ret
495
496 ! Routine that 'tabs' to next col.
497
    ! display a dot '.' and four spaces
498 spcing: mov     al,#0x2e         ! a dot '.'
499         call    prnt1
500         mov     al,#0x20
501         call    prnt1
502         mov     al,#0x20
503         call    prnt1
504         mov     al,#0x20
505         call    prnt1
506         mov     al,#0x20
507         call    prnt1
508         ret
509
510 ! Routine to print asciiz-string at DS:SI
511
512 prtstr: lodsb
513         and     al,al
514         jz      fin
515         call    prnt1            ! print a char in al
516         jmp     prtstr
517 fin:    ret
518
519 ! Routine to print a decimal value on screen, the value to be
520 ! printed is put in al (i.e 0-255).
521
522 dprnt:  push    ax
```

```
        push    cx
        mov     ah,#0x00
        mov     cl,#0x0a
        idiv    cl
        cmp     al,#0x09
        jbe     lt100
        call    dprnt
        jmp     skip10
lt100:  add     al,#0x30
        call    prnt1
skip10: mov     al,ah
        add     al,#0x30
        call    prnt1
        pop     cx
        pop     ax
        ret

! Part of above routine, this one just prints ascii al
! This subroutine uses the interrupt 0x10 function 0x0E to write a character on the screen
! by telex. The cursor will automatically move to the next position. If a line is written,
! the cursor will move to the beginning of the next line. If the last line of a screen has
! been written, the entire screen will scroll up one line. The characters 0x07 (BEL), 0x08 (BS),
! 0x0A (LF), and 0x0D (CR) are not displayed as commands.
! Input: AL -- character; BH -- page number; BL -- foreground color (in graphic mode).

prnt1:  push    ax
        push    cx
        mov     bh,#0x00        ! page number.
        mov     cx,#0x01
        mov     ah,#0x0e
        int     0x10
        pop     cx
        pop     ax
        ret

! Prints <CR> + <LF>

docr:   push    ax
        push    cx
        mov     bh,#0x00
        mov     ah,#0x0e
        mov     al,#0x0a
        mov     cx,#0x01
        int     0x10
        mov     al,#0x0d
        int     0x10
        pop     cx
        pop     ax
        ret


! Start here is the global descriptor table GDT. It consists of multiple 8-byte long
! descriptor entries. Three descriptor items are given here. The first entry is useless
```

```
    ! (568 lines), but it must exist. The second item is the system code segment descriptor
    ! (lines 570-573), and the third is the system data segment descriptor (lines 575-578).
567 gdt:
568         .word   0,0,0,0         ! dummy
569
    ! The offset here in GDT is 0x08. It happens to be the value of kernel code selector.
570         .word   0x07FF          ! 8Mb - limit=2047  (0--2047, so 2048*4096=8Mb)
571         .word   0x0000          ! base address=0
572         .word   0x9A00          ! code read/exec
573         .word   0x00C0          ! granularity=4096, 386
574
    ! The offset here in GDT is 0x10. It happens to be the value of kernel data selector.
575         .word   0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
576         .word   0x0000          ! base address=0
577         .word   0x9200          ! data read/write
578         .word   0x00C0          ! granularity=4096, 386
579
    ! Below is the 6-byte operand required by the instruction lidt that loads the interrupt
    ! descriptor table register. The first 2 bytes are the limit of the IDT table, and the
    ! last 4 bytes are the 32-bit base address of the idt table in the linear address space.
    ! The CPU requires that the IDT table be set before entering the protection mode, so here
    ! an empty table of length 0 is set first.
580 idt_48:
581         .word   0               ! idt limit=0
582         .word   0,0             ! idt base=0L
583
    ! This is the 6-byte operand required by the instruction lgdt to load the global
    ! descriptor table register. The first 2 bytes are the limit length of the gdt table, and
    ! the last 4 bytes are the linear base address of the gdt table. The global table size is
    ! set to 2KB (0x7ff). Since every 8 bytes constitutes a segment descriptor item, there are
    ! a total of 256 entries in the table.
    ! The 4-byte linear base address is 0x0009<<16 + 0x0200 + gdt, which is 0x90200 + gdt.
    ! (The symbol gdt is the offset address of the global table in this block, see line 205)
584 gdt_48:
585         .word   0x800           ! gdt limit=2048, 256 GDT entries
586         .word   512+gdt,0x9     ! gdt base = 0X9xxxx
587
588 msg1:   .ascii  "Press <RETURN> to see SVGA-modes available or any other key to continue."
589                 db      0x0d, 0x0a, 0x0a, 0x00
590 msg2:           .ascii  "Mode:  COLSxROWS:"
591                 db      0x0d, 0x0a, 0x0a, 0x00
592 msg3:           .ascii  "Choose mode by pressing the corresponding number."
593                 db      0x0d, 0x0a, 0x00
594
    ! Below are the feature data strings for the four display cards.
595 idati:          .ascii  "761295520"
596 idcandt:        .byte   0xa5                    ! idcandt means “ID of Chip AND Tech.”
597 idgenoa:        .byte   0x77, 0x00, 0x66, 0x99
598 idparadise:     .ascii  "VGA="
599

    ! The following is a list of the number of extended modes and corresponding mode numbers
    ! that can be used by various display cards. The first byte of each line is the number of
    ! modes, and the subsequent values are the mode numbers that can be used by interrupt 0x10
    ! function 0 (AH=0). For example, from line 602, for the ATI brand card, two extended
    ! modes can be used in addition to the standard mode: 0x23 and 0x33.
600 ! Manufacturer:   Numofmodes:   Mode:
601
602 moati:          .byte   0x02,   0x23, 0x33
603 moahead:        .byte   0x05,   0x22, 0x23, 0x24, 0x2f, 0x34
604 mocandt:        .byte   0x02,   0x60, 0x61
605 mocirrus:       .byte   0x04,   0x1f, 0x20, 0x22, 0x31
606 moeverex:       .byte   0x0a,   0x03, 0x04, 0x07, 0x08, 0x0a, 0x0b, 0x16, 0x18, 0x21, 0x40
607 mogenoa:        .byte   0x0a,   0x58, 0x5a, 0x60, 0x61, 0x62, 0x63, 0x64, 0x72, 0x74, 0x78
608 moparadise:     .byte   0x02,   0x55, 0x54
609 motrident:      .byte   0x07,   0x50, 0x51, 0x52, 0x57, 0x58, 0x59, 0x5a
610 motseng:        .byte   0x05,   0x26, 0x2a, 0x23, 0x24, 0x22
611 movideo7:       .byte   0x06,   0x40, 0x43, 0x44, 0x41, 0x42, 0x45
612
    ! Below is a list of columns and rows for the modes that can be used with various brands
    !  of VGA cards. For example, line 615 indicates that the column and row values of the
    !  two extension modes for ATI card are 132 x 25 and 132 x 44, respectively.
613 !                       msb = Cols   lsb = Rows:
614
615 dscati:         .word   0x8419, 0x842c
616 dscahead:       .word   0x842c, 0x8419, 0x841c, 0xa032, 0x5042
617 dsccandt:       .word   0x8419, 0x8432
618 dsccirrus:      .word   0x8419, 0x842c, 0x841e, 0x6425
619 dsceverex:      .word   0x5022, 0x503c, 0x642b, 0x644b, 0x8419, 0x842c, 0x501e, 0x641b, 0xa040, 0x841e
620 dscgenoa:       .word   0x5020, 0x642a, 0x8419, 0x841d, 0x8420, 0x842c, 0x843c, 0x503c, 0x5042, 0x644b
621 dscparadise:    .word   0x8419, 0x842b
622 dsctrident:     .word   0x501e, 0x502b, 0x503c, 0x8419, 0x841e, 0x842b, 0x843c
623 dsctseng:       .word   0x503c, 0x6428, 0x8419, 0x841c, 0x842c
624 dscvideo7:      .word   0x502b, 0x503c, 0x643c, 0x8419, 0x842c, 0x841c
625
626 .text
627 endtext:
628 .data
629 enddata:
630 .bss
631 endbss:
```

6.3.3 Reference information

In order to get the basic parameters of the machine and display messages of the boot process to user, this program calls interrupt services in the BIOS multiple times and starts to involve some access operations to the hardware ports. The following briefly describes several BIOS interrupt services used and explains the cause of the A20 address line problem. Finally we also mentioned the issues of the 80X86 CPU 32-bit protection mode operation.

6.3.3.1 Current memory image

After the execution of the setup.s program, the system module is moved to the beginning of the physical memory at address 0x00000, and from the location 0x90000, some basic system parameters that the kernel will use are stored, as shown in Figure 6-6.

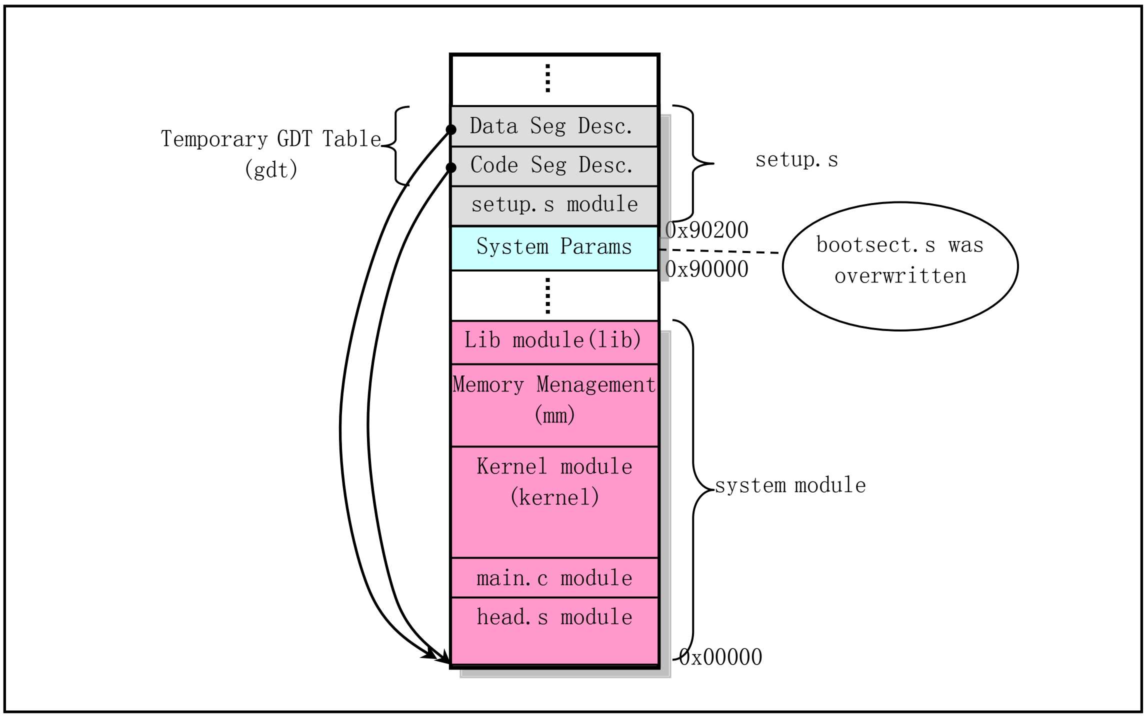


Figure 6-6 Diagram of memory map after setup.s ends

At this point, there are three descriptors in the temporary global table GDT. The first one is NULL not used, the other two are code and data segment descriptors. They all point to the beginning of the system module, which is the physical memory address 0x00000. Thus, when the last instruction 'jmp 0,8 ' (line 193) is executed, it will jump to the beginning of the head.s program to continue execution. The '8' in this instruction is the value of the segment selector, which is used to specify the descriptor item to be used. This is the code segment descriptor in the GDT. '0' is the offset in the code segment specified by the descriptor.

6.3.3.2 BIOS Video Interrupt 0x10

This section describes the ROM BIOS video interrupt service function used in the above program. See Table 6-3 for a description of the function of obtaining display card information (other auxiliary function selection). Other display service functions are given in the program comments.

Table 6-3 Obtain display card information (function: ah = 0x12, bl = 0x10)

Input/Return	Register	Description
Input Information	ah	Function No. = 0x12, Obtain display card information.
	bl	Sub-Function No. = 0x10
Return Information	bh	Video Status: 0x00 - Color mode (the video hardware I/O port base address is 0x3DX); 0x01 - Mono mode (the video hardware I/O port base address is 0x3BX); (where the X value in the port address can be 0 -- F)
	bl	Installed video memory size: 00 = 64K, 01 = 128K, 02 = 192K, 03 = 256K

	ch	Feature connector bit information:
		Bits 0-1 Feature line 1-0, Status 2;
		Bits 2-3 Feature line 1-0, Status 1;

		Bits 4-7 Not used (set to 0)
	cl	Video switch settings: Bits 0-3 correspond to switches 1-4. Bits 4-7 is not used. Original EGA/VGA switch settings: 0x00 MDA/HGC; 0x01-0x03 MDA/HGC; 0x04 CGA 40x25; 0x05 CGA 80x25; 0x06 EGA + 40x25; 0x07-0x09 EGA + 80x25; 0x0A EGA + 80x25Mono; 0x0B EGA + 80x25Mono.

6.3.3.3 Hard Drive Basic Parameter Table ("INT 0x41")

In the ROM BIOS interrupt vector table, the interrupt vector location of INT 0x41 (4 * 0x41 =0x0000:0x0104) stores not the address of the interrupt program, but the address of the basic parameter table of the first hard disk. For the BIOS of an IBM PC fully compatible machine, the address stored here is F000h:E401h. The basic parameter table address for the second hard disk is stored at the INT 0x46 interrupt vector location.

Table 6-4 Hard disk basic parameter table

Offset	Size	Name	Description
0x00	word	cyl	Number of cylinders
0x02	byte	head	Number of heads
0x03	word		Start cylinder to reducing the write current (only for PC/XT, others are 0)
0x05	word	wpcom	Pre-compensation cylinder number before start writing (multiplied by 4)
0x07	byte		Maximum ECC burst size (only for PC/XT, others are 0)
0x08	byte	ctl	Control byte (driver step selection): Bit 0 - Not used (0); Bit 1 - Reserved (0) (Close IRQ) Bit 2 - Allow reset; Bit 3 - Set if number of heads great than 8 Bit 4 - Not used (0); Bit 5 - Set if there is bad map at cylinder number +1 Bit 6 - Disable ECC retry; Bit 7 - Disable Access retry.
0x09	byte		Standard timeout value (only for PC/XT, others are 0)
0x0A	byte		Format timeout value (only for PC/XT, others are 0)
0x0B	byte		Detect drive timeout value (only for PC/XT, others are 0)
0x0C	word	lzone	Head landing (stop) cylinder number
0x0E	byte	sect	Number of sectors per track
0x0F	byte		Reserved.

6.3.3.4 A20 address line problem

In August 1981, IBM's original personal computer IBM PC used a 16-bit Intel 8088 CPU. The CPU has a 16-bit internal (8-bit external) data bus and a 20-bit address bus width. Therefore, there are only 20 address lines (A0 – A19) in this PC, and the CPU can address only up to 1MB of memory range. At the time when the popular machine memory capacity was only a few tens of KB and several hundred KB, 20 address lines were enough to address the memory. The highest address that it can address is 0xffff:0xffff, which is 0x10ffef. For memory addresses that exceed 0x100000 (1MB), the CPU will default wrap around to the 0x0ffef position.

When IBM introduced the new PC/AT model in 1985, it used the Intel 80286 CPU. It has 24 address lines, can address up to 16MB of memory, and has a real-mode of operation that is fully compatible with the 8088. However, when the addressing value exceeds 1MB, it cannot implement addressing surround like the 8088 CPU. But at the time there were programs that were designed to work with this address wrapping mechanism. Therefore, in order to achieve full compatibility with the original PC, IBM invented the use of a switch to enable or disable the 0x100000 address bit. Since there was just a free port pin (output port P2, pin P21) on the keyboard controller 8042 at the time, this pin was used as an AND gate to control this address bit. This signal is called A20. If it is zero, then bits 20 and above are cleared, thus achieved the compatibility of memory addressing. IBM's 80X86-based machines since then have also inherited this feature. For details on the keyboard controller 8042 chip, see the description after the kernel/chr_drv/keyboard.S program.

For compatibility, the A20 address line is disabled by default when the machine is booted, so the operating system of the 32-bit machine must use the appropriate method to enable it. However, due to the different chipsets used by various compatible machines, it is very troublesome to do this. Therefore, it is usually necessary to choose among several control methods.

A common method of controlling the A20 signal line is to set the port value of the keyboard controller. The typical control method is used by the setup.s program (lines 138-144). For other compatible microcomputers, other methods can be used to control the A20 line. Some operating systems use the A20's enable and disable as part of the standard process of converting between real mode and protected mode of operation. Since the controller of the keyboard is very slow, it is not possible to operate the A20 line using the keyboard controller. To this end, an A20 fast door option (Fast Gate A20) was introduced. It uses I/O port 0x92 to handle the A20 signal line, eliminating the need for slow keyboard controller operation. For systems without a keyboard controller, only the 0x92 port can be used for control. However, the port may also be used by devices on other compatible microcomputers (such as display chips), resulting in system error operation. Another way is to open the A20 signal line by reading the 0xee port, and writing the port will disable the A20 signal line.

6.3.3.5 Programming method of 8259A interrupt controller

In Chapter 2 we have outlined the basic workings of the interrupt mechanism and the hardware interrupt subsystem used in PC/AT compatible computers. Here we first introduce the working principle of the 8259A chip, and then explain in detail the programming method of the 8259A chip and how the Linux kernel works.

1. 8059A chip working principle

As mentioned earlier, a cascade of two 8259A programmable controller (PIC) chips is used in the PC/AT series compatible machines to manage a total of 15 interrupt vectors, as shown in Figure 2-20. It is connected from the INT pin of slave chip to the IR2 pin of the master chip. The port base address of the master 8259A is 0x20, and the slave chip is 0xA0. The logic block diagram of an 8259A chip is shown in Figure 6-7.

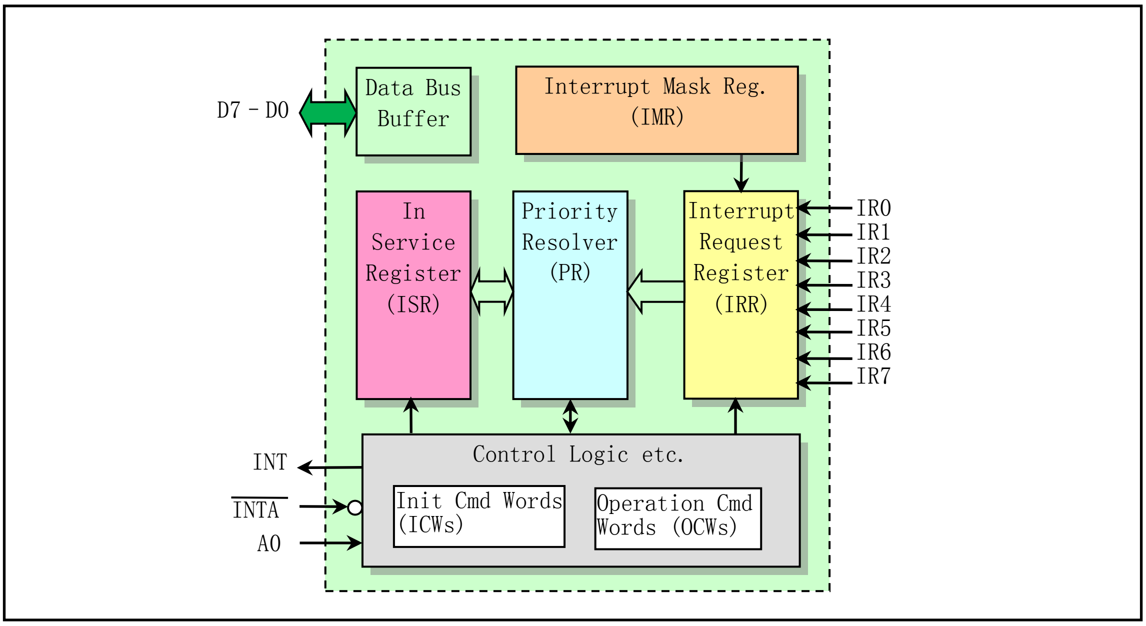


Figure 6-7 Programmable Interrupt Controller 8259A Chip Diagram

In the figure, the Interrupt Request Register (IRR) is used to store all the requested service interrupt levels on the interrupt request input pin. The 8 bits (D7-D0) of the register correspond to the pins IR7-IR0. The Interrupt Mask Register (IMR) is used to store the bits corresponding to the masked interrupt request line. The 8 bits of the register also correspond to 8 interrupt levels. Which bit is set to 1 masks which level of interrupt request. That is, the IMR processes the IRR, each bit of which corresponds to each request bit of the IRR. Masking high priority input lines does not affect the input of low priority interrupt request lines. The priority resolver (PR) is used to determine the priority of the bits set in the IRR, and the highest priority interrupt request is strobed into the in-service register (ISR). The ISR holds an interrupt request that is receiving service. The register set in the control logic block is used to accept two types of commands generated by the CPU. Before the 8259A can operate normally, the contents of the Initialization Command Word (ICW) registers must be set first. In the course of its work, you can use the Operation Command Words (OCW) registers to set and manage the 8259A's working mode at any time. The A0 line is used to select the register for the operation. In the PC/AT microcomputer system, when the A0 line is 0, the port address of the chip is 0x20 (master chip) and 0xA0 (slave chip), and when A0=1, the port is 0x21 and 0xA1.

The interrupt request lines from each device are connected to the IR0-IR7 interrupt request pin of the 8259A. When one or more interrupt request signals arrive on these pins, the corresponding bit in the interrupt request register IRR is set and latched. At this time, if the corresponding bit in the interrupt mask register IMR is set, the corresponding interrupt request will not be sent to the priority parser. After an unmasked interrupt request is sent to the priority resolver, the highest priority interrupt request is selected. At this point, the 8259A will send an INT signal to the CPU, and the CPU will return an INTA to the 8259A to respond to the interrupt signal after executing the current instruction. After receiving the response signal, the 8259A saves the selected highest priority interrupt request to the service register ISR, that is, the bit corresponding to the interrupt request level in the ISR is set. At the same time, the corresponding bit in the interrupt request register IRR is reset, indicating that the interrupt request is starting to be processed.

After that, the CPU will send a second INTA pulse signal to the 8259A, which is used to inform the 8259A to send the interrupt number. Therefore, during the pulse signal, 8259A will send an 8-bit data representing the interrupt number to the data bus for reading by the CPU.

At this point, the CPU interrupt period ends. If the 8259A is using the Automatic End of Interrupt (AEOI) mode, the current service interrupt bit in the service register ISR at the end of the second INTA pulse is reset. Otherwise, if the 8259A is in the non-automatic end mode, then at the end of the interrupt service routine the program will need to send an End of Interrupt (EOI) command to the 8259A to reset the bits in the ISR. If the interrupt request comes from the second 8259A chip that is connected, then the EOI command needs to be sent to both chips. After that, 8259A will check the next highest priority interrupt and repeat the above process. Below we first give the programming method of the initialization command word and the operation command word, and then further explain some of the operation methods used therein.

2. Initialization command word programming

The programmable controller 8259A mainly has four working modes: (1) full nested mode; (2) rotating priority mode; (3) special mask mode and (4) program polled mode. By programming the 8259A, we can choose the current working mode of the 8259A. Programming is divided into two phases. The first is to write the programming of the 4 initialization command words (ICW1 - ICW4) register of each 8259A chip before the 8259A works; the second is to program the 8 operation command words (OCW1 - OCW3) of the 8259A at any time during the work. After initialization, the contents of the operation command word can be written to the 8259A at any time. Below we explain the programming operation of the 8259A initialization command word.

The programming operation flow of the initialization command word is shown in Figure 6-8. As can be seen from the figure, the settings for ICW1 and ICW2 are required. ICW3 needs to be set only when the system includes multiple 8259A chips and is connected. This needs to be clearly stated in the settings of ICW1. In addition, whether you need to set ICW4 also needs to be specified in ICW1.

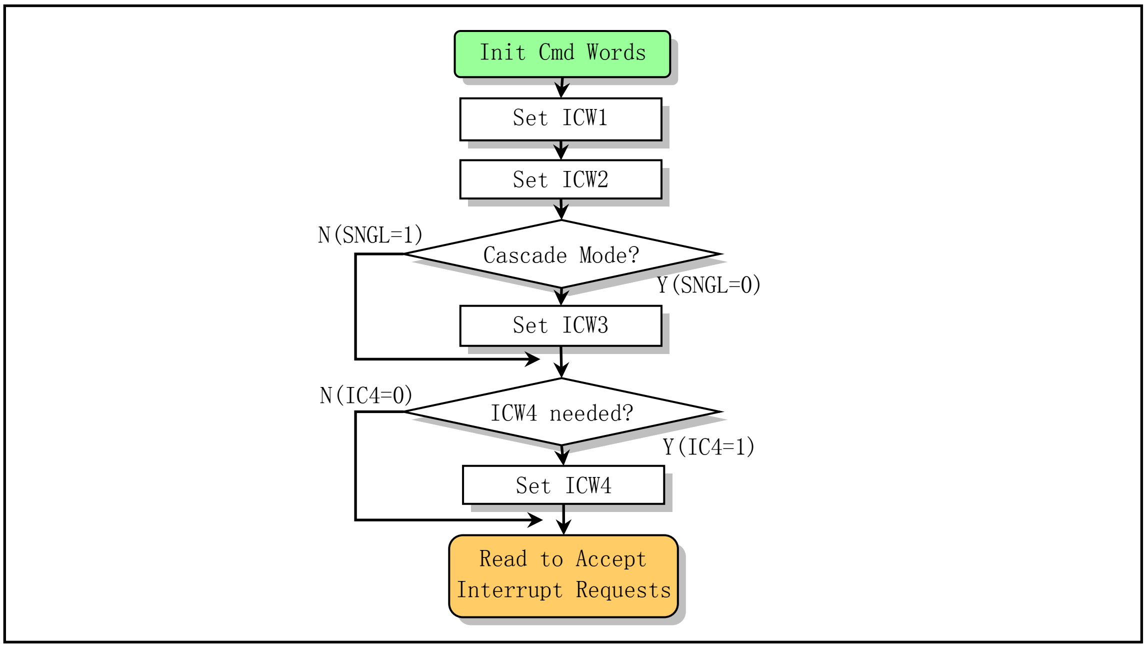


Figure 6-8 8259A Initialization Sequence

(1) ICW1 When the transmitted byte 5th bit (D4) = 1 and the address line A0 = 0, it indicates that ICW1 is programmed. At this time, for the multi-chip cascading case of the PC/AT microcomputer system, the port address of the 8259A main chip is 0x20, and the port address of the slave chip is 0xA0. The format of ICW1 is shown in Table 6–5.

Table 6-5 Interrupt initialization command word ICW1

Bit	Name	Description
D7	A7	A7-A5 indicates the page start address used in the MCS80/85 for the interrupt service process. They are combined with A15-A8 in ICW2. These are not used in 8086/88.
D6	A6	
D5	A5	
D4	1	Always 1
D3	LTIM	1 - Level triggered interrupt mode; 0 – Edge triggered mode.
D2	ADI	The MCS80/85 used for the CALL instruction address interval. Not used in 8086/88.
D1	SNGL	1 - Single 8259A; 0 - Cascade mode.
D0	IC4	1 – requires ICW4; 0 – not required.

In the Linux 0.12 kernel, ICW1 is set to 0x11. It indicates that the interrupt request is edge triggered, multiple slices of 8259A are cascaded, and finally ICW4 needs to be sent.

(2) ICW2 This initialization command word is used to set the upper 5 bits of the interrupt number sent by the chip. After the ICW1 is set, the interrupt number indicates that ICW2 is set when A0=1. At this time, for the multi-chip cascading of the PC/AT microcomputer system, the port address of the 8259A main chip is 0x21, and the port address of the slave chip is 0xA1. The ICW2 format is shown in Table 6–6.

Table 6-6 Interrupt initialization command word ICW2

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	A15/T7	A14/T6	A13/T5	A12/T4	A11/T3	A10	A9	A8

In the MCS80/85 system, the A15-A8 indicated by bits D7-D0 and the A7-A5 set by ICW1 form the interrupt service program page address. In a system or compatible system using the 8086/88 processor, T7-T3 is the upper 5 bits of the interrupt number, and the lower 3 bits automatically set by the 8259A chip form an 8-bit interrupt number. When the 8259A receives the second interrupt response pulse INTA, it will be sent to the data bus for the CPU to read.

The Linux 0.12 system sets the ICW2 of the main slice to 0x20, indicating that the main chip interrupt request is 0 level - the corresponding interrupt number range of the 7th level is 0x20-0x27. The ICW2 of the slave slice is set to 0x28, indicating that the interrupt number range corresponding to the 8-level to 15-level slave interrupt request is 0x28-0x2f.

(3) ICW3 This command word is used to load an 8-bit slave register when multiple 8259A chips are cascaded. The port address is the same as above. The ICW3 format is shown in Table 6–7.

Table 6-7 Interrupt initialization command word ICW3

	A0	D7	D6	D5	D4	D3	D2	D1	D0
Master	1	S7	S6	S5	S4	S3	S2	S1	S0

	A0	D7	D6	D5	D4	D3	D2	D1	D0
Slave	1	0	0	0	0	0	ID2	ID1	ID0

The master chip bits S7_S0 correspond to the cascaded slaves. Which bit is 1 means that the signal on the interrupt request pin IR of the master is from the slave, otherwise the corresponding IR pin is not connected to the slave. The slave chip bits of ID2_ID0 correspond to the identification numbers of the slave chips, that is, the interrupt level connected to the master chip. When a slave receives a cascading line (CAS2 - CAS0) input value equal to its own ID2 - ID0, it means that the slave is selected. At this point, the slave should send the interrupt number of the interrupt request currently selected from the slave chip to the data bus.

The Linux 0.12 kernel sets the ICW3 of the 8259A main chip to 0x04, that is, S2=1, and the remaining bits are 0. Indicates that the IR2 pin of the master chip is connected to a slave chip. The ICW3 of the slave chip is set to 0x02, that is, its identification number is 2. Represents the IR2 pin from the slave chip connected to the main chip. Therefore, the order of the interrupt priorities is the highest at level 0, followed by the level 8-15 on the chip, and finally the level 3-7.

(4) ICW4 When IC0 bit 0 (IC4) is set, it indicates that ICW4 is required. Address line A0=1. The port address is the same as above. The ICW4 format is shown in Table 6–8.

Table 6-8 Interrupt initialization command word ICW4

Bit(s)	Name	Description
D7-5		Always 0
D4	SFNM	1 - special fully nested mode; 0 - not a special fully nested mode.
D3	BUF	1 - buffer mode; 0 - unbuffered mode.
D2	M/S	1 - Buffered mode /Slave; 0 - Buffered mode /Master.
D1	AEOI	1 - Auto End of Interrupt mode; 0 - Normal End of Interrupt mode.
D0	μPM	1 – 8086/88 processor system; 0 – MCS80/85 system.

The value of the ICW4 command word sent to the 8259A master chip and the slave chip by the Linux 0.12 core is 0x01. Indicates that the 8259A chip is set to a normal fully nested, unbuffered, non-automatic end interrupt mode and is used in the 8086 and its compatible systems.

3. Operation command word programming

After setting the initialization command word register to the 8259A, the chip is ready to receive interrupt request signal from the device. However, during the 8259A operation, we can also use the operation command word OCW1-OCW3 to monitor the working status of the 8259A, or change the working mode of the 8259A set at the time of initialization.

(1) OCW1 This operation command word is used to read/write the interrupt mask register IMR. Address line A0 needs to be 1. The port address description is the same as above. The OCW1 format is shown in Table 6–9.

Table 6-9 Interrupt operation command word OCW1

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	M7	M6	M5	M4	M3	M2	M1	M0

Bits D7-D0 correspond to 8 interrupt requests, 7 levels - 0 level mask bits M7 - M0. If M=1, the corresponding interrupt request level is masked; if M=0, the corresponding interrupt request level is allowed. In addition, masking high priority does not affect other low priority interrupt requests.

During the Linux 0.12 kernel initialization process, the code uses the operation command word to modify the relevant interrupt request mask bit after setting the relevant device driver. For example, at the end of the floppy disk driver initialization, in order to allow the floppy device to issue an interrupt request, port 0x21 is read to obtain the current mask byte of the 8259A chip. Then, with AND ~0x40 operation, the mask bit of the interrupt request 6 connected to the corresponding floppy disk controller is reset. Finally write back to the interrupt mask register. See line 461 of the kernel/blk_drv/floppy.c program.

(2) OCW2 is used to send EOI commands or set the automatic rotate mode for interrupt priority. When the bit D4D3 = 00, the address line A0 = 0 indicates that the OCW2 is programmed. The format of the operation command word OCW2 is shown in Table 6–10.

Table 6-10 Interrupt operation command word OCW2

Bit(s)	Name	Description
D7	R	Priority rotation state.
D6	SL	Priority setting flag.
D5	EOI	Non-Automatic End of Interrupt flag.
D4-3		Always 0
D2	L2	L2 -- L0 - 3 bits form the level number, corresponding to the interrupt request level IRQ0--IRQ7 (or IRQ8-IRQ15).
D1	L1	
D0	L0	

The roles and meanings of the combination of bits D7-D5 are shown in Table 6–11. Those with an * can specify the priority to reset the ISR by setting L2--L0, or select the special rotate priority to become the current lowest priority.

Table 6-11 OCW2 bit D7--D5 combination meaning

R(D7)	SL(D6)	EOI(D5)	Description	Type
0	0	1	Non-specific EOI command (fully nested mode).	End of Interrupt
0	1	1	Specific EOI command (not fully nested).	
1	0	1	Rotate on non-specific EOI command.	Automatic Rotation
1	0	0	Rotate in Automatic EOI mode (Set).	
0	0	0	Rotate in Automatic EOI mode (Clean).	
1	1	1	Rotate on Specific EOI command.	Specific rotation
1	1	0	Set priority command.	
0	1	0	No operation.	

The Linux 0.12 kernel uses only the operational command word to send an end interrupt EOI command to the 8259A before the end of the interrupt processing. The OCW2 value used is 0x20, indicating a non-special end interrupt EOI command in full nested mode.

(3) OCW3 is used to set the special mask mode and read register status (IRR and ISR). When D4D3=01 and address line A0=0, it means that OCW3 is programmed (read/write). However, this operation command word is not used in the Linux 0.12 kernel. The format of OCW3 is shown in Table 6–12.

Table 6-12 Interrupt operation command word OCW3

Bit	Name	Description
D7		Always 0
D6	ESMM	Operate in a special mask mode: D6 -- D5: 11 - Set special mask; 10 - Reset special mask; 00,01 - No action.
D5	SMM	
D4		Always 0
D3		Always 1
D2	P	1 - Poll command; 0 - No poll command.
D1	RR	Read register status command on the next RD pulse: D1 -- D0: 11 - Read In Service Reg. ISR; 10 - Read Interrupt Reg. IRR
D0	RIS	

4. 8259A operation mode description

In the programming process of the 8259A initialization command word and the operation command word, some working methods are mentioned. The following is a detailed description of several common ways to better understand how the 8259A chip operates.

(1) Full nested mode

After initialization, the operation automatically enters this fully nested mode unless the operation command word has been used to change the way the 8259A works. In this mode, the order of interrupt request priority is from level 0 to level 7 (level 0 has the highest). When the CPU responds to an interrupt, the highest priority interrupt request is determined and the interrupt request's interrupt number is placed on the data bus. In addition, the corresponding bit in the interrupt service register ISR will be set and the set state of the bit will remain until the end of the interrupt EOI command is sent before returning from the interrupt service procedure. If the automatic interrupt end AEOI bit is set in the ICW4, the bit in the ISR will be reset at the end edge of the second interrupt response pulse INTA issued by the CPU. During the ISR with a set bit, all interrupt requests of the same priority and low priority will be temporarily disabled, but higher priority interrupt requests are allowed to respond and be processed. Furthermore, the corresponding bits of the interrupt mask register IMR can mask 8-level interrupt requests, respectively, but masking any one of the interrupt requests does not affect the operation of other interrupt requests. Finally, after initializing the command word programming, the 8259A pin IR0 has the highest priority, while IR7 has the lowest priority. The Linux 0.12 kernel code works with the machine's 8259A chip set in this mode.

(2) End of Interrupt (EOI) method

As described above, the bit corresponding to the interrupt request being processed in the service register ISR can be reset in two ways. One is that when the automatic interrupt end bit AEOI in ICW4 is set, it is reset by the end edge of the second interrupt response pulse INTA issued by the CPU. This method is called the Automatic End of Interrupt (AEOI) method. The second is to send an end interrupt EOI command to reset interrupt before returning from the interrupt service process. This method is called the End of Program Interrupt (EOI) method. In cascading system, the slave interrupt service routine needs to send two EOI commands, one for the slave chip and one for the master chip.

There are two ways in which a program can issue an EOI command. One is called a special EOI command, and the other is called a non-special EOI command. The special EOI command is used in non-fully nested mode and can be used to specify the interrupt level bits for the specific reset of the EOI command. That is, when sending a special EOI command to the chip, it is necessary to specify the priority in the reset ISR. The special EOI command is sent using OCW2, the upper 3 bits are 011, and the lowest 3 bits are used to specify the priority. This special EOI command is used in current Linux systems. The non-special EOI command for the fully nested mode automatically resets the highest priority bit currently in the service register ISR. Because in the fully nested mode, the highest priority bit in the ISR is definitely the priority of the last response and service. It is also sent using OCW2, but the highest 3 bits need to be 001. This non-special EOI command is used in the Linux 0.12 system discussed in this book.

(3) Special full nested mode

The special full nesting mode (D4=1) set in ICW4 is mainly used in large cascading systems, and the priority in each slave chip needs to be saved. This approach is similar to the normal full nesting approach described above, with the following two exceptions:

A. When an interrupt request from a slave chip is being serviced, the slave chip is not excluded by the priority of the master chip. Therefore, other higher priority interrupt requests issued from the chip will be recognized by the master chip, and the master chip will immediately issue an interrupt to the CPU. In the above conventional full nesting mode, when a slave interrupt request is being serviced, the slave chip is masked by the master chip. Therefore, a higher priority interrupt request issued from the slave chip cannot be processed.

B. When exiting the interrupt service routine, the program must check if the current interrupt service is the only interrupt request issued from the slave chip. The method of checking is to first issue a non-special interrupt EOI command to the slave chip and then read the value of its service register ISR. Check if the value is 0 at this time. If it is 0, it means that a non-special EOI command can be sent to the main chip. If it is not 0, there is no need to send an EOI command to the main chip.

(4) Cascade mode method

The 8259A can be easily connected into a master and a number of slave chips. If you use 8 slave chips, you can control up to 64 interrupt priorities. The master chip controls the slaves through three cascaded lines. These three cascaded lines are equivalent to the chip selection signal from the chip. In the cascade mode, the interrupt output of the slave chip is connected to the interrupt request input pin of the master chip. When an interrupt request line from the chip is processed and responded, the master chip selects the slave chip to place the corresponding interrupt number on the data bus.

In a cascading system, each 8259A chip must be initialized independently and can operate in different ways. In addition, the initialization command word ICW3 of the master chip and the slave chip are separately programmed. It is also necessary to send 2 interrupt end EOI commands during operation, one for the main chip and the other for the slave chip.

(5) Automatic rotation priority mode

When we are managing devices with the same priority, we can use OCW2 to set the 8259A chip to automatic rotation priority mode. That is, after a device receives a service, its priority automatically becomes the lowest. The priorities are cyclically changed in sequence. The most unfavorable situation is that when an interrupt request comes in, it needs to wait for 7 devices before it can get the service.

(6) Interrupt mask mode

The interrupt mask register, IMR, controls the masking of each interrupt request. The 8259A can be set to two mask methods. For general normal masking, use OCW1 to set the IMR. The IMR bits (D7--D0) are applied to the respective interrupt request pins IR7-IR0. Masking an interrupt request does not affect other priority interrupt requests. This normal masking mode causes the 8259A to mask all low priority interrupt requests for an interrupt request during the response and service (before the EOI command is sent). However, in some applications it may be necessary for interrupt service process to dynamically change the system's priority. In order to solve this problem, the special masking method was introduced in the 8259A. We need to first set this mode (D6, D5 bits) using OCW3. In this special masking mode, the masking information set by OCW1 causes all unmasked priority interrupts to be responded to during an interrupt.

(7) Read register status

There are three registers (IMR, IRR, and ISR) in the 8259A that allow the CPU to read its status. The current mask information in the IMR can be obtained by directly reading OCW1. Before reading the IRR or ISR, you need first use OCW3 to output reading command of IRR or ISR before you can read them.

6.4 head.s

6.4.1 Function description

The head.s program is linked to the system module along with the target files of other programs in the kernel after being compiled to an object file, and is located at the beginning of the system module. This is why it is called a head program. The system module will be placed in the sector starting after the setup module on the disk, starting from the sixth sector on the disk. Under normal circumstances, the system module of the Linux 0.12 kernel is about 120 KB in size, so it accounts for about 240 sectors on the disk.

From here on, the kernel is completely running in protected mode. The heads.s assembly file is different from the previous assembly syntax. It uses AT&T's assembly language format and requires GNU's gas and gld to compile and link. So please note that the direction of the assignment in the code is from left to right.

This program is actually at the beginning of the absolute address 0 of the memory. 这个程序的功能比较单一。First, it loads each data segment register, re-establishes the interrupt descriptor table IDT, a total of 256 items (boot/head.s, 78), and makes each entry point to a dummy interrupt subroutine ignore_int that only reports errors. This dummy interrupt vector points to a default "ignore interrupt" process (boot/head.s, 150). The message "Unknown interrupt" is displayed when an interrupt occurs and the interrupt vector has not been setup. All 256 items are set here to prevent the occurrence of a general protection fault (Exception 13). Otherwise, if the IDT is set to less than 256 entries, the CPU will generate a general protection fault (Exception 13) when the descriptor entry specified by a required interrupt is greater than the maximum descriptor entry set. In addition, if there is a problem with the hardware and the device vector is not placed on the data bus, the CPU will usually read all 1 (0xff) as a vector from the data bus, so it will read the 256th item in the IDT table. Therefore, it also causes general protection errors if the vector is not set.

For some interrupts that need to be used in the system, the kernel will setup the interrupt descriptor items for these interrupts during the process of their initialization (init/main.c), and point them to the corresponding actual interrupt hander procedures. Usually, the exception interrupt handler (int0 -- int 31) is re-installed in the initialization function of traps.c (kernl/traps.c, line 185), and the system call interrupt int 0x80 is installed in the scheduler initialization function (kernel/sched.c, line 417).

Each descriptor item in the interrupt descriptor table IDT also occupies 8 bytes, and its format is shown in Figure 6-9. Where P is the segment presence flag; DPL is the priority of the descriptor.

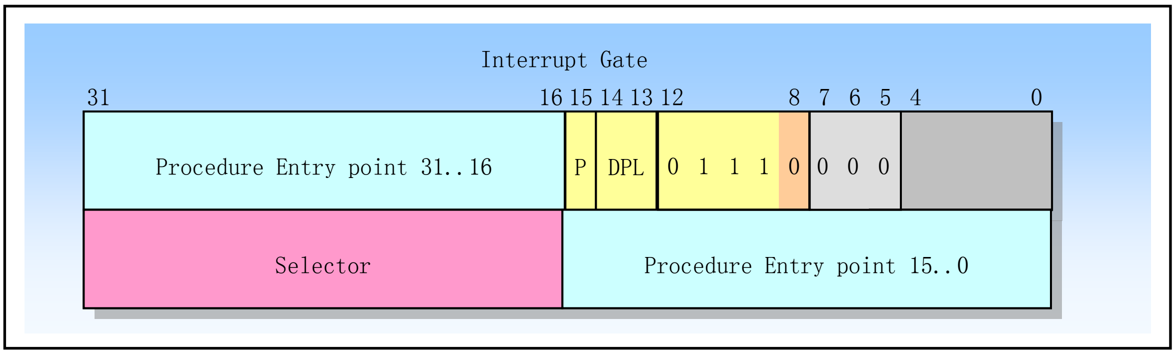


Figure 6-9 Interrupt Gate Descriptor Format in IDT

In the head.s program, the segment selector field in the interrupt gate descriptor is set to 0x0008, which indicates that the dummy interrupt service routine ignore_int is in the kernel code. and the offset is set to the offset of ignore_int interrupt service handler in the head.s program. Since the head.s program is moved to the beginning of memory address 0, the offset of the interrupt service handler is also the offset in the kernel code. Since the kernel code segment is always in memory and the privilege level is 0 (ie, P=1, DPL=00), it can be seen from the figure that the value of byte 5 and byte 4 of the interrupt gate descriptor should be 0x8E00.

After setting the interrupt descriptor table, the program rebuilds the global segment descriptor table GDT. In fact, descriptors in the newly created GDT table is not much different from that in the original GDT table. Except for some differences in the limit (originally 8MB, now 16MB), the other content is exactly the same. So we can also set the segment limit of the descriptor directly to 16MB in setup.s, and then directly move the original GDT table to the appropriate location in the memory. So the main reason for re-create a new GDT here is to put the GDT table in a reansonable place in the kernel space. The previously set GDT table is at location 0x902XX in memory. This place will be used as part of the memory cache after the kernel is initialized.

Then it is detected whether the A20 address line is turned on. The method is to compare the contents starting at the memory address 0 with the contents beginning from address 1 MB. If the A20 line is not enabled, the CPU will cyclically access the contents at address (address MOD 1MB) when accessing more than 1MB of physical memory, that is, the same as accessing the corresponding byte starting from address 0. If the program detects that it is not open, it enters an infinite loop. Otherwise the program will continue to test whether the PC contains a math coprocessor chip (80287, 80387 or its compatible chip) and set the corresponding flag in the control register CR0.

Then head.s code sets the paging mechanism for managing memory, placing the page directory table at the beginning of the physical address 0 (also the memory area this program is located, so the code area that has finished execution will be overwritten). Immediately afterwards, four page tables with a total addressable 16 MB of memory are placed, and their entries are set separately. The page directory entry and page table entry format are shown in Figure 6-10. Where P is page exists flag; R/W is read/write flag; U/S is user/super user flag; A is page visited flag; D is page content modified flag; and the leftmost 20 bits are the upper 20 bits of the page address in the memory corresponding to the page entry.

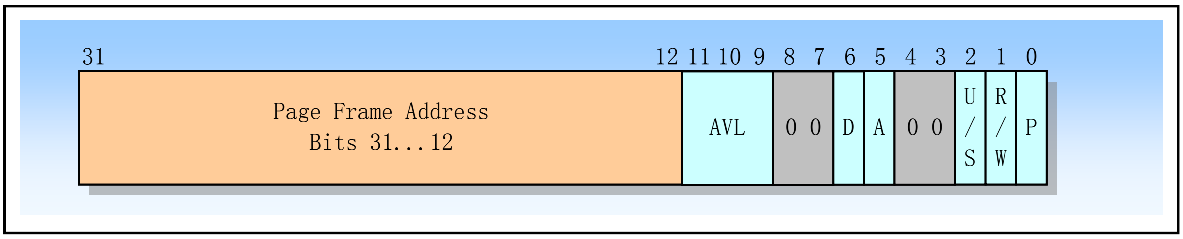


Figure 6-10 Entry structure of page directory and page table

Here, the attribute flag of each entry is set to 0x07 (P=1, U/S=1, R/W=1), indicating that the page exists and the user can read and write. The reason for setting the kernel page table attribute in this way is that both the segmentation and the paging management have protection methods. The protection flags (U/S, R/W) set in the page directory and page table entries need to be combined with the privilege level (PL) protection in the segment descriptor. But the PL in the segment descriptor plays a major role. The CPU will first check the segment protection and then check the page protection. If the current privilege level CPL < 3 (for example, 0), the CPU is running as a supervisor. At this point all pages can be accessed, and free to read and write. If CPL = 3, the CPU is running as user (user). At this point only the pages belonging to user (U/S=1) are accessible, and only pages marked as readable and writable (W/R = 1) are writable. At this time, the page belonging to the super user (U/S=0) can neither be written nor read. Because the kernel code is a bit special, it contains the code and data for task 0 and task 1. So setting the page property to 0x7 here will ensure that the two tasks can be executed in user mode, but they cannot access kernel resources arbitrarily.

Finally, the head.s program uses the return instruction to pop out the entry address of the /init/main.c program pre-placed on the stack, and transfer the execution rights to the main() code.

6.4.2 Code Comments

Program 6-3 linux/boot/head.s

```
1 /*
2  *  linux/boot/head.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 /*
8  *  head.s contains the 32-bit startup code.
9  *
10  * NOTE!!! Startup happens at absolute address 0x00000000, which is also where
11  * the page directory will exist. The startup code will be overwritten by
12  * the page directory.
13  */
14 .text
15 .globl _idt,_gdt,_pg_dir,_tmp_floppy_area
16 _pg_dir:                   # The page directory will be stored here.

   # Note again!! This is already in 32-bit mode, so 0x10 is now a selector for a descriptor
   # and the instruction loads the contents of the corresponding descriptor into the segment
   # register. Here, the meaning of 0x10 is: request privilege level RPL is 0 (bit 0-1 = 0),
   # select global descriptor table GDT (bit 2 = 0), and select the second item in the table
   # (bits 3-15 = 2). It just points to the data segment descriptor item in the table (see
   # the 575--578 lines in setup.s for the specific value of the descriptor).
   #
   # The following code means: set ds, es, fs, gs to the kernel data segment constructed in
   # setup.s with selector=0x10 (corresponding to item 3 of the global descriptor table),
   # and place the stack in the user_stack array area pointed to by stack_start. Then use
   # the new interrupt descriptor table (line 232) and the global segment description table
   # (lines 234--238) defined later in this program. The initial content in the new GDT table
   # is basically the same as in setup.s, and only the segment length is changed from 8MB to 16MB.
   # Stack_start is defined in kernel/sched.c, lines 82-87. It is a long pointer to the end
   # of the user_stack array. The code on line 23 sets the stack used here, which we now call
   # the system stack. But after moving to task 0 execution (137 lines in init/main.c), the
   # stack is used as the user stack for task 0 and task 1.

17 startup_32:                  # set each data segment registers.
18         movl $0x10,%eax      # direct operand starts with '$', otherwise it's an address.
19         mov %ax,%ds
20         mov %ax,%es
21         mov %ax,%fs
22         mov %ax,%gs
23         lss _stack_start,%esp      # _stack_start -> ss:esp, set system stack.
24         call setup_idt             # line 67--93
25         call setup_gdt             # line 95--107
26         movl $0x10,%eax            # reload all the segment registers
27         mov %ax,%ds                # after changing gdt. CS was already
28         mov %ax,%es                # reloaded in 'setup_gdt'
29         mov %ax,%fs                # GDT changed, all segs need to be reloaded.
30         mov %ax,%gs

   # Since the segment length in the descriptor has been changed from 8MB to 16MB (see
   # setup.s line 567-578 and lines 235-236 later in this program), therefore, the load
   # operation must be performed on all segment registers again. In addition, by using the
   # bochs simulation software to track the code, if the CS is not loaded again, the limit
   # length in the invisible portion of CS is still 8 MB when executing to line 26. It seems
   # that the CS should be reloaded here. However, since the code segment descriptor only
   # changes the segment length, the rest is exactly the same, so the 8MB limit length does
   # not cause problems during the kernel initialization phase. In addition, the inter-segment
   # jump instruction will reload CS during the kernel execution process, so not loading it
   # here will not cause future kernel errors.
   # In response to this problem, the current kernel has added a long jump instruction after
   # the 25th line: 'ljmp $(__KERNEL_CS), $1f', jump to line 26 to ensure that the CS is
   # indeed reloaded.

31         lss _stack_start,%esp

   # Lines 32-36 are used to test if the A20 address line is enabled. The method used is to
   # write any value to the memory address beginning at 0x000000, and then see if the value
   # at the cooresponding address 0x100000 (1M) contains the same value. If they are always
   # the same, they will continue to compare, that is, infinite loops and crashes. This means
   # that the address A20 line is not strobed, and the kernel cannot use more than 1MB of
   # memory.
   #
   # '1:' on line 33 is a label consisting of a local symbol. At this point the symbol
   # represents the current value of the Active Location Counter and can be used as the
   # operand of the instruction. Local symbols are used to help compilers and programmers
   # temporarily use some names. There are a total of 10 local labels that can be reused
   # throughout the program. These labels are referenced using the names '0', '1', ..., '9'.
   # In order to define a local symbol, the label is written in the form 'N:' (where N
   # represents a number). In order to reference this previously defined label, it needs to
   # be written as 'Nb'. In order to reference the next definition of a local label, it needs
   # to be written as 'Nf'. Above 'b' means "backwards" and 'f' means "forwards". At some
   # point in the assembly file, we can refer up to 10 labels backwards/forward.

32          xorl %eax,%eax
33 1:       incl %eax                   # check that A20 really IS enabled
34          movl %eax,0x000000          # loop forever if it isn't
35          cmpl %eax,0x100000
36          je 1b                       # '1b' means backward label 1.
                                        # '5f' means forward 5.
37 /*
38  * NOTE! 486 should set bit 16, to check for write-protect in supervisor
39  * mode. Then it would be unnecessary with the "verify_area()"-calls.
40  * 486 users probably want to set the NE (#5) bit also, so as to use
41  * int 16 for math errors.
42  */
   # The bit 16 of the CR0 control register in the 486 CPU mentioned in the previous comment
   # is the write-protection flag (WP), which is used to prohibit the super-user-level
   # program from writing to the general user read-only page. This flag is mainly used by
   # the operating system to implement a copy-on-write method when creating a new process.
   #
   # The following code (lines 43-65) is used to check if the math coprocessor chip is
   # present. The method is to modify the control register CR0 and execute a coprocessor
   # instruction assuming a coprocessor is present. If something goes wrong, the coprocessor
   # chip does not exist. The coprocessor emulation bit EM (bit 2) in CR0 needs to be set and
   # the coprocessor presence flag MP (bit 1) is reset.

43          movl %cr0,%eax              # check math chip
44          andl $0x80000011,%eax       # Save PG,PE,ET
45 /* "orl $0x10020,%eax" here for 486 might be good */
46          orl $2,%eax                 # set MP
47          movl %eax,%cr0
48          call check_x87
49          jmp after_page_tables       # line 135
50
51 /*
52  * We depend on ET to be correct. This checks for 287/387.
53  */

   # The following fninit and fstsw are instructions for the math coprocessor (80287/80387).
   # finit issues an initialization command to the coprocessor, which places the coprocessor
   # in a known state that is not affected by previous operations, sets its control word to
   # the default value, clears the status word and all floating point stack registers. This
   # non-waiting form of fninit also causes the coprocessor to terminate execution of any
   # previous arithmetic operations currently in progress. The fstsw instruction gets the
   # status word of the coprocessor. If there is a coprocessor in the system, then the status
   # low byte must be 0 after the fninit instruction is executed.

54 check_x87:
55          fninit
56          fstsw %ax                   # get status word -> ax
57          cmpb $0,%al                 # status word should be 0 after fninit if has a math.
58          je 1f                       /* no coprocessor: have to set bits */
59          movl %cr0,%eax
60          xorl $6,%eax                /* reset MP, set EM */
```

```
        movl %eax,%cr0
        ret
```

```
# .align is an assembly indicator. Its meaning refers to the storage boundary alignment
# adjustment. Here, '2' indicates that the offset position of the subsequent code or data
# is adjusted to the position where the last 2 bits of the address value are zero (2^2),
# that is, the memory address is aligned in a 4-byte manner. (But now GNU as is writing
# the aligned value directly instead of the power of 2). The purpose of using this
# directive to achieve memory alignment is to increase the speed and efficiency of 32-bit
# CPU access to code or data in memory.
# The following two byte are the machine code of 80287 instruction fsetpm. Its role is to
# set 80287 to protection mode. The 80387 does not require this instruction and will treat
# it as a nop.
```

```
.align 2
1:      .byte 0xDB,0xE4            /* fsetpm for 287, ignored by 387 */
        ret

/*
 *  setup_idt
 *
 *  sets up a idt with 256 entries pointing to
 *  ignore_int, interrupt gates. It then loads
 *  idt. Everything that wants to install itself
 *  in the idt-table may do so themselves. Interrupts
 *  are enabled elsewhere, when we can be relatively
 *  sure everything is ok. This routine will be over-
 *  written by the page tables.
 */
```

```
# Each item in the interrupt descriptor table (IDT) is composed of 8 bytes, but its format
# is different from that in the GDT table, and is called a gate descriptor. Its 0-1, 6-7
# bytes are offsets, 2-3 bytes are selectors, and 4-5 bytes are some flags.
# This code first sets the default interrupt descriptor value of 8 bytes in EDX and EAX,
# and then places the descriptor in each item of the idt table, a total of 256 items. EAX
# contains a descriptor lower 4 bytes and EDX contains a height of 4 bytes. During the
# subsequent initialization process, the kernel replaces the current default settings with
# those really useful interrupt descriptor entries.
```

```
setup_idt:
        lea ignore_int,%edx        # effective addr of ignore_int -> edx
        movl $0x00080000,%eax      # store selector 0x0008 to high word of eax
        movw %dx,%ax               /* selector = 0x0008 = cs */
                                   # store offset low 16 bits to low word of eax.
        movw $0x8E00,%dx           /* interrupt gate - dpl=0, present */
                                   # edx contains high 4 bytes of gate descriptor.
        lea _idt,%edi              # _idt is IDT table address (offset).
        mov $256,%ecx
rp_sidt:
        movl %eax,(%edi)           # store the dummy descriptor into IDT table
        movl %edx,4(%edi)          # store eax to [edi+4]
        addl $8,%edi               # edi point to the next item by plus 8.
        dec %ecx
        jne rp_sidt
```

```
 92         lidt idt_descr              # Load the interrupt descriptor table register.
 93         ret
 94
 95 /*
 96  *  setup_gdt
 97  *
 98  *  This routines sets up a new gdt and loads it.
 99  *  Only two entries are currently built, the same
100  *  ones that were built in init.s. The routine
101  *  is VERY complicated at two whole lines, so this
102  *  rather long comment is certainly needed :-).
103  *  This routine will beoverwritten by the page tables.
104  */
105 setup_gdt:
106         lgdt gdt_descr              # contents on lines 234-238.
107         ret
108
109 /*
110  * I put the kernel page tables right after the page directory,
111  * using 4 of them to span 16 Mb of physical memory. People with
112  * more than 16MB will have to expand this.
113  */

    # Each page table is 4KB in size (1 page memory), and each page table entry requires 4
    # bytes, so a page table can store a total of 1024 entries. If a page table entry
    # addresses a 4KB address space, a page table can address 4 MB of physical memory. The
    # format of the page table entry is: the first 0-11 bits of the item store some flags,
    # such as whether it is in memory (P bit 0), read and write permission (R/W bit 1), normal
    # user or super user (U/S, Bit 2), whether it has been modified or dirty (D bit 6), etc.;
    # bits 12-31 are page frame address, which are used to indicate the physical start address
    # of a page.

114 .org 0x1000      # here begins the first page table. page directory stored at 0.
115 pg0:
116
117 .org 0x2000
118 pg1:
119
120 .org 0x3000
121 pg2:
122
123 .org 0x4000
124 pg3:
125
126 .org 0x5000        # the code or data below starts from offset 0x5000.
127 /*
128  * tmp_floppy_area is used by the floppy-driver when DMA cannot
129  * reach to a buffer-block. It needs to be aligned, so that it isn't
130  * on a 64kB border.
131  */
132 _tmp_floppy_area:
133         .fill 1024,1,0              # 1024 bytes filled with 0.
134
```

```
    # The following push operations are used to prepare for the jump to the main() function in
    # the init/main.c. The instruction on line 139 pushes the return address (label L6) on the stack,
    # while line 140 pushes the address of the main() function code. When head.s finally executes
    # the ret instruction on line 218, it will pop up the address of main() and transfer control
    # to the init/main.c program. See the description of C function call mechanism in Chapter 3.

    # The first three pop-up 0 values represent the main function arguments envp, argv pointer,
    # and argc, respectively, but main() does not use them. The 139-line push operation
    # simulates the return address of the call to main(). So if the main program really exits,
    # it will return to the label L6 to continue, that is, performs the infinite loop. Line 140
    # pushes the address of main() onto the stack, so that when the 'ret' instruction is
    # executed after setting the paging processing (setup_paging), the address of the main()
    # is popped out of the stack and the main() is executed.
135 after_page_tables:
136         pushl $0                    # These are the parameters to main :-)
137         pushl $0
138         pushl $0
139         pushl $L6                   # return address for main, if it decides to.
140         pushl $_main                # '_main' is the internal representation of main().
141         jmp setup_paging            # jump to line 198.
142 L6:
143         jmp L6                      # main should never return here, but
144                                     # just in case, we know what happens.
145
146 /* This is the default interrupt "handler" :-) */
147 int_msg:
148         .asciz "Unknown interrupt\n\r"     # 定义字符串“未知中断(回车换行)”。
149 .align 2                    # alignment with 4 bytes in memory.
150 ignore_int:
151         pushl %eax
152         pushl %ecx
153         pushl %edx
154         push %ds            # ds,es,fs,gs still occupy 2 words each when on stack.
155         push %es
156         push %fs
157         movl $0x10,%eax     # set selector (ds,es,fs points data descriptor in gdt)
158         mov %ax,%ds
159         mov %ax,%es
160         mov %ax,%fs
    # Put printk() function's parameter pointer onto the stack. Note that if '$' is not added
    # before int_msg, it means that the long word ('Unkn') at the int_msg symbol is pushed
    # onto the stack. This function is in /kernel/printk.c. '_printk' is the internal
    # representation in the printk compiled module.
161         pushl $int_msg
162         call _printk
163         popl %eax           # /kernel/printk.c
164         pop %fs
165         pop %es
166         pop %ds
167         popl %edx
168         popl %ecx
169         popl %eax
170         iret                # iret pop out CFLAGS too.
```

```
171
172
173 /*
174  * Setup_paging
175  *
176  * This routine sets up paging by setting the page bit
177  * in cr0. The page tables are set up, identity-mapping
178  * the first 16MB. The pager assumes that no illegal
179  * addresses are produced (ie >4Mb on a 4Mb machine).
180  *
181  * NOTE! Although all physical memory should be identity
182  * mapped by this routine, only the kernel page functions
183  * use the >1Mb addresses directly. All "normal" functions
184  * use just the lower 1Mb, or the local data space, which
185  * will be mapped to some other place - mm keeps track of
186  * that.
187  *
188  * For those with more memory than 16 Mb - tough luck. I've
189  * not got it, why should you :-) The source is here. Change
190  * it. (Seriously - it shouldn't be too difficult. Mostly
191  * change some constants etc. I left it at 16Mb, as my machine
192  * even cannot be extended past that (ok, but it was cheap :-)
193  * I've tried to show which constants to change by having
194  * some kind of marker at them (search for "16Mb"), but I
195  * won't guarantee that's all :-( )
196  */
    # The meaning of the second paragraph of the original comment above means that more than
    # 1MB of memory space in the machine is mainly used for the main memory area. This main
    # memory area is managed by the mm module, which involves page mapping operations. All
    # other functions in the kernel are the general (ordinary) functions referred to here. To
    # use the page in the main memory area, you need to use the get_free_page() function to
    # get it. Because the memory pages in the main memory area are shared resources, there
    # must be a program for unified management to avoid resource contention.
    #
    # The 1-page page directory table and the 4-page page table are stored at the physical
    # address 0x0 of the memory. The page directory table is common to all processes in the
    # system, and the 4-page page table here is kernel-specific. The initial 16MB linear
    # address space is mapped one by one to the physical memory space. For newly created
    # processes, the system will store the page table for its application in the main memory
    # area. In addition.

197 .align 2                          # align memory boundaries in 4-byte.
198 setup_paging:
199         movl $1024*5,%ecx         /* 5 pages - pg_dir+4 page tables */
200         xorl %eax,%eax
201         xorl %edi,%edi            /* pg_dir is at 0x000 */
202         cld;rep;stosl             # eax -> [es:edi], edi increased by 4

    # The following statements sets the items in the page directory. Since the kernel uses a
    # total of 4 page tables, only 4 items need to be set. The structure of the page directory
    # entry is the same as the structure of the entries in the page table, with 4 bytes being
    # one item. See the descriptions under line 113 above.
    # For example, "$pg0+7" means: 0x00001007, which is the first item in the page directory,
    # then the address of the first page table = 0x00001007 & 0xfffff000 = 0x1000; the
    # attribute flag of the first page table = 0x00001007 & 0x00000fff = 0x07, indicating that
    # the page exists and the user can read and write.
203         movl $pg0+7,_pg_dir        /* set present bit/user r/w */
204         movl $pg1+7,_pg_dir+4      /*  --------- " " --------- */
205         movl $pg2+7,_pg_dir+8      /*  --------- " " --------- */
206         movl $pg3+7,_pg_dir+12     /*  --------- " " --------- */

    # The following 6 lines of code fill in the contents of all the items in the four page
    # tables, the total number of items: 4 (page table) * 1024 (item / page table) = 4096
    # items (0 - 0xfff), that is, it can map physical memory 4096 * 4Kb = 16Mb. The content of
    # each item is: the physical memory address mapped by the current item + page flags (all 7).
    # The method used to fill in is to fill in the reverse order from the last item in the
    # last page table. The position of the last item in each page table is 1023*4 = 4092, so
    # the last item on the last page is $pg3 + 4092.

207         movl $pg3+4092,%edi        # edi -> points to the last item in the last page
208         movl $0xfff007,%eax        /*  16Mb - 4096 + 7 (r/w user,p) */
                                       # the last item map to physical mem addr 0xfff000 + 7
209         std                        # set direction flag, edi is decreased (by 4 bytes).
210 1:      stosl                      /* fill pages backwards - more efficient :-) */
211         subl $0x1000,%eax          # each time an item is filled,addr is reduced by 0x1000.
212         jge 1b                     # if less than 0, all filled.
    # Now set the page directory base register cr3, it contains the physical address of the
    # page directory table. Then set to start using paging (the PG flag of cr0, bit 31).
213         xorl %eax,%eax             /* pg_dir is at 0x0000 */
214         movl %eax,%cr3             /* cr3 - page directory start */
215         movl %cr0,%eax
216         orl $0x80000000,%eax       # add PG flag
217         movl %eax,%cr0             /* set paging (PG) bit */
218         ret                        /* this also flushes prefetch-queue */

    # After changing the paging flag, it is required to use the branch instruction to refresh
    # the prefetch instruction queue. The ret instruction is used here. Another function of
    # this return instruction is to pop the address of the main() pushed by instructions on
    # line 140, and jump to the /init/main.c program to run. This program really ends here.

219
220 .align 2                           # align memory boundaries in 4-byte.
221 .word 0                            # skip a word, so that line 224 are 4-byte aligned.

    # The following is the 6-byte operand required by the instruction LIDT to load the
    # interrupt descriptor table register. The first 2 bytes are the limit of the IDT table,
    # and last 4 bytes are the 32-bit base address of the IDT table in linear address space.
222 idt_descr:
223         .word 256*8-1              # idt contains 256 entries
224         .long _idt
225 .align 2
226 .word 0

    # The 6-byte operand required by the LGDT instruction of the global descriptor table
    # register is loaded below. The first 2 bytes are the limit of the GDT, and the last 4
    # bytes are the linear base address of the GDT. Here the global table size is set to 2KB
    # bytes (ie 0x7ff). Since each descriptor item has 8 bytes, there are a total of 256
    # entries in the table. The symbol _gdt is the offset position of the global table in the
    # program, see line 234.
227 gdt_descr:
228         .word 256*8-1                # so does gdt (note that that's any
229         .long _gdt                   # magic number, but it works for me :^)
230
231         .align 3                     # align memory boundaries by 8 (2^3) bytes.
232 _idt:   .fill 256,8,0                # idt is uninitialized
233
    # Global descriptor table GDT. The first four items are: empty item (not used), kernel
    # code segment descriptor, kernel data segment descriptor, system call segment descriptor.
    # The system call segment descriptor is not used. Linus might have wanted to put the
    # system call code in this separate segment. A space of 252 items is reserved later for
    # placing the local descriptor table (LDT) of the newly created task and the descriptor of
    # the task state segment TSS.
    # (0-nul, 1-cs, 2-ds, 3-syscall, 4-TSS0, 5-LDT0, 6-TSS1, 7-LDT1, 8-TSS2 etc...)

234 _gdt:   .quad 0x0000000000000000        /* NULL descriptor */
235         .quad 0x00c09a0000000fff        /* 16Mb */    # 0x08, kernel code seg.
236         .quad 0x00c0920000000fff        /* 16Mb */    # 0x10, kernel data seg.
237         .quad 0x0000000000000000        /* TEMPORARY - don't use */
238         .fill 252,8,0                   /* space for LDT's and TSS's etc */
```

6.4.3 Reference Information

6.4.3.1 Memory map after the end of head.s program execution

After the execution of head.s program, the kernel code has officially completed the settings of the memory page directory and the page tables, and re-created the interrupt descriptor table IDT and the global descriptor table GDT. In addition, the program also opened a 1KB byte buffer for the floppy disk driver. At this time the detailed image of the system module in memory is shown in Figure 6-11.

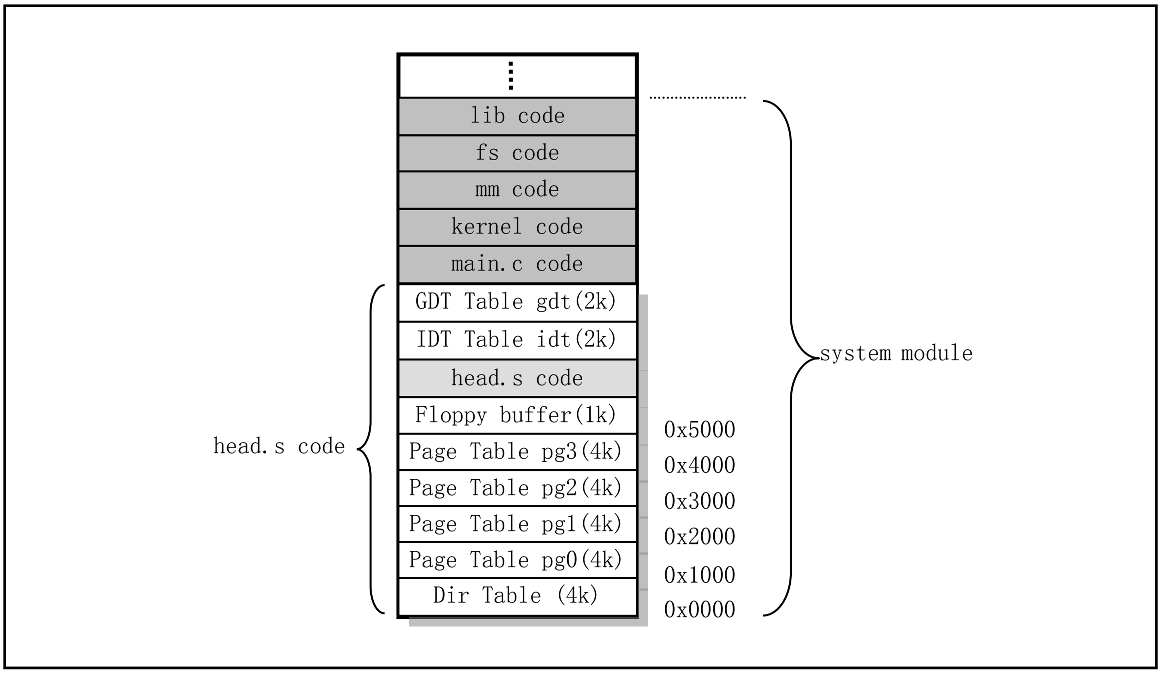


Figure 6-11 名 Map of the system module in memory

6.4.3.2 Intel 32-bit protection operation mechanism

The key to understanding this program is to know the operating mechanism of the Intel 80X386 32-bit protection mode。In order to be compatible with the 8086 CPU, the 80X86 protected mode was designed to be more complicated. See Chapter 4 for a detailed description of how the protected mode operates. Here we make a brief introduction to the protection mode by comparing the real mode and the protection mode.

When the CPU is running in real mode, the segment register is used to place the base address of a memory segment (for example, 0x9000). The size of the memory segment is fixed at 64KB. Up to 64KB of memory can be addressed in this segment. However, when entering the protection mode, the segment register does not contains segment base address in memory, but the selector of the descriptor item corresponding to the segment in the descriptor table. The 8-byte size descriptor contains the 'segment' base address and segment length of the segment linear address, as well as other bits describing the segment feature. Therefore, the memory location addressed at this time can be specified by the base address of the segment plus the current offset value. Of course, the actual physical memory address that is addressed needs to be transformed by the memory paging mechanism. In short, the memory addressing mode in 32-bit protected mode requires one more procedure, which is determined by using descriptors in the descriptor table and memory paging management.

Descriptor tables are divided into three types for different purposes: Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), and Local Descriptor Table (LDT). When the CPU is running in protected mode, there can only be one GDT and IDT at a time, and their table base addresses are specified by the registers GDTR and IDTR respectively. The number of local descriptor tables can be zero or up to 8191, as determined by the number of unused items in the GDT table and the specific system being designed. At some point, the base address of the current LDT table is specified by the contents of the LDTR register, and the contents of the LDTR are loaded using a descriptor in the GDT, that is, the LDT is also specified by the descriptor in the GDT.

In general, the kernel uses one LDT for each task (process). At runtime, the program can use the descriptors in the GDT as well as the descriptors in the LDT of the current task. For the Linux 0.12 kernel, there are 64 tasks that can be executed at the same time, so there are up to 64 descriptor entries for the LDT table in the GDT table.

The structure of the interrupt descriptor table IDT is similar to that of the GDT, which is located just in front of the GDT table in the Linux kernel. It contains a total of 256 8-byte descriptors. However, the format of each descriptor item is different from that of GDT, which stores the offset (0-1, 6-7 bytes) of the corresponding interrupt handler procedure, the selector of the segment (2-3 bytes), and Some flags (4-5 bytes).

Figure 6-12 is a schematic diagram of the descriptor table used in the Linux kernel. In the figure, each task occupies two descriptor items in the GDT. The LDT0 descriptor entry in the GDT table is the descriptor of the local descriptor table of the first task (process), and TSS0 is the descriptor of the task state segment (TSS) of the first task. Each LDT contains three descriptors, the first of which is not used, the second is the descriptor of the task code segment, and the third is the descriptor of the task data segment and the stack segment. When the DS segment register is the data segment selector of the first task, DS:ESI points to a certain data in the task data segment.

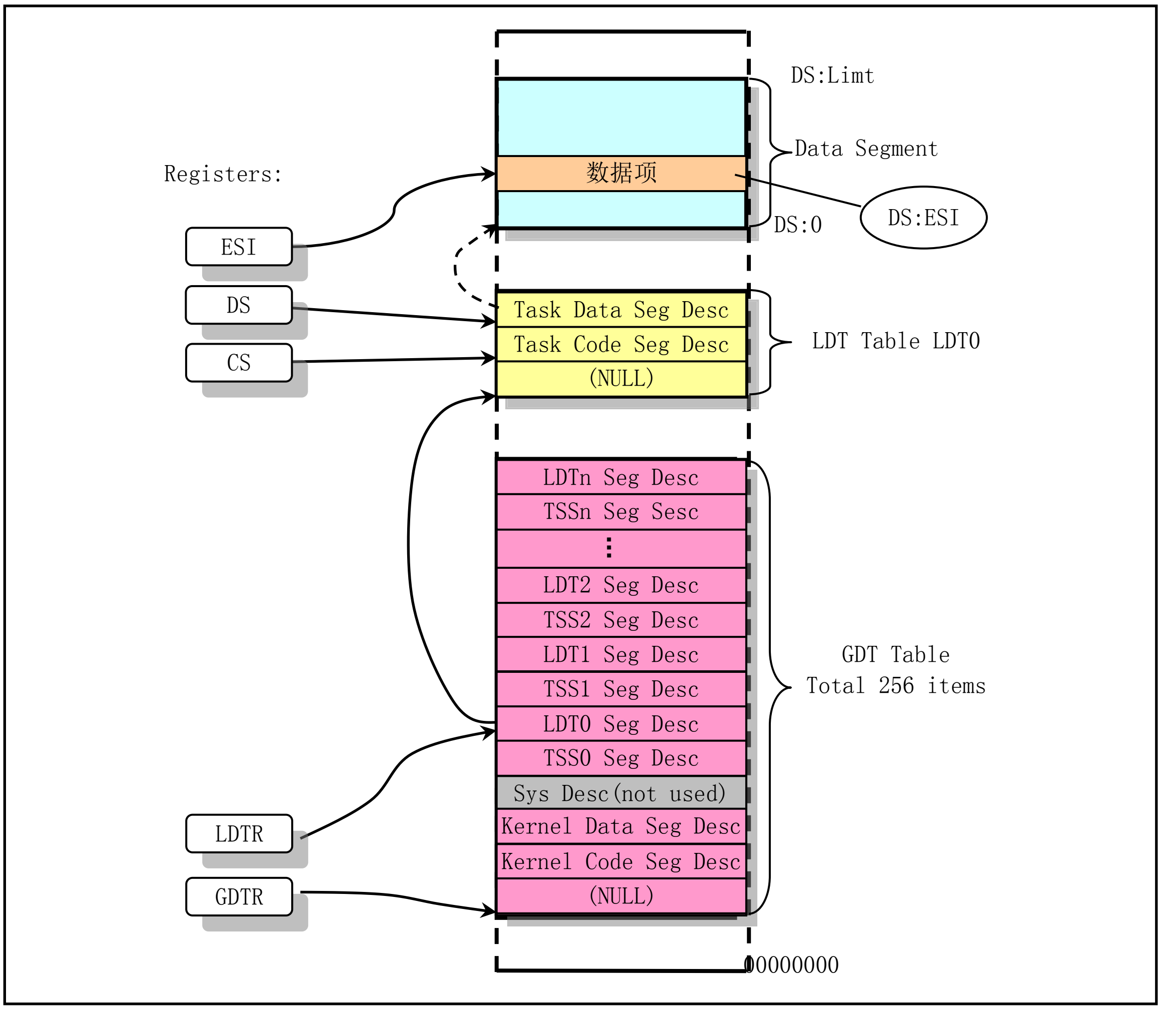


Figure 6-12 Descriptor table used by the Linux kernel

6.4.3.3 6.4.3.3 align directive

We have already explained the align directive when we introduced the assembler in Chapter 3. Here we will summarize it. The use of the directive .align is to instruct the compiler to populate the location counter (like the instruction counter) to a specified memory boundary at compile time. The goal is to increase the speed and efficiency with which the CPU can access code or data in memory. Its full format is:

.align val1, val2, val3

The first parameter value val1 is the required alignment ; the second is specified by the padding byte. The padding value can be omitted, and if omitted, the compiler is padded with a value of 0. The third optional parameter value val3 is used to indicate the maximum number that can be used for padding or skipping. If the boundary alignment exceeds the maximum number of bytes specified by val3, then no alignment is done at all. If you need to omit the second parameter val2 but still need to use the third parameter val3, you only need to put two commas.

For programs that now use the ELF object format, the first parameter, val1, is the number of bytes that need to be aligned. For example, '.align 8' means to adjust the position counter until it points to a multiple of 8 boundaries. If it is already on the multiple of 8, then the compiler does not have to change. However, for the system using the a.out object format here, the first parameter val1 is the number of the lower 0 bits, that is, the power of 2 (2^Val1). For example, '.align 3' in the previous program head.s means that the position counter needs to be on the multiple of 8 boundaries. Again, if it is already on the multiple of 8 boundaries, then the directive does nothing. GNU as(gas) treats these two target formats differently because Gas is formed to mimic the behavior of the assembler that comes with various architecture systems.

6.5 Summary

The boot loader bootsect.S loads the setup.s code and the system module into memory, and moves its self and setup.s code to physical memory 0x90000 and 0x90200 respectively, and then delegates the execution to the setup program. The header of the system module contains the header.s code.

The main function of the setup program is to use the ROM BIOS interrupt program to get some basic parameters of the machine, and save it in the memory block starting at 0x90000 for later programs. At the same time, move the system module down to the beginning of the physical address 0x00000, so the head.s code in the system is at the beginning of 0x00000. The descriptor table base address is then loaded into the descriptor table register to prepare for operation in 32-bit protected mode. Next, re-build the interrupt control hardware. Finally, set the machine control register CR0 and jump to the head.s code of the system module to start the CPU in 32-bit protected mode.

The main function of the Head.s program is to initially initialize the 256-item descriptor in the interrupt descriptor table, check if the A20 address line is already open, and test whether the system contains a math coprocessor. Then initialize the memory page directory table to prepare for the paging management of the memory. Finally, jump to the initialization program init/main.c in the system module to continue execution.

The main content of the next chapter is to describe in detail the function of the init/main.c program.

7 Initialization program (init)

There is only one main.c file in the init/ directory of the kernel source. The system module will pass execution rights to main.c after executed the code of boot/head.s. Although the program is not long, it includes all the work of kernel initialization. Therefore, when reading the kernel source, you need to refer to the initialization part of many other programs. If you can fully understand all the functions called here, then after reading this chapter you should have a general understanding of how the Linux kernel works.

Starting from this chapter, we will encounter a large number of C language programs, so readers should already have a certain C language knowledge. The best reference book on C should be the "C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie. The understanding of pointers and arrays in Chapter 5 of the book can be said to be the key to understanding the C language. In addition, you need to have the GNU gcc manual as a reference, because gcc's extended features, such as inline functions, inline assembly statements, etc., are used in many places in the kernel source code.

When annotating a C language program, we use '//' as the starting symbol of the comment statement in order to distinguish it from the original comment in the program. The translation of the original comments uses the same comment mark. For the header file (*.h) included in the program, only the meaning of the summary is gaven. The specific detailed comments on the header file will be given in the corresponding section of the comment header file.

7.1 main.c

7.1.1 Function description

The main.c program first uses the machine parameters obtained from the previous setup.s to set the system's root file device number and some memory global variables. These memory variables indicate the starting address of the main memory area, the amount of memory the system has, and the end address of the cache memory. If a virtual disk (RAMDISK) is also defined, the main memory area will be appropriately reduced. A schematic diagram of the entire memory space is shown in Figure 7-1. In the figure, the cache portion also needs to deduct the portions occupied by video memory and its BIOS of the display card. The high-speed cache is used for temporarily storing data from or to a block device such as a disk, and takes 1K (1024) bytes as a block unit. The main memory area is managed and allocated by the memory management module mm through the paging mechanism, and 4K bytes is a memory page unit. The kernel program has free access to the data in the cache, but it needs to pass mm to use the allocated memory page.

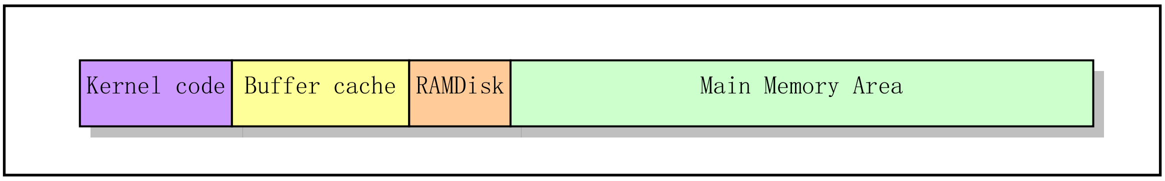


Figure 7-1 Diagram of memory function partitioning in the system

Then, the kernel code performs various aspects of hardware initialization work. This includes trap gates, block devices, character devices, and ttys, as well as manually setting up the first task (task 0). After all initialization work is completed, the kernel sets the interrupt enable flag to turn on the interrupt and switches to task 0 to run. At this point, it can be said that the kernel has basically completed all the setup work. The kernel then creates several initial tasks through task 0, runs the shell program and displays the command line prompt, so that the Linux system is in normal operation.

Note that when reading these initialization subroutines, it is best to go deeper into the program being called. If you really can't understand it, just put it aside and continue to look at the next initialization function. After some understanding, continue to study the places that have not been read.

7.1.1.1 Kernel initialization flow diagram

After the entire kernel has finished initializing, the kernel switches execution control to user mode (task 0), that is, the CPU switches from privilege level 0 to privilege level 3. At this point, the main program runs in task 0. Then, the system first calls the process creation function fork() to create a child process (init process) for running init(). The entire initialization process of the system is shown in Figure 7-2.

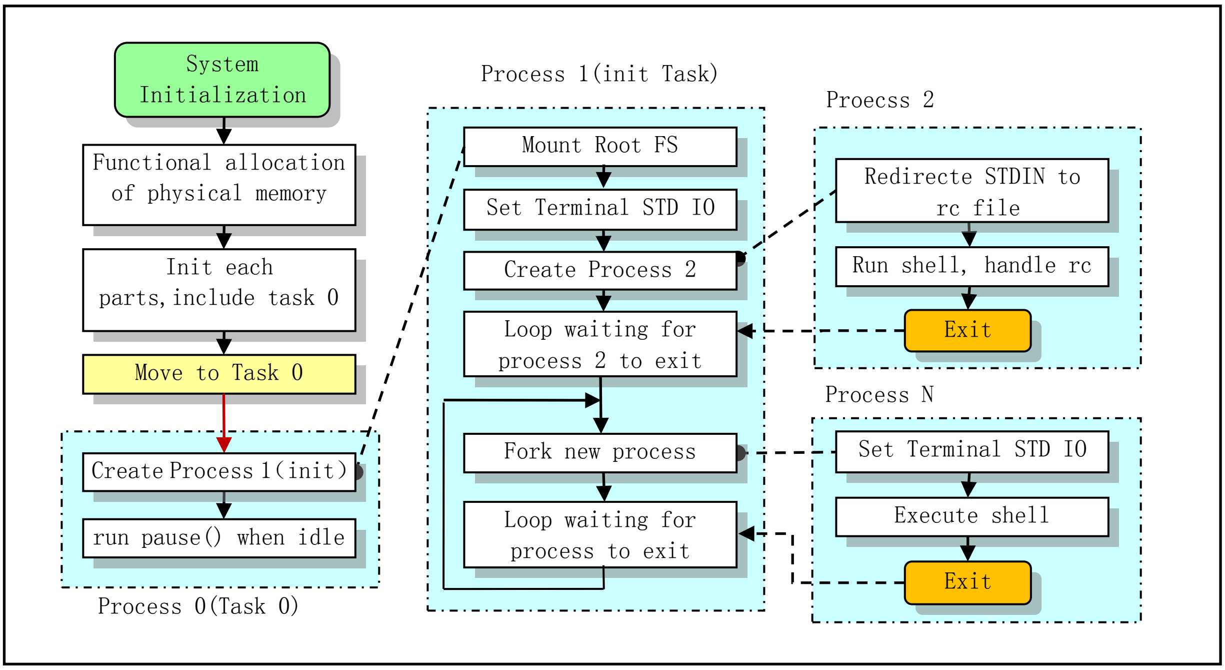


Figure 7-2 Kernel initialization process flow diagram

As can be seen from the figure, the main.c program first determines how to allocate the system physical memory, and then calls the initialization functions of each part of the kernel to initialize the memory management, interrupt processing, block device and character device, process management, and hard disk and floppy disk hardware. After these operations are completed, the various parts of the system are already operational. The program then moves itself "manually" to task 0 (process 0) and uses the fork() call to create process 1 (init process) for the first time and calls the init() function there. In this function the program will continue to initialize the application environment and execute the shell login program. The original process 0 is scheduled to execute when the system is idle, so process 0 is also commonly referred to as the idle process. At this point, process 0 only executes the pause() system call and calls the scheduler function.

The function of the init() can be divided into four parts: (1) to install the root file system; (2) to display system information; (3) to run the system initial resource configuration file rc; (4) to execute the user login shell program.

1. Install root file system

   The code first calls the syscall setup() to collect the hard disk device partition table information and install the root file system. Before installing the root file system, the system will first determine if you need to create a virtual disk first. If the size of the virtual disk is set when the kernel is compiled, and a memory has been opened for use as a virtual disk during the kernel initialization process, the kernel will first try to load the root file system into the virtual extent of the memory.

2. Display system information

   Then init() opens a terminal device tty0 and copies its file descriptor or handler to produce standard input stdin, standard output stdout, and error output stderr device. The kernel then uses these file descriptors to display some system information on the terminal, such as the total number of buffer blocks in the cache, the total number of bytes of free memory in the main memory area, and so on.

3. Run resource configuration file

   Next, init() creates a new process (Process 2) and performs some initial configuration operations in it to establish a user interaction environment. That is, before the user can use the shell command line environment, the kernel uses the /bin/sh program to run the commands set in the configuration file etc/rc. The role of the rc file is similar to the AUTOEXEC.BAT file on the root directory of the DOS operating system. This code first directs the standard input stdin to the etc/rc file by closing file descriptor 0 and immediately opening the file /etc/rc so that all standard input data will be read from the file. The kernel then executes /bin/sh in a non-interactive manner to implement the running of commands in the /etc/rc file. When the command in the file is executed, /bin/sh will exit immediately, so process 2 will end.

4. Execute user login shell

   The last part of the init() function is used to create a new session for the user in the new process and run the user login shell /bin/sh. When the system executes the program in process 2, the parent process (init process) waits for its end. With the exit of process 2, the parent process enters an infinite loop. In this loop, the parent process will generate a new process again, then create a new session in the process, and execute the program /bin/sh again for login to create a user interaction shell environment. The parent process then continues to wait for the child process. Although the login shell is the same program /bin/sh as the previous non-interactive shell, the command line arguments used (argv[]) are different. The first character of the 0th command line argument of the login shell must be a minus sign '-'. This particular flag tells /bin/sh that it is not a normal run, but runs /bin/sh as the login shell. From this point on, the user can use the Linux command line environment normally, and the parent process will enter the wait state. Thereafter, if the user executes the exit or logout command on the command line, after displaying a information of about current login shell exit, the system will repeat the process of creating the login shell process again in this infinite loop.

The last two parts of the init() function running in task 1 should actually be the functions of the independent environment initialization program init. See the description of this after the program list.

7.1.1.2 Operations of the initial user stack

Since the process of creating a new process is implemented by completely copying the parent process code and data segment, when the new process init is created using fork() for the first time, in order to ensure that there is no redundant information of process 0 in the user process stack of the new process, Process 0 should not use its user-mode stack until the first new process (Process 1) is created, that is, Task 0 is not required to call the function. Therefore, after the main.c program moves to the execution of task 0, the code fork() in task 0 should not be called as a function. The method implemented in the program is to execute this syscall using the gcc function inline form as shown below (see line 23 of the program):

> By declaring an inline function, you can have gcc integrate the code of the function into the code that called it. This will speed up code execution because it saves the overhead of function calls. In addition, if any of the actual arguments is a constant, then these known values at compile time may make the code simpler without including all the code for the inline function. See the relevant instructions in Chapter 3.

```
 23 static inline _syscall0(int,fork)
```

Where _syscall0() is the inline macro code in unistd.h, which calls the Linux system call interrupt int 0x80 in the form of embedded assembly. According to the macro definition on line 150 of the include/unistd.h file, we expand this macro and replace it with the above line. It can be seen that this statement is actually an int fork() creation process system call, as shown below.

```
 // The definition of _syscall0() in the unistd.h file. That is, the system calls the macro
 // function without parameters: type name(void).
150 #define _syscall0(type,name) \
151 type name(void) \
152 { \
153 long __res; \
154 __asm__ volatile ("int $0x80" \          // syscall INT 0x80
155         : "=a" (__res) \                 // return value -> eax(__res)
156         : "0" (__NR_##name)); \          // input is syscall number __NR_name.
157 if (__res >= 0) \                        // If return value >=0, the value is returned.
158         return (type) __res; \
159 errno = -__res; \                        // Otherwise set error number and return -1.
160 return -1; \
161 }
```

According to the above definition, we can expand _syscall0(int, fork). After substituting the 23rd line, we can get the following statement:

```
static inline int fork(void)
{
    long __res;
    __asm__ volatile ("int $0x80" : "=a" (__res) : "0" (__NR_fork));
    if (__res >= 0)
        return (int) __res;
    errno = -__res;
    return -1;
}
```

It can be seen that this is an inline function definition. gcc will insert the statement in the above "function" body directly into the code that calls the fork() statement, so executing fork() will not cause a function call. In addition, the last 0 in the macro name string "syscall0" means no parameter, and 1 means 1 parameter. If the system call has 1 parameter, then the macro _syscall1() should be used.

Although the above syscall execution interrupt instruction INT can not avoid using the stack, but the syscall uses the kernel state stack of the task instead of the user stack, and each task has its own independent kernel state stack, so the syscall will not affect the user state stack discussed here.

In addition, during the process of creating new process init (process 1), the system has some special handling. In fact, process 0 and process init use the same code and data memory pages (640KB) in the kernel, but the code that is executed is not in one place, so in fact they also use the same user stack area at the same time. When copying the page directory and page table entries of its parent process(process 0) for the new process init, the 640KB page table entry's attribute of process 0 has not been changed (still readable and writable), but the corresponding attribuite of process 1 is set to read-only. Therefore, when process 1 begins execution, its access to the user stack will result in a page write protection exception, which will cause the kernel's memory manager (mm) to allocate a memory page for process 1 in the main memory area, and copy the corresponding pages in task 0's stack to this new page. From this point on, the user mode stack of task 1 begins to have its own independent memory page. That is, after accessing the stack from task 1, the user stacks of task 0 and task 1 become independent of each other. Therefore, in order not to cause conflicts, task 0 must be required to prohibit the use of the user stack area before task 1 performs the stack operation.

In addition, because the order in which the kernel schedules each process is random, it is possible to run task 0 first after task 1 is created for task 1. Therefore, after task 0 executes the fork() operation, the subsequent pause() function must also be implemented in the form of an inline function to avoid task 0 using the user stack before task 1.

When a process in the system (such as the child process of the init process, process 2) has executed the execve() call, the code and data of process 2 will be located in the main memory area, so the system can use the copy-on-write technology to handle the creation and execution of other new processes at any time thereafter.

For Linux, all tasks run in user mode, including many system applications, such as shell programs, network subsystem programs, and so on. The library files in the kernel source code lib/ directory (except the string.c program) are designed to provide function support for the newly created processes. The kernel code itself at the 0 privilege level does not use these library functions.

7.1.2 **Program Annotations**

Program 7-1 linux/init/main.c

```
/*
 *  linux/init/main.c
 *
 *  (C) 1991  Linus Torvalds
 */

// unistd.h is a standard symbol constant and type file. It defines various symbol consts
// and types and declares various functions. If the symbol __LIBRARY__ is also defined,
// the system call number and the inline assembly code syscall0() are also included.
#define __LIBRARY__
#include <unistd.h>
#include <time.h>        // time type header file. tm structure, time function prototypes.

```

```
/*
 * we need this inline - forking from kernel space will result
 * in NO COPY ON WRITE (!!!), until an execve is executed. This
 * is no problem, but for the stack. This is handled by not letting
 * main() use the stack at all after fork(). Thus, no function
 * calls - which means inline code for fork too, as otherwise we
 * would use the stack upon exit from 'fork()'.
 *
 * Actually only pause and fork are needed inline, so that there
 * won't be any messing with the stack from main(), but we define
 * some others too.
 */
// Linux does not use copy-on-write when creating processes in kernel space. main() uses
// the inline fork() and pause() after moving to user mode (task 0), so it is guaranteed
// that the user stack of task 0 is not used.
// After executing moveto_user_mode(), the main() program is running as task 0. Task 0 is
// the parent of all the child processes that will be created. When it creates a child
// process (task or process 1, init), since the task 1 code is also in kernel space, no
// copy-on-write functionality is used. At this point, task 0 and task 1 use the same user
// stack space together, so you don't want to have any operations on the stack when running
// in task 0 environment to avoid messing up the stack. After executing fork() again and
// executing the execve() function, the loaded program is no longer in kernel space, so you
// can use the copy-on-write technique. See section 5.3, "How the Linux Kernel Uses Memory".

// The following _syscall0() is the inline macro code defined in unistd.h. It calls Linux's
// system call interrupt 0x80 in embedded assembly form. This interrupt is the entry point
// for all syscalls. The statement is actually a  creating process syscall int fork().You
// can expand it and you will understand it immediately. The last 0 in the syscall0 name
// indicates it contains no parameters, and 1 indicates 1 parameter. See include/unistd.h,
// lines 150-161.
// syscall pause() : Suspend process execution until a signal is received.
// syscall setup (void * BIOS): linux initialization (only called in this program).
// syscall sync(): update or synchronize the file system.
static inline _syscall0(int,fork)
static inline _syscall0(int,pause)
static inline _syscall1(int,setup,void *,BIOS)
static inline _syscall0(int,sync)


// <linux/tty.h>  defines parameters and constants for tty_io, serial communication.
// <linux/sched.h> scheduler header file defines the task structure task_struct, the data
//    of the initial task 0, and some embedded assembly function macro statements about
//    the descriptor parameter settings and acquisition.
// <linux/head.h> head file. A simple structure for the segment descriptor is defined,
//    along with several selector constants.
// <asm/system.h> system header file. An embedded assembly macro that defines or modifies
//    descriptors/interrupt gates, etc. is defined.
// <asm/io.h> io header file. Defines the function that operates on the io port in the
//    form of a macro's embedded assembler.
// <stddef.h> standard definition header file. NULL, offsetof(TYPE, MEMBER) is defined.
// <stdarg.h> Standard parameter file. Define a list of variable parameters in the form
//    of macros. It mainly describes one type (va_list) and three macros (va_start, va_arg
//    and va_end) for the vsprintf, vprintf, and vfprintf functions.
```

```
   // <unistd.h> Linux standard header file. Various symbol constants and types are defined
   //   and various functions are declared. If __LIBRARY__ is defined, it also includes the
   //   system call number and the inline assembly _syscall0().
   // <fcntl.h> File control header file. The definition of the operation control constant
   //   symbol used for the file and its descriptors.
   // <sys/types.h> type header file. The basic system data types are defined.
   // <linux/fs.h> fs header file. Define file table structure (file, buffer_head, m_inode, etc.)
   // <string.h> string header file. mainly defines embedded functions for string operations.
28 #include <linux/tty.h>     // teletype terminal.
29 #include <linux/sched.h>   // scheduler.
30 #include <linux/head.h>    // kernel head.
31 #include <asm/system.h>    // system machine.
32 #include <asm/io.h>        // port io.
33
34 #include <stddef.h>        // standard definition.
35 #include <stdarg.h>        // standard arguments.
36 #include <unistd.h>
37 #include <fcntl.h>         // file control.
38 #include <sys/types.h>     // date types.
39
40 #include <linux/fs.h>      // file system.
42 #include <string.h>        // string operations.
43
44 static char printbuf[1024];           // cache for kernel to display information.
45
46 extern char *strcpy();                // external functions defined elsewhere.
47 extern int vsprintf();                // Formatted output into a string (vsprintf.c, 92).
48 extern void init(void);               // Function prototype, init (168).
49 extern void blk_dev_init(void);       // Block dev init (blk_drv/ll_rw_blk.c,210)
50 extern void chr_dev_init(void);       // Char dev init(chr_drv/tty_io.c, 402)
51 extern void hd_init(void);            // Hard disk init(blk_drv/hd.c, 378)
52 extern void floppy_init(void);        // Floppy init (blk_drv/floppy.c, 469)
53 extern void mem_init(long start, long end);      // Memory init (mm/memory.c, 443)
54 extern long rd_init(long mem_start, int length); // Ramdisk init (blk_drv/ramdisk.c,52)
55 extern long kernel_mktime(struct tm * tm);       // Calc boot time (kernel/mktime.c, 41)
56
   // kernel-specific sprintf() function. This function is used to generate a formatted
   // message and output it to the specified buffer str. The parameter '*fmt' specifies the
   // format in which the output will be used, refer to the standard C language book.  This
   // function uses vsprintf() to put the formatted string into the str buffer, see the
   // printf() function on line 179.
57 static int sprintf(char * str, const char *fmt, ...)
58 {
59         va_list args;
60         int i;
61
62         va_start(args, fmt);
63         i = vsprintf(str, fmt, args);
64         va_end(args);
65         return i;
66 }
67
68 /*
```

```
69  * This is set up by the setup-routine at boot-time
70  */
   // The following three lines forcibly convert the specified linear address to a pointer of
   // the given data type and obtain the contents pointed. Since the kernel code segments are
   // mapped to locations starting from the physical address zero, these linear addresses are
   // also the corresponding physical addresses. See Table 6-3 in Chapter 6 for the meaning of
   // the memory values at these specified addresses.  See line 125 below for the drive_info
   // structure.
71 #define EXT_MEM_K (*(unsigned short *)0x90002)      // Extented Mem size(KB) beyond 1MB
72 #define CON_ROWS ((*(unsigned short *)0x9000e) & 0xff)  // Console rows and colums.
73 #define CON_COLS (((*(unsigned short *)0x9000e) & 0xff00) >> 8)
74 #define DRIVE_INFO (*(struct drive_info *)0x90080)  // Harddisk parameter table(32 bytes)
75 #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC)  // Root fs dev number.
76 #define ORIG_SWAP_DEV (*(unsigned short *)0x901FA)  // Swap file dev number.
77
78 /*
79  * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
80  * and this seems to work. I anybody has more info on the real-time
81  * clock I'd be interested. Most of this was trial and error, and some
82  * bios-listing reading. Urghh.
83  */
84
   // This macro reads the CMOS real time clock information. Outb_p and inb_p are port input
   // and output macros defined in include/asm/io.h. 0x70 is the write address port, and 0x71
   // is read data port. 0x80|addr is the CMOS memory address to be read.
85 #define CMOS_READ(addr) ({ \
86 outb_p(0x80|addr,0x70); \        // Output CMOS addr (0x80|addr) to be read to port 0x70
87 inb_p(0x71); \                   // Read 1 byte from port 0x71 and return the byte.
88 })
89
   // Define a macro to convert a BCD code to a binary value. The BCD code uses a half byte
   // (4 bits) to represent a decimal number, so one byte can represent two decimal numbers.
   // '(val)&15' takes the decimal digits represented by BCD, and '((val)>>4)*10' takes the
   // decimal tens digit of BCD and multiplies it by 10. The last two are added together,
   // which is the actual binary value of a byte BCD code.
90 #define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)
91
   // This function takes CMOS real clock information as the boot time and saves it to the
   // global variable startup_time (seconds). See the description of CMOS memory list later.
   // The function kernel_mktime() is used to calculate the number of seconds elapsed from
   // 0:00 on January 1, 1970 to the date of boot, as the boot time.
92 static void time_init(void)
93 {
94         struct tm time;                    // defined in include/time.h
95
   // CMOS access is very slow. In order to reduce the time error, after reading all the
   // values in the following loop, if the second value changes in CMOS at this time, then
   // all the values are read again. This allows the core to control the CMOS time error to
   // within 1 second.
96         do {
97                 time.tm_sec = CMOS_READ(0);     // current time seconds (BCD code).
98                 time.tm_min = CMOS_READ(2);     // current minutes.
99                 time.tm_hour = CMOS_READ(4);    // current hours.
100                 time.tm_mday = CMOS_READ(7);    // day in a month.
101                 time.tm_mon = CMOS_READ(8);     // current month (1-12).
102                 time.tm_year = CMOS_READ(9);    // current year.
103         } while (time.tm_sec != CMOS_READ(0));
104         BCD_TO_BIN(time.tm_sec);                // BCD to binary value.
105         BCD_TO_BIN(time.tm_min);
106         BCD_TO_BIN(time.tm_hour);
107         BCD_TO_BIN(time.tm_mday);
108         BCD_TO_BIN(time.tm_mon);
109         BCD_TO_BIN(time.tm_year);
110         time.tm_mon--;                          // range of months in tm_mon is 0-11.
111         startup_time = kernel_mktime(&time);    // calc boot time (kernel/mktime.c, 41)
112 }
113
    // Defines some static variables that can only be accessed by main.c source file.
114 static long memory_end = 0;                 // machine physical memory size (bytes).
115 static long buffer_memory_end = 0;          // buffer end address.
116 static long main_memory_start = 0;          // the main memory start position.
117 static char term[32];                       // terminal settings (environment parameter).
118
    // Command line and environment parameters used when manipulating the /etc/rc file.
119 static char * argv_rc[] = { "/bin/sh", NULL };        // parameter string array
120 static char * envp_rc[] = { "HOME=/", NULL ,NULL };   // environment string array
121
    // The command line arguments and environment parameters used to run the login shell.
    // The character "-" in argv[0] on line 122 is a flag passed to the shell program sh. By
    // identifying this flag, the sh program is executed as a login shell. Its execution is
    // not the same as executing sh at the shell prompt.
122 static char * argv[] = { "-/bin/sh",NULL };           // as above.
123 static char * envp[] = { "HOME=/usr/root", NULL, NULL };
124
125 struct drive_info { char dummy[32]; } drive_info;     // hard disk parameter table.
126
    // Main() is the kernel initialization main program. After the initialization is
    // completed, it will run in task 0 (idle task is idle task).
127 void main(void)         /* This really IS void, no error here. */
128 {                       /* The startup routine assumes (well, ...) this */
129 /*
130  * Interrupts are still disabled. Do necessary setups, then
131  * enable them
132  */
    // First save the root file system device number and the swap file device number. Then set
    // the console screen row and column number environment variable TERM according to the
    // information obtained in the setup.s program. Then use it to set the environment
    // variables used by the etc/rc file and the shell program in the init process. Then copy
    // the hard disk parameter table at memory 0x90080.
    // ROOT_DEV has been declared as extern int on line 206 in linux/fs.h file, and SWAP_DEV
    // has the same declaration in the linux/mm.h file. The mm.h file here is not explicitly
    // listed in front of the program, as it is already included in the linux/sched.h file
    // included earlier.
133         ROOT_DEV = ORIG_ROOT_DEV;                 // ROOT_DEV defined in fs/super.c, 29
134         SWAP_DEV = ORIG_SWAP_DEV;                 // SWAP_DEV defined in mm/swap.c, 36
135         sprintf(term, "TERM=con%dx%d", CON_COLS, CON_ROWS);
```

```
136         envp[1] = term;
137         envp_rc[1] = term;
138         drive_info = DRIVE_INFO;                // Copy hd parameter table at 0x90080

    // The location of cache and main memory area is then set according to the physical memory
    // capacity of the machine.
    // Cache end address -> buffer_memory_end; Machine memory capacity -> memory_end;
    // Main memory start address -> main_memory_start.
139         memory_end = (1<<20) + (EXT_MEM_K<<10); // mem size = 1MB + extended mem (bytes)
140         memory_end &= 0xfffff000;               // Ignore less than 4KB (1 page)
141         if (memory_end > 16*1024*1024)          // If mem size > 16MB, calculated as 16MB
142                 memory_end = 16*1024*1024;
143         if (memory_end > 12*1024*1024)          // If mem size >12MB, set buffer end = 4MB
144                 buffer_memory_end = 4*1024*1024;
145         else if (memory_end > 6*1024*1024)      // if size >6Mb, set buffer end = 2Mb
146                 buffer_memory_end = 2*1024*1024;
147         else
148                 buffer_memory_end = 1*1024*1024; // Otherwise set buffer end = 1Mb
149         main_memory_start = buffer_memory_end;

    // If symbol RAMDISK is defined in the Makefile, the virtual disk is initialized.  At this
    // point, the main memory area will be reduced. See kernel/blk_drv/ramdisk.c.
150 #ifdef RAMDISK
151         main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
152 #endif
    // The following is the initialization of all aspects of the kernel. It is better to
    // follow into each init function when reading.
153         mem_init(main_memory_start,memory_end);  // Main mem area (mm/memory.c,443)
154         trap_init();                // trap gate (hw vector) init (kernel/traps.c, 181)
155         blk_dev_init();             // block dev init (blk_drv/ll_rw_blk.c, 210)
156         chr_dev_init();             // char dev init (chr_drv/tty_io.c, 402)
157         tty_init();                 // tty init (chr_drv/tty_io.c, 105)
158         time_init();                // setting boot time (line 92)
159         sched_init();               // scheduler init (kernel/sched.c, 385)
160         buffer_init(buffer_memory_end); // buffer init(fs/buffer.c, 348)
161         hd_init();                  // harddisk init (blk_drv/hd.c, 378)
162         floppy_init();              // floppy init(blk_drv/floppy.c, 469)
163         sti();                      // All init has completed, so enable interrupt.

    // The task 0 is started by manipulating data in the stack and using interrupt return
    // instruction. Then immediately derive a new task 1 (known as init process) in task 0
    // and execute init() in the new task. For the child process being created, fork() will
    // return 0, and for the original process (parent process) it will return the process
    // number pid of the child process.
164         move_to_user_mode();    // see head file include/asm/system.h.
165         if (!fork()) {          /* we count on this going ok */
166                 init();         // run init() in task 1 (init process).
167         }

    // The following code runs in task 0.
168 /*
169  *   NOTE!!   For any other task 'pause()' would mean we have to get a
170  * signal to awaken, but task0 is the sole exception (see 'schedule()')
```

```
171  * as task 0 gets activated at every idle moment (when no other tasks
172  * can run). For task0 'pause()' just means we go check if some other
173  * task can run, and if not we return here.
174  */
    // The pause() syscall converts task 0 into an interruptible wait state before executing
    // the scheduler function. However, the scheduler will switch back to task 0 as long as it
    // finds that no other tasks in the system can run, and does not depend on the state of
    // task 0. See (kernel/sched.c,144).
175         for(;;)
176                 __asm__("int $0x80"::"a" (__NR_pause):"ax");        // syscall pause()
177 }
178
    // The printf() function produces formatting information and outputs it to the standard
    // output device stdout(1) for display. The parameter '*fmt' specifies the format used for
    // the output, see the standard C language book. The program uses vsprintf() to put the
    // formatted string into the printbuf buffer and then use write() to output the contents
    // to stdout. See the kernel/vsprintf.c for the implementation of vsprintf() function.
179 static int printf(const char *fmt, ...)
180 {
181         va_list args;
182         int i;
183
184         va_start(args, fmt);
185         write(1,printbuf,i=vsprintf(printbuf, fmt, args));
186         va_end(args);
187         return i;
188 }
189

    // The init() function runs in the newly created process 1 (or init process). It first
    // initializes the environment of the first program to be executed (shell), then loads the
    // program as a login shell and executes it.
190 void init(void)
191 {
192         int pid,i;
193
    // setup() is a syscall that reads the hard disk parameters including the partition
    // table information and loads the virtual disk (if it exists) and installs the root file
    // system device. This function is defined with a macro on line 25, and the corresponding
    // function is sys_setup(). See kernel/blk_drv/hd.c, line 74 for its implementation.
194         setup((void *) &drive_info);

    // The terminal console device "/dev/tty0" is opened in read/write mode. Since this is the
    // first time the system opened a file, the resulting file handle number (file descriptor)
    // is definitely 0. The file handle (0) corresponds to the default console standard input
    // device stdin of the UNIX-like operating system. It is copied here and opened separately
    // in read and write to produce a standard output handle stdout (1) and a standard error
    // output handle stderr (2). The "(void)" prefix in front of the function is used to force
    // the function to return no value.
195         (void) open("/dev/tty1",O_RDWR,0);
196         (void) dup(0);              // duplicate file handle 0 to produce stdout (1).
197         (void) dup(0);              // duplicate file handle 0 to produce stderr (2).
```

```
    // Print the buffer blocks (1024 bytes per block) and total number of bytes, as well as
    // free bytes in the main memory area.
198         printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
199                 NR_BUFFERS*BLOCK_SIZE);
200         printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);

    // Next create a child process (task 2) and run the commands in the /etc/rc file in that
    // child process. For the child process being created, fork() will return 0, and for the
    // original process it will return the process number pid of the child process. So lines
    // 202-206 are the code that is executed in the child process. The child process code
    // first redirects the standard input (stdin) to the /etc/rc file, then runs the /bin/sh
    // program with the execve().The program reads the commands in the rc file from stdin and
    // executes them in an interpreted manner. The parameters and environment variables used
    // by the sh are given by the argv_rc and envp_rc arrays, respectively.
    // Closing handle 0 and immediately opening the /etc/rc file redirects the standard input
    // stdin to the /etc/rc file. This allows you to read the contents of the /etc/rc file
    // through a console read operation. Since sh runs non-interactively, it will exit
    // immediately after executing the rc file, and process 2 will end. For a description of
    // the execve(), see the fs/exec.c program, line 207. Error code when function _exit()
    // exits: 1- Operation is not permited; 2-File or directory does not exist.
201         if (!(pid=fork())) {
202                 close(0);
203                 if (open("/etc/rc",O_RDONLY,0))
204                         _exit(1);                    // open failed, (lib/_exit.c,10)
205                 execve("/bin/sh",argv_rc,envp_rc);   // execute program /bin/sh
206                 _exit(2);
207         }

    // Below is the statement executed by process 1. wait() waits for the child process to
    // stop or terminate, and the return value should be the process identifier (pid) of the
    // child process. The following code acts as the parent process waiting for the end of the
    // child process. &i stores the return status information. If the return value of wait()
    // is not equal to the child process id, continue to wait.
208         if (pid>0)
209                 while (pid != wait(&i))
210                         /* nothing */;

    // If the code is executed here, the child process just created has finished executing rc
    // file (or file does not exist), so the child process automatically stops or terminates.
    // In the loop below, a child process is created again to run login and console shell program.
    // The new child process will first close all previously leftover handles and create a new
    // session. Then reopen /dev/tty0 as stdin and copy it into stdout and stderr, and execute
    // the /bin/sh program again. But this time the arguments and environment arrays used by
    // the shell are another set (see lines 122--123 above). At this point the parent process
    // (process 1) runs wait() again. If the child process stops once again, an error message
    // is displayed, and then the code continue to try..., forming a "big" infinite running loop.
211         while (1) {
212                 if ((pid=fork())<0) {
213                         printf("Fork failed in init\r\n");
214                         continue;
215                 }
216                 if (!pid) {                           // new process
217                         close(0);close(1);close(2);
218                     setsid();                   // create a new session.
219                     (void) open("/dev/tty1",O_RDWR,0);
220                     (void) dup(0);
221                     (void) dup(0);
222                     _exit(execve("/bin/sh",argv,envp));
223             }
224             while (1)
225                     if (pid == wait(&i))
226                             break;
227             printf("\n\rchild %d died with code %04x\n\r",pid,i);
228             sync();                             // flush the buffer.
229     }
230     _exit(0);       /* NOTE! _exit, not exit() */
   // Both _exit() and exit() are used to terminate a function normally. But _exit() is
   // directly a sys_exit syscall, and exit() is usually a function in the normal library.
   // It performs some cleanup operations, such as executing each termination code, closing
   // all standard IOs, etc., and then calling sys_exit.
231 }
232
```

7.1.3 Reference Information

7.1.3.1 CMOS

The CMOS memory of a PC is a 64- or 128-byte memory block powered by a battery, usually part of the real-time clock (RTC) chip. Some machines have a larger CMOS memory capacity. The 64-byte CMOS was originally used on IBM PC-XT machines to store clock and date information in a BCD format. Since this information only uses 14 bytes, the remaining bytes can be used to store some system configuration data.

The CMOS address space is outside the base memory address space, so executable code is not included. To access it you need to go through I/O ports. 0x70 is the address port and 0x71 is the data port. In order to read the byte of the specified offset, the OUT instruction must first be used to send the offset value of the byte to port 0x70, and then use IN instruction to read the byte from data port 0x71. Similarly, for a write operation, it is first necessary to send the offset of the byte to port 0x70, and then write the data to port 0x71.

The main.c program line 86 statement does not need to perform an OR operation for the byte address with 0x80. Because the CMOS memory capacity at that time has not exceeded 128 bytes, it does not have any effect on OR operation with 0x80. After the kernel 1.0, the operation is removed (see the code from line 42 of the kernel driver/block/hd.c in v1.0). Table 7–1 is a short list of CMOS memory information.

Table 7-1 CMOS 64-byte information profile

Address	Description	Address	Description
0x00	Current Seconds (real clock)	0x11	Reserved
0x01	Alarm Seconds	0x12	Hard drive type
0x02	Current Minutes (real clock)	0x13	Reserved
0x03	Alarm Seconds	0x14	Device byte
0x04	Current Hours (real clock)	0x15	Basic memory (low byte)
0x05	Alarm Hours	0x16	Basic memory (high byte)
0x06	Current day of the week (real clock)	0x17	Extended memory (low byte)

0x07	Date of the day in a month (real clock)	0x18	Extended memory (high byte)
0x08	Current month (real clock)	0x19-0x2d	Reserved
0x09	Current year (real time clock)	0x2e	Checksum (low byte)
0x0a	RTC Status Register A	0x2f	Checksum (high byte)
0x0b	RTC Status Register B	0x30	Extended memory above 1MB (low byte)
0x0c	RTC Status Register C	0x31	Extended memory above 1MB (high byte)
0x0d	RTC Status Register D	0x32	Current century
0x0e	POST diagnostic status byte	0x33	Information flag
0x0f	Shutdown status byte	0x34-0x3f	Reserved
0x10	Floppy disk drive type		

7.1.3.2 Forking a new process

Fork() is a system call function that copies the current process and creates a new entry in the process table that is almost identical to the original process (called the parent process) and executes the same code. But the new process (child process) has its own data space and environment parameters. The main purpose of creating a new process is to run the program concurrently, or use the exec() cluster function to execute other different programs in the new process.

At the return position of the fork, the parent process will resume execution, and the child process will just begin execution. In the parent process, the call to fork() returns the process ID of the child process, and in the child process fork() returns a value of 0. In this way, although it is still executed in the same program at this time, they have started to fork and each executes its own code. If the fork() call fails, a value less than 0 is returned. As shown in Figure 7-3.

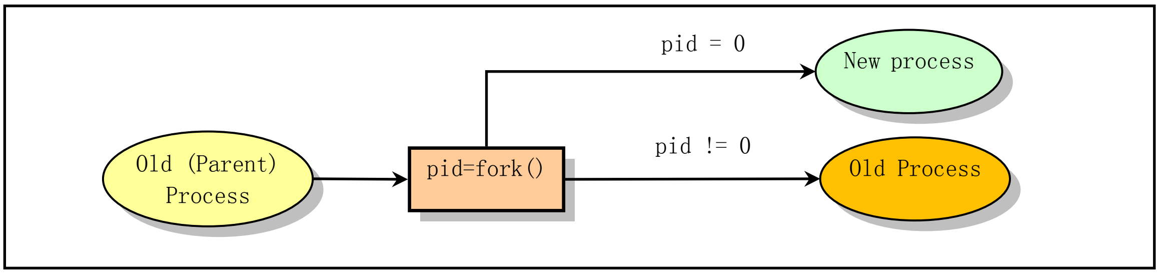


Figure 7-3 Call fork() to create a new process

The init() uses the return value of the fork() to distinguish and execute different code parts. Lines 201 and 216 in the main.c program are the code blocks that the child process starts executing (using the execve() system call to execute other programs, where sh is executed), and lines 208 and 224 are the code blocks executed by the parent process.

exit() can be called to exit the execution of the program when the program has finished or is necessary to terminate. This function terminates the process and releases the kernel resources it occupies. The parent process can use the wait() call to view or wait for the child process to exit, and obtain the exit status information of the terminated process.

7.1.3.3 Concept of session

As we said earlier, a program is an executable file, and a process is an instance of a program that is executing. In the kernel, each process is identified by a different positive integer greater than zero, called the process identification number pid (Porcess ID). A process can create one or more child processes through fork(), and these processes can form a process group. For example, for a pipe command typed on the shell command line,

```
[root]# cat main.c | grep for | more
```

Each of these commands: cat, grep, and more belong to a process group.

A process group is a collection of one or more processes. Similar to a process, each process group has a unique process group identification number gid (Group ID), and it is also a positive integer. Each process group has a process called a group leader. The group leader process is a process whose pid is equal to the process group number gid. A process can participate in an existing process group or create a new process group by calling setpgid().The concept of process group has many uses, but the most common one is that we issue a termination signal to the foreground execution program on the terminal (usually by pressing the Ctrl-C key) and terminate all processes in the entire process group. For example, if we issue a termination signal to the above pipe command, the three commands will terminate execution at the same time.

A session is a collection of one or more process groups. Normally, all the programs executed after the user logs in belong to a session, and the login shell is the session leader, and the terminal used by it is the control terminal of the session. Therefore, the first process of a session is also commonly referred to as a Controlling process. When we log out, all processes belonging to our session will be terminated, and this is one of the main uses of the concept of the session. The setsid() function is used to create a new session, usually called by the environment initializer. The relationship between processes, process groups, and session periods is shown in Figure 7-4.

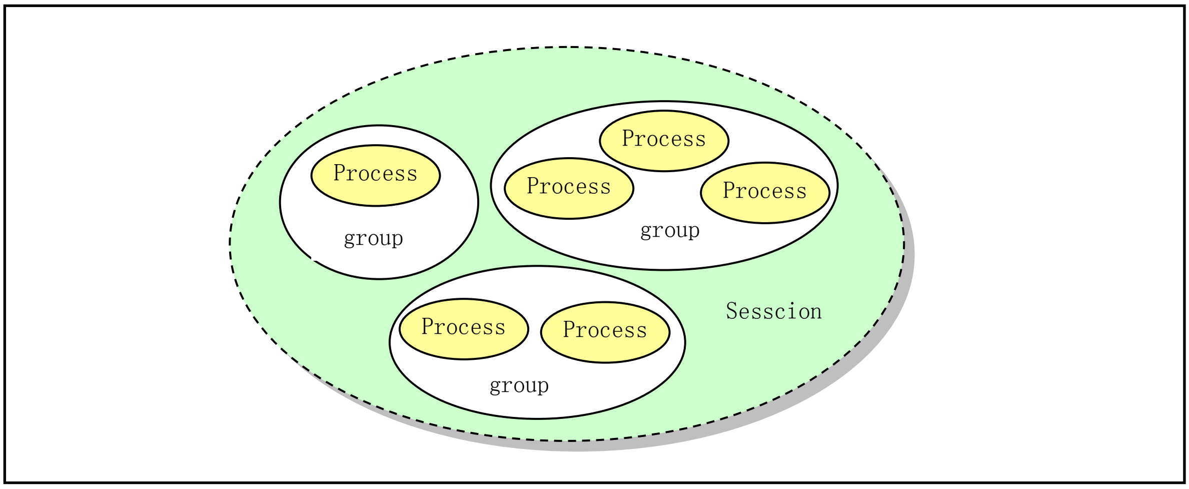


Figure 7-4 Relationship between processes, process groups, and sessions

Several process groups in a session are divided into a foreground process group and one or several background process groups. A terminal can only act as a control terminal for a session. A foreground process group is a process group that has a control terminal in the session, and other process groups in the session become a background process group. The control terminal corresponds to the /dev/tty device file, so if a process needs to access the control terminal, you can directly read and write the /dev/tty file.

7.2 Environment initialization

After the kernel is initialized, the system also needs to perform further environment initialization according to the specific configuration, in order to truly have the working environment of a common system. On lines 205 and 222 above, the init() function directly executes the command interpreter (shell) /bin/sh, which is not the case in actually available systems. In order to have the login function and the ability of multiple people to use the system at the same time, the usual system is to execute the system environment initialization program init.c here or in a similar place, and the program will be based on the setting information of the configuration file in the /etc/ directory. Each terminal device supported in the system creates a child process, and runs the terminal initialization setting program agetty (collectively called getty) in the child process. The getty will display the user login prompt message "login:" on the terminal. When the user types in the username, getty is replaced with the login program. After verifying the correctness of the user's input password, the login program finally calls the shell program and enters the shell interaction interface. The execution relationship between them is shown in Figure 7-5.

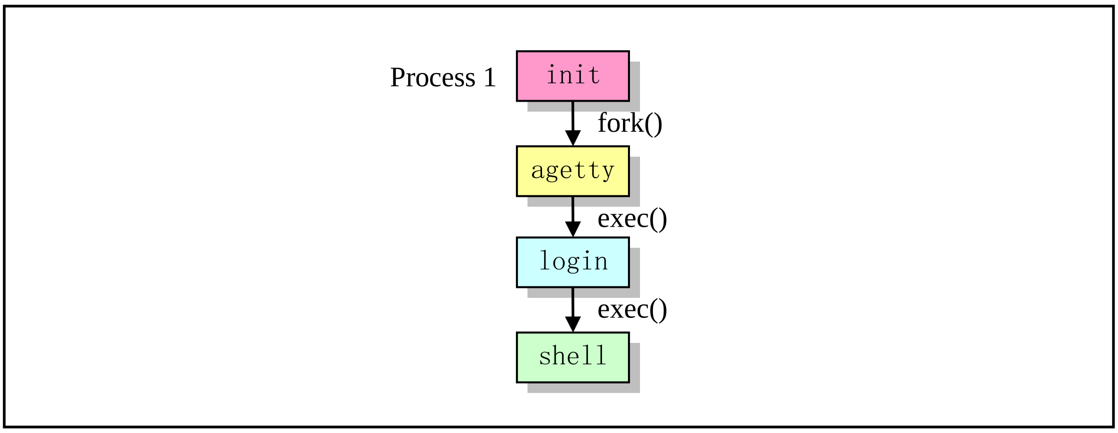


Figure 7-5 Procedures for environment initialization

Although these programs (init, getty, login, shell) are not part of the kernel, a basic understanding of their role will facilitate understanding of the many functions that the kernel provides.

The main job of init process is to execute the commands set in /etc/rc file, and then create a child process for each terminal device that is allowed to log in using fork() according to the information in the /etc/inittab file, and run the agetty (getty) program in each newly created child process. At this point, the init process calls wait() and waits for the end of the child process. Whenever one of its child processes ends and exits, it will know which sub-process of the corresponding terminal is finished according to the pid returned by wait(), so a new sub-process will be created for the corresponding terminal device, and re-execute the agetty program in the child process. In this way, each allowed terminal device always has a corresponding process waiting for processing.

Under normal operation, init determines that agetty is working to allow the user to log in and to collect orphaned processes. An orphan process is one that has its parent process ended. All processes in Linux must belong to a single process tree, so orphaned processes must be gathered. When the system is shut down, init is responsible for killing all other processes, unmounting all filesystems and stopping the processor, as well as any other work that is configured to do.

The main task of the getty program is to set the terminal type, properties, speed, and line discipline. It opens and initializes a tty port, displays a prompt, and waits for the user to type in the username. This program can only be executed by the superuser. Usually, if the /etc/issue text file exists, getty will first display the text message, then display the login prompt information (for example: plinux login:), read the login name entered by the user, and then execute the login program.

The login program is mainly used to require the login user to enter a password. According to the user name entered by the user, it obtains the corresponding user's login item from the password file passwd, then calls getpass() to display the "password:" prompt message, reads the password typed by the user, and then uses the encryption algorithm to the input password, and compared to the pw_passwd field in the user entry in the password file. If the password entered by the user is invalid, the login program will exit with the error code 1, indicating that the login process failed. At this time, the wait() of the parent process (process init) will return the pid of the exit process, so init will create a child process again according to the recorded information, and execute the agetty program again for the terminal device in the child process. This repeats the above process.

If the password entered by the user is correct, login will change the current working directory to the user's starting working directory specified in the password file, and modify the access rights to the terminal device to user read/write and group write, and set the group ID of the process. Then use the information obtained to initialize environment variable information, such as the starting directory (HOME=), the shell program used (SHELL=), the user name (USER= and LOGNAME=), and the default path sequence of the system executive (PATH=). The text message in the /etc/motd file (message-of-the-day) is then displayed and checked to see if the user has mail information. Finally, the login program changes to the user ID of the logged in user and executes the shell program specified in the user item in the password file, such as bash or csh.

If the shell name in the password file /etc/passwd does not specify which shell to use, the system will use the default /bin/sh program. If the user's home directory is not specified, the default root directory / will be used. For a description of some of the execution options and special access restrictions for the login program, see the online manual page (man 8 login) on Linux.

A shell program is a complex command-line interpreter that is executed when a user logs into the system for interactive operations. It is where the user interacts with the computer. It takes the information entered by the user and then executes the command. The user can interact directly to the shell on the terminal, or use the shell script file to input to the shell.

When starts executing the shell during login, the first character of the parameter argv[0] is '-', indicating that the shell is executed as a login shell. At this point, the shell program will perform some operations corresponding to the login process based on the character. The login shell will first read the command from the /etc/profile file and the .profile file (if it exists) and execute it. If the ENV environment variable is set when entering the shell, or if the variable is set in the .profile file when logging in to the shell, the shell will next read the command from the file and execute it. Therefore, the user should put the command to be executed at login time in the .profile file, and put the command to be executed every time the shell is executed in the file specified by the ENV variable. The way to set the ENV environment variable is to put the following statement in the .profile file in your home directory:

```
    ENV=$HOME/.anyfilename; export ENV
```

When executing the shell, in addition to some of the specified options, if the command line argument is also specified, the shell will treat the first argument as a script filename and execute the command. The rest of the parameters are treated as shell parameters ($1, $2, etc.). Otherwise the shell program will read the command from its standard input.

There are many options for executing a shell program, see the instructions in the online manual page for sh on a Linux system.

7.3 Summary

Through the code analysis in this chapter, we know that for the kernel 0.12, as long as the root file system is a MINIX, and it contains files /etc/rc, /bin/sh, /dev/*, and some directories (/etc/, /dev/, /bin/, /home/, /home/root/), we can form a simple root file system to make the Linux system run.

From here on, for the reading of subsequent chapters, the main.c program can be used as a main line, and does not need to be read in chapter order. If the reader does not understand the memory paging mechanism, it is recommended to first read the contents of chapter 10 memory management.

In order to understand the contents of the following chapters smoothly, the author strongly hopes that readers can review the mechanism of 32-bit protection mode operation at this time. Read the relevant contents in the appendix in detail, or refer to the Intel 80x86 books for protection mode.

If you get here in order of chapters, then you should have a general understanding of the initialization process of the Linux kernel. But you might also ask the question: "After generating a series of processes, how does the system run these processes in a time-sharing manner or how to schedule them to run? That is how the 'wheels' turn around?" . The answer is not complicated: the kernel is run by executing the schedule() and the timer clock interrupt handler _timer_interrupt. The kernel sets a clock interrupt every 10 milliseconds, and during the interrupt processing, the next state of the process is determined by calling the do_timer() function to check the current execution of all processes. According to the state of each process, the scheduler schedules each process to execute sequentially.

When the process needs to wait for a while due to the temporary lack of resources, it will indirectly call the schedule() through the sleep_on() to voluntarily hand over the CPU rights to the other processes. As for which process the system will run next, it is completely determined by schedule() according to the current state and priority of all processes. For a process that is always in a runnable state, when the clock interrupt process determines that the time slice it is running has been used up, the process switch operation is performed in do_timer(), and the CPU usage rights of the process are reluctantly deprived, and the kernel will schedule other processes to run.

The scheduling function schedule() and the clock interrupt handler procedure are one of the important topics in the next chapter.

8 Kernel Code (kernel)

The linux/kernel/ directory contains 10 C files and 2 assembly language files, as well as a Makefile, as shown in List 8-1. Comments on the code in three subdirectories will be made in subsequent chapters. This chapter mainly comments on these 12 code files. First, we give a general introduction to the basic functions of all programs, so that we can get a general understanding of the functions implemented by these kernel code and the calling relationship between them, and then make detailed comments on each code file.

List 8-1 linux/kernel/

	Filename	Size	Last Modified Time (GMT)	Description
📁	blk_drv/		1992-01-16 14:39:00	
📁	chr_drv/		1992-01-16 14:37:00	
📁	math/		1992-01-16 14:37:00	
📄	Makefile	4034 bytes	1992-01-12 19:49:12	
📄	asm.s	2422 bytes	1991-12-18 16:40:03	
📄	exit.c	10554 bytes	1992-01-13 21:28:02	
📄	fork.c	3951 bytes	1992-01-13 21:52:19	
📄	mktime.c	1461 bytes	1991-10-02 14:16:29	
📄	panic.c	448 bytes	1991-10-17 14:22:02	
📄	printk.c	537 bytes	1992-01-10 23:13:59	
📄	sched.c	9296 bytes	1992-01-12 15:30:13	
📄	signal.c	5265 bytes	1992-01-10 00:30:25	
📄	sys.c	12003 bytes	1992-01-11 00:15:19	
📄	sys_call.s	5704 bytes	1992-01-06 21:10:59	
📄	traps.c	5090 bytes	1991-12-18 19:14:43	
📄	vsprintf.c	4800 bytes	1991-10-02 14:16:29	

8.1 Main Functions

The code files in this directory can be divided into three categories, one is the hardware (exception) interrupt handler files; the other is the system call service handler files; the third is the general function files such as process scheduling, see Figure 8-1. We now provide a more detailed explanation of the functionality implemented based on this classification.

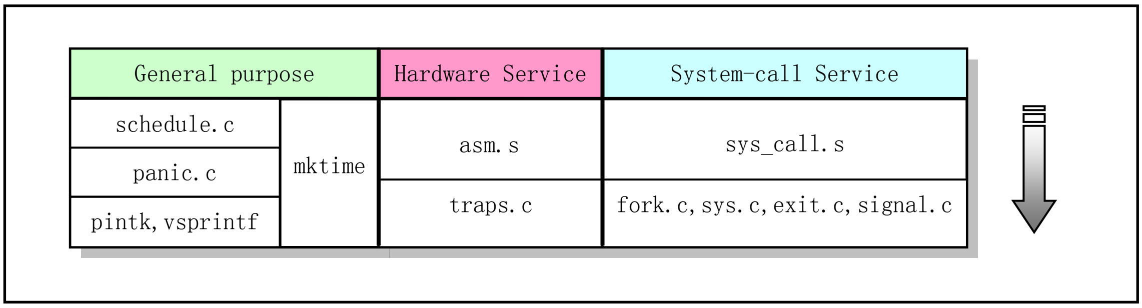


Figure 8-1 Invocation hierarchical relationships between each files

8.1.1 Interrupt Processing

There are two files related to interrupt handling: asm.s and traps.c. Asm.s is used to implement the assembly processing part of the interrupt service caused by most hardware exceptions, while the traps.c implements the C function part called in the asm.s interrupt processing. This C function part sometimes called Bottom-halfs of the interrupt handling. In addition, several other hardware interrupt handlers are implemented in the files sys_call.s and mm/page.s. See Figure 5-21 for the connection and function of the 8259A programmable interrupt control chip in the PC.

In Linux systems, the interrupt service function is provided by the kernel, so the interrupt handler uses the kernel state stack of the process. Before the user program (process) passes control to the interrupt handler, the CPU first pushes at least 12 bytes (EFLAGS, CS, and EIP) into the stack of the interrupt handler, that is the kernel state stack of the process. See Figure 8-2(a). This situation is similar to a far call (inter-segment call). The CPU pushes the code segment selector and offset of the return address onto the stack. Another point that is similar to inter-segment calls is that the 80X86 CPU pushes information onto the stack of destination code (interrupt handler code) instead of the interrupted code stack. If the priority level changes, such as from user level to kernel level, the CPU also pushes the stack segment SS and stack pointer ESP of the original code onto the stack of the interrupt handler. But after the kernel is initialized, the kernel code is executed using the kernel state stack of the process, so the stack of the destination code here refers to the kernel state stack of the process, and the stack of interrupted code is of course the user state stack of the process. So when an interrupt occurs, the interrupt handler uses the kernel state stack of the process. In addition, the CPU always pushes the contents of the EFLAGS onto the stack. A schematic diagram of the contents of the stack with priority changes is shown in Figure 8 (c) and (d).

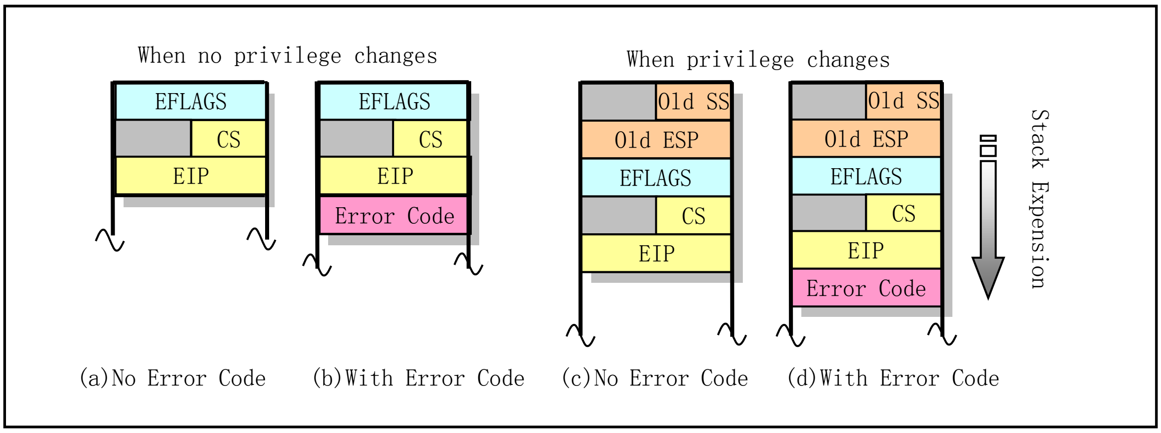
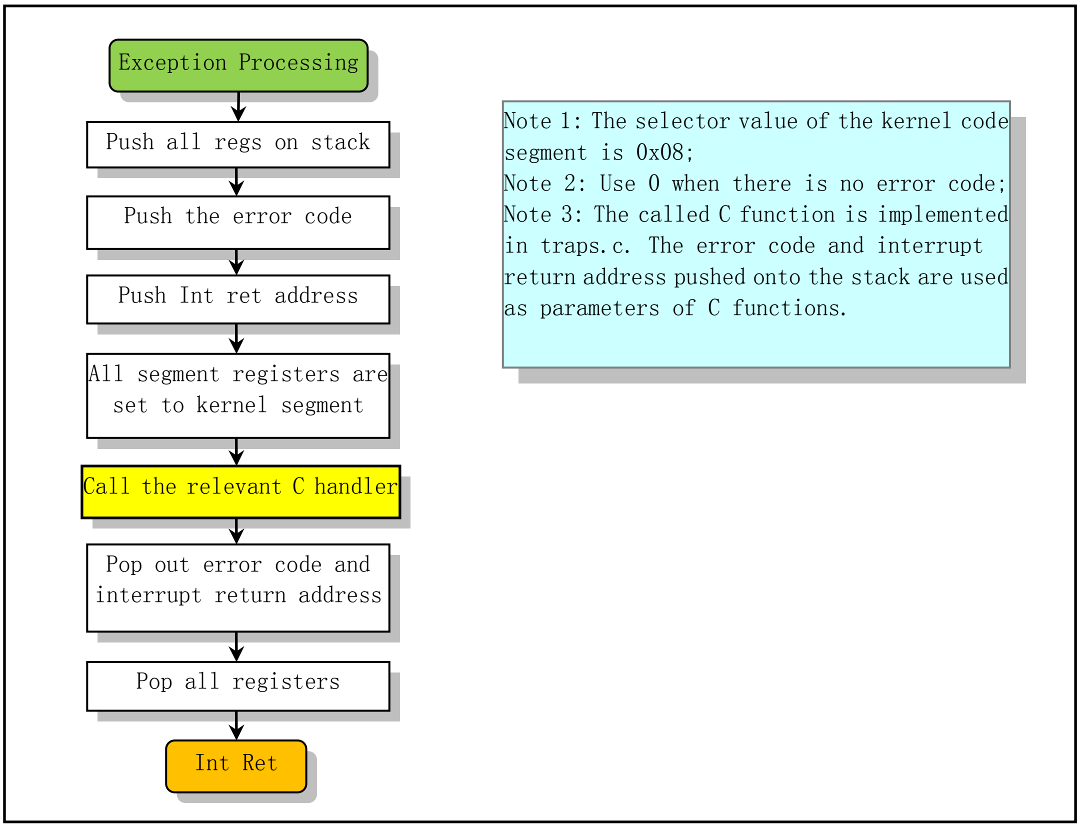


Figure 8-2 The contents of the stack when an interrupt occurs

The asm.s code file mainly deals with the processing of Intel's reserved interrupt INT0--INT16, and the remaining reserved interrupts (INT17--INT31) are reserved for future expansion by Intel Corporation. The processing of 16 interrupts (INT32--INT47) corresponding to the IRQ pins of the PIC chip will be set in the initialization routines of various hardware such as clock, keyboard, floppy disk, math coprocessor, hard disk, etc. . The handler for the Linux system call interrupt INT128 (INT0x80) will be given in kernel/sys_call.s. The specific definition of each interrupt is described in the Reference Information section given after the code file comment.

When some exceptions cause an interrupt, the CPU internally generates an error code that is pushed onto the stack (such as the exception interrupt INT8 and INT10 -- INT14, as shown in Figure 8-2 (b)), while the other interrupts do not carry this error code (for example, a divide-by-zero error and a boundary check error), therefore, the asm.s divides the interrupt into two types according to whether or not the error code is carried. But the process is the same as without the error code. The processing of an interrupt caused by a hardware exception is shown in Figure 8-3.


Exception Processing
Push all regs on stack
Push the error code
Push Int ret address
All segment registers are set to kernel segment
Call the relevant C handler
Pop out error code and interrupt return address
Pop all registers
Int Ret

Note 1: The selector value of the kernel code segment is 0x08;
Note 2: Use 0 when there is no error code;
Note 3: The called C function is implemented in traps.c. The error code and interrupt return address pushed onto the stack are used as parameters of C functions.

Figure 8-3 Hardware exception (fault, trap) processing flow

8.1.2 System-call handling

In Linux, the application needs to use the interrupt INT 0x80 when using system resources, and the function call number needs to be placed in the register eax. If you need to pass parameters to the interrupt handler, you can use the registers ebx, ecx, and edx to store the parameters. Therefore the interrupt call is called a system call (syscall). The relevant files for implementing system calls include sys_call.s, fork.c, signal.c, sys.c, and exit.c files.

The sys_call.s acts like the asm.s program in hardware interrupt handling. In addition, the program handles clock interrupts and hard disk and floppy disk interrupts. The interrupt service functions in fork.c and signal.c are similar to the traps.c program, which provide C handlers for system interrupt calls. The fork.c provides two C handlers for creating processes: find_empty_process() and copy_process(). The signal.c provides a function do_signal() on the processing of the process signal, which is used during system call interrupt processing. It also includes implementations of four system calls.

The sys.c and exit.c programs implement some other sys_xxx() system call functions. These sys_xxx() functions are the handlers that are called for the corresponding system call. Some functions are implemented in assembly language, such as sys_execve(), while others are implemented in C (for example, four syscalls functions in signal.c).

Based on the simple naming conventions for these interrupt-related function's names, we can understand this: Usually the C functions starting with 'do_' is either a function common to the system calls or a function specific to a system call; , and the functions starting with 'sys_' is usually a special handler for the specified system call. For example, do_signal() is basically a function that all system calls must execute, while sys_pause() and sys_execve() are C-processor functions specific to a system call.

8.1.3 Other general-purpose programs

These programs include schedule.c, mktime.c, panic.c, printk.c, and vsprintf.c.

Schedule.c contains the most frequently used functions schedule(), sleep_on(), and wakeup(). It is the kernel's core scheduler, which is used to switch the execution of a process or change the execution state of a process. It also includes functions for system clock interrupts and floppy drive timing. The mktime.c contains only one time function mktime() used by the kernel, which is called only once in init/main.c. panic.c includes a panic() function that displays an error message and stops the kernel when an error occurs in kernel. printk.c and vsprintf.c are kernel supporting programs that implement the kernel-specific display function printk() and the string-formatted output function vsprintf().

8.2 asm.s

8.2.1 Function description

The asm.s assembly file contains the underlying code of handler procedures for most of the exceptions detected by the CPU, as well as the exception handling code for the math coprocessor (FPU). This program has a close relationship with traps.c. The main processing method of this program is to call the corresponding C function program in traps.c in the interrupt handler, display the error location and error code, and then exit the interrupt.

It is helpful to refer to the kernel stack change diagram for the current task in Figure 8-4 when reading this code, where each row represents 4 bytes. For the interrupt process without an error code, refer to Figure 8 4(a) for the changes of the stack pointer position. The stack pointer esp refers to the interrupt return address (esp0 in the figure) before starting the execution of the corresponding interrupt service routine. When the do_divide_error() or other C function address is pushed onto the stack, the pointer position is at esp1. At this point, the program uses the swap instruction to put the address of the function into the eax register, and the original eax value is saved to the stack. After the program puts some registers onto the stack, the stack pointer position is at esp2. Before the formal call to do_divide_error(), the program will push the address of the original eip onto the stack (that is, placed at location esp3) when the interrupt handler begins executing, and by plus 8 to go back to the location esp2 before pop out all registers

For the interrupt that the CPU generates an error code, refer to Figure 8-4(b) for the changes of the stack pointer position. The stack pointer points to esp0 in the figure just before the interrupt service routine is executed. After the do_double_fault() or other C function address to be called is pushed onto the stack, the stack pointer location is at esp1. At this time, the program saves the values of the eax and ebx registers at the positions of esp0 and esp1 by using two exchange instructions, and exchanges the error code into the eax registers; the function address is swapped into ebx register. Subsequent processing is the same as in Figure 8-4(a) above.

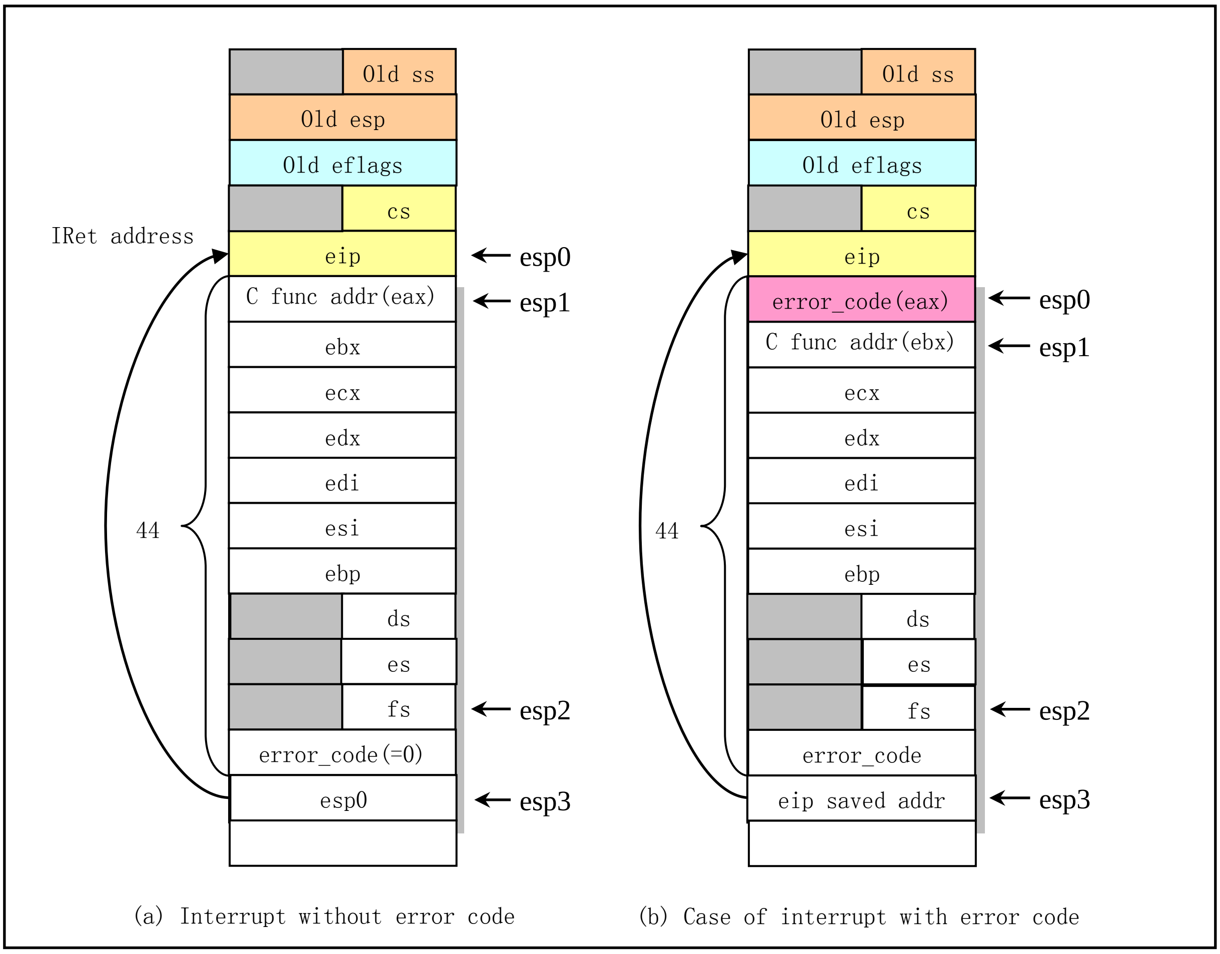


Figure 8-4 Schematic diagram of kernel stack changes during error handling

The reason for putting the error code and esp0 on the stack before the formal invocation to do_divide_error() is to use error code and esp0 as parameters do_divide_error().In traps.c, the function signature is:

```
    void do_divide_error(long esp, long error_code)
```

Therefore, the position and error code of the error can be printed in this C function. The processing of the remaining exceptions in the file is basically similar to the process described here.

8.2.2 Code Annotation

Program 8-1 linux/kernel/asm.s

```
 1 /*
 2  *  linux/kernel/asm.s
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /*
 8  * asm.s contains the low-level code for most hardware faults.
 9  * page_exception is handled by the mm, so that isn't here. This
10  * file also handles (hopefully) fpu-exceptions due to TS-bit, as
11  * the fpu must be properly saved/resored. This hasn't been tested.
12  */
   # TS - Task Switched, bit-3 in CR0. refer to section 4.1.3
13
   # This file mainly deals with the processing of Intel reserved interrupts INT0 -- INT16.
   # The following are some global function declarations, actually in traps.c.
14 .globl _divide_error,_debug,_nmi,_int3,_overflow,_bounds,_invalid_op
15 .globl _double_fault,_coprocessor_segment_overrun
16 .globl _invalid_TSS,_segment_not_present,_stack_segment
17 .globl _general_protection,_coprocessor_error,_irq13,_reserved
18 .globl _alignment_check
19
   # The following procedure deals with exceptions without error codes.
   # Int0 -- Handles errors of divided by zero.      Type: error;    Error code: None.
   # When executing a DIV or IDIV instruction, if the divisor is 0, the CPU will generate
   # this exception. This exception is also generated when EAX (or AX, AL) does not hold the
   # result of a legal divide operation. The label '_do_divide_error' on line 21 is actually
   # the corresponding name in the object compiled from C function do_divide_error(). This
   # function is in traps.c, line 101.
20 _divide_error:
21         pushl $_do_divide_error    # first push the C function address.
22 no_error_code:                     # no error code processing entry.
23         xchgl %eax,(%esp)          # _do_divide_error in stack exchanged with EAX
24         pushl %ebx
25         pushl %ecx
26         pushl %edx
27         pushl %edi
28         pushl %esi
29         pushl %ebp
30         push %ds                   # occupied 4 bytes in stack.
31         push %es
32         push %fs
33         pushl $0                   # "error code"       # 0 as error code
34         lea 44(%esp),%edx          # get effective addr in esp to edx, points to the
35         pushl %edx                 # ip of original interrupted code (at esp0).
36         movl $0x10,%edx            # Initialize ds, es, fs to kernel data seg selector.
37         mov %dx,%ds
38         mov %dx,%es
```

```
39        mov %dx,%fs
   # The ’*’ in the following statement indicates the register value as an address. This
   # statement represents invocation to a routine whos address is specified by the operand.
   # This is an indirect call. The meaning of this sentence is to call the C routine
   # specified, such as do_divide_error(). After the C function returns, a number 8 is added
   # to the stack pointer (line 41), which is equivalent to executing twice pop instructions,
   # used to discard the C routine’s two parameters (pushed on stack at line 33 and 35), and
   # let the stack pointer esp point back to the location of register fs.
40        call *%eax                 # like: do_divide_error(long esp, long error_code)
41        addl $8,%esp
42        pop %fs
43        pop %es
44        pop %ds
45        popl %ebp
46        popl %esi
47        popl %edi
48        popl %edx
49        popl %ecx
50        popl %ebx
51        popl %eax                  # recover original EAX
52        iret
53
   # Int1 -- debug interrupt entry point.   Type: Error/Trap (Fault/Trap); No error code.
   # This exception is raised when the TF flag in EFLAGS is set. The CPU generates this
   # exception when a hardware breakpoint is found, or an instruction trace trap or task
   # swap trap is turned on, or debug register access is invalid (error).
54 _debug:
55        pushl $_do_int3            # _do_debug  # push C routine address
56        jmp no_error_code          # line 22.
57
   # Int2 - Non Maskable Interrupt entry point.   Type: Trap; No error code.
   # This is the only hardware interrupt that is given a fixed interrupt vector. Whenever an
   # NMI signal is received, the CPU internally generates an interrupt vector 2 and performs
   # a standard interrupt acknowledge cycle, thus saving time. NMI is usually reserved for
   # use with extremely important hardware events. When the CPU receives an NMI signal and
   # begins executing its interrupt handler, all subsequent hardware interrupts are ignored.
58 _nmi:
59        pushl $_do_nmi
60        jmp no_error_code
61
   # Int3 - The breakpoint instruction entry point. Type: Trap; No error code.
   # The interrupt caused by the int 3 instruction is independent of the hardware interrupt.
   # This instruction is usually inserted into the code by the debuger, and the processing
   # method is same as _debug.
62 _int3:
63        pushl $_do_int3
64        jmp no_error_code
65
   # Int4 -- Overflow error interrupt entry point.   Type: Trap; no error code.
   # This interrupt is raised by the CPU executing the INTO instruction when the OF flag is
   # set in EFLAGS. Usually used by compilers to track arithmetic calculation overflows.
66 _overflow:
67        pushl $_do_overflow
```

```
68         jmp no_error_code
69
   # Int5 - Bounds check error interrupt entry point.   Type: Error; No error code.
   # An interrupt that is raised when the operand is outside the valid range. This interrupt
   # is generated when the BOUND instruction fails. The BOUND instruction has 3 operands. If
   # the first one is not between the other two, an exception of 5 is generated.
70 _bounds:
71         pushl $_do_bounds
72         jmp no_error_code
73
   # Int6 -- Invalid opcode interrupt entry point.   Type: Error; no error code.
   # The interrupt caused by the CPU actuator detecting an invalid opcode.
74 _invalid_op:
75         pushl $_do_invalid_op
76         jmp no_error_code
77
   # Int9 -- The coprocessor segment overrun entry point.  Type: Abandon; No error code.
   # This exception is basically equivalent to coprocessor error protection. Because when
   # the floating-point instruction operand is too large, we have the opportunity to load or
   # save a floating-point value that exceeds the data segment.
78 _coprocessor_segment_overrun:
79         pushl $_do_coprocessor_segment_overrun
80         jmp no_error_code
81
   # Int15 -- The entry point for other interrupts reserved by Intel.
82 _reserved:
83         pushl $_do_reserved
84         jmp no_error_code
85
   # Int45 -- (0x20 + 13) Math coprocessor hardware interrupts set by the Linux kernel.
   # When the coprocessor performs an operation, it will issue an IRQ13 interrupt signal to
   # inform CPU that the operation is complete. When the 80387 is performing calculations,
   # CPU waits for its operation to complete. On line 89 below, 0xF0 is the co-processing
   # port used to clear the latch. By writing to this port, this interrupt will eliminate the
   # CPU's BUSY continuation signal and reactivate the 80387 processor extension request pin
   # PEREQ. This operation is mainly to ensure that the CPU responds to this interrupt before
   # continuing to execute any instruction of 80387.
86 _irq13:
87         pushl %eax
88         xorb %al,%al
89         outb %al,$0xF0
90         movb $0x20,%al
91         outb %al,$0x20            # sent EOI (End of Interrupt) to 8259's master chip.
92         jmp 1f                    # delay a while.
93 1:      jmp 1f
94 1:      outb %al,$0xA0            # sent EOI to 8259's slave chip.
95         popl %eax
96         jmp _coprocessor_error    # code in system_call.s
97
   # When the following interrupt is called, the CPU will push the error code onto the stack
   # after interrupting the return address, so the error code will also need to be popped up
   # when returning (see Figure 5.3(b)).
# Int8 -- double fault. Type: Abandon; There is an error code.
# Usually when the CPU invokes a exception handler and detects a new exception, the two
# exceptions can be handled serially. However, there are few situations in which the CPU
# cannot handle them serially, and the double fault exception is triggered at this time.
_double_fault:
        pushl $_do_double_fault    # addr of C routine pushed onto stack.
error_code:
        xchgl %eax,4(%esp)         # error code <-> %eax, original eax is in stored stack.
        xchgl %ebx,(%esp)          # &function <-> %ebx, original ebx is stored in stack.
        pushl %ecx
        pushl %edx
        pushl %edi
        pushl %esi
        pushl %ebp
        push %ds
        push %es
        push %fs
        pushl %eax                 # error code
        lea 44(%esp),%eax          # offset
        pushl %eax
        movl $0x10,%eax            # set kernel data segment selector.
        mov %ax,%ds
        mov %ax,%es
        mov %ax,%fs
        call *%ebx                 # indirect invocation to C routine
        addl $8,%esp               # discard used parameters.
        pop %fs
        pop %es
        pop %ds
        popl %ebp
        popl %esi
        popl %edi
        popl %edx
        popl %ecx
        popl %ebx
        popl %eax
        iret

# Int10 -- Invalid task status segment (TSS).   Type: Error; there is an error code.
# The CPU attempts to switch to a process, and the TSS of the process is invalid.
# Depending on which part of the TSS caused an exception, when the TSS length exceeds 104
# bytes, this exception is generated in the current task, and the handover is terminated.
# Other problems can cause this exception to occur in new tasks after the switch.
_invalid_TSS:
        pushl $_do_invalid_TSS
        jmp error_code

# Int11 -- The segment does not present. Type: Error; there is an error code.
# The referenced segment is not in memory. The flag in the segment descriptor indicates
# that the segment is not in memory.
_segment_not_present:
        pushl $_do_segment_not_present
        jmp error_code
```

```

# Int12 -- Stack exception. Type: Error; there is an error code.
# The instruction operation attempted to exceed the stack segment range, or the stack
# segment is not in memory. This is a special case of exceptions 11 and 13. Some operating
# systems can use this exception to determine when more stack space should be allocated
# for the program.
_stack_segment:
        pushl $_do_stack_segment
        jmp error_code

# Int13 -- General protection exception. Type: Error; there is an error code.
# Indicates a protection violation that does not belong to any other class. If an exception
# is generated without a corresponding exception vector (0-16), it will usually fall back
# to this class.
_general_protection:
        pushl $_do_general_protection
        jmp error_code

# Int17 -- Boundary alignment check error.
# This exception is raised when privilege level 3 (user-level) data is non-boundically
# aligned when memory boundary checking is enabled.
_alignment_check:
        pushl $_do_alignment_check
        jmp error_code

# int7  -- _device_not_available in file kernel/sys_call.s, line 158.
# int14 -- _page_fault in file mm/page.s, line 14.
# int16 -- _coprocessor_error in file kernel/sys_call.s, line 140.
# int 0x20 -- _timer_interrupt in file kernel/sys_call.s, line 189.
# int 0x80 -- _system_call in file kernel/sys_call.s, line 84.
```

8.2.3 Information

8.2.3.1 Intel reserved interrupt vector definition

Here is a summary of the Intel reserved interrupt vector, as shown in Table 8–1.

Table 8-1 Exceptions and Interrupts reserved by Intel Co.

Vecter No	Name	Type	Error Code	Signal	Source
0	Devide error	Fault (Error)	No	SIGFPE	DIV and IDIV instructions.
1	Debug	Fault/ Trap	No	SIGTRAP	Any code or data reference or the INT instruction.
2	nmi	Interrupt	No		Non maskable external interrupt.
3	Breakpoint	Trap	No	SIGTRAP	INT 3 instruction.
4	Overflow	Trap	No	SIGSEGV	INTO instruction.
5	Bounds check	Fault	No	SIGSEGV	BOUND instruction.
6	Invalid Opcode	Fault	No	SIGILL	UD2 instruction or reserved opcode.
7	Device not available	Fault	No	SIGSEGV	Floating-point or WAIT/FWAIT instruction.

8	Double fault	Abort	Yes(0)	SIGSEGV	Any instruction that can generate an exception, NMI, or an INTR.
9	Coprocessor seg overrun	Abort	No	SIGFPE	Floating-point instruction.
10	Invalid TSS	Fault	Yes	SIGSEGV	Task switch or TSS access.
11	Segment not present	Fault	Yes	SIGBUS	Loading segment registers or accessing system segments.
12	Stack segment	Fault	Yes	SIGBUS	Stack operations and SS register loads.
13	General protection	Fault	Yes	SIGSEGV	Any memory reference and other protection checks.
14	Page fault	Fault	Yes	SIGSEGV	Any memory reference.
15	Intel reserved		No		
16	Coprocessor error	Fault	No	SIGFPE	Floating-point or WAIT/FWAIT
17	Alignment check	Fault	Yes(0)		Any data reference in memory.
20-31	Intel reserved.				
32-255	User Defined interrupts	Interrupt			External interrupt or INT n instruction.

8.3 traps.c

8.3.1 Function Description

The traps.c program mainly includes some C functions used in exception handling, which are used to be called by the exception handling low-level code asm.s. to display debugging message such as error location and error code. The die() generic function is used to display detailed error information in interrupt processing. The final initialization function trap_init() of the program is called in the previous init/main.c to initialize the hardware exception handling interrupt vector (trap gate) and enable the interrupt request signal to arrive. Please refer to the previous asm.s file when reading this program.

From the beginning of this program, we will encounter many assembly statements embedded in C language programs. See Section 3.3.2 for the basic syntax of embedded assembly statements.

8.3.2 Code Annotation

Program 8-2 linux/kernel/traps.c

```
/*
 *  linux/kernel/traps.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 'Traps.c' handles hardware traps and faults after we have saved some
 * state in 'asm.s'. Currently mostly a debugging-aid, will be extended
 * to mainly kill the offending process (probably by giving it a signal,
 * but possibly by killing it outright if necessary).
 */
// <string.h> header file. Mainly defines some embedded functions about string operations.
```

```
   // <linux/head.h> Head header file. A simple structure for the segment descriptor is
   //     defined, along with several selector constants.
   // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
   //     data of the initial task 0, and some embedded assembly function macro statements
   //     about the descriptor parameter settings and acquisition.
   // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
   //     commonly used functions of the kernel.
   // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
   //     descriptors/interrupt gates, etc. is defined.
   // <asm/segment.h> Segment operation header file. An embedded assembly function is defined
   //     for segment register operations.
   // <asm/io.h> Io header file. Defines the function that operates on the io port in the form
   //     of a macro's embedded assembler.
13 #include <string.h>
14
15 #include <linux/head.h>
16 #include <linux/sched.h>
17 #include <linux/kernel.h>
18 #include <asm/system.h>
19 #include <asm/segment.h>
20 #include <asm/io.h>
21
   // The following statements define three embedded assembly macro functions.
   // A compond statement enclosed in parentheses (statements in curly braces) can be used as
   // an expression, with the last __res being its output value. Line 23 defines a register
   // variable __res. This variable will be saved in a register for quick access and operation.
   // If you want to specify a register (such as eax), then we can write the sentence as
   // "register char __res asm("ax");". Refer to section 3.3.2 for detailed descriptions.
   //
   // Function: Take a byte at address addr in segment seg.
   // Parameters: seg - the segment selector; addr - the specified address in the segment.
   // Output: %0 - eax (__res);   Input: %1 - eax (seg); %2 - memory address (*(addr)).
22 #define get_seg_byte(seg,addr) ({ \
23 register char __res; \
24 __asm__("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs" \
25         :"=a" (__res):"0" (seg),"m" (*(addr))); \
26 __res;})
27
   // Function: Take a long word (4 bytes) at the address addr in the segment seg.
   // Parameters: seg - the segment selector; addr - the specified address in the segment.
   // Output: %0 - eax (__res); Input: %1 - eax (seg); %2 - memory address (*(addr)).
28 #define get_seg_long(seg,addr) ({ \
29 register unsigned long __res; \
30 __asm__("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
31         :"=a" (__res):"0" (seg),"m" (*(addr))); \
32 __res;})
33
   // Function: Take the value (selector) of the fs segment register .
   // Output: %0 - eax (__res).
34 #define _fs() ({ \
35 register unsigned short __res; \
36 __asm__("mov %%fs,%%ax":"=a" (__res):); \
37 __res;})
```

```
// Some function prototypes are defined below.
void page_exception(void);                      // page_fault (mm/page.s,14).

void divide_error(void);                        // Int0 (kernel/asm.s, 20).
void debug(void);                               // int1 (kernel/asm.s, 54).
void nmi(void);                                 // int2 (kernel/asm.s, 58).
void int3(void);                                // int3 (kernel/asm.s, 62).
void overflow(void);                            // int4 (kernel/asm.s, 66).
void bounds(void);                              // int5 (kernel/asm.s, 70).
void invalid_op(void);                          // int6 (kernel/asm.s, 74).
void device_not_available(void);                // int7 (kernel/sys_call.s, 158).
void double_fault(void);                        // int8 (kernel/asm.s, 98).
void coprocessor_segment_overrun(void);         // int9 (kernel/asm.s, 78).
void invalid_TSS(void);                         // int10 (kernel/asm.s, 132).
void segment_not_present(void);                 // int11 (kernel/asm.s, 136).
void stack_segment(void);                       // int12 (kernel/asm.s, 140).
void general_protection(void);                  // int13 (kernel/asm.s, 144).
void page_fault(void);                          // int14 (mm/page.s, 14).
void coprocessor_error(void);                   // int16 (kernel/sys_call.s,140).
void reserved(void);                            // int15 (kernel/asm.s, 82).
void parallel_interrupt(void);                  // int39 (kernel/sys_call.s, 295).
void irq13(void);                          // int45 (kernel/asm.s,86) Coprocessor handling.
void alignment_check(void);                     // int46 (kernel/asm.s, 148).

// This subroutine is used to print the error name, error code, program's CS:EIP, EFLAGS,
// ESP, fs segment, segment messages, process pid, task no, and 10-byte instruction code.
// If the stack is in user data segment, then 16 bytes of stack content is also printed
// out. These information can be used for debugging.
// Parameter:
// str - Error name string pointer;
// esp_ptr - Pointer to info of the interrupted program on stack (see esp0 in Figure 8-4);
// nr - Error code. For exception with no error code, this parameter is always 0.
static void die(char * str,long esp_ptr,long nr)
{
        long * esp = (long *) esp_ptr;
        int i;

        printk("%s: %04x\n\r",str,nr&0xffff);
// The next statement prints CS:EIP, EFLAGS, and SS:ESP for the currently calling process.
// As can be seen from Figure8-4, here esp[0] is esp0 in the figure. So we take this
// statement apart and look at it as:
// (1) EIP:\t%04x:%p\n  -- esp[1] segment selector(CS), esp[0] is EIP;
// (2) EFLAGS:\t%p      -- esp[2] is EFLAGS;
// (3) ESP:\t%04x:%p\n  -- esp[4] is SS, esp[3] is ESP
        printk("EIP:\t%04x:%p\nEFLAGS:\t%p\nESP:\t%04x:%p\n",
                esp[1],esp[0],esp[2],esp[4],esp[3]);
        printk("fs: %04x\n",_fs());
        printk("base: %p, limit: %p\n",get_base(current->ldt[1]),get_limit(0x17));
        if (esp[4] == 0x17) {              // if SS == 0x17, means in user segment,
                printk("Stack: ");         // print 16 bytes data in user stack too.
                for (i=0;i<4;i++)
                        printk("%p ",get_seg_long(0x17,i+(long *)esp[3]));
                printk("\n");
        }
        str(i);                                 // get task no.(include/linux/sched.h, 210)
        printk("Pid: %d, process nr: %d\n\r",current->pid,0xffff & i);
        for(i=0;i<10;i++)
                printk("%02x ",0xff & get_seg_byte(esp[1],(i+(char *)esp[0])));
        printk("\n\r");
        do_exit(11);            /* play segment exception */
}

// The following functions are called by the interrupt handler in asm.s.
void do_double_fault(long esp, long error_code)
{
        die("double fault",esp,error_code);
}

void do_general_protection(long esp, long error_code)
{
        die("general protection",esp,error_code);
}

void do_alignment_check(long esp, long error_code)
{
    die("alignment check",esp,error_code);
}

void do_divide_error(long esp, long error_code)
{
        die("divide error",esp,error_code);
}

// These parameters are the register values that are sequentially pushed onto the stack
// after entering the interrupt. See lines 24-35 in asm.s file.
void do_int3(long * esp, long error_code,
                long fs,long es,long ds,
                long ebp,long esi,long edi,
                long edx,long ecx,long ebx,long eax)
{
        int tr;

        __asm__("str %%ax":"=a" (tr):"" (0));           // get task register TR -> tr
        printk("eax\t\tebx\t\tecx\t\tedx\n\r%8x\t%8x\t%8x\t%8x\n\r",
                eax,ebx,ecx,edx);
        printk("esi\t\tedi\t\tebp\t\tesp\n\r%8x\t%8x\t%8x\t%8x\n\r",
                esi,edi,ebp,(long) esp);
        printk("\n\rds\tes\tfs\ttr\n\r%4x\t%4x\t%4x\t%4x\n\r",
                ds,es,fs,tr);
        printk("EIP: %8x   CS: %4x  EFLAGS: %8x\n\r",esp[0],esp[1],esp[2]);
}

void do_nmi(long esp, long error_code)
{
        die("nmi",esp,error_code);
```

```c
}

void do_debug(long esp, long error_code)
{
        die("debug",esp,error_code);
}

void do_overflow(long esp, long error_code)
{
        die("overflow",esp,error_code);
}

void do_bounds(long esp, long error_code)
{
        die("bounds",esp,error_code);
}

void do_invalid_op(long esp, long error_code)
{
        die("invalid operand",esp,error_code);
}

void do_device_not_available(long esp, long error_code)
{
        die("device not available",esp,error_code);
}

void do_coprocessor_segment_overrun(long esp, long error_code)
{
        die("coprocessor segment overrun",esp,error_code);
}

void do_invalid_TSS(long esp,long error_code)
{
        die("invalid TSS",esp,error_code);
}

void do_segment_not_present(long esp,long error_code)
{
        die("segment not present",esp,error_code);
}

void do_stack_segment(long esp,long error_code)
{
        die("stack segment",esp,error_code);
}

void do_coprocessor_error(long esp, long error_code)
{
        if (last_task_used_math != current)
                return;
        die("coprocessor error",esp,error_code);
}
```

```
179
180 void do_reserved(long esp, long error_code)
181 {
182         die("reserved (15,17-47) error",esp,error_code);
183 }
184
    // The following are exception (trap) initializers for setting their interrupt call gates
    // (vectors) separately. Both set_trap_gate() and set_system_gate() use the Trap Gate in
    // the interrupt descriptor table IDT. The main difference between them is that the former
    // sets the privilege level to 0 and the latter to 3. Therefore, breakpoint trap int3,
    // overflow interrupt, and bounds error interrupt can be called by any program. Both of
    // these functions are embedded assembly macros, see include/asm/system.h, lines 36, 39.
185 void trap_init(void)
186 {
187         int i;
188
189         set_trap_gate(0,&divide_error);
190         set_trap_gate(1,&debug);
191         set_trap_gate(2,&nmi);
192         set_system_gate(3,&int3);              /* int3-5 can be called from all */
193         set_system_gate(4,&overflow);
194         set_system_gate(5,&bounds);
195         set_trap_gate(6,&invalid_op);
196         set_trap_gate(7,&device_not_available);
197         set_trap_gate(8,&double_fault);
198         set_trap_gate(9,&coprocessor_segment_overrun);
199         set_trap_gate(10,&invalid_TSS);
200         set_trap_gate(11,&segment_not_present);
201         set_trap_gate(12,&stack_segment);
202         set_trap_gate(13,&general_protection);
203         set_trap_gate(14,&page_fault);
204         set_trap_gate(15,&reserved);
205         set_trap_gate(16,&coprocessor_error);
206         set_trap_gate(17,&alignment_check);

    // The trap gates of int17--int47 are all set to reserved first, and they will be re-set
    // after each hardware initialization.
207         for (i=18;i<48;i++)
208                 set_trap_gate(i,&reserved);

    // Set the coprocessor int45 (0x20+13) trap gate descriptor below and allow it to generate
    // an interrupt request. The coprocessor IRQ13 is connected to the 8259 slave chip's IR5 pin.
    // Lines 210-211 are used to allow the coprocessor to send an interrupt request signal. In
    // addition, the gate descriptor of the int39 (0x20+7) of the parallel port 1 is also set
    // here, and the interrupt request number IRQ7 is connected to the IR7 pin of the 8259
    // main chip.Refer to figure 2-6.
    //
    // The line 210 statement sends an operation command word OCW1 to 8259. This command is
    // used to set the 8259 Interrupt Mask Register IMR. 0x21 is the main chip port. The mask
    // code is read from it and written immediately after AND 0xfb (0b11111011), indicating
    // that the interrupt request mask bit M2 corresponding to the interrupt request IR2 is
    // cleared. As shown in figure 2-6, the request pin INT of the slave chip is connected to
    // the IR2 pin of the master chip, so the statement indicates that the interrupt request
    // signal from the slave chip is enabled.
    // similarly, the line 211 statement performs a similar operation for the slave chip. 0xA1
    // is the slave chip port. Read the mask code from it and write it immediately after AND
    // 0xdf (0b11011111), indicating that the interrupt mask bit M2 for the IR5 on slave chip
    // is cleared. Since the coprocessor is connected to the IR5 pin, this statement enables
    // the coprocessor to send interrupt request signal IRQ13.
209         set_trap_gate(45,&irq13);
210         outb_p(inb_p(0x21)&0xfb,0x21);          // enable IRQ2 of master chip.
211         outb(inb_p(0xA1)&0xdf,0xA1);            // enable IRQ13 of slave chip.
212         set_trap_gate(39,&parallel_interrupt);  // set parallel 1 gate
213 }
214
```

8.4 sys_call.s

Linux uses the interrupt invocation method to implement the access interface between the user and the kernel resources. The sys_call.s program mainly implements the system call INT 0x80 entry processing and signal detection processing, and gives the underlying interfaces of the two system functions, namely sys_execve and sys_fork. Interrupt handlers for coprocessor errors (INT 16), device not exist (INT7), clock interrupt (INT32), hard disk interrupt (INT46), floppy disk interrupt (INT38) are also listed.

8.4.1 Function descriptions

In Linux 0.12, application programs use INT 0x80 and function number in register EAX to use various services provided by the kernel, and these services are called system call (syscall) services. Usually users do not use the system call service directly, but use it through the interface functions provided in general libraries (such as libc). For example, the system call fork that creates the process can directly use the function fork() in the library. The INT 0x80 invocation is executed in this function and the result is returned to the user program.

In the kernel, the C function implementation code for all system call services is distributed throughout the kernel. The kernel sequentially arranges them into a function pointer (address) table according to the system call function number, and then calls the corresponding system service function during INT 0x80 processing.

In addition, this source file also includes the entry processing code of several other interrupt calls. The implementation process and steps of these interrupt entry codes are basically the same. For soft interrupts (system_call, coprocessor_error, device_not_available), the processing can be basically divided into the following two steps: First prepare for calling the corresponding C function handler, pushing some parameters onto the stack. The system call can take up to 3 parameters, which are passed in via the registers EBX, ECX and EDX. Then call the C function to process the corresponding function. After the processing returns, the signal bitmap of the current task is detected, and a signal with the smallest value (highest priority) is processed and the signal in the signal bitmap is reset.

For the interrupt sent by the hardware IRQ, the processing first sends an end of interrupt instruction EOI to the interrupt control chip 8259A, then calls the corresponding C function program. For the clock interrupt, the signal bitmap of the current task is also detected.

8.4.1.1 Interrupt Service Entry Processing

For the interrupt handling of the system call (int 0x80), the program can be thought of as an "interface". In fact, the processing of each system call is basically done by calling the corresponding C function. The entire process of the system call is shown in Figure 8-5.

This program will first check if the syscall function number in EAX is valid (within a given range), and then save some of the registers that will be used onto the stack. By default, the Linux kernel uses the segment registers DS, ES for kernel data segments and FS for user data segments. Then call the C function of the corresponding syscall through the above address jump table (sys_call_table). After the C function returns, the program pushes the return value onto the stack and saves it.

Next, the program looks at the status of the process that performed this invocation. If the process state changes from the running state to another state due to the operation of the above C function or other conditions, or because the time slice has run out (counter==0), then the process scheduling function schedule() ('jmp _schedule') is called. . Since the return address 'ret_from_sys_call' has been pushed onto the stack before executing "jmp _schedule", it will eventually return to 'ret_from_sys_call' and continue execution after end of schedule().

The code starting at the 'ret_from_sys_call' label performs some post-processing. The main operation is to determine whether the current process is the initial process 0, and if so, directly exit the system call, and the interrupt returns. Otherwise, according to the code segment descriptor and the stack used, it is judged whether the process is a normal process, if not, then it is a kernel process (for example, initial process 1) or the like. The stack content is also immediately popped out and the syscall interrupt is exited. A piece of code at the end is used to handle the signal of the process. If the signal bitmap of the process structure indicates that the process has received a signal, then the signal handler do_signal() is called.

Finally, the program restores the contents of the saved registers, exits the interrupt processing and returns to the interrupted program. If there is a signal, the program will first "return" to the corresponding signal processing function to execute, and then return to the program that calls system_call.

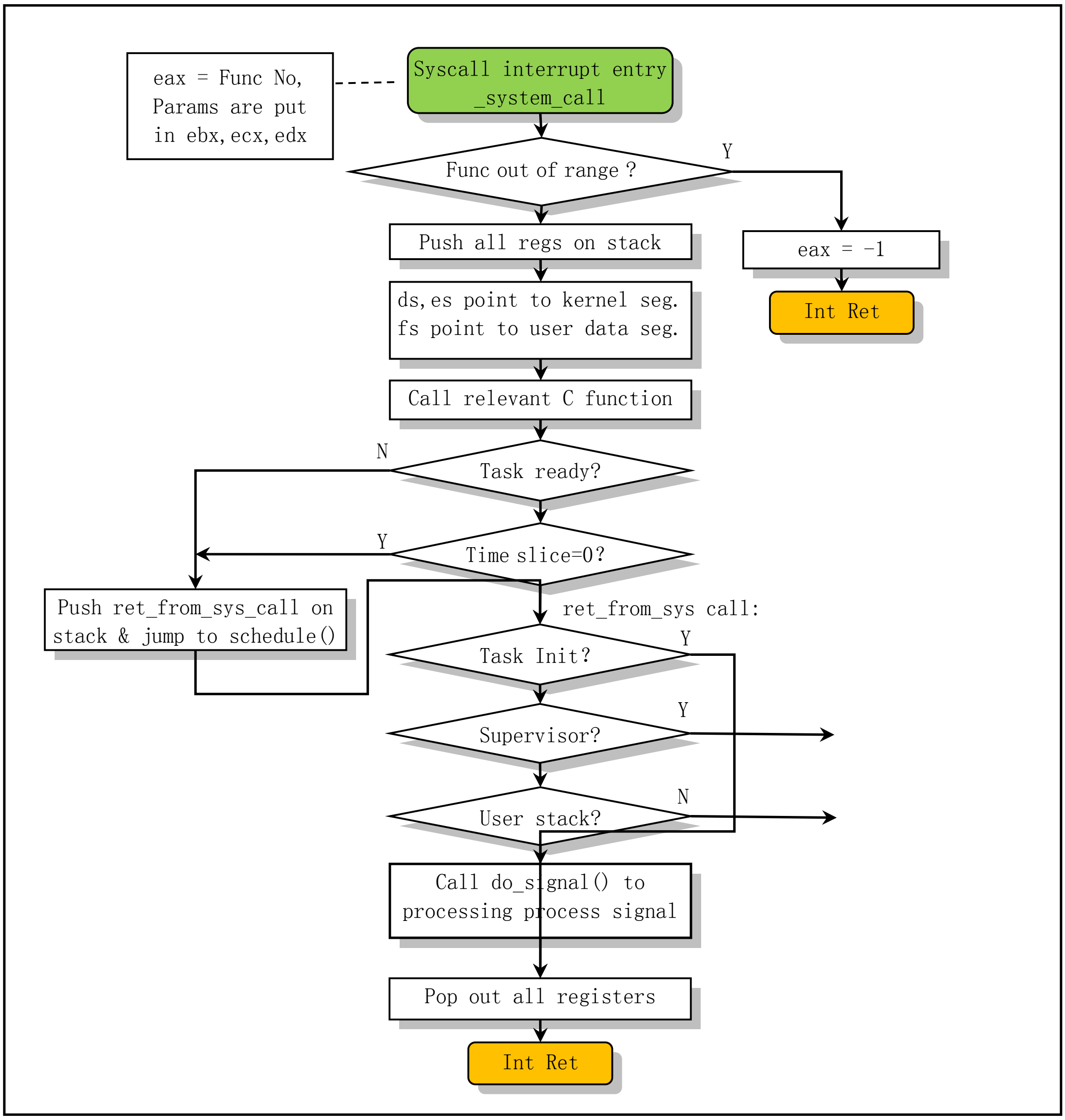


Figure 8-5 System call processing flow diagram

8.4.1.2 Syscall parameter passing method

Regarding the parameter transfer issue in the system call INT 0x80, the Linux kernel uses several general-purpose registers as a channel for parameter passing. In the Linux 0.12 system, the program uses the registers EBX, ECX, and EDX to pass parameters, and can pass three parameters directly to the system call service procedure (not including syscall function number in the EAX register). If a pointer to a user-space data block is used, the user program can pass more data information to the system call procedure.

As mentioned above, during the system call processing, the segment registers DS and ES point to the kernel data space and the FS is set to user data space. Therefore, in the actual data block transfer procedure, the Linux kernel can use the FS register to perform data copying between the kernel data space and the user data space, and the kernel program does not need to perform any check operation on the data boundary range during the copying process. The boundary check is done automatically by CPU. The actual data transfer work in the kernel can be done using functions such as get_fs_byte() and put_fs_byte(), see the implementations for these functions in the include/asm/segment.h file.

This method of using registers to pass parameters has a distinct advantage. That is, when the system interrupt service routine is entered, the registers that pass the parameters are also automatically placed on the kernel state stack, and when the process exits from the interrupt call, the kernel state stack is also popped, so the kernel does not have to specialize them. This method is the simplest and fastest method of parameter transfer that Mr. Linus knew at the time.

8.4.2 Code Annotation

Program 8-3 linux/kernel/sys_call.s

```
/*
 *  linux/kernel/system_call.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  system_call.s  contains the system-call low-level handling routines.
 * This also contains the timer-interrupt handler, as some of the code is
 * the same. The hd- and flopppy-interrupts are also here.
 *
 * NOTE: This code handles signal-recognition, which happens every time
 * after a timer-interrupt and after each system call. Ordinary interrupts
 * don't handle signal-recognition, as that would clutter them up totally
 * unnecessarily.
 *
 * Stack layout in 'ret_from_system_call':
 *
 *       0(%esp) - %eax
 *       4(%esp) - %ebx
 *       8(%esp) - %ecx
 *       C(%esp) - %edx
 *      10(%esp) - original %eax        (-1 if not system call)
 *      14(%esp) - %fs
 *      18(%esp) - %es
 *      1C(%esp) - %ds
 *      20(%esp) - %eip
 *      24(%esp) - %cs
 *      28(%esp) - %eflags
 *      2C(%esp) - %oldesp
 *      30(%esp) - %oldss
 */
# The general interrupt procedure in the original Linus comment above refers to interrupts
# other than the system-call (int 0x80) and clock interrupt (int 0x20). These interrupts
# occur randomly in the kernel state or user state. If the signal recognition is also
# handled during these interrupts, it may conflict with the process of identifying the
# signal during the system-call and clock interrupt. This violates the non-preemptive
# principle of kernel code. It is therefore not necessary for the system to process the
# signals in these "other" interrupts, nor to do so.

SIG_CHLD        = 17                # signal SIG_CHLD (child stop or end).

EAX             = 0x00              # offset of each register in the stack.
```

```
37 EBX            = 0x04
38 ECX            = 0x08
39 EDX            = 0x0C
40 ORIG_EAX       = 0x10            # If not a syscall (other interrupts), the value is -1
41 FS             = 0x14
42 ES             = 0x18
43 DS             = 0x1C
44 EIP            = 0x20            # Line 44-48 is automatically pushed onto stack by CPU.
45 CS             = 0x24
46 EFLAGS         = 0x28
47 OLDESP         = 0x2C            # old SS:ESP is also pushed when privilege level changed
48 OLDSS          = 0x30
49
   # To facilitate access to the data structure in assembly, the offsets of the fields in
   # structure of task and signal are given here.
   # These are the field offsets in the task_struct, see include/linux/sched.h, line 105.
50 state   = 0          # these are offsets into the task-struct.
51 counter = 4          # task runtime counts (ticks), time slice.
52 priority = 8         # counter=priority when task starts running, the longer it runs.
53 signal  = 12         # signal bitmap, signal = bit offset + 1
54 sigaction = 16       # MUST be 16 (=len of sigaction)
55 blocked = (33*16)    # blocked signal offset
56
57 # offsets within sigaction         # see include/signal.h, line 55.
58 sa_handler = 0
59 sa_mask = 4
60 sa_flags = 8
61 sa_restorer = 12                  # refer to the description of kernel/signal.c
62
63 nr_system_calls = 82              # total number of system calls in Linux 0.12.
64
65 ENOSYS = 38                       # system-call number error code
66
67 /*
68  * Ok, I get parallel printer interrupts while using the floppy for some
69  * strange reason. Urgel. Now I just ignore them.
70  */
71 .globl _system_call,_sys_fork,_timer_interrupt,_sys_execve
72 .globl _hd_interrupt,_floppy_interrupt,_parallel_interrupt
73 .globl _device_not_available, _coprocessor_error
74
   # The error code -ENOSYS will be returned if the system-call number is incorrect.
75 .align 2                          # Memory is 4 bytes aligned.
76 bad_sys_call:
77         pushl $-ENOSYS            # set -ENOSYS in eax
78         jmp ret_from_sys_call

   # scheduler re-execute entry. The scheduler starts at (kernel/sched.c, line 119).
   # When scheduler schedule() returns, it continues running from ret_from_sys_call.
79 .align 2
80 reschedule:
81         pushl $ret_from_sys_call
82         jmp _schedule
```

```
    #### Int 0x80 -- Linux system call entry point (int 0x80, call number in eax).
83  .align 2
84  _system_call:
85          push %ds                  # save original seg registers.
86          push %es
87          push %fs
88          pushl %eax                # save the orig_eax

    # A system-call can take up to 3 parameters or no parameters. The EBX, ECX, and EDX pushed
    # onto the stack are loaded with parameters of the corresponding C function (see line 99).
    # The order in which these registers are pushed is specified by GNU gcc. The first parameter
    # can be stored in EBX, the second is in ECX, and the third is in EDX.
    # The system-calls can be found in macros on lines 150--200 in file include/unistd.h.
89          pushl %edx
90          pushl %ecx                # push %ebx,%ecx,%edx as parameters
91          pushl %ebx                # to the system call

    # After saving segment registers above, here we set DS, ES to point to the kernel data
    # segment, and FS to the current local data segment of the user program that performs
    # this syscall. Note that in Linux 0.12, the code and data memory segments allocated to
    # the task is overlapped, their segment base address and limits are the same.
92          movl $0x10,%edx           # set up ds,es to kernel space
93          mov %dx,%ds
94          mov %dx,%es
95          movl $0x17,%edx           # fs points to local data space
96          mov %dx,%fs
97          cmpl _NR_syscalls,%eax    # syscall nr is valid ?
98          jae bad_sys_call

    # The meaning of the operand in the following sentence is: [_sys_call_table + %eax * 4].
    # See the explanation after the program and section 3.2.3. Sys_call_table[] is an array
    # of pointers defined in include/linux/sys.h. This array contains the addresses of all 82
    # syscall C handlers.
99          call _sys_call_table(,%eax,4)    # call C function indirectly.
100         pushl %eax                       # the return value.

    # Lines 101-106 below check the status of the current task. If it is not in the running
    # state (state is not equal to 0) then execute the scheduler. If the task is in the
    # running state, but its time slice has been used up (counter=0), then the scheduler is
    # also executed. For example, when a process in the background process group performs
    # control terminal read/write operations, all processes in the background group will
    # receive a SIGTTIN or SIGTTOU signal by default, causing all processes in the process
    # group to be in a stopped state, and the current process will return immediately.
101 2:
102         movl _current,%eax        # structure pointer -> eax
103         cmpl $0,state(%eax)       # state
104         jne reschedule
105         cmpl $0,counter(%eax)     # counter
106         je reschedule

    # The following code executes the recognition of the signal after returning from the C
    # function. When other interrupt service routines exit, they will also jump to here for
    # processing before exiting the interrupt process. For example, the processor error
    # interrupt int 16 on the following 131 lines.
    # Here, it is first determined whether the current task is the initial task0, and if so,
    # it is not necessary to perform signal processing on it and return directly. Note that
    # _task on line 109 corresponds to task[] array in the C program, and direct reference
    # to it is equivalent to referencing task[0].
107 ret_from_sys_call:
108         movl _current,%eax
109         cmpl _task,%eax             # task[0] cannot have signals
110         je 3f                       # forward jump to label 3 (line 129), exit

    # Check whether the program is a user task by checking the code segment selector of the
    # original calling program, and if not, exit the interrupt directly (task cannot be
    # preempted while in kernel mode). Otherwise, the signal of the task is checked.
    # Here, the selector is compared with 0x000f to determine whether it is a user task.
    # Value 0x000f represents the selector of the user code segment (RPL=3, local table,
    # code segment). If not, it means that an interrupt handler (such as INT 16) jumps to
    # line 107 and runs here. For this case, jump and exit the interrupt. In addition, if
    # the original stack segment selector is not 0x17 (not in the user segment), it also
    # indicates that the caller of the system-call is not a user task, and also exits.
111         cmpw $0x0f,CS(%esp)         # was old code segment supervisor ?
112         jne 3f
113         cmpw $0x17,OLDSS(%esp)      # was stack segment = 0x17 ?
114         jne 3f

    # The following code (lines 115-128) is used to process the signal of the current task.
    # Here, the signal bitmap (32 bits, each bit represents 1 kind of signal) in the current
    # task structure is first obtained, and then the signal block code is used to block the
    # impermissible signals. Then take the signal with the smallest value and reset the
    # corresponding bit of the signal in the original bitmap. Finally, using this signal as
    # one of the parameters, call do_signal()(in kernel/signal.c, 128). After the do_signal()
    # or signal handler returns, if the return value is not 0, then see if you need to switch
    # processes or continue processing other signals.
115         movl signal(%eax),%ebx      # signal bitmap -> ebx.
116         movl blocked(%eax),%ecx     # signals blocked -> ecx.
117         notl %ecx
118         andl %ebx,%ecx              # get a bitmap of permissible signals
119         bsfl %ecx,%ecx              # scan the bitmap from bit0, located none zero bit.
120         je 3f                       # exit if none.
121         btrl %ecx,%ebx              # reset the signal.
122         movl %ebx,signal(%eax)      # store the new bitmap -> current->signal
123         incl %ecx                   # adjust signal starting from 1 (1--32).
124         pushl %ecx                  # as parameter.
125         call _do_signal             # do_signal()(kernel/signal.c,128)
126         popl %ecx                   # discard the parameter.
127         testl %eax, %eax            # check return value.
128         jne 2b                      # see if we need to switch tasks, or do more signals

129 3:      popl %eax                   # contains ret code pushed at line 100.
130         popl %ebx
131         popl %ecx
132         popl %edx
133         addl $4, %esp               # skip orig_eax
```

```
134         pop %fs
135         pop %es
136         pop %ds
137         iret
138
    #### Int16 -- Processor error interrupt.   Type: Error; no error code.
    # This is an external hardware exception. When the coprocessor detects that it has an
    # error, it notifies the CPU via the ERROR pin. The following code is used to process
    # the error signal issued by the coprocessor and jump to execute the C function
    # math_error(). After returning, it will jump to the label 'ret_from_sys_call' to
    # continue execution.
139 .align 2
140 _coprocessor_error:
141         push %ds
142         push %es
143         push %fs
144         pushl $-1                   # fill in -1 for orig_eax    # not an syscall
145         pushl %edx
146         pushl %ecx
147         pushl %ebx
148         pushl %eax
149         movl $0x10,%eax            # ds,es point to kernel data seg.
150         mov %ax,%ds
151         mov %ax,%es
152         movl $0x17,%eax            # fs point to(user data seg)
153         mov %ax,%fs
154         pushl $ret_from_sys_call
155         jmp _math_error            # math_error()(kernel/math/error.c, 11).
156
    #### Int7 -- The device or coprocessor does not exist.   Type: Error; no error code.
    # If the EM (analog) flag in control register CR0 is set, the interrupt is raised when
    # CPU executes a coprocessor instruction, so the CPU has a chance to have the interrupt
    # handler emulate the coprocessor instruction (line 181).
    # The flag TS of CR0 is set when the CPU performs task switch. TS can be used to determine
    # when the content in the coprocessor does not match the task being executed by the CPU.
    # This interrupt is raised when the CPU is running a coprocessor escape instruction and
    # finds that TS is set. At this point you can save the coprocessor content of the previous
    # task and restore coprocessor execution status of the new task (line 176). See kernel/sched.c,
    # line 92. The interrupt will eventually be transferred to the label 'ret_from_sys_call'
    # for execution (detection and processing of the signal).
157 .align 2
158 _device_not_available:
159         push %ds
160         push %es
161         push %fs
162         pushl $-1                  # fill in -1 for orig_eax
163         pushl %edx
164         pushl %ecx
165         pushl %ebx
166         pushl %eax
167         movl $0x10,%eax            # ds,es to kernel data seg.
168         mov %ax,%ds
169         mov %ax,%es
```

```
        movl $0x17,%eax            # fs to user data seg.
        mov %ax,%fs

# The following code clears the flag TS and get CR0. If the coprocessor emulation flag EM
# is not set, indicating that it is not an interrupt caused by EM, the task coprocessor
# state is restored, the C function math_state_restore() is executed, and the code at
# ret_from_sys_call is executed upon return.
        pushl $ret_from_sys_call
        clts                       # clear TS so that we can use math
        movl %cr0,%eax
        testl $0x4,%eax            # EM (math emulation bit)
        je _math_state_restore

# If the EM flag is set, execute the math simulation function math_emulate().
        pushl %ebp
        pushl %esi
        pushl %edi
        pushl $0                   # temporary storage for ORIG_EIP
        call _math_emulate         # (math/math_emulate.c, line 476)
        addl $4,%esp               # discard temporary data.
        popl %edi
        popl %esi
        popl %ebp
        ret                        # ret to ret_from_sys_call

#### Int32 -- (int 0x20) Clock interrupt handler.
# The clock interrupt frequency is set to 100Hz (include/linux/sched.h, 4). The timing
# chip 8253/8254 is initialized at (kernel/sched.c, 438). Here jiffies add 1 every 10
# milliseconds. This code increments jiffies by 1, sends EOI to 8259 controller, and then
# calls the C function do_timer(long CPL) with current privilege level as a parameter.
.align 2
_timer_interrupt:
        push %ds                   # save ds,es and put kernel data space
        push %es                   # into them. %fs is used by _system_call
        push %fs
        pushl $-1                  # fill in -1 for orig_eax
        pushl %edx                 # we save %eax,%ecx,%edx as gcc doesn't
        pushl %ecx                 # save those across function calls. %ebx
        pushl %ebx                 # is saved as we use that in ret_sys_call
        pushl %eax
        movl $0x10,%eax            # ds,es to kernel
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax            # fs to user
        mov %ax,%fs
        incl _jiffies
        movb $0x20,%al             # EOI to interrupt controller #1
        outb %al,$0x20

# The current privilege level (0 or 3) in the selector (CS segment) that executes the
# system-call is fetched and pushed onto the stack as a parameter to do_timer. The
# do_timer() function performs task switching, timing, etc., and is implemented in
# kernel/sched.c, line 324.
```

```
        movl CS(%esp),%eax
        andl $3,%eax               # %eax is CPL (0 or 3, 0=supervisor)
        pushl %eax
        call _do_timer             # 'do_timer(long CPL)' does everything from
        addl $4,%esp               # task switching to accounting ...
        jmp ret_from_sys_call

#### This is sys_execve() syscall. The C function do_execve() is called with the caller's
# code pointer as a parameter.  function do_execve() is in fs/exec.c, line 207.
.align 2
_sys_execve:
        lea EIP(%esp),%eax         # eax points to old eip on the stack.
        pushl %eax
        call _do_execve
        addl $4,%esp               # discard pushed eip.
        ret

#### The sys_fork() call, used to create a child process, is syscall function 2.
# First call the C function find_empty_process() to get a process last_pid. If a negative
# number is returned, the current task array is full. Otherwise call copy_process() to
# copy the process.
.align 2
_sys_fork:
        call _find_empty_process   # get last_pid (kernel/fork.c, 143)
        testl %eax,%eax            # pid in eax, if negative then ret.
        js 1f
        push %gs
        pushl %esi
        pushl %edi
        pushl %ebp
        pushl %eax
        call _copy_process         # copy_process()(kernel/fork.c, 68).
        addl $20,%esp              # discard.
1:      ret

#### Int 46 -- (int 0x2E) Harddisk interrupt handler, which responds to IRQ14.
# This interrupt occurs when the requested hard disk operation is completed or an error
# occurs. (See kernel/blk_drv/hd.c).
# The code first sends an EOI instruction to 8259A slave chip, then takes function pointer
# in variable do_hd into EDX, and sets the do_hd to NULL. Then check if the edx function
# pointer is null. If it is null, let edx point to unexpected_hd_interrupt() to display
# the error message. The EOI instruction is then sent to the 8259A master chip, and the
# function pointed to by EDX is called: read_intr(), write_intr(), or unexpected_hd_interrupt().
_hd_interrupt:
        pushl %eax
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%eax            # ds,es poing to kernel data seg.
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax          # fs point to user data seg.
        mov %ax,%fs
        movb $0x20,%al
        outb %al,$0xA0           # EOI to interrupt controller #1
        jmp 1f                   # give port chance to breathe
1:      jmp 1f
# do_hd is defined as a function pointer that will be assigned the address of read_intr()
# or write_intr() function. The do_hd pointer variable is set to NULL after being placed
# in the edx register. Then the resulting function pointer is tested. If the pointer is
# NULL, the pointer is assigned to C function unexpected_hd_interrupt() to handle the
# unknown hard disk interrupt.
1:      xorl %edx,%edx
        movl %edx,_hd_timeout    # hd_timeout set to 0,controller produces INT in time.
        xchgl _do_hd,%edx
        testl %edx,%edx
        jne 1f                   # if null, point to unexpected_hd_interrupt().
        movl $_unexpected_hd_interrupt,%edx
1:      outb %al,$0x20           # send EOI to 8259A master chip.
        call *%edx               # "interesting" way of handling intr.
        pop %fs
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret

#### Int38 -- (int 0x26) floppy drive interrupt handler, handles the interrupt request IRQ6.
# The processing is basically the same as the above of the hard disk. (kernel/blk_drv/floppy.c).
# The following code first sends an EOI instruction to the 8259A interrupt controller
# master chip. Then take the function pointer in the variable do_floppy into the eax register,
# and set do_floppy to NULL. Then check if the eax function pointer is NULL. If it is NULL,
# then let eax point to unexpected_floppy_interrupt () to display the error message. Then
# call the function pointed to by eax: rw_interrupt, seek_interrupt, recal_interrupt,
# reset_interrupt or unexpected_floppy_interrupt.
_floppy_interrupt:
        pushl %eax
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%eax          # ds,es point kernel data seg.
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax          # fs point to user data seg.
        mov %ax,%fs
        movb $0x20,%al           # send EOI to 8259A master chip.
        outb %al,$0x20           # EOI to interrupt controller #1
        xorl %eax,%eax
        xchgl _do_floppy,%eax
        testl %eax,%eax          # function pointer NULL ?
        jne 1f                   # yes, point to unexpected_floppy_interrupt()
```

```
            movl $_unexpected_floppy_interrupt,%eax
    1:      call *%eax                  # "interesting" way of handling intr.
            pop %fs                     # function pointed by do_floppy
            pop %es
            pop %ds
            popl %edx
            popl %ecx
            popl %eax
            iret

    #### Int 39 -- (int 0x27) Parallel port interrupt handler, corresponding to IRQ7.
    # The kernel has not implemented this handler, only the EOI instruction is sent here.
    _parallel_interrupt:
            pushl %eax
            movb $0x20,%al
            outb %al,$0x20
            popl %eax
            iret
```

8.4.3 Reference Information

8.4.3.1 32-bit addressing for GNU assembly language

The GNU assembly language uses AT&T's assembly syntax. See Section 3.2.3 for a detailed introduction and examples of this. Here is just an introduction to the addressing method and some examples. The AT&T and Intel assembly language addressing operand formats are as follows:

```
AT&T:  disp (base, index, scale)
Intel: [base + index * scale + disp]
```

Where disp is a optional offset, base is 32-bit base address, index is 32-bit index register, and scale is scale factor (1, 2, 4, 8, default is 1). Although the two assembly language addressing formats are slightly different, the specific addressing locations are actually identical. The addressing position calculation method in the above format is: disp + base + index * scale

You don't need to write all of these fields when you apply, but there must be one in disp and base. Here are some examples.

Table 8-2 Memory addressing examples

Addressing requirements	AT&T format	Intel format
Addressing a specified C variable 'booga'	_booga	[_booga]
Addressing the location pointed to by register	(%eax)	[eax]
Addressing a variable by using the contents of the register as the base address	_variable(%eax)	[eax + _variable]
Address a value in an int array (scale value is 4)	_array(,%eax,4)	[eax*4 + _array]
Use direct addressing offset *(p+1), where p is the char's pointer, placed in %eax.	1(%eax)	[eax+1]
Addresses the specified character in an 8-byte array of records. Where eax is the index, and ebx is the	_array(%ebx,%eax,8)	[ebx + eax * 8 + _array]

offset of the specified character in the record.		

8.4.3.2 Adding a System Call

To add a new system-call to your kernel, we should first decide what its exact purpose is. Linux systems do not promote a system call for multiple purposes (except for ioctl() system calls). In addition, we need to determine the parameters, return values, and error code for the new system-call. The interface of the system call should be as simple as possible, so the parameters should be as few as possible. Also, the versatility and portability of system-calls should be considered in design. If we want to add a new system call to Linux 0.12, then we need to do the following things.

First, write the handler for the new system call in the relevant program, such as the function named sys_sethostname().This function is used to modify the computer name of the system. Usually this C function can be placed in the kernel/sys.c program. In addition, since the thisname structure is used, it is also necessary to move the thisname structure (lines 218-220) in sys_uname() outside of the function.

```
#define MAXHOSTNAMELEN 8
int sys_sethostname(char *name, int len)
{
        int     i;

        if (!suser())
                return -EPERM;
        if (len > MAXHOSTNAMELEN)
                return -EINVAL;
        for (i=0; i < len; i++) {
                if ((thisname.nodename[i] = get_fs_byte(name+i)) == 0)
                        break;
        }
        if (thisname.nodename[i]) {
                thisname.nodename[i>MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] = 0;
        }
        return 0;
}
```

Then add the new system call number and prototype definition in the include/unistd.h file. For example, you can add a function number after line 149 and add a prototype definition after line 279:

```
// The new system call number.
#define __NR_sethostname    87
// The new system calls function prototype.
int sethostname(char *name, int len);
```

Then add the external function declaration in the include/linux/sys.h file and insert the name of the new system call handler at the end of the function pointer table sys_call_table, as shown below. Note that the function names must be arranged in strict order of syscall function numbers.

```
extern int sys_sethostname();
// Function pointer array table.
fn_ptr sys_call_table[] = { sys_setup, sys_exit,sys_fork, sys_read,
   ...,
sys_lstat, sys_readlink, sys_uselib, sys_sethostname };
```

Then modify line 63 of the sys_call.s file to increase the total number of nr_system_calls by one. At this point you can recompile the kernel. Finally, refer to the implementation of the library function in the lib/ directory to add a new system call library function sethostname() to the libc library.

```
#define __LIBRARY__
#include <unistd.h>

_syscall2(int, sethostname, char *, name, int, len);
```

8.4.3.3 Using System-Calls Directly in Assembly File

Below is a simple assembly example asm.s given by Mr. Linus in explaining the relationship and difference between as86 and GNU as. This example shows how to program a stand-alone program in assembly language on a Linux system. That is, it is not necessary to use a start code module (such as crt0.o) and a function in the library. The procedure is as follows:

```
.text
_entry:
    movl $4,%eax                        # syscall nr, write op.
    movl $1,%ebx                        # paras: fhandle, stdout.
    movl $message,%ecx                  # paras: buff pointer.
    movl $12,%edx                       # paras: size.
    int $0x80
    movl $1,%eax                        # syscall no, exit.
    int $0x80

message:
    .ascii "Hello World\n"
```

There are two system-calls used: 4 - write file operation sys_write() and 1 - exit program sys_exit().The C function executed by the write system-call is declared as sys_write(int fd, char *buf, int len), see the program fs/read_write.c, starting at line 83. It comes with 3 parameters. These three parameters are stored in registers EBX, ECX, and EDX before calling the system call. The steps to compile and execute the program are as follows:

```
[/usr/root]# as -o asm.o asm.s
[/usr/root]# ld -o asm asm.o
[/usr/root]# ./asm
Hello World
[/usr/root]#
```

8.5 mktime.c

The mktime.c program is used to calculate the boot time of the kernel-specific UNIX calendar time.

8.5.1 Functions

The program has only one function kernel_mktime(), which is only used by the kernel and is used to calculate the number of seconds (calendar time) from 0:00 on January 1, 1970 to the date of booting, as the boot time. This function is exactly the same as the mktime() function provided in the standard C library, which converts the time represented by the tm structure into UNIX calendar time. However, since the kernel is not a normal program, you cannot call functions in the development environment library, so you must write one yourself.

8.5.2 Code Annotation

Program 8-4 linux/kernel/mktime.c

```
/*
 *  linux/kernel/mktime.c
 *
 *  (C) 1991  Linus Torvalds
 */

// Time type header file. The most important of these is the definition of the tm
// structure and some function prototypes related to time.
#include <time.h>

/*
 * This isn't the library routine, it is only used in the kernel.
 * as such, we don't care about years<1970 etc, but assume everything
 * is ok. Similarly, TZ etc is happily ignored. We just do everything
 * as easily as possible. Let's find something public for the library
 * routines (although I think minix times is public).
 */
/*
 * PS. I hate whoever though up the year 1970 - couldn't they have gotten
 * a leap-year instead? I also hate Gregorius, pope or no. I'm grumpy.
 */
#define MINUTE 60                    // 1 minute in seconds.
#define HOUR (60*MINUTE)             // 1 hour in seconds.
#define DAY (24*HOUR)                // 1 day in seconds.
#define YEAR (365*DAY)               // 1 year in seconds.

/* interestingly, we assume leap-years */
// The start time seconds at the beginning of each month is defined in a year limit.
static int month[12] = {
        0,
        DAY*(31),
        DAY*(31+29),
        DAY*(31+29+31),
        DAY*(31+29+31+30),
        DAY*(31+29+31+30+31),
        DAY*(31+29+31+30+31+30),
        DAY*(31+29+31+30+31+30+31),
        DAY*(31+29+31+30+31+30+31+31),
        DAY*(31+29+31+30+31+30+31+31+30),
        DAY*(31+29+31+30+31+30+31+31+30+31),
        DAY*(31+29+31+30+31+30+31+31+30+31+30)
};

// This function calculates the number of seconds elapsed from 0:00 on January 1, 1970 to
// the date of machine boot, as the boot time. The fields in the tm have been assigned in
// init/main.c and the information is taken from CMOS.
long kernel_mktime(struct tm * tm)
{
        long res;
        int year;

// First calculate the number of years that have passed since 1970. Because there is a
// 2-digit representation here, there will be a year 2000 problem. We can simply solve
// this problem by adding a statement to the front: if (tm->tm_year<70) tm->tm_year += 100;
// Since the year y of UNIX is calculated from 1970. It is a leap year until 1972, so the
// third year (71, 72, 73) is the first leap year, so the calculation method of the leap
// year from 1970 should be 1 + (y - 3) / 4, that is (y + 1)/4.
// res = the nr of secs in these years + the nr of secs in each leap year + the nr of secs
// from current year to current month. In addition, the number of days in February in the
// month[] array contains the number of days in leap year, that is, the number of days in
// February is one day more. Therefore, if the year is not a leap year and the current
// month is greater than February, we will subtract this day. Since we counted from 1970,
// the detection method for the leap year is: (y + 2) can be divided by 4. If not, it's
// not a leap year.
// if (tm->tm_year<70) tm->tm_year += 100;
        year = tm->tm_year - 70;
/* magic offsets (y+1) needed to get leapyears right.*/
        res = YEAR*year + DAY*((year+1)/4);
        res += month[tm->tm_mon];
/* and (y+2) here. If it wasn't a leap-year, we have to adjust */
        if (tm->tm_mon>1 && ((year+2)%4))
                res -= DAY;
        res += DAY*(tm->tm_mday-1);          // nr of days in the past month in secs.
        res += HOUR*tm->tm_hour;             // the past hours of the day in secs.
        res += MINUTE*tm->tm_min;            // the past minutes of the hour in secs.
        res += tm->tm_sec;                   // nr of secs that have passed in 1 minute.
        return res;                          // the nr of secs elapsed since 1970.
}

```

8.5.3 Information

8.5.3.1 Calculation method for leap year

The basic calculation method for leap years is:

If y can be divisible by 4 and cannot be divisible by 100, or can be divisible by 400, then y is a leap year.

8.6 sched.c

8.6.1 Function description

The sched.c source file contains codes for scheduling tasks in the kernel. It includes several basic functions for scheduling (sleep_on(), wakeup(), schedule(), etc.), as well as some simple system-call functions (such as getpid()).The timer function do_timer() of the system clock interrupt service routine is also in this program. In addition, in order to facilitate the floppy disk drive timing processing programming, Mr. Linus also put several functions related to floppy disk timing into this program.

The code for these basic functions is not long, but it is somewhat abstract and difficult to understand. Fortunately, there are already many textbooks that have a more in-depth introduction and discussion. Therefore, you can refer to other books to describe these functions when you study. Before we start to annotate and analyze the code, let's take a look at the principle of the scheduler, sleep, and wake-up functions.

8.6.1.1 Schedule function

The schedule function schedule() is responsible for selecting the next task (process) to run in the system. It first checks all tasks and wakes up any task that has received signals. The specific method is to check the alarm timing value 'alarm' for each task in the task array. If the task's alarm time has expired (jiffies > alarm), set the SIGALRM signal in its signal bitmap and clear the alarm value. Jiffies is the number of ticks from machine boot time (10ms/tick, defined in sched.h). If there are other signals besides the blocked signal in the signal bitmap of the task and the task is in an interruptible sleep state (TASK_INTERRUPTIBLE), then the task is set to ready state (TASK_RUNNING).

This is followed by the core processing part of the scheduling function. This part of the code selects the task to be executed later based on the time slice and priority mechanism of the task. It first loops through all the tasks in the task array, selects the task with the largest counter value of the remaining execution time, and switches to the task with the switch_to() function.

If the counter value of all ready-to-run tasks is equal to zero, it means that the time slice of all tasks has been run out at the moment. Then, according to the task priority value 'priority', reset the running time slice value 'counter' of each task, and then cycle through the execution time slice values of all tasks again.

8.6.1.2 Sleep and wake-up functions

The other two functions worth mentioning are the sleep function sleep_on() and the wake-up function wake_up().Although these two functions are very short, they are harder to understand than the schedule() function. Before we look at the code, let's make some explanations by means of the diagram. Simply put, the main function of the sleep_on() is to temporarily switch the process out onto the waiting queue for a period of time when the resource requested by the process (or task) is being used or not in memory. When the process is switched back, it will continue to run. The way to put in the wait queue takes advantage of the tmp pointer in the function as the link for each waiting task.

The function involves the operation of three task pointers: *p, tmp, and current. *p is the wait queue head pointer, such as the i_wait pointer of the file system memory i node, the buffer_wait pointer in the memory buffer operation, etc.; tmp is a temporary pointer established on the function stack, stored on the current task kernel state stack; 'current' is a pointer to the current task. For the changes of these pointers in memory, we can use the schematic diagram of Figure 8-6 to illustrate. The long bars in the figure represent a sequence of memory bytes.

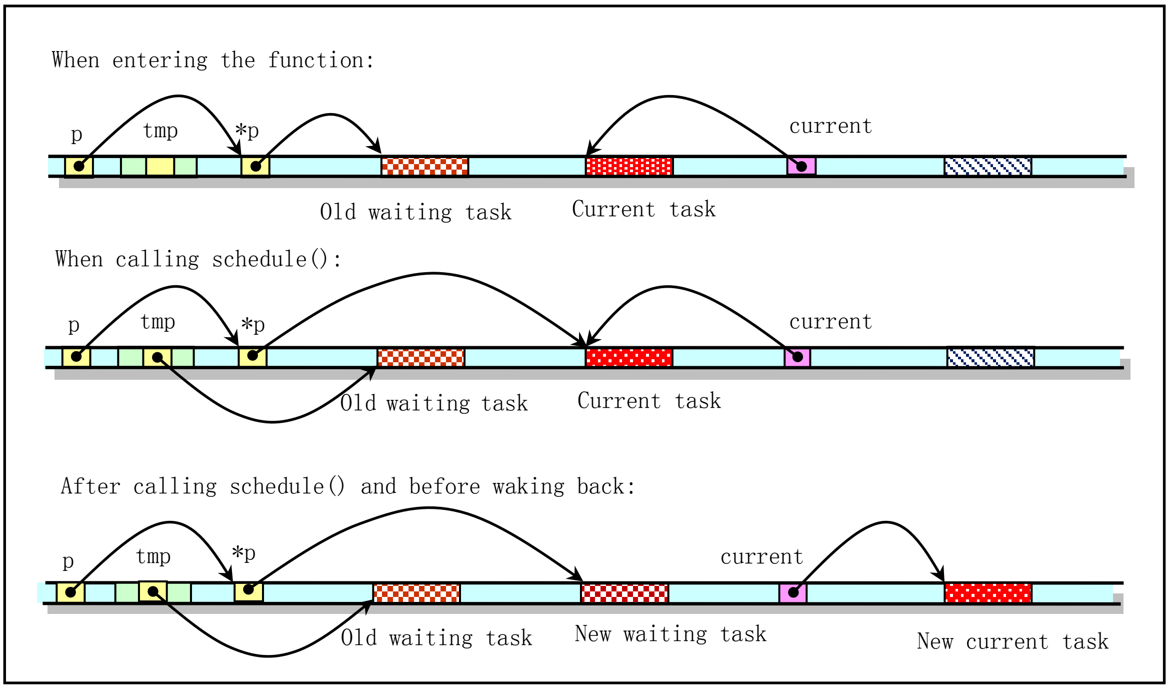


Figure 8-6 Schematic diagram of pointer changes in the sleep_on().

When entering the function, the queue head pointer *p points to the task structure (process descriptor) that has been waiting in the wait queue. Of course, there is no waiting task on the wait queue when the system first starts executing. Therefore, the original waiting task in the above figure does not exist at the beginning, and *p points to NULL.

Through the pointer operations, before the scheduler function is called, the queue head pointer points to the current task structure, and the temporary pointer 'tmp' in the function points to the original waiting task. Before executing the scheduler and before the task is woken up and returned to execution, the current task pointer is directed to the new current task, and the CPU switches to execute in the new task. In this way, the execution of the sleep_on() function causes the tmp pointer to point to the original waiting task pointed to by the queue head pointer in the queue, and the queue head pointer points to the newly added waiting task, that is, the task of calling this function. Thus, by the link function of the temporary pointer tmp on the stack, when several processes call the function for waiting for the same resource, the kernel program implicitly constructs a waiting queue. See the waiting queue diagram in Figure 8-7. The figure shows the situation when a third task is inserted into the head of the queue. From the figure we can more easily understand the wait queue formation process of the sleep_on() function.

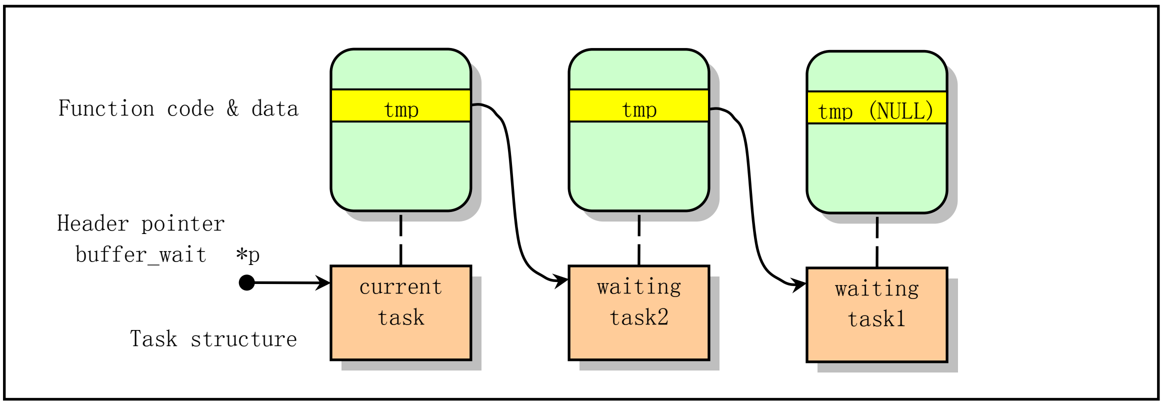


Figure 8-7 The implicit task wait queue for the sleep_on()

After inserting the process into the wait queue, the sleep_on() function calls the schedule() function to execute another process. When the process is awakened and re-executed, the subsequent statements are executed, and a process that enters the waiting queue earlier than it wakes up. Note that the so-called wake-up here does not mean that the process is in the execution state, but in the ready state that can be scheduled to execute.

The wakeup function wake_up() is used to put the specified task waiting for available resources into a ready state (TASK_RUNNING). This function is a generic wakeup function. In some cases, such as reading a block of data on a disk, since any task in the wait queue may be awakened first, it is also necessary to empty the pointer of the wake-up task structure. In this way, when the process that goes to sleep is awakened and the sleep_on() is re-executed, there is no need to wake up the process.

There is also a function interruptible_sleep_on(), whose structure is basically similar to sleep_on(), except that the current task is set to an interruptible wait state before scheduling, and after the task is awakened, it is necessary to determine whether there is any task entered later. If so, schedule them to run first. Starting at kernel 0.12, the two functions are combined into one and only use the state of the task as a parameter to distinguish between the two cases.

When reading the code in this file, it is best to refer to the comments in the include/linux/sched.h file for a more complete understanding of the kernel's scheduling mechanism.

8.6.2 Code Annotation

Program 8-5 linux/kernel/sched.c

```
/*
 *  linux/kernel/sched.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 'sched.c' is the main kernel file. It contains scheduling primitives
 * (sleep_on, wakeup, schedule etc) as well as a number of simple system
 * call functions (type getpid(), which just extracts a field from
 * current-task
 */
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the
//     data of the initial task 0, and some embedded assembly function macro statements
//     about the descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
//     commonly used functions of the kernel.
// <linux/sys.h> The system calls the header file. Contains 72 system call C function
//     handlers, starting with 'sys_'.
// <linux/fdreg.h> Floppy disk file. Contains some definitions of floppy disk controller
//     parameters.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//     descriptors/interrupt gates, etc. is defined.
// <asm/io.h> Io header file. Defines the function that operates on the io port in the form
//     of a macro's embedded assembler.
```

```
   // <asm/segment.h> Segment operation header file. An embedded assembly function is defined
   //     for segment register operations.
   // <signal.h> Signal header file. Define signal symbol constants, signal structures, and
   //     signal manipulation function prototypes.
13 #include <linux/sched.h>
14 #include <linux/kernel.h>
15 #include <linux/sys.h>
16 #include <linux/fdreg.h>
17 #include <asm/system.h>
18 #include <asm/io.h>
19 #include <asm/segment.h>
20
21 #include <signal.h>
22
   // The macro takes the binary value of the corresponding bit of the signal nr in the
   // signal bitmap. The signal number range is 1-32. For example, the bitmap value of
   // signal 5 is 1<<(5-1) = 16 = 00010000b.
   // All signals except the SIGKILL and SIGSTOP signals are blockable:
   // BLOCKABLE = (11111111,11111011,11111110,11111111b).
23 #define _S(nr) (1<<((nr)-1))
24 #define _BLOCKABLE (~(_S(SIGKILL) | _S(SIGSTOP)))
25
   // Kernel debugging function. prints the pid, process status, kernel stack free bytes
   // (approximately) and task's younger & older siblings for the specified task nr.
   // Because task's data and its kernel state stack are on the same memory page (4096 bytes
   // per page), and the kernel state stack begins from the end of the page downward, so
   // variable j on line 28 represents the maximum kernel stack capacity, or the lowest top
   // position of the task's kernel stack.
   // Parameters:
   // nr - task no;    p - task structure pointer.
26 void show_task(int nr,struct task_struct * p)
27 {
28         int i,j = 4096-sizeof(struct task_struct);
29
30         printk("%d: pid=%d, state=%d, father=%d, child=%d,  ",nr,p->pid,
31                 p->state, p->p_pptr->pid, p->p_cptr ? p->p_cptr->pid : -1);
32         i=0;
33         while (i<j && !((char *)(p+1))[i])  // Detects nr of zero bytes after task struct.
34                 i++;
35         printk("%d/%d chars free in kstack\n\r",i,j);
36         printk("   PC=%08X.", *(1019 + (unsigned long *) p));
37         if (p->p_ysptr || p->p_osptr)
38                 printk("   Younger sib=%d, older sib=%d\n\r",
39                         p->p_ysptr ? p->p_ysptr->pid : -1,
40                         p->p_osptr ? p->p_osptr->pid : -1);
41         else
42                 printk("\n\r");
43 }
44
   // Displays status information for all tasks in the system.
   // NR_TASKS is the max nr of tasks in the system (64), defined in line 6 of linux/sched.h.
45 void show_state(void)
46 {
```

```
47         int i;
48
49         printk("\rTask-info:\n\r");
50         for (i=0;i<NR_TASKS;i++)
51                 if (task[i])
52                         show_task(i,task[i]);
53 }
54
   // The input clock frequency of 8253 counter/timer chip is about 1.193180 MHz. The Linux
   // kernel expects the timer interrupt frequency to be 100 Hz, that is, a clock interrupt
   // is issued every 10 ms. So here LATCH is the initial value of setting the 8253 chip, see
   // line 438.
55 #define LATCH (1193180/HZ)
56
57 extern void mem_use(void);          // [??] not defined anywhere.
58
59 extern int timer_interrupt(void);   // kernel/system_call.s, 189
60 extern int system_call(void);       // kernel/system_call.s, 84
61
   // Each task (process) has its own kernel-state stack. This defines the task union,
   // consisting of the task structure and the stack array. Because the data structure of a
   // task and its kernel-state stack are placed in the same memory page, its data segment
   // selector can be obtained from the stack segment register SS.
   // Line 67 below sets data for the initial task (initial data is in linux/sched.h, 156).
62 union task_union {
63         struct task_struct task;
64         char stack[PAGE_SIZE];
65 };
66
67 static union task_union init_task = {INIT_TASK,};
68
   // The nr of ticks from system start (10ms/tick). A tick is made every time the timer
   // chip's interrupt occurs.
   // The qualifier 'volatile', its English meaning is easy to change, unstable. The meaning
   // of this qualifier is to indicate to the compiler that the contents of the variable may
   // change as a result of modifications by other programs. Usually when a variable is
   // declared in a program, the compiler will try to put it in a general-purpose register,
   // such as EBX, to improve access efficiency. After that, it generally does not care about
   // the original value of the variable in memory. If other programs or devices modify the
   // value of this variable in memory at this time, the value in EBX will not be updated. To
   // solve this issue, a volatile qualifier is created, so that the code must take its value
   // from the specified memory location when referring to the variable. Here, gcc is required
   // not to optimize the jiffies, nor to move the location, and to take its value from memory.
   // Because the counter/timer chip interrupt processing process and other programs will modify
   // its value.
69 unsigned long volatile jiffies=0;        // kernel pulse (ticks)
70 unsigned long startup_time=0;            // total seconds from 1970:0:0:0
71 int jiffies_offset = 0;          /* # clock ticks to add to get "true
72                                     time".  Should always be less than
73                                     1 second's worth.  For time fanatics
74                                     who like to syncronize their machines
75                                     to WWV :-) */
76
```

```
// current task pointer and points to task 0 during initialization.
struct task_struct *current = &(init_task.task);
struct task_struct *last_task_used_math = NULL;

// Define an array of task pointers. The first item is initialized to the task data
// structure of the initial task (task 0).
struct task_struct * task[NR_TASKS] = {&(init_task.task), };

// Define the user stack (array), a total of 1K items, size 4K bytes. Used as a kernel
// stack during kernel initialization. After initialization is complete, it will be used
// as the user mode stack for task 0.  It is the kernel stack before running task 0 and is
// later used as the user state stack for tasks 0 and 1.
// The following structure is used to set the stack SS: ESP, see head.s, line 23. SS is
// set to the kernel data segment selector (0x10), and ESP is set to point to the end of
// the last item in the user_stack array. This is because Intel CPU performs the stack
// operation by first decrementing the stack pointer SP and then saving the contents of
// the stack at the SP pointer.
long user_stack [ PAGE_SIZE>>2 ] ;

struct {
        long * a;
        short b;
        } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
/*
 *  'math_state_restore()' saves the current math information in the
 * old math state array, and gets the new ones from the current task
 */
// After the task is scheduled to be exchanged, this function is used to save the math
// coprocessor state (context) of the original task and restore the coprocessor context
// of the new task scheduled.
void math_state_restore()
{
// Return if the task has not changed (the previous task is the current task). Here
// "previous task" refers to the task that has just been exchanged out.
// In addition, the WAIT instruction must be executed before the coprocessor instructions.
// If the previous task used a coprocessor, its state is saved to task's field of TSS.
        if (last_task_used_math == current)
                return;
        __asm__("fwait");
        if (last_task_used_math) {
                __asm__("fnsave %0"::"m" (last_task_used_math->tss.i387));
        }
// Now, 'last_task_used_math' points to the current task, in case the current task is
// swapped out. At this point, if the current task has used the coprocessor, its state is
// restored. Otherwise, it is the first time to use, so the initialization command is sent
// to the coprocessor, and the coprocessor flag is set.
        last_task_used_math=current;
        if (current->used_math) {
                __asm__("frstor %0"::"m" (current->tss.i387));
        } else {
                __asm__("fninit"::);           // send initial cmd to the math.
                current->used_math=1;          // set used math flag.
        }
```

```
}

/*
 *  'schedule()' is the scheduler function. This is GOOD CODE! There
 * probably won't be any reason to change this, as it should work well
 * in all circumstances (ie gives IO-bound processes good response etc).
 * The one thing you might take a look at is the signal-handler code here.
 *
 *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
void schedule(void)
{
        int i,next,c;
        struct task_struct ** p;              // pointer's pointer of task struct.

/* check alarm, wake up any interruptible tasks that have got a signal */

// Start checking the alarm from the last task in the task array. Skip empty pointer items
// when looping.
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if (*p) {
// If the task timeout is set and has expired (jiffies>timeout), then timeout is reset to
// 0, and if the task is in TASK_INTERRUPTIBLE sleep state, then it is put into the ready
// state (TASK_RUNNING).
                        if ((*p)->timeout && (*p)->timeout < jiffies) {
                                (*p)->timeout = 0;
                                if ((*p)->state == TASK_INTERRUPTIBLE)
                                        (*p)->state = TASK_RUNNING;
                        }
// If the timeout alarm value of the SIGALRM signal of the task is set and has expired
// (alarm<jiffies), the SIGALRM signal is set in the signal bitmap, that is, the SIGALARM
// signal is sent to the task, and then the alarm is cleared. The default action for this
// signal is to terminate the task.
                        if ((*p)->alarm && (*p)->alarm < jiffies) {
                                (*p)->signal |= (1<<(SIGALRM-1));
                                (*p)->alarm = 0;
                        }
// If there are other signals in the signal bitmap in addition to the blocked signal and
// the task is in an interruptible state, then set the task to ready state (TASK_RUNNING).
// Where '~(_BLOCKABLE & (*p)->blocked)' is used to ignore blocked signals, but SIGKILL
// and SIGSTOP signals cannot be blocked.
                        if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
                        (*p)->state==TASK_INTERRUPTIBLE)
                                (*p)->state=TASK_RUNNING;     // set ready.
                }

/* this is the scheduler proper: */

        while (1) {
                c = -1;
                next = 0;
147                 i = NR_TASKS;
148                 p = &task[NR_TASKS];
    // This code is also looped from the last task of the task array and skips the empty
    // slots. It compares the counter (the countdown number of task run time) of each ready
    // state task. Which value is large, it means that there are still many running times of
    // the task, and next points to the task number of that task.
149                 while (--i) {
150                         if (!*--p)
151                                 continue;
152                         if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
153                                 c = (*p)->counter, next = i;
154                 }
    // If the comparison results in a result with a counter value not equal to 0, or if there
    // is no executable task in the system (c is still -1, next=0), then exit the outer while
    // loop (144 lines), execute the latter task switching macro(line 161).  Otherwise, the
    // counter value of each task is updated according to the priority of each task, and then
    // back to 144 lines for re-comparison. The counter value is calculated as
    //     counter = counter /2 + priority
    // Note that the calculation process here does not consider the state of the process.
155                 if (c) break;
156                 for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
157                         if (*p)
158                                 (*p)->counter = ((*p)->counter >> 1) +
159                                                 (*p)->priority;
160         }
    // The following macro (sched.h) takes the selected task next as the current task and
    // switches to the task to run. Because next is initialized to 0 on line 146, next is
    // always 0 if there are no other tasks in the system to run. Therefore, the scheduler
    // will run task 0 when the system is idle. At this time, task 0 only executes the pause()
    // syscall, and causes this function to be called again.
161         switch_to(next);
162 }
163
    // This is the pause() system-call, which is used to convert the current task's state to
    // an interruptible wait state (TASK_INTERRUPTIBLE) and reschedule.
    // This system-call will cause the process to go to sleep until a signal is received. This
    // signal is used to terminate the process or cause the process to call a signal capture
    // function. Pause() returns only if a signal is caught and the signal capture handler
    // returns. At this point the pause() return value should be -1 and errno is set to EINTR.
    // It has not been fully implemented yet (until kernel 0.95).
164 int sys_pause(void)
165 {
166         current->state = TASK_INTERRUPTIBLE;
167         schedule();
168         return 0;
169 }
170
    // The following function sets the current task to an interruptible or uninterruptible
    // sleep state and has the sleep queue head pointer point to the current task.
    // The function parameter p is the wait pointer of the task queue; the parameter state is
    // the state used by the task sleep: TASK_UNINTERRUPTIBLE or TASK_INTERRUPTIBLE. Tasks in
    // the uninterruptible sleep state require the kernel to explicitly wake up using the
    // wake_up() function; tasks in the interruptible sleep state can be woken up by signal,
    // task timeout, etc. (set to ready state TASK_RUNNING).
    // *** Note that because this code is not very mature, there are some issues with it.
    static inline void __sleep_on(struct task_struct **p, int state)
    {
            struct task_struct *tmp;

    // First, if the pointer is invalid, it will exit. (The object pointed to by the pointer
    // can be NULL, but the pointer itself will not be 0). If the current task is task 0, the
    // kernel panic.
            if (!p)
                    return;
            if (current == &(init_task.task))
                    panic("task[0] trying to sleep");
    // Then let tmp point to the task already on waiting queue (if any), such as inode->i_wait,
    // and point the sleep queue head to the current task. This inserts the current task into
    // the wait queue of *p. The current task is then placed in the specified wait state and
    // rescheduling is performed.
            tmp = *p;
            *p = current;
            current->state = state;
    repeat: schedule();

    // Only when this waiting task is awakened will the program continue to execute from
    // here. Indicates that the process has been explicitly woken up and executed.
    // If there are still waiting tasks in the queue, and the task pointed to by the queue
    // header *p is not the current task, then there is still tasks entering the queue after
    // the task is inserted.  Therefore, we should also awaken these subsequent tasks entered
    // laterly. So the task indicated by the queue header is set to the ready state first, and
    // the current task itself is set to the uninterruptible wait state. That is, after waiting
    // for these subsequent queued tasks to be awakened, the current task itself can then be
    // woken up by using the wake_up() function. Then jump to label repeat and re-execute the
    // schedule() function.
            if (*p && *p != current) {
                    (**p).state = 0;
                    current->state = TASK_UNINTERRUPTIBLE;
                    goto repeat;
            }

    // Execution here, indicating that this task is really awakened to execute. At this point
    // the queue head should point to this task. If it is empty, it indicates that there is a
    // problem with the schedule, and a warning message is displayed. Finally, we let the head
    // point to the task that entered the queue in front of us (*p = tmp). If there is such a
    // task, that is, there are tasks in the queue (tmp is not empty), it will be woken up.
    // Therefore, the task that first enters the queue will eventually set the wait queue
    // header to NULL when it is run after wakeup.
            if (!*p)
                    printk("Warning: *P = NULL\n\r");
            if (*p = tmp)
                    tmp->state=0;
    }

    // Set the current task to the interruptible wait state (TASK_INTERRUPTIBLE) and put it
    // into the wait queue specified by the head pointer *p. This wait state task can be
```

```
    // awakened by means of signals, task timeouts, and the like.
194 void interruptible_sleep_on(struct task_struct **p)
195 {
196         __sleep_on(p,TASK_INTERRUPTIBLE);
197 }
198

    // Set the current task to the interruptible wait state (TASK_UNINTERRUPTIBLE) and put
    // it into the wait queue specified by the head pointer *p. This wait state task can only
    // be awakened by wait_up() function.
199 void sleep_on(struct task_struct **p)
200 {
201         __sleep_on(p,TASK_UNINTERRUPTIBLE);
202 }
203

    // Wake up the uninterruptible wait task. *p is the task wait queue head pointer. Since
    // the new wait task is inserted at the wait queue head, the wake up is the last task to
    // enter the wait queue. If the task is already in a stopped or zombie state, a warning
    // message is displayed.
204 void wake_up(struct task_struct **p)
205 {
206         if (p && *p) {
207                 if ((**p).state == TASK_STOPPED)
208                         printk("wake_up: TASK_STOPPED");
209                 if ((**p).state == TASK_ZOMBIE)
210                         printk("wake_up: TASK_ZOMBIE");
211                 (**p).state=0;                          // TASK_RUNNING
212         }
213 }
214
215 /*
216  * OK, here are some floppy things that shouldn't be in the kernel
217  * proper. They are here because the floppy needs a timer, and this
218  * was the easiest way of doing it.
219  */
    // The following 220--281 lines of code are used to handle the floppy drive timing.
    // Before reading this code, please take a look at the instructions in the chapter on
    // block devices for floppy drivers (floppy.c), or look at this code when reading the
    // floppy block device driver.
    //
    // The array wait_motor[] is used to store the process pointer waiting for drive motor
    // to start up to normal speed. The array index 0-3 corresponds to the floppy drive A-D.
    // The array mon_timer[] stores the number of ticks required for each floppy drive motor
    // to start. The default startup time in the program is 50 ticks (0.5 seconds).
    // The array moff_timer[] stores the time each floppy drive needs to maintain before the
    // motor stalls. The program is set to 10,000 ticks (100 seconds).
220 static struct task_struct * wait_motor[4] = {NULL,NULL,NULL,NULL};
221 static int  mon_timer[4]={0,0,0,0};
222 static int moff_timer[4]={0,0,0,0};

    // The following variables correspond to the current digital output register (DOR) in the
    // floppy drive controller. The definition of each bit of this register is as follows:
    // Bits 7-4: control the activation of the drive D-A motor separately. 1-Start; 0-Close.
    // Bit 3:1 - enable DMA and interrupt requests; 0  - disable DMA and interrupt requests.
    // Bit 2:1 - start floppy drive controller(FDC); 0 - reset FDC.
    // Bits 1-0: Used to select the floppy drive A-D.
    // The initial value set here is: Allow DMA and interrupt request, start FDC.
    unsigned char current_DOR = 0x0C;

    // Specifies the wait time for the floppy drive from begin to normal operation.
    // Parameter nr is floppy drive number (0--3), the return value is the nr of ticks.
    // The variable selected is the selected floppy drive flag (blk_drv/floppy.c, line 123).
    // The mask is the start motor bits in the selected floppy drive DOR, and the upper 4 bits
    // are the floppy drive start motor flags.
    int ticks_to_floppy_on(unsigned int nr)
    {
            extern unsigned char selected;
            unsigned char mask = 0x10 << nr;

    // The system has up to 4 floppy drives. First, set the time (100 seconds) that the
    // specified floppy drive nr needs to stop before it stops. Then take the current DOR
    // value into variable mask, and set the motor start flag of the specified floppy drive
    // in it.
            if (nr>3)
                    panic("floppy_on: nr>3");
            moff_timer[nr]=10000;           /* 100 s = very big :-) */
            cli();                          /* use floppy_off to turn it off */
            mask |= current_DOR;
    // If the floppy drive is not currently selected, first reset the selection bits of other
    // floppy drive, then set the floppy drive selection bit.
            if (!selected) {
                    mask &= 0xFC;
                    mask |= nr;
            }
    // If the current value of DOR is different from the required value, a new value (mask) is
    // output to the FDC digital output port, and if the motor that is required to start is
    // not yet started, the motor start timer value of the corresponding floppy drive is set
    // (HZ/2 = 0.5) Seconds or 50 ticks). If it has been started, set the startup timing to 2
    // ticks, which can meet the requirements of the following depreciation in do_floppy_timer().
    // The current digital output register current_DOR is updated thereafter.
            if (mask != current_DOR) {
                    outb(mask,FD_DOR);
                    if ((mask ^ current_DOR) & 0xf0)
                            mon_timer[nr] = HZ/2;
                    else if (mon_timer[nr] < 2)
                            mon_timer[nr] = 2;
                    current_DOR = mask;
            }
            sti();                          // enable int.
            return mon_timer[nr];           // return time value required to start motor.
    }

    // wait for a period of time required to start the floppy drive motor.
    // Sets the delay time required for the motor of the specified floppy drive from start to
    // normal speed, then sleeps. During the timer interrupt process, the delay value set here
    // is decremented. When the delay expires, it will wake up the waiting process here.
```

```
251 void floppy_on(unsigned int nr)
252 {
    // Disable interrupt. If the motor start timer has not expired, the current process is
    // always placed in an uninterruptible sleep state and placed in a queue waiting for the
    // motor to run. Then open the interrupt.
253         cli();
254         while (ticks_to_floppy_on(nr))
255                 sleep_on(nr+wait_motor);
256         sti();
257 }
258
    // Set to turn off the motor stall timer (3 seconds).
    // If you do not use this function to explicitly turn off the specified floppy drive
    // motor, it will be turned off after the motor is turned on for 100 seconds.
259 void floppy_off(unsigned int nr)
260 {
261         moff_timer[nr]=3*HZ;
262 }
263
    // The floppy disk timer subroutine. Update the motor start timing value and the motor off
    // stall count value. This subroutine is called during the system timer interrupt, so the
    // system is called once every time a tick (10ms) is passed, and the value of the motor on
    // or off timer is updated at each time. If a motor stall timing expires, the DOR motor
    // start bit is reset.
264 void do_floppy_timer(void)
265 {
266         int i;
267         unsigned char mask = 0x10;
268
    // For the four floppy drives that the system have, check the floppy drives in use one by
    // one. Skip if it is not the motor specified by DOR.  If the motor start timer expires,
    // the process is woken up. If the motor off timer expires, reset motor's start bit.
269         for (i=0 ; i<4 ; i++,mask <<= 1) {
270                 if (!(mask & current_DOR))
271                         continue;
272                 if (mon_timer[i]) {
273                         if (!--mon_timer[i])            // if motor on timer expires
274                                 wake_up(i+wait_motor); // wake up the process.
275                 } else if (!moff_timer[i]) {
276                         current_DOR &= ~mask;          // reset motor start bit
277                         outb(current_DOR,FD_DOR);      // update DOR.
278                 } else
279                         moff_timer[i]--;
280         }
281 }
282
    // Below is the code for the kernel timer. There can be up to 64 timers.
    // Lines 285-289 defines linked timer list structure and timer array. The linked timer
    // list is dedicated to the floppy drive to turn on and off motor for timing operation.
    // This type of timer is similar to the dynamic timer in modern Linux systems and is
    // intended for use only by the kernel.
283 #define TIME_REQUESTS 64
284
```

```
static struct timer_list {
        long jiffies;                   // Timer ticks.
        void (*fn)();                   // Timer handler.
        struct timer_list * next;       // points to the next timer.
} timer_list[TIME_REQUESTS], * next_timer = NULL;  // next_timer is timer queue head.

// Add timer subroutine. The input parameters are the specified timing values (ticks) and
// the associated handler. The floppy disk driver uses this function to perform a delay
// operation to start or shut down the motor.
// jiffies - number of timed ticks; *fn() - function to be executed when the time is up.
void add_timer(long jiffies, void (*fn)(void))
{
        struct timer_list * p;

// If the timer handler pointer to be added is null, the function is exited.
        if (!fn)
                return;
        cli();
// If the timer time value <=0, its handler is called immediately and the timer is not
// added to the linked list.
        if (jiffies <= 0)
                (fn)();
        else {
// Otherwise, find a free entry from the timer array.
                for (p = timer_list ; p < timer_list + TIME_REQUESTS ; p++)
                        if (!p->fn)
                                break;
// If the timer array has been used up, the system crashes :-). Otherwise, the timer data
// structure is filled with information and linked into the list header.
                if (p >= timer_list + TIME_REQUESTS)
                        panic("No more time requests free");
                p->fn = fn;
                p->jiffies = jiffies;
                p->next = next_timer;
                next_timer = p;
// The linked list items are sorted from early to late according to the time value.
// Subtract the number of ticks needed before sorting. In this way, when processing the
// timer, it is only necessary to check whether the timing of the first item expires.
                while (p->next && p->next->jiffies < p->jiffies) {
                        p->jiffies -= p->next->jiffies;
                        fn = p->fn;
                        p->fn = p->next->fn;
                        p->next->fn = fn;
                        jiffies = p->jiffies;
                        p->jiffies = p->next->jiffies;
                        p->next->jiffies = jiffies;
                        p = p->next;
                }
        }
        sti();
}

//// The C function called in the timer interrupt handler. Called in _timer_interrupt
```

```
// (line 189, 209) in the sys_call.s file. The parameter cpl is the current privilege
// level 0 or 3, which is the privilege level in the code selector being executed when
// the interrupt occurs. Cpl=0 means that the kernel code is being executed when the
// interrupt occurs; cpl=3 means that the user code is being executed when the interrupt
// occurs. For a task, if its execution time slice is used up, the task is switched. At
// this point the function will perform a timing update.
void do_timer(long cpl)
{
        static int blanked = 0;

// First determine if you need to perform a screen blankout operation. If the blankcount
// is not zero, or the black screen delay interval blankinterval is 0, then if the screen
// is already in a black screen (black screen flag blanked = 1), the screen is restored.
// If the blankcount is not zero, it is decremented and the black screen flag is reset.
        if (blankcount || !blankinterval) {
                if (blanked)
                        unblank_screen();
                if (blankcount)
                        blankcount--;
                blanked = 0;
// Otherwise, if the black screen flag is not set, the screen will be blank and the flag
// will be set.
        } else if (!blanked) {
                blank_screen();
                blanked = 1;
        }

// Next, we handle the hard disk operation timeout issue. If the hard disk timeout count
// is decremented to 0, the hard disk access timeout processing is performed.
        if (hd_timeout)
                if (!--hd_timeout)
                        hd_times_out();            // blk_drv/hdc, line 318

// If the beep counts is reached, the beep is turned off.(cmd is sent to port 0x61, reset
// bits 0 and 1. Bit 0 controls the counter 2 of 8253 chip, bit 1 controls the speaker ).
        if (beepcount)                       // beep ticks (chr_drv/console.c, 950)
                if (!--beepcount)
                        sysbeepstop();       // chr_drv/console.c, 944.

// If the current privilege level (cpl) is 0 (the highest, indicating that the kernel
// program is working), then the kernel code runtime stime is incremented; if cpl > 0,
// it means that the general user program is working, adding utime.
        if (cpl)
                current->utime++;
        else
                current->stime++;

// If a timer exists, the value of the first timer in the linked list is decremented by
// one. If it is equal to 0, the corresponding handler is called, and the handler pointer
// is set to null, and then the timer is removed.
        if (next_timer) {                       // timer list header.
                next_timer->jiffies--;
                while (next_timer && next_timer->jiffies <= 0) {
                        void (*fn)(void);        // a function pointer definition.

                        fn = next_timer->fn;
                        next_timer->fn = NULL;
                        next_timer = next_timer->next;
                        (fn)();                 // call the timer handler.
                }
        }
// If the motor enable bit in the DOR of the current floppy disk controller FDC is set,
// the floppy disk timer routine is executed.
        if (current_DOR & 0xf0)
                do_floppy_timer();
// If the task still has run time, exit here to continue running the task. Otherwise, the
// current task running count is set to 0. And if it is running in the kernel code when
// the interrupt occurs, it returns, otherwise it means that the user program is being
// executed, so the scheduler is called to try to perform task switching operation.
        if ((--current->counter)>0) return;
        current->counter=0;
        if (!cpl) return;                       // kernel code
        schedule();
}

// System-call function - Sets the alarm timer value (in seconds).
// If the parameter seconds > 0, the new timing is set, and the remaining interval is
// returned to the original timing, otherwise it returns 0.
// The unit of the alarm field in the process data structure is tick, which is the sum of
// the system tick value jiffies and the timing value, ie 'jiffies + HZ* seconds', where
// the constant HZ = 100. The main operation of this function is to set the alarm field
// and convert between two time units.
int sys_alarm(long seconds)
{
        int old = current->alarm;

        if (old)
                old = (old - jiffies) / HZ;
        current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
        return (old);
}

// Get current process pid.
int sys_getpid(void)
{
        return current->pid;
}

// Get parent pid - ppid.
int sys_getppid(void)
{
        return current->p_pptr->pid;
}

// Get current user id.
int sys_getuid(void)
```

```
{
        return current->uid;
}

// Get effective user id - euid.
int sys_geteuid(void)
{
        return current->euid;
}

// Get group id - gid
int sys_getgid(void)
{
        return current->gid;
}

// Get effective group id  -  egid.
int sys_getegid(void)
{
        return current->egid;
}

// System call function -- Reduce the priority of using the CPU (someone will use it?).
// parameter increment should be limited to a value greater than 0.
int sys_nice(long increment)
{
        if (current->priority-increment>0)
                current->priority -= increment;
        return 0;
}

// The initialization subroutine of the kernel scheduler.
void sched_init(void)
{
        int i;
        struct desc_struct * p;        // descriptor structure pointer

// At the beginning of Linux development, the kernel was not mature. The kernel code is
// often modified. Mr. Linus feared that he had inadvertently modified these critical
// data structures, causing incompatibility with the POSIX standard, so add the
// following statement here. This is not necessary, it is purely to remind himself and
// others who modify the kernel code.
        if (sizeof(struct sigaction) != 16)      // signal struct
                panic("Struct sigaction MUST be 16 bytes");

// The task state segment (TSS) descriptor and the local data table (LDT) descriptor of
// the initial task (task 0) are set in the global descriptor table (GDT).
// The value of FIRST_TSS_ENTRY and FIRST_LDT_ENTRY is 4 and 5 respectively, defined in
// file linux/sched.h. gdt is a descriptor array (linux/head.h), it associated with the
// base address _gdt in file head.s, line 234. Therefore, gdt + FIRST_TSS_ENTRY is
// gdt[FIRST_TSS_ENTRY] (ie gdt[4]), which is the address of item 4 of the gdt array.
// See asm/system.h, line 65.
        set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
        set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
// Clear the task array and descriptor table entries (note that starting with i=1, so the
// descriptor for the initial task is still there).
        p = gdt+2+FIRST_TSS_ENTRY;
        for(i=1;i<NR_TASKS;i++) {
                task[i] = NULL;
                p->a=p->b=0;
                p++;
                p->a=p->b=0;
                p++;
        }
/* Clear NT, so that we won't have troubles with that later on */
// The NT flag in EFLAGS is used to control nested calls to tasks. When NT is set, the
// current interrupt task will cause a task switch when the IRET instruction is executed.
// NT indicates whether the back_link field in the TSS is valid. Invalid when NT=0.
        __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");     // reset NT

// The TSS segment selector of task 0 is loaded into the task register (TR). The LDT
// segment selector is loaded into the local descriptor table register (LDTR). Note! The
// selector of the corresponding LDT descriptor in the GDT is loaded into the LDTR. It
// only explicitly loads this time. Later, the loading of the new task LDT is
// automatically loaded by the CPU according to the LDT entry in the TSS.
        ltr(0);                         // include/linux/sched.h, 157-158
        lldt(0);                        // 0 is task no.

// The following code is used to initialize the 8253 timer. Channel 0, select working
// mode 3, binary counting mode. The output pin of channel 0 is connected to the IRQ0 of
// the interrupt control master chip, which issues an IRQ0 request every 10 milliseconds.
// LATCH is the initial timing count value.
        outb_p(0x36,0x43);              /* binary, mode 3, LSB/MSB, ch 0 */
        outb_p(LATCH & 0xff , 0x40);    /* LSB */
        outb(LATCH >> 8 , 0x40);        /* MSB */

// Set the timer interrupt handler. Modify the interrupt controller mask code to enable
// timer interrupt occurs. Then set system-call interrupt gate. The macro definitions of
// the descriptors are at lines 33 and 39 of file asm/system.h.
        set_intr_gate(0x20,&timer_interrupt);
        outb(inb_p(0x21)&~0x01,0x21);          // change int mask, enable timer.
        set_system_gate(0x80,&system_call);
}

```

8.6.3 Information

8.6.3.1 Floppy Drive Controller

For the application in the above program, only the I/O port used by the floppy disk controller (FDC) is briefly introduced here. For a detailed description of FDC programming, see the explanations in Chapter 9 after floppy.c. Four ports need to be accessed when programming the FDC. These ports correspond to one or more registers on the controller. For a normal floppy disk controller there are some ports shown in Table 8-3.

Table 8-3 Floppy drive controller ports

I/O port	Port name	Read/Write	Register name

0x3f2	FD_DOR	Write only	Digital output register (digiter controller register)
0x3f4	FD_STATUS	Read only	FDC main status register
0x3f5	FD_DATA	Read/Write	FDC data register
0x3f7	FD_DIR	Read only	Digital input register
0x3f7	FD_DCR	Write only	Drive control register (transfer rate control)

The digital output register DOR (or digital control) is an 8-bit register that controls driver motor turn-on, driver select, start/reset FDC, and enable/disable DMA and interrupt requests.

The FDC's main status register is also an 8-bit register that reflects the basic state of the FDC and floppy disk drive FDD. Typically, the status bits of the main status register are read before the CPU sends a command to the FDC or before the FDC obtains the result of the operation to determine if the current FDC data register is ready and to determine the direction of data transfer.

The data port of the FDC corresponds to multiple registers (write-only command register and parameter register, read-only result register), but only one register can appear on data port 0x3f5 at any one time. When accessing a write-only register, the DIO direction bit of the main state control must be 0 (CPU -> FDC), and vice versa when accessing the read-only register. When reading the result, the result is only read after the FDC is not busy. Usually, the result data has a maximum of 7 bytes.

The floppy disk controller can accept a total of 15 commands. Each command goes through three phases: the command phase, the execution phase, and the results phase.

The command phase is that the CPU sends command bytes and parameter bytes to the FDC. The first byte is always the command byte (command code) followed by a parameter of 0-8 bytes.

The execution phase is the operation specified by the FDC execution command. In the execution phase, the CPU does not intervene. Generally, the FDC issues an interrupt request to know the end of the command execution. If the FDC command sent by the CPU is to transfer data, the FDC can be performed in an interrupt mode or in a DMA manner. The interrupt mode transfers 1 byte at a time. The DMA mode is under the management of the DMA controller, and the FDC and the memory transfer data until all the data is transmitted. At this time, the DMA controller notifies the FDC of the transmission byte count termination signal, and finally the FDC issues an interrupt request signal to inform the CPU that the execution phase is over.

The result phase is that the CPU reads the FDC data register return value to obtain the result of the FDC command execution. The result data returned is 0--7 bytes in length. For commands that do not return result data, the FDC should be sent a detect interrupt status command to get the status of the operation.

8.6.3.2 Programmable Timer/Counter Controller

1. Intel 8253 (8254) chip

The Intel 8253 (or 8254) is a programmable Timer/Counter chip that solves the time control problems typically encountered in computers, ie, produces precise time delays under software control. The chip provides three independent 16-bit counter channels. Each channel can work in different operating modes, and these can be set using software. The 8254 is an updated product of the 8253 chip. The main functions are basically the same, except that the 8254 chip adds a readback command. In the following description, we use 8253 to refer to the 8253 and 8254 chips, and only point out where they differ in their functions.

The programming of the 8253 chip is relatively simple and can produce the desired delays of various lengths of time. A block diagram of the 8253 (8254) chip is shown in Figure 8-8.

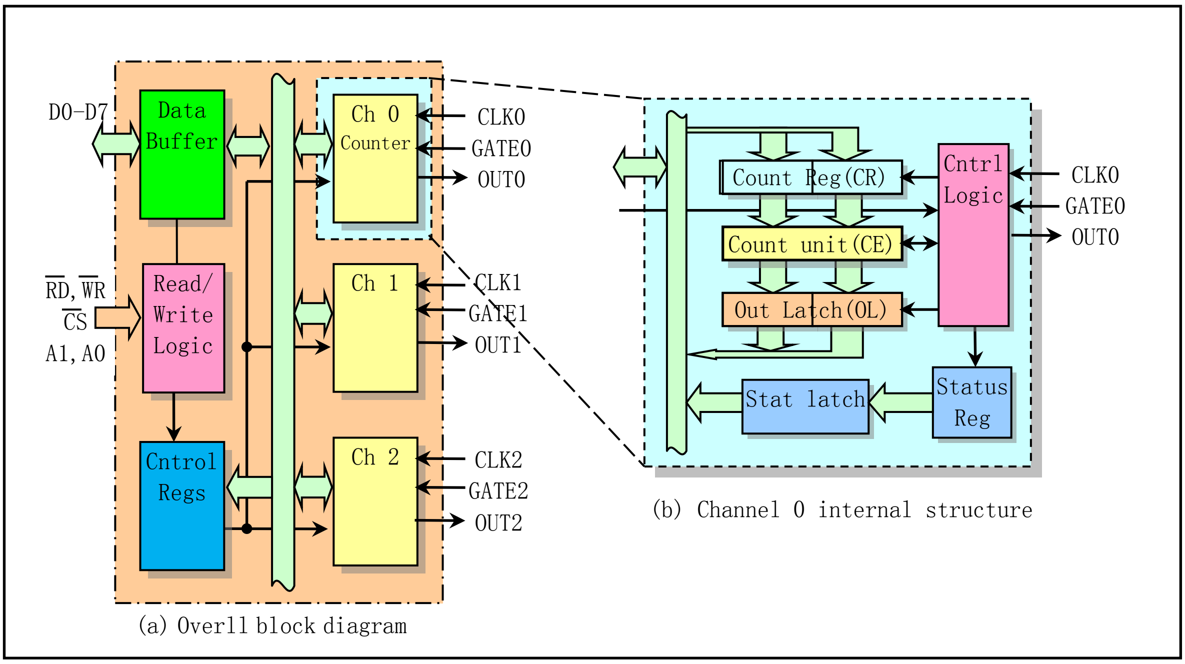


Figure 8-8 8253 (8254) Timer/Counter chip internal structure

The 3-state, bidirectional 8-bit Data Bus Buffer is used to interface with the system data bus. Read/Write Logic is used to receive input signals from the system bus and generate control signals that are input to other parts. Address lines A1, A0 are used to select one of the three counter channels or Control Word Registers that need to be read/written. Usually they are connected to the A0, A1 address line of the system. The read and write pins RD, WR and chip select pin CS are used by the CPU to control the read and write operations of the 8253 chip. The Control Word Register is used by the CPU to set how the specified counter works. It is a write-only register. But for the 8254 chip, you can use the Read-Back Command to read the status information. The three independent counter channels function exactly the same, each of which can work in different ways. The Control Word register will determine how each counter works. The CLK pin of each counter is connected to the clock frequency generator (crystal oscillator). The 8253 has a clock input frequency of up to 2.6MHz, while the 8254 can be up to 10MHz. Pin GATE is the gate control input of the counter, which is used to control the start and stop of the counter and the output state of the counter. Pin OUT is the output signal terminal of the counter.

Figure 8-8(b) is an internal logic block diagram of one of the counter channels. The status register will contain the current contents of the control word register and the status of the output and the Null Count Flag when locked. The actual counter is the CE (counting unit) in the figure. It is a 16-bit pre-settable synchronous down counter. The output latch OL (Output Latch) is composed of two 8-bit latches, OLm and OL1, which represent the high byte and low byte of the latch, respectively. Usually the contents of the two output latches change following the content change of the counting unit CE, but if the chip receives a counter latch command, then their contents will be locked. Until the CPU reads their contents, they will continue to follow the CE content changes. Note that the value of CE is unreadable. Whenever you need to read the count value, the contents of the latch OL are always output. The other two in Figure 8-8(b) are 8-bit registers called the Count Register (CR). When the CPU writes a new count value to the counter channel, the initial count value is stored in these two registers and then copied to the count unit CE. These two registers will be cleared when the counter is programmed. Therefore, after the initial count value is saved in the count register CR, it is sent to the counting unit CE. When GATE is enabled, the counting unit performs a countdown operation under the action of the clock pulse CLK. Each time it is decremented by one, until the count value is decremented to zero, a signal is sent to the OUT pin.

2. 8253 (8254) chip programming

When the system is just powered up, the state of the 8253 is unknown. By writing a control word and an initial count value to the 8253, we can program a counter we want to use. For counters that are not used we don't have to program them. Table 8–4 shows the format of the contents of the control registers.

Table 8-4 8253 (8254) chip control word format

Bit	Name	Description
7	SC1	SC1, SC0 are used to select counter channel 0-2, or to read back the command. 00 - Channel 0; 01 - Channel 1; 02 - Channel 2; 11 - Readback command (only in 8254).
6	SC0	
5	RW1	RW1 and RW0 are used for counter read/write operation selection. 00 - indicates a register latch command; 01 - Reads/writes the low byte (LSB); 10 - Read/write high byte (MSB); 11 - Read/write low byte first, then high byte.
4	RW0	
3	M2	M2-M0 is used to select the working mode of the specified channel. 000 - mode 0; 001 - mode 1; 010 - mode 2; 011 - mode 3; 100 - mode 4; 101 - mode 5.
2	M1	
1	M0	
0	BCD	Count value format selection. 0 - 16 bit binary count; 1 - 4 BCD code counts.

When the CPU performs a write operation, if the A1 and A0 lines are 11 (in this case, the corresponding port 0x43 on the PC microcomputer), the control word is written into the control word register. The contents of the control word specify the counter channel being programmed. The initial count value is written to the specified counter. When A1 and A0 are 00, 01, and 10 (corresponding to PC ports 0x40, 0x41, and 0x42, respectively), one of the three counters is selected. In a write operation, the control word must be written first and then the initial count value. The initial count value must be written in the format set in the control word (binary or BCD code format). When the counter starts working, we can still rewrite the new initial value to the specified counter at any time. This does not affect how the counters that have been set work.

During the read operation, there are three ways to read the counter's current count value for the 8254 chip: (1) simple read operation; (2) use counter latch command; (3) use readback command. The first method must temporarily stop the clock input of the counter using the GATE pin or the corresponding logic circuit during reading. Otherwise the counting operation may be in progress, resulting in an incorrect result of the reading. The second method is to use the counter latch command. The command is first sent to the control word register before the read operation, and the two bits (00) of D5, D4 indicate that the counter latch command is sent instead of the control word command. When the counter receives the command, it latches the count value in the counting unit CE into the output latch register OL. At this point, if the CPU does not read the contents of the OL, the value in the OL will remain the same, even if you send another counter latch command. Only after the CPU performs the reading of the counter operation, the contents of the OL will automatically follow the counting unit CE to change. The third method is to use the readback command. But only the 8254 has this feature. This command allows the program to detect the current count value, how the counter is run, and the current output status and NULL count flag. Similar to the second method, after the count value is locked, the contents of the OL will automatically follow the counting unit CE again after the CPU performs the reading of the counter operation.

3. Counter working mode

The 3 counter channels of 8253/8254 can work independently. There are 6 ways to choose from.

(1) Mode 0 - Interrupt on terminal count

After this mode is set, the output pin OUT is low and remains low until the count is decremented to zero. At this time, OUT goes high and stays high until a new count value is written or the control word is reset to mode 0. This method is usually used for event counting. This mode is characterized by the use of the GATE pin to control the count pause; the output goes high at the end of the count as an interrupt signal; the initial count value can be reloaded during the count and re-executed after the count high byte is received.

(2) Mode 1 - Hardware Retriggerable One-shot

When working in this mode, OUT is initially at a high level. The counter is ready after the CPU has written the control word and the initial count value. At this point, the GATE pin rising edge triggers the counter to start operation and OUT turns low. Until the end of the count (0), OUT goes high. During the counting period or after the counting is completed, GATE will go high again and trigger the counter to load the initial count value and restart the counting operation. For this mode of operation, the GATE signal does not work.

(3) Mode 2 - Rate Generator

The function of this mode is similar to an N divider. Usually used to generate real-time clock interrupts. OUT is high in the initial state. When the count value is decremented to 1, OUT goes low and then goes high. The interval is one CLK pulse width. At this point the counter will reload the initial value and repeat the above process. Therefore, for the case where the initial count value is N, a low-level pulse signal is output every N clock pulses. In this way GATE can control the pause and continuation of the count. When GATE goes high, the counter is reloaded with the initial value and begins to recount.

(4) Mode 3 - Square Wave Mode

This method is usually used for the baud rate generator. This mode is similar to mode 2, but the OUT output is a square wave. If the initial count value is N, the frequency of the square wave is one-N of the input clock CLK. The characteristic of this mode is that the square wave duty cycle is about 1 to 1 (slightly different when N is odd), and if the new initial value is reset during the counter decrement, the new initial value takes effect only after the previous count is completed.

(5) Mode 4 - Software Triggered Strobe

OUT is high in the initial state. When the count ends, OUT will output a low level of the clock pulse width and then go high (low level gate). The counting operation is "triggered" by writing the initial count value. In this mode of operation, the GATE pin can control the count pause (1 allow count), but does not affect the state of OUT. If a new initial value is written during the counting process, the counter will use the new value to recount after one clock pulse.

(6) Mode 5 - Hardware Triggered Strobe

OUT is high in the initial state. The counting operation will be triggered by the rising edge of the GATE pin. When the count is over, OUT will output a low level of the clock CLK pulse width and then go high. After writing the control word and the initial value, the counter does not immediately load the initial count value and starts working. It will only be triggered to start operation after a CLK clock pulse after the GATE pin goes high.

For PC/AT and its compatible microcomputer system, the 8254 chip is used. Three timer/counter channels are used for the clock timing interrupt signal, the dynamic memory DRAM refresh timing circuit, and the host speaker tone synthesis. The input clock frequencies of the three counters are all 1.193180MHz. The connection diagram of the 8254 chip in the PC/AT microcomputer is shown in Figure 8-9. The A1 and A0 pins are connected to the system address lines A1 and A0, and the 8254 chip is selected when the system address line A9--A2 signal is 0b0010000. Therefore, the I/O port range of 8254 chip is 0x40--0x43. Among them, 0x40--0x42 corresponds to select counter channel 0--2, and 0x43 corresponds to the control word register write port.

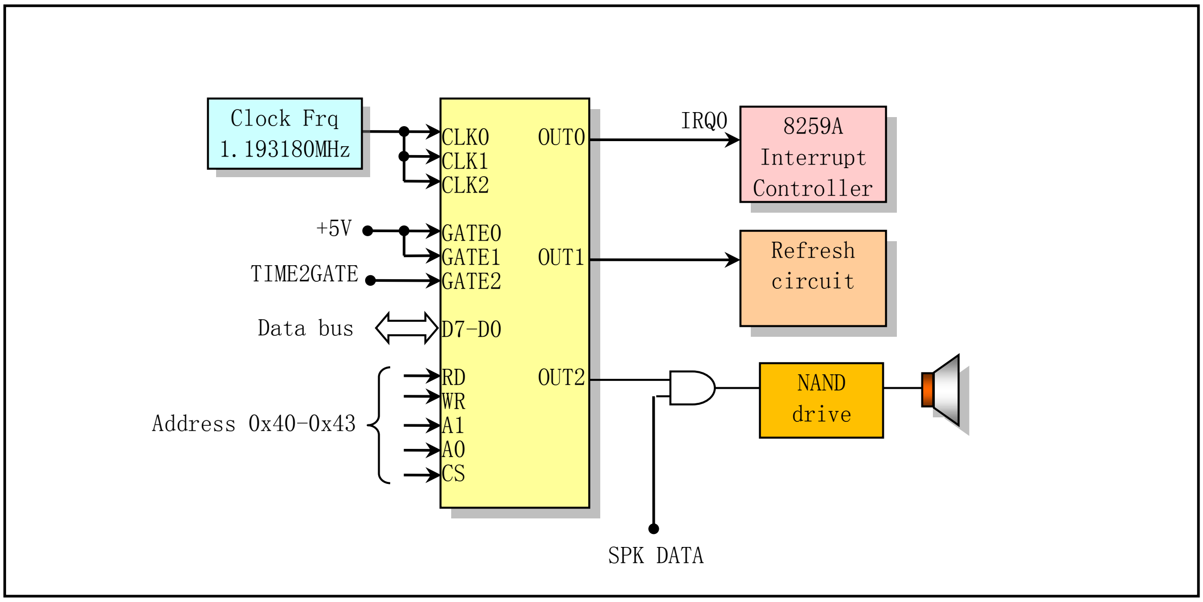


Figure 8-9 Timer/counter chip connection diagram in PC

For counter channel 0, its GATE pin is fixed high. When the system is powered on, it is set to work in mode 3 (square wave generator mode), and the initial count value is set to 0 by default, which means that the count value is 65536 (0--65535). Therefore, the OUT0 pin emits a square wave signal with a frequency of 18.2HZ (1.193180MHz/65536) every second. OUT0 is connected to the level 0 interrupt requester of the programmable interrupt controller 8259 chip. Therefore, using the rising edge of the square wave can trigger an interrupt request, causing the system to issue an interrupt request every 54.9ms (1000ms/18.2).

The GATE pin of counter channel 1 is also directly connected to the high level, so it is in the allowable count state. It works in mode 2 (frequency generator mode) and the initial value is usually set to 18. This counter is used to send a RAM refresh signal to the refresh circuit of the DMA controller channel 2 of the PC/XT system or the PC/AT system. A signal is output approximately every 15 microseconds with an output frequency of 1.19318/18 = 66.288 KHz.

The GATE pin (TIME2GATE) of counter channel 2 is connected to the D0 pin of the 8255A chip port B or equivalent logic. The SPK DATA in Figure 8-9 is connected to the D1 pin or equivalent logic of the 8255A chip port B (0x61). This counter channel is used to make the main speaker sound, but it can also be used as a normal timer with the 8255A chip (or its equivalent).

In Linux 0.12, the kernel only reinitializes counter channel 0 of the 8254 so that the counter operates in mode 3, the initial count value is binary, and the initial count value is set to LATCH (1193180/100). That is, the counter 0 sends a square wave rising edge signal every 10 milliseconds to generate an interrupt request signal (IRQ0). Therefore, the control word written to the 8254 is 0x36 (0b00110110), and then the low byte and the high byte of the initial count value are written. The initial count value low byte and high byte value are (LATCH & 0xff) and (LATCH >> 8), respectively. The interrupt request generated by this interval timing is the pulse of the Linux 0.12 kernel working. It is used to periodically switch the currently executing tasks, count the amount of system resources (time) used by each task, and implement kernel timing operations.

8.7 signal.c

8.7.1 Function Description

The signal.c program involves all the functions related to signal processing in the kernel. In UNIX-like systems, signals are a "software interrupt" processing mechanism. There are many more complicated programs that use signals. The signaling mechanism provides a way to handle asynchronous events, which can be used as a simple message mechanism for communication between processes, allowing one process to send signals to another. The signal is usually a positive integer, which does not carry any other information except to indicate its own signal class. For example, when a child process terminates or ends, a SIGCHILD signal is sent to the parent process to inform it about the current state of the child process; use the system function kill() to send termination execution signal to all child processes in the same group; typing ctrl-C will generate a SIGINT signal that is sent to the foreground process to terminate it. In addition, when an alarm timer set by the process expires, the system sends a SIGALRM signal to the process; when a hardware exception occurs, the system also sends a corresponding signal to the executing process.

Signal processing mechanisms existed in very early UNIX systems, but the methods of signal processing in earlier UNIX kernels were not as reliable. Signals may be lost, and it is sometimes difficult for a process to close a specified signal while processing a critical area code. Later POSIX provided a way to reliably process signals. In order to maintain compatibility, the Linux kernel provides both methods for processing signals.

8.7.1.1 Signals in Linux

In Linux kernel code, bits in an unsigned long integer (32 bits) are typically used to represent a variety of different signals, so there can be up to 32 different signals in the system. In this version of the Linux kernel, 22 different signals are defined. 20 of these signals are those specified in the POSIX.1 standard, and the other 2 are Linux-specific signals: SIGUNUSED (undefined) and SIGSTKFLT (stack error). The former can represent all other signal types that the system does not currently support. For the specific names and definitions of these 22 signals, please refer to Table 8-4 of the signal list after the program. Also refer to the contents of the include/signal.h header file.

When a process receives a signal, there are three different ways of processing or operation: one is to ignore the signal, but two signals cannot be ignored (SIGKILL and SIGSTOP). The second approach is that the process defines its own signal handler to process the signal. The third is to perform the system's default signal processing operations.

1. Ignore this signal. Most signals can be ignored by the process. But there are two signals that can't be ignored: SIGKILL and SIGSTOP. The reason is to give the superuser a definite way to terminate or stop any process specified. In addition, if the signal generated by some hardware exceptions is ignored (for example, divided by 0), the behavior or state of the process may become agnostic.
2. Capture the signal. In order to perform the capture operation, we must first tell the kernel to call our custom signal handler when the specified signal occurs. In this handler we can do anything. Of course, you can do nothing and play the same role of ignoring the signal. An example of a custom signal processing function is: If we create some temporary files during program execution, then we can define a function to capture the SIGTERM (terminate execution) signal and do some cleanup in the function. The SIGTERM signal is the default signal sent by the kill command.
3. Perform the default action. The process does not process the signal, and the signal is processed by the system's corresponding default signal handler. The kernel provides a default action for each type of signal. Usually these default actions are to terminate the execution of the process. See the description in the

post-program signal list (Table 8-4).

8.7.1.2 Signal Processing Implementation

The signal.c program mainly includes: 1) the access signal blocking code system-calls sys_ssetmask() and sys_sgetmask(), 2) signal handling syscall sys_signal() (the traditional signal handling function), 3) handler for modification signal actions syscall sys_sigaction() (the reliable signal handler), 4) and the function do_signal() that processes the signal in the system call interrupt handler. The send signal function send_sig() and the notification parent process function tell_father() are included in another source file (exit.c). In addition, the name prefix 'sig' in the code is short for signal.

The functions of signal() and sigaction() are similar, and can be used to change the signal handler. However, signal() is the traditional way for the kernel to process signals, which can cause signal loss at certain special times. When the user wants to use his own signal handler (signal handle), the user needs to use the signal() or sigaction() syscall to first set the sigaction[] structure array item in the task's data structure, and put the signal handler and some attributes in the structure for itself. When the kernel returns from a system-call or some interrupt procedures, it will detect if the current process receives a signal. If a specific signal specified by the user is received, the kernel executes the user-defined signal processing service program according to the structure item in sigaction[] in the process task data structure.

1. signal() fuction

On line 62 of the include/signal.h header file, the signal() function signature is declared as follows:

```
    void (*signal(int signr, void (*handler)(int)))(int);
```

This signal() function takes two arguments. One is to specify the signal signr to be captured; the other is the new signal handler pointer void (*handler)(int). This new signal handler is a function pointer with no return value and an integer parameter that is passed to the handler when the specified signal occurs.

The prototype declaration of the signal() function seems complicated, but if we define a type like this:

```
    typedef  void sigfunc(int);
```

Then we can rewrite the prototype of the signal() function to the following simple form:

```
    sigfunc *signal(int signr, sigfunc *handler);
```

The signal() function installs a new signal handler for the signal signer. The signal handler can be a signal processing function specified by the user, or it can be a specific function pointer SIG_IGN or SIG_DFL provided by the kernel. When the specified signal arrives, the signal is ignored if the associated signal processing handler is set to SIG_IGN. If the signal handler is SIG_DFL, then the default operation of the signal is performed. Otherwise, if the signal handler is set to a user's signal handler, the kernel first resets the signal handle to its default handle, performs an implementation-related signal blocking operation, and then invokes the specified signal handler.

The signal() function returns the original signal handler, and the returned handler is also a function pointer with no return value and with an integer argument, and after the new handler is called once, it is restored to the default processing handler value SIG_DFL.

In the include/signal.h file (starting on line 46), the default handle SIG_DFL and the ignore handle SIG_IGN are defined as:

```
    #define SIG_DFL      ((void (*)(int))0)
    #define SIG_IGN      ((void (*)(int))1)
```

They all represent function pointers with no return value, respectively, as required by the second parameter in the signal() function. The pointer values are 0 and 1, respectively. These two pointer values are logically the function address values that are not possible in the actual program. Therefore, in the signal() function, it is possible to judge whether to use the default signal processing handle or ignore the processing of the signal based on the two special pointer values. Of course, SIGKILL and SIGSTOP cannot be ignored. See the processing statements on lines 155-162 in the program listing below for related code.

When a program starts executing, the system sets its way to process all signals as SIG_DFL or SIG_IGN. In addition, when the program fork() a child process, the child process inherits the signal processing mode (signal mask) of the parent process. Therefore, the way the parent process sets and processes the signal is equally valid in the child process.

In order to continuously capture a specified signal, an example of the usual use of the signal() function in the user program is as follows.

```
void sig_handler(int signr)
{
    signal(SIGINT, sig_handler);    // re-install the handler for the next capture.
    ...                             // do something.
}

main ()
{
    signal(SIGINT, sig_handler);    // set signal handler in main.
    ...
}
```

The reason that the signal() function is unreliable is that when the signal has already occurred and enters its own set of signal processing functions, it is possible that another signal will occur during this time before re-setting its own handler. But at this point the system has set the handle to the default value. Therefore, it is possible to cause signal loss.

2. sigaction() function

The sigaction() function uses the sigaction data structure to hold the information of the specified signal. It is a reliable mechanism for the kernel to process signals. It allows us to easily view or modify the processing handle of a given signal. This function is a superset of the signal(). The declaration of this function in the include/signal.h header file (line 73) is:

```
    int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
```

The parameter sig is the signal we need to view or modify the signal processing handler, and the last two parameters are pointers to the sigaction structure. When the parameter act pointer is not NULL, the behavior of the specified signal can be modified according to the information in the act structure. When oldact is not empty, the kernel will return the original settings of the signal in the structure. The sigaction structure is as follows:

```
struct sigaction {
        void (*sa_handler)(int);           // signal handler.
        sigset_t sa_mask;                  // signal mask.
        int sa_flags;
        void (*sa_restorer)(void);         // internal restorer pointer.
};
```

When modifying a signal processing method, if the processing handler sa_handler is not the default SIG_DFL or the ignore handler SIG_IGN, then before the sa_handler can be called, the sa_mask field specifies one signal set that needs to be added to the process signal mask bitmap. If the signal handler returns, the system will restore the original signal mask bitmap of the process. This way we can block some of the specified signals when a signal handler is called. When the signal handler is called, the new signal mask bitmap automatically includes the currently transmitted signal, blocking the continued transmission of the signal. Thus, we can ensure that the same signal is blocked without being lost during the processing of a specified signal until the processing is completed. In addition, when a signal is blocked and occurs multiple times, usually only one sample is saved, that is, when the blocking is released, the signal processing handler is called again only once for the same multiple signals that are blocked. After we modify the processing handler of a signal, the handler is used until it is changed again. This is not the same as the traditional signal() function. The signal() function will restore it to the default handler of the signal after the handler processing has finished.

The sa_flags in the sigaction structure are used to specify other options for processing signals. These options are used to change some of the default processes in the signal handling. For their definitions, see the description in the include/signal.h file (lines 37-40). .

```
  // The symbol constant value that the sigaction structure sa_flags field can take.
37 #define SA_NOCLDSTOP    1             // ignore SIGCHLD if child stopped.
38 #define SA_INTERRUPT    0x20000000    // not restart syscall after interrupt by sig.
39 #define SA_NOMASK       0x40000000    // do not mask the current signal.
40 #define SA_ONESHOT      0x80000000    // restore to defulat handler after finished.
```

The last field in the sigaction structure and the parameter restorer of the sys_signal() function are both function pointers. It is provided by the libc library when compiling and linking programs, to clean up the user-mode stack after the signal handler ends, and to restore the return value of the system-call stored in eax, as detailed below.

3. do_signal() function

The do_signal() function is a preprocessor for the signal in the kernel system-call (int 0x80) interrupt handler. Each time a process calls a system-call or a timer interrupt occurs, if the process has received a signal, the function inserts the signal's processing handle (ie, the signal handler) into the user program stack. In this way, the signal handler is executed immediately after the current system-call returns, and then the user's program is executed, as shown in Figure 8-10.

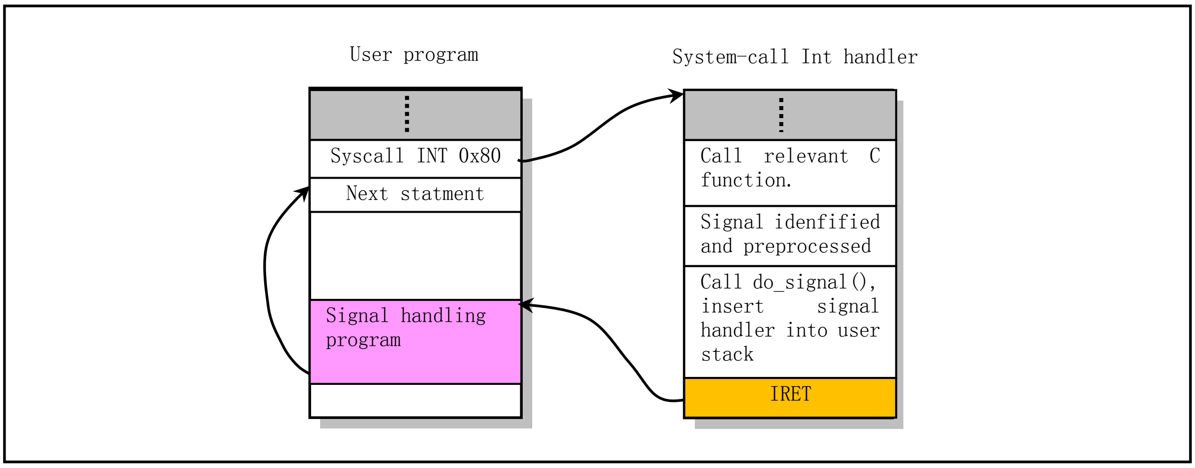

User program
System-call Int handler
Syscall INT 0x80
Next statment
Signal handling program
Call relevant C function.
Signal idenfified and preprocessed
Call do_signal(), insert signal handler into user stack
IRET

Figure 8-10 How the signal handler is called.

Before inserting the parameters of the signal handler into the user stack, the do_signal() function first expands the user program stack pointer down longs (see line 195 in the program below) and then adds the relevant parameters to it. See Figure 8-11. Since the code starting with line 193 of the do_signal() function is hard to understand, we will describe it in detail below.

When the user program calls a system-call just entering the kernel, the kernel state stack of the process is automatically pushed into the content by the CPU, as shown in Figure 8-11. That is: the SS, ESP and the CS and EIP of the next instruction in the user program. After processing the specified system-call and preparing to call do_signal() (that is, after the 124 lines in file sys_call.s), the contents of the kernel state stack are shown in the left side of Figure 8-12. So the arguments to do_signal() is what are on the kernel-state stack.

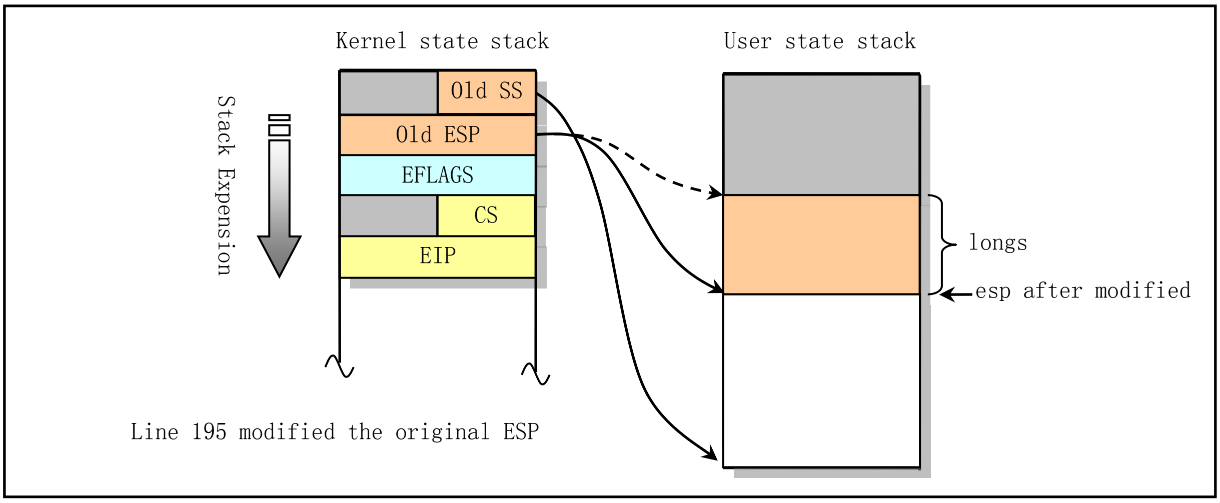


Figure 8-11 Modification of the user stack by the do_signal()

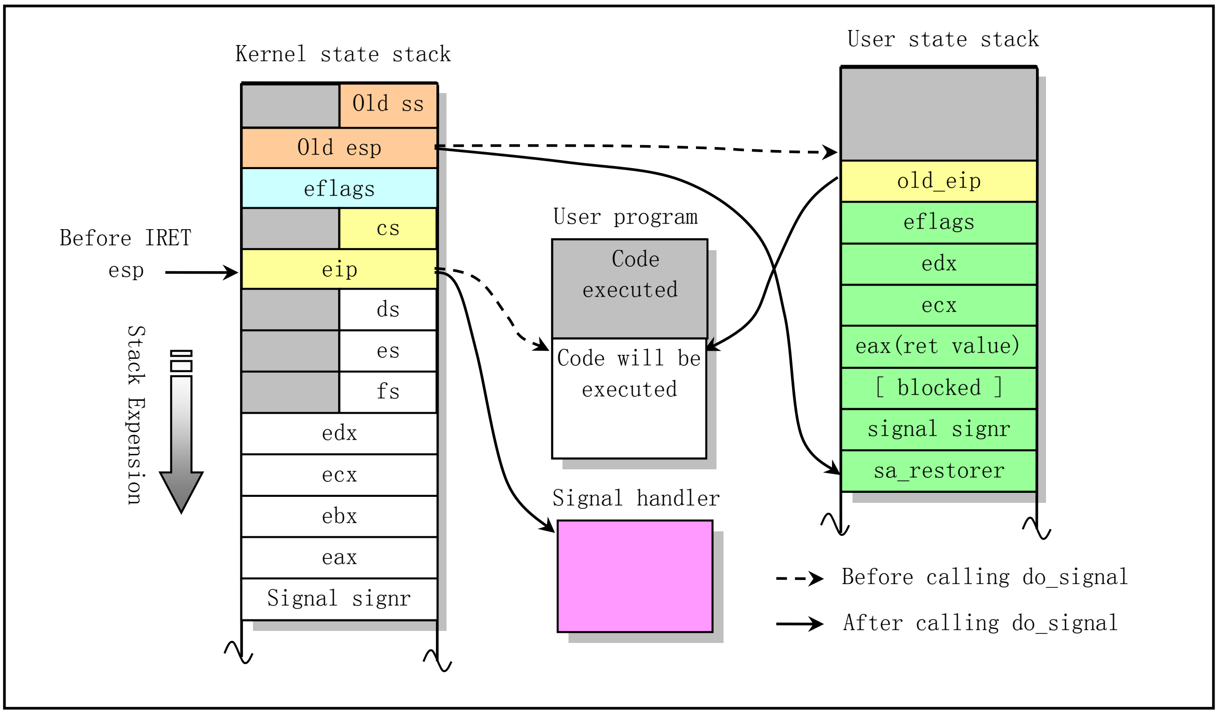


Figure 8-12 The specific process of modifying the user state stack

After do_signal() determines and processes the two default signal handlers (SIG_IGN and SIG_DFL), if the user customizes the signal handler, then from the 193th line, do_signal() starts to prepare inserting the user-defined handler into the user state stack. It first saves the return execution point pointer eip of the original user program in the kernel state stack as old_eip in user stack, and then replaces the eip with the custom handler sa_handler, that is, the eip in the kernel state stack in the figure points to sa_handler. Next, the user state stack is extended downward by 7 or 8 long words by subtracting the longs value from the "original esp" saved in the kernel state. Finally, some of the register contents on the kernel stack are copied into this space, as shown in the right side of Figure 8-12.

The kernel code puts a total of 7 to 8 values onto the user state stack. Let us now explain what these values mean and why they are placed. old_eip is the return address of the original user program, which is reserved before the eip is replaced with the signal handler address on the kernel stack. eflags, edx, and ecx are the values of the original user program before the system-call is invoked. They are basically the parameters used by the system-call. After the system-call returns, these register values of the user programs still need to be restored. The return value of the system-call is stored in eax. If the processed signal also allows itself to be received, the blocked code blocked for that process is also stored on the stack. The next one is the signal signr.

The last one is the pointer sa_restorer of the signal activity recovery function. This recovery function is not set by the user because only one signal value signr and one signal handler are provided when the user defines the signal() function.

The following is a simple example of setting a custom signal processing handler for the SIGINT. By default, pressing the Ctrl-C key combination will generate a SIGINT signal.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void handler(int sig)                        // user defined signal handler.
```

```
{
    printf("The signal is %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);          // restore default handler of SIGINT
}                                            // some system will restored automatically

int main()
{
    (void) signal(SIGINT, handler);          // set user defined  handler for SIGINT.
    while (1) {
        printf("Signal test.\n");
        sleep(1);                            // wait a second.
    }
}
```

Among them, the signal handler function () will be called when the signal SIGINT appears, and then return to the main program to continue execution. The function first outputs a message and then sets the processing of the SIGINT signal to the default signal handler. So when you press the Ctrl-C key combination a second time, SIG_DFL will let the program finish running.

4. sa_restorer function

So, where does the sa_restorer function come from? In fact, it is provided by the function library. This function is available in the Linux Libc-2.2.2 library file (misc/subdir) and is defined as follows:

```
.globl ____sig_restore
.globl ____masksig_restore
# Use this restorer function if there is no blocked
____sig_restore:
    addl $4,%esp           # discard the signal signr
    popl %eax              # restore system-call ret value in eax
    popl %ecx              # restore user original registers
    popl %edx
    popfl                  # restore user eflags.
    ret
# If there is blocked, use the following restorer function. Blocked for use by ssetmask.
____masksig_restore:
    addl $4,%esp           # discard signal signr
    call ____ssetmask      # set signal mask old blocking
    addl $4,%esp           # discard blocked.
    popl %eax
    popl %ecx
    popl %edx
    popfl
    ret
```

The main purpose of this function is to restore the return value and some register contents after the user program executes the syscall after the signal handler ends, and clear the signal value signr. When compiling a user-defined signal handler, the compiler invokes the signal syscall in the libc library to insert the sa_restorer() function into the user program. The function implementation of the signal syscall in the library file is shown below.

```
#define __LIBRARY__
#include <unistd.h>

extern void ___sig_restore();
extern void ___masksig_restore();

// The signal() wrapper function called by the user in the library.
void (*signal(int sig, __sighandler_t func))(int)
{
        void (*res)();
        register int __fooebx __asm__ ("bx") = sig;
        __asm__("int $0x80":"=a" (res):
        "0" (__NR_signal),"r" (__fooebx),"c" (func),"d" ((long)___sig_restore));
        return res;
}

// The sigaction() function called by the user.
int sigaction(int sig,struct sigaction * sa, struct sigaction * old)
{
        register int __fooebx __asm__ ("bx") = sig;
        if (sa->sa_flags & SA_NOMASK)
                sa->sa_restorer=___sig_restore;
        else
                sa->sa_restorer=___masksig_restore;
        __asm__("int $0x80":"=a" (sig)
                :"0" (__NR_sigaction),"r" (__fooebx),"c" (sa),"d" (old));
        if (sig>=0)
                return 0;
        errno = -sig;
        return -1;
}
```

The sa_restorer() is responsible for cleaning up the register value of the user program and the return value of the system-call after the signal handler is executed, as if it had not run the signal handler, but returned directly from the system-call.

Finally, let us explain the flow of the signal being processed. After do_signel() is executed, sys_call.s will pop the stack all values below the eip on the process kernel state stack. After executing the IRET instruction, the CPU will pop the cs:eip, eflags, and ss:esp on the kernel state stack, and return to the user state to execute the program. But since the eip has now been replaced with a pointer to the signal handler, the user-defined signal handler is executed immediately. After the signal handler is executed, the CPU will transfer control to the recovery program pointed to by sa_restorer through the RET instruction. The sa_restorer program will do some user-level stack cleanup, which will skip the signal value signr on the stack and pop the return value eax and the registers ecx, edx and the flag register eflags after the system-call. The state of each register and CPU after the system call is completely restored. Finally, the eip of the original user program (that is, old_eip on the stack) is popped up by the RET instruction of sa_restorer, and the user program is returned to execute.

8.7.1.3 Process suspension

The signal.c program also includes a sys_sigsuspend() syscall implementation that temporarily replaces the process signal mask with the given set in the argument and then suspends the process until a signal is received. The syscall is declared as a signature with three parameters:

```
int sys_sigsuspend(int restart, unsigned long old_mask, unsigned long set)
```

Where restart is a restart indicator. If it's 0, then we save the current mask in the oldmask, and then block the process until we receive a signal; if it's non-zero, then we restore the original mask from the saved oldmask, and return normally.

Although the syscall has three parameters, the general user program will call through the library when using it. This function in the library only uses a form with a set parameter:

```
int sigsuspend(unsigned long set)
```

This is the form in which the syscall is used in the C library. The first two parameters will be processed by the sigsuspend() library function. The general implementation of this library function is similar to the following code:

```
#define __LIBRARY__
#include <unistd.h>

int sigsuspend(sigset_t *sigmask)
{
        int res;

        register int __fooebx __asm__ ("bx") = 0;
        __asm__("int $0x80"
                :"=a" (res)
                :"0" (__NR_sigsuspend), "r" (__fooebx), "c" (0), "d" (*sigmask)
                :"bx","cx");
        if (res >= 0)
                return res;
        errno = -res;
        return -1;
}
```

Here, the register variable __fooebx is the above 'restart'. When the syscall is called for the first time, it is 0, and the original blocking code is saved (old_mask), and 'restart' is set to a non-zero value. So when the process calls the syscall for the second time, it will restore the blocking code that the process originally saved in old_mask.

8.7.1.4 Restart of a system-call interrupted by signal

If a process receives a signal while it is blocked while executing a slow system-call, the system-call is interrupted by the signal and no longer continues. At this point, the system-call will return an error message, and the corresponding global error code variable errno is set to EINTR, indicating that the system call is interrupted by a signal. For example, when reading or writing pipes, terminal devices or network devices, if the read data does not exist or the device cannot accept the data immediately, the calling program of the system-call will be blocked all the time. Therefore, for some slow system-calls, signals can be used to interrupt them and return to the user program when necessary. This also includes system-calls such as pause() and wait().

However, in some cases it is not necessary for the user program to personally handle the system-call that was interrupted by the signal. Because sometimes the user does not know if the device is a low speed device. If the program can run interactively, it may read and write low-speed devices. If the signal is captured in such a program and the system does not provide an automatic restart function for the system-call, then the program needs to detect the error return code each time the system-call is read or written. If it is interrupted by a signal, it needs to be read and written again. For example, when performing a read operation, if it is interrupted by a signal, then in order for it to continue the read operation, the user will be asked to write the following code:

```
again:
       if (( n = read(fd, buff, BUFFSIZE)) < 0 ) {
                if (errno == EINTR)
                        goto again;          /* an interrupted syscall */
        }
```

In order to prevent the user program from having to deal with certain interrupted system-call situations, a restart (re-execute) function for some interrupted system-calls is introduced while processing the signal. System-calls that are automatically restarted include: ioctl, read, write, wait, and waitpit. The first three system-calls are interrupted by the signal only when operating on low-speed devices, while wait and waitpid are always interrupted when the signal is captured.

When handling a signal, depending on the flag set in the sigaction structure, it is possible to select whether to restart the interrupted system-call. In the Linux 0.12 kernel, if the SA_INTERRUPT flag is set in the sigaction structure (the system-call can be interrupted) and the relevant signals are not SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU, the system-call will be interrupted when a signal is received. Otherwise the kernel will automatically re-execute the interrupted system-call. The method of execution is to first restore the value of the original register eax when the system-call is invoked, and then subtract two bytes from the eip of user program, that is, let eip redirect to the system-call int 0x80 instruction.

For the current Linux system, the flag SA_INTERRUPT has been discarded. Instead, the flag SA_RESTART, which has the opposite meaning, is required to restart the interrupted system-call after the signal handler is executed.

8.7.2 Code Comments

Program 8-6 linux/kernel/signal.c

```
/*
 *  linux/kernel/signal.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <linux/sched.h> The scheduler header file defines the task structure task_struct, the
//     data of the initial task 0, and some embedded assembly function macro statements
//     about the descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
//     commonly used functions of the kernel.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//     descriptors/interrupt gates, etc. is defined.
// <signal.h> Signal header file. Define signal symbol constants, signal structures, and
//      signal manipulation function prototypes.
// <errno.h> Error number header file. Contains various error numbers in the system.
//      (Linus was introduced from minix).
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>

#include <signal.h>
#include <errno.h>

// Get the current task signal mask bitmap (mask, block code). The abbreviation 'sgetmask'
// can be broken down into 'signal-get-mask'.
int sys_sgetmask()
{
        return current->blocked;
}

// Set a new signal mask bitmap. The signals SIGKILL and SIGSTOP cannot be masked.
// The return value is the original signal mask bitmap.
int sys_ssetmask(int newmask)
{
        int old=current->blocked;

        current->blocked = newmask & ~(1<<(SIGKILL-1)) & ~(1<<(SIGSTOP-1));
        return old;
}

// Detects and acquires signals received but blocked. Bitmaps that have not yet processed
// the signal will be placed in the set.
int sys_sigpending(sigset_t *set)
{
    /* fill in "set" with signals pending but blocked. */
// First verify that the user storage space provided should have 4 bytes. The bitmap of
// the signal that has not been handled and blocked is then filled in at the position
// indicated by the set pointer.
    verify_area(set,4);
    put_fs_long(current->blocked & current->signal, (unsigned long *)set);
    return 0;
}

/* atomically swap in the new signal mask, and wait for a signal.
 *
 * we need to play some games with syscall restarting.  We get help
 * from the syscall library interface.  Note that we need to coordinate
 * the calling convention with the libc routine.
 *
 * "set" is just the sigmask as described in 1003.1-1988, 3.3.7.
 *       It is assumed that sigset_t can be passed as a 32 bit quantity.
 *
 * "restart" holds a restart indication.  If it's non-zero, then we
 *       install the old mask, and return normally.  If it's zero, we store
 *       the current mask in old_mask and block until a signal comes in.
 */
```

```
// The syscall temporarily replaces the signal mask with the given set in the parameter
// and then suspends the process until a signal is received.
// restart is an interrupted system-call restart indicator. When the syscall is invoked
// for the first time, it is 0, and the original blocking code is saved (old_mask), and
// restart is set to a non-zero value. Therefore, when the process invokes the syscall
// for the second time, it will restore the blocking code that the process originally
// saved in old_mask.
// The pause() syscall will cause the process that called it to go to sleep until it
// receives a signal. This signal may either terminate the execution of the process or
// cause the process to execute the corresponding signal capture function.
int sys_sigsuspend(int restart, unsigned long old_mask, unsigned long set)
{
    extern int sys_pause(void);

// If the restart flag is not 0, it means that the program is to be re-run. The original
// process blocking code previously saved in the old_mask is restored, and the code -EINTR
// is returned (the system call is interrupted by the signal).
    if (restart) {
        /* we're restarting */
        current->blocked = old_mask;
        return -EINTR;
    }
// Otherwise, the restart flag is 0. Indicates that it is the first call. So first set the
// restart flag (set to 1), save the current blocking code to old_mask, and replace the
// process's blocking code with set. Then call sys_pause() to let the process sleep and
// wait for the signal to arrive. When the process receives a signal, pause() will return,
// and the process will execute the signal handler, then the call returns the -ERESTARTNOINTR
// code and exits. This return code indicates that after the signal is handled, it is
// required to return to the system-call to continue running.
    /* we're not restarting.  do the work */
    *(&restart) = 1;
    *(&old_mask) = current->blocked;
    current->blocked = set;
    (void) sys_pause();                 /* return after a signal arrives */
    return -ERESTARTNOINTR;             /* handle the signal, and come back */
}

// Copy the sigaction data to the fs data segment to. That is, copy from the kernel space
// to the user (task) data segment.
// First verify that the mem space at variable to is large enough. Then copy the sigaction
// structure data into the fs segment (user) space. The macro put_fs_byte() is implemented
// in include/asm/segment.h.
static inline void save_old(char * from,char * to)
{
        int i;

        verify_area(to, sizeof(struct sigaction));
        for (i=0 ; i< sizeof(struct sigaction) ; i++) {
                put_fs_byte(*from,to);
                from++;
                to++;
        }
}
```

```

// Copy the sigaction data from the fs data segment 'from' to 'to'. That is, copy from the
// user data space to the kernel data segment.
static inline void get_new(char * from,char * to)
{
        int i;

        for (i=0 ; i< sizeof(struct sigaction) ; i++)
                *(to++) = get_fs_byte(from++);
}

// The signal() syscall, similar to sigaction(). Install a new signal handler for the
// specified signal. The signal handler can be a user-specified function, or it can be
// SIG_DFL (the default handler) or SIG_IGN (ignored).
// Parameters: signum - the specified signal; handler - the specified signal handler;
// restorer - the recovery function pointer. This function is provided by libc library.
// It is used to restore the original value of several registers and the return value of
// the syscall when the syscall returns after the end of signal handler, just as if the
// syscall does not execute the signal handler and returns directly to the user program.
// The function returns the original signal handler.
int sys_signal(int signum, long handler, long restorer)
{
        struct sigaction tmp;

// First verify that the signal is within the valid range (1--32) and must not be signal
// SIGKILL (and SIGSTOP). Because these two signals cannot be captured by the process.
// Then build the sigaction structure content according to the provided parameters.
// sa_handler is the specified signal handler. sa_mask is the signal mask. sa_flags is
// some combination of flags when executed. Here, the signal handler is set to the default
// after only one use, and the signal is allowed to be received in its own handler.
        if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
                return -EINVAL;
        tmp.sa_handler = (void (*)(int)) handler;
        tmp.sa_mask = 0;
        tmp.sa_flags = SA_ONESHOT | SA_NOMASK;
        tmp.sa_restorer = (void (*)(void)) restorer;
// Then take the original signal handler and set the sigaction structure. Finally return
// the original signal handler.
        handler = (long) current->sigaction[signum-1].sa_handler;
        current->sigaction[signum-1] = tmp;
        return handler;
}

// Sigaction() system-call. Change the operation of the process when it receives a signal.
// Signum is any signal other than SIGKILL. [If the new action is not empty] the new
// operation is installed. If the oldaction pointer is not empty, the original operation
// is retained to the oldaction. Returns 0 if successful, otherwise -EINVAL.
int sys_sigaction(int signum, const struct sigaction * action,
        struct sigaction * oldaction)
{
        struct sigaction tmp;

// First verify that the signal is within the valid range (1--32) and must not be signal
// SIGKILL (and SIGSTOP). Because these two signals cannot be captured by the process.
// Then set a new action in the sigaction structure of the signal. If the oldaction
// pointer is not empty, the original operation pointer is saved to the location pointed
// to by the oldaction.
        if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
                return -EINVAL;
        tmp = current->sigaction[signum-1];
        get_new((char *) action,
                (char *) (signum-1+current->sigaction));
        if (oldaction)
                save_old((char *) &tmp,(char *) oldaction);
// If we allow the signal to be received in its own signal handler, then the mask is 0,
// otherwise the mask is set to mask this signal.
        if (current->sigaction[signum-1].sa_flags & SA_NOMASK)
                current->sigaction[signum-1].sa_mask = 0;
        else
                current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
        return 0;
}

/*
 * Routine writes a core dump image in the current directory.
 * Currently not implemented.
 */
int core_dump(long signr)
{
        return(0);      /* We didn't do a dump */
}

// The real signal pre-processing code in system-call interrupt handler.
// The main purpose of this code is to insert the signal processing handler into the user
// program stack, and immediately execute the signal handler after system-call returns,
// and then continue to execute the user program. The parameters of this function include
// all values pushed step by step onto the stack from entering the system-call handler to
// the location before calling this function (sys_call.s, line 125). These values include
// (the line number in sys_call.s):
// (1)The user stack address ss and esp, the eflags, and the return addresses cs and eip
//    that are pushed onto kernel-state stack by the interrupt instruction;
// (2)The values of the segment registers ds, es, fs and registers eax (orig_eax), edx,
//    ecx, and ebx pushed onto the stack just after entering system_call on lines 85-91;
// (3)After the sys_call_table is called on line 100, the return value (eax) of the
//    system-call is pushed.
// (4)The currently processed signal (signer) is pushed onto the stack on line 124.
int do_signal(long signr,long eax,long ebx, long ecx, long edx, long orig_eax,
        long fs, long es, long ds,
        long eip, long cs, long eflags,
        unsigned long * esp, long ss)
{
        unsigned long sa_handler;
        long old_eip=eip;
        struct sigaction * sa = current->sigaction + signr - 1;
        int longs;                                   // current->sigaction[signr-1]

        unsigned long * tmp_esp;

// The debug statement. The relevant information is printed when notdef is defined.
#ifdef notdef
        printk("pid: %d, signr: %x, eax=%d, oeax = %d, int=%d\n",
                current->pid, signr, eax, orig_eax,
                sa->sa_flags & SA_INTERRUPT);
#endif
// The orig_eax value will be -1 if it's not a system-call intterrupt but is called during
// other interrupt service (see line 144 of sys_call.s). So when orig_eax is not equal to -1,
// it means that this function was called in the handling of a system-call. In the waitpid()
// function in kernel/exit.c, if SIGCHLD signal is received, or if reading data from a
// pipeline but no data is read, and if the process receives any non-blocking signal, it
// will be returned with a return value of -ERESTARTSYS. It indicates that the process can
// be interrupted, but the system-call is restarted after execution continues. Return code
// -ERESTARTNOINTR indicates that after the signal is handled, it is required to return to
// the system-call to continue running, that is, the system-call will not be interrupted.
// Therefore, the following statement shows that if this function is called in the system-call,
// and the return code eax of the system-call is equal to -ERESTARTSYS or -ERESTARTNOINTR,
// the following processing is performed (actually, it has not returned to user program).
        if ((orig_eax != -1) &&
            ((eax == -ERESTARTSYS) || (eax == -ERESTARTNOINTR))) {

// If the system-call return code is -ERESTARTSYS, and the sigaction contains flag
// SA_INTERRUPT or the signal is less than SIGCONT or greater than SIGTTOU (ie the signal
// is not SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU), the modified return value of the
// system-call is eax = -EINTR, which is the system-call interrupted by the signal.
                if ((eax == -ERESTARTSYS) && ((sa->sa_flags & SA_INTERRUPT) ||
                    signr < SIGCONT || signr > SIGTTOU))
                        *(&eax) = -EINTR;
                else {
// Otherwise, the eax is restored to the value before the system-call is called, and the
// original program instruction pointer is subtracted by 2 bytes. That is, when returning
// to the user program, let the program restart to execute the system-call that was
// interrupted by the signal.
                        *(&eax) = orig_eax;
                        *(&eip) = old_eip -= 2;
                }
        }
// If the signal handle is SIG_IGN (1, the handle is ignored by default), the signal is
// not processed and returned directly.
        sa_handler = (unsigned long) sa->sa_handler;
        if (sa_handler==1)
                return(1);   /* Ignore, see if there are more signals... */
// If the handle is SIG_DFL (0, default processing), the default method is used for
// processing according to the specific signal.
        if (!sa_handler) {
                switch (signr) {
// If the signal is the following two, it is also ignored and returned.
                case SIGCONT:
                case SIGCHLD:
                        return(1);  /* Ignore, ... */

// If the signal is one of the following four signals, the current process state is set to
// the stop state TASK_STOPPED. If the parent process of the current process does not set
// the sigaction flag SA_NOCLDSTOP of the SIGCHLD signal, then the SIGCHLD signal is sent
// to the parent process. The SA_NOCLDSTOP flag indicates that the SIGCHLD signal is not
// generated when the child process stops executing or continues execution.
                case SIGSTOP:
                case SIGTSTP:
                case SIGTTIN:
                case SIGTTOU:
                        current->state = TASK_STOPPED;
                        current->exit_code = signr;
                        if (!(current->p_pptr->sigaction[SIGCHLD-1].sa_flags &
                                        SA_NOCLDSTOP))
                                current->p_pptr->signal |= (1<<(SIGCHLD-1));
                        return(1);  /* Reschedule another event */

// If the signal is one of the following 6 signals, if the signal generates a core dump,
// call do_exit() to exit with the exit code as signr|0x80. Otherwise the exit code is the
// signal value. The parameters of do_exit() are the return code and the exit status
// information provided by the program. It can be used as status information for the
// wait() or waitpid() functions. See lines 13-19 of the sys/wait.h file. wait() or
// waitpid() can use these macros to get the exit status code of the child process or the
// reason (signal) of the child process termination.
                case SIGQUIT:
                case SIGILL:
                case SIGTRAP:
                case SIGIOT:
                case SIGFPE:
                case SIGSEGV:
                        if (core_dump(signr))
                                do_exit(signr|0x80);
                        /* fall through */
                default:
                        do_exit(signr);
                }
        }
        /*
         * OK, we're invoking a handler
         */
// If the signal handler only needs to be called once, the handler is left blank. Note
// that the signal handler has been previously saved in the sa_handler pointer.
// When the system-call enters the kernel code, the user program return address (eip, cs)
// is saved in the kernel state stack. The following code modifies the code pointer eip on
// the kernel state stack to point to the signal handler, and also pushed sa_restorer,
// signr, process mask (if SA_NOMASK is not set), eax, ecx, edx as parameters onto the
// user stack. The original program return pointer and eflag register are also pushed onto
// the user stack. Therefore, when the syscall returns to the user program, the user's
// signal handler is executed first, and then the user program is continued.
        if (sa->sa_flags & SA_ONESHOT)
                sa->sa_handler = NULL;

// Let the user's next code instruction pointer eip on the kernel state stack point to the
// signal handler. Since the C function is passed by value, you need to use the form
// "*(&eip)" when assigning values to eip.
// The sa_mask field of the sigaction structure gives the set of signals that should be
// masked during the execution of the current signal handler. At the same time, the current
// signal will also be blocked. However, if the SA_NOMASK flag is used in sa_flags, the
// current signal will not be masked. If the signal handler is allowed to receive its own
// signal, the signal blocking code of the process also needs to be pushed onto the stack.
        *(&eip) = sa_handler;
        longs = (sa->sa_flags & SA_NOMASK)?7:8;
// Extend the user stack pointer of the original user program by 7 (or 8) long words (used
// to store the parameters of the signal handler, etc.) and check the memory usage (if the
// memory is out of bounds, allocate a new page, etc.).
        *(&esp) -= longs;
        verify_area(esp,longs*4);
// Store sa_restorer, signal signer, mask code blocked (if SA_NOMASK is set), eax, ecx,
// edx, eflags, and user program code pointer from bottom to top in the user stack.
        tmp_esp=esp;
        put_fs_long((long) sa->sa_restorer,tmp_esp++);
        put_fs_long(signr,tmp_esp++);
        if (!(sa->sa_flags & SA_NOMASK))
                put_fs_long(current->blocked,tmp_esp++);
        put_fs_long(eax,tmp_esp++);
        put_fs_long(ecx,tmp_esp++);
        put_fs_long(edx,tmp_esp++);
        put_fs_long(eflags,tmp_esp++);
        put_fs_long(old_eip,tmp_esp++);
        current->blocked |= sa->sa_mask;    // Fill in with sa_mask bitmap
        return(0);              /* Continue, execute handler */
}
```

8.7.3 Information

8.7.3.1 Signal Description

The signal in the process is a simple message used for communication between processes, usually a label value between 1 to 31, and does not carry any other information. The signals supported by the Linux 0.12 kernel are shown in Table 8–5.

Table 8-5 Process signals

Sig	Name	Description	Default operation
1	SIGHUP	(Hangup) The kernel generates this signal when you no longer have a control terminal, or when you turn off Xterm or disconnect the modem. Since the background programs do not have control terminals, they often use SIGHUP to signal that they need to re-read their configuration files.	(Abort) Hang up control terminal or process.
2	SIGINT	(Interrupt) An interrupt from the keyboard. Usually the terminal driver will bind it to ^C.	(Abort) Terminate program.
3	SIGQUIT	(Quit) The exit interrupt from keyboard. Usually the terminal driver will bind it to ^\\.	(Dump) Terminated and a dump core file is generated.
4	SIGILL	(Illegal Instruction) The program has an error or an illegal operation	(Dump)

		command has been executed.	
5	SIGTRAP	(Breakpoint/Trace Trap) Used for debugging, tracking breakpoints.	
6	SIGABRT	(Abort) Abandon execution and end abnormally.	(Dump)
6	SIGIOT	(IO Trap) Same as SIGABRT	(Dump)
7	SIGUNUSED	(Unused) Not used.	
8	SIGFPE	(Floating Point Exception) Floating exception.	(Dump)
9	SIGKILL	(Kill) The program was terminated. This signal cannot be captured or ignored. If you want to terminate a process immediately, you will send a signal 9. Note that the program will have no chance to do the cleanup.	(Abort)
10	SIGUSR1	(User defined Signal 1) User defined signal 1.	(Abort)
11	SIGSEGV	(Segmentation Violation) This signal is generated when the program references invalid memory. For example: addressing unmapped memory; addressing unlicensed memory.	(Dump)
12	SIGUSR2	(User defined Signal 2) Reserved for user programs for IPC or other purposes.	(Abort)
13	SIGPIPE	(Pipe) This signal is generated when a program writes to a socket or pipe, and there is no reader.	(Abort)
14	SIGALRM	(Alarm) This signal is generated after the delay time set by the user using alarm syscall. This signal is often used to determine syscall timeouts.	(Abort)
15	SIGTERM	(Terminate) Used to kindly require a program to terminate. It is the default signal for kill. Unlike SIGKILL, this signal can be captured so that it can be cleaned up before exiting.	(Abort)
16	SIGSTKFLT	(Stack fault on coprocessor) Coprocessor stack error.	(Abort)
17	SIGCHLD	(Child) The child process issued. The child process has been stopped or terminated. You can change its meaning and use it for other usage.	(Ignore) The child process stops or ends.
18	SIGCONT	(Continue) This signal causes the process stopped by SIGSTOP to resume operation. Can be captured.	(Continue) Resume process running.
19	SIGSTOP	(Stop) Stop process running. This signal cannot be captured or ignored.	(Stop) Stop process running.
20	SIGTSTP	(Terminal Stop) Send a stop key sequence to the terminal. This signal can be captured or ignored.	(Stop)
21	SIGTTIN	(TTY Input on Background) The background process attempts to read data from a terminal that is no longer under control, at which point the process will be stopped until the SIGCONT signal is received. This signal can be captured or ignored.	(Stop)
22	SIGTTOU	(TTY Output on Background) The background process attempts to output data to a terminal that is no longer under control, at which point the process will be stopped until the SIGCONT signal is received. This signal can be captured or ignored.	(Stop)

8.8 exit.c

8.8.1 Function Description

The exit.c program mainly implements the processing related to the termination and exit of process. These include process release, session (process group) termination, and program exit handlers, as well as system-calls such as killing processes, terminating processes, and suspending processes. It also includes signal sending function send_sig(), and function tell_father() that notifies parent the termination of child process.

The release process function release() deletes the specified task pointer in the task array, releases the relevant memory page according to the specified task pointer, and immediately causes the kernel to reschedule the running task. The process group termination function kill_session() is used to send a signal to the process whose session number is the same as the current process ID. The system-call sys_kill() is used to send any specified signal to a process. Depending on the parameter pid, the system-call sends a signal to a different process or group of processes. The handling of the various situations is listed in the program comments.

The program exit handling function do_exit() is called in the interrupt handler of the exit system-call. It first releases the memory page occupied by the code and data segments of the current process. If the current process has child process, set the father field of the child process to 1, that is, change the parent of the child process to process 1 (init process). If the child process is already in a zombie state, the child termination signal SIGCHLD is sent to process 1. Then close all files opened by the current process, release the used terminal device, and coprocessor device. If the current process is the first process of the process group, then all related processes need to be terminated. The current process is then set to a zombie state, the exit code is set, and the child termination signal SIGCHLD is sent to its parent process. Finally let the kernel reschedule the running task.

The system-call waitpid() is used to suspend current process until the child process specified by pid exits (terminates) or receives a signal requesting termination of the process, or a signal handler needs to be called. If the child process pointed to by pid has already exited (which has become a so-called zombie process), this call will return immediately. All resources used by the child process will be released. The specific operation of this function is also handled differently according to its parameters. See the relevant comments in the code for details.

8.8.2 Code Comments

Program 8-7 linux/kernel/exit.c

```
/*
 *  linux/kernel/exit.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define DEBUG_PROC_TREE

// <errno.h> Error number header file. Contains various error numbers in the system.
// <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
//      manipulation function prototypes.
// <sys/wait.h> Wait header file. Define the system call wait() core waitpid() and related
//     constant symbols.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
//     communication.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined
//     for segment register operations.
#include <errno.h>
#include <signal.h>
#include <sys/wait.h>

#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/tty.h>
#include <asm/segment.h>

int sys_pause(void);        // put in sleep until receive a signal (kernel/sched.c, 164).
int sys_close(int fd);      // close a file (fs/open.c, 219).

// Frees task slot occupied by the process and mem page occupied by its task structure. The
// parameter p is a pointer to the task data structure. This function is called in the sys_kill()
// and sys_waitpid() that follow.
// The program scans the task pointer array task[] to find the specified task. If found, the
// task slot is first emptied and then the memory page occupied by the task data structure is
// released. Finally execute the scheduler and exit immediately upon return. If the item
// corresponding to the specified task is not found in the task array, the kernel panic.
void release(struct task_struct * p)
{
        int i;

// Exit if the given task structure pointer is NULL. If the pointer points to the current process,
// a warning message is displayed to exit. Suicide is not allowed here even you are not a creature!
        if (!p)
                return;
        if (p == current) {
                printk("task releasing itself\n\r");
                return;
        }
// The following loop statement scans the array of task structure pointers to find the specified
// task p. If found, the corresponding item in the task pointer array is set to NULL, and the
// associated pointer between the task is updated, and the memory page occupied by the task
// p data structure is released. Finally, exit after scheduler returns. If task p is not found,
// the kernel code is wrong, and an error message is displayed and the kernel crashes. In addition,
// the code that updates the links removes the task p from the doubly linked list.
        for (i=1 ; i<NR_TASKS ; i++)
                if (task[i]==p) {
                        task[i]=NULL;
                        /* Update links */
// The following code operates on the linked list. If p is not the last (oldest) child process,
// let the old sibling (neighbor) points to the young sibling. If p is not the latest child
// process, let the newer sibling points to the older sibling. If task p is the latest child
// process, you need to update its parent's latest child pointer cptr to point to its old sibling.
// Refer to figure 5-20.
// osptr (old sibling pointer) points to the sibling process created earlier than p.
// ysptr (younger sibling pointer) points to the sibling process created after p.
// pptr (parent pointer) points to the parent process of p.
// cptr (child pointer) parent process points to the last created child.
                        if (p->p_osptr)
                                p->p_osptr->p_ysptr = p->p_ysptr;
                        if (p->p_ysptr)
                                p->p_ysptr->p_osptr = p->p_osptr;
                        else
                                p->p_pptr->p_cptr = p->p_osptr;
                        free_page((long)p);
                        schedule();
                        return;
                }
        panic("trying to release non-existent task");
}

#ifdef DEBUG_PROC_TREE
// If symbol DEBUG_PROC_TREE is defined, the following code is included at compile time.
/*
 * Check to see if a task_struct pointer is present in the task[] array
 * Return 0 if found, and 1 if not found.
 */
int bad_task_ptr(struct task_struct *p)
{
        int     i;

        if (!p)
                return 0;
        for (i=0 ; i<NR_TASKS ; i++)
                if (task[i] == p)
                        return 0;
        return 1;
}

/*
 * This routine scans the pid tree and make sure the rep invarient still
 * holds.  Used for debugging only, since it's very slow....
 *
 * It looks a lot scarier than it really is.... we're doing nothing more
 * than verifying the doubly-linked list found in p_ysptr and p_osptr,
 * and checking it corresponds with the process tree defined by p_cptr and
 * p_pptr;
 */
void audit_ptree()
{
        int     i;

// The loop scans all tasks except task 0 in the system and checks the correctness of the
// four pointers (pptr, cptr, ysptr, and osptr).  Skip if the task array slot is empty.
        for (i=1 ; i<NR_TASKS ; i++) {
                if (!task[i])
                        continue;
// If the task's parent pointer p_pptr does not point to any process (that is, it does not
// exist in the task array), a warning message is displayed. The following statements perform
// similar operations on cptr, ysptr, and osptr.
                if (bad_task_ptr(task[i]->p_pptr))
                        printk("Warning, pid %d's parent link is bad\n",
                                task[i]->pid);
                if (bad_task_ptr(task[i]->p_cptr))
                        printk("Warning, pid %d's child link is bad\n",
                                task[i]->pid);
                if (bad_task_ptr(task[i]->p_ysptr))
                        printk("Warning, pid %d's ys link is bad\n",
                                task[i]->pid);
                if (bad_task_ptr(task[i]->p_osptr))
                        printk("Warning, pid %d's os link is bad\n",
                                task[i]->pid);
// If the task's parent pointer p_pptr points to itself, a warning message is displayed.
// The following statements perform similar operations on cptr, ysptr, and osptr.
                if (task[i]->p_pptr == task[i])
                        printk("Warning, pid %d parent link points to self\n");
                if (task[i]->p_cptr == task[i])
                        printk("Warning, pid %d child link points to self\n");
                if (task[i]->p_ysptr == task[i])
                        printk("Warning, pid %d ys link points to self\n");
                if (task[i]->p_osptr == task[i])
                        printk("Warning, pid %d os link points to self\n");
// If the task has a old sibling process (that was created earlier than itself), then check
// if they have a common parent and check if the ysptr pointer of the buddy points to this process
// correctly, otherwise a warning message is displayed.
                if (task[i]->p_osptr) {
                        if (task[i]->p_pptr != task[i]->p_osptr->p_pptr)
                                printk(
                        "Warning, pid %d older sibling %d parent is %d\n",
                                task[i]->pid, task[i]->p_osptr->pid,
                                task[i]->p_osptr->p_pptr->pid);
                        if (task[i]->p_osptr->p_ysptr != task[i])
                                printk(
                "Warning, pid %d older sibling %d has mismatched ys link\n",
                                task[i]->pid, task[i]->p_osptr->pid);
                }
// If the task has a young sibling process (that was created later than itself), then check
// if they have a common parent and check if the osptr pointer of the younger correctly points
// to this process, otherwise a warning message is displayed.
                if (task[i]->p_ysptr) {
                        if (task[i]->p_pptr != task[i]->p_ysptr->p_pptr)
                                printk(
                        "Warning, pid %d younger sibling %d parent is %d\n",
                                task[i]->pid, task[i]->p_osptr->pid,
                                task[i]->p_osptr->p_pptr->pid);
                        if (task[i]->p_ysptr->p_osptr != task[i])
```

```
                                printk(
                "Warning, pid %d younger sibling %d has mismatched os link\n",
                                task[i]->pid, task[i]->p_ysptr->pid);
                }
// If the task's latest child pointer cptr is not empty, then check if the child's parent is
// the process, and check if the child's younger pointer yspter is empty. If not, a warning
// message is displayed.
                if (task[i]->p_cptr) {
                        if (task[i]->p_cptr->p_pptr != task[i])
                                printk(
                        "Warning, pid %d youngest child %d has mismatched parent link\n",
                                task[i]->pid, task[i]->p_cptr->pid);
                        if (task[i]->p_cptr->p_ysptr)
                                printk(
                        "Warning, pid %d youngest child %d has non-NULL ys link\n",
                                task[i]->pid, task[i]->p_cptr->pid);
                }
        }
}
#endif /* DEBUG_PROC_TREE */

// Send a signal sig to task p with privilege priv.
// sig - the signal; p - a pointer to the task; priv - a flag that forces the signal to be sent,
// ie the right to send signal without regard to process user attribute or level.
// The function first checks the correctness of the parameters and then determines if the
// condition is met. If it is satisfied, it sends a signal sig to the process and exits,
// otherwise it returns an unlicensed error code.
static inline int send_sig(long sig,struct task_struct * p,int priv)
{
// If there is no permission, and the effective user ID (euid) of the current process is
// different from the process p, and is not a superuser, then there is no right to send a
// signal to p. suser() is defined as (current->euid==0), used to check if it's a superuser.
        if (!p)
                return -EINVAL;
        if (!priv && (current->euid!=p->euid) && !suser())
                return -EPERM;
// If the signal to be send is SIGKILL or SIGCONT, then if the process p receiving signal
// is stopped at this time, it is set to the ready state (TASK_RUNNING). Then modify the
// signal bitmap process p to remove (reset) the signals SIGSTOP, SIGTSTP, SIGTTIN, and
// SIGTTOU that will cause the process to stop.
        if ((sig == SIGKILL) || (sig == SIGCONT)) {
                if (p->state == TASK_STOPPED)
                        p->state = TASK_RUNNING;
                p->exit_code = 0;
                p->signal &= ~( (1<<(SIGSTOP-1)) | (1<<(SIGTSTP-1)) |
                                (1<<(SIGTTIN-1)) | (1<<(SIGTTOU-1)) );
        }
        /* If the signal will be ignored, don't even post it */
        if ((int) p->sigaction[sig-1].sa_handler == 1)
                return 0;
        /* Depends on order SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU */
// If the signal is one of SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU, then it is necessary to
// stop process p from running. Therefore (if SIGCONT is set in the signal bitmap of p), it
// is necessary to reset the SIGCONT bit in the bitmap.
        if ((sig >= SIGSTOP) && (sig <= SIGTTOU))
                p->signal &= ~(1<<(SIGCONT-1));
        /* Actually deliver the signal */
        p->signal |= (1<<(sig-1));
        return 0;
}

// Get the session id based on the process group id pgrp.
// The code scans task array, looks for process with the group id pgrp, and returns its session
// id. If no process is found for the specified group pgrp, -1 is returned.
int session_of_pgrp(int pgrp)
{
        struct task_struct **p;

        for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if ((*p)->pgrp == pgrp)
                        return((*p)->session);
        return -1;
}

// Kill the process group (send a signal to the group).
// Parameters: grp - process group id;  sig - signal;  priv - privilege.
// That is, the signal sig is sent to each process in the process group pgrp. As long as it
// is successfully sent to a process, it will return 0. Otherwise, if no process is found for
// the group pgrp, the error code -ESRCH is returned. If the process whose group is pgrp is
// found, but the signal transmission fails, the error code is returned.
int kill_pg(int pgrp, int sig, int priv)
{
        struct task_struct **p;
        int err,retval = -ESRCH;          // ESRCH  -  error search.
        int found = 0;

// First check if the given signal and process group are valid, then scan all tasks in the
// system. If the process with the group id pgrp is scanned, the signal sig is sent to it.
// As long as the signal is sent successfully, the function will return 0 at the end.
        if (sig<1 || sig>32 || pgrp<=0)
                return -EINVAL;
        for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if ((*p)->pgrp == pgrp) {
                        if (sig && (err = send_sig(sig,*p,priv)))
                                retval = err;
                        else
                                found++;
                }
        return(found ? 0 : retval);
}

// Kill the process (send a signal to the process).
// Parameters: pid - the process id;  sig - the signal;  priv - the privilege.
// That is, the signal sig is sent to the process with the pid. If the process of the pid is
// found, then if the signal is sent successfully, it returns 0, otherwise return error code.
// If the process with the pid is not found, the error code -ESRCH is returned
```

```
int kill_proc(int pid, int sig, int priv)
{
        struct task_struct **p;

        if (sig<1 || sig>32)
                return -EINVAL;
        for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if ((*p)->pid == pid)
                        return(sig ? send_sig(sig,*p,priv) : 0);
        return(-ESRCH);
}

/*
 * POSIX specifies that kill(-1,sig) is unspecified, but what we have
 * is probably wrong.  Should make it like BSD or SYSV.
 */
// The system-call kill() can be used to send any signal to process or process group.
// Parameters: pid - the process id; sig - the signal that needs to be sent.
// If pid > 0, the signal is sent to the process whose process id is pid.
// If pid = 0, the signal is sent to all processes in the group of the current process.
// If pid = -1, the signal is sent to all processes except the first (initial) process.
// If pid < -1, the signal will be sent to all processes in the group -pid.
// If sig is 0, no signal is sent, but error check is still performed. Ret 0 if succeed.
//
// This function scans the task array and sends the signal sig to the process that satisfies
// the condition according to pid. If pid is equal to 0, it indicates that the current process
// is the group leader, so it is necessary to force the signal sig to be sent to all processes
// in the group.
int sys_kill(int pid,int sig)
{
        struct task_struct **p = NR_TASKS + task;     // points to the last item.
        int err, retval = 0;

        if (!pid)
                return(kill_pg(current->pid,sig,0));
        if (pid == -1) {
                while (--p > &FIRST_TASK)
                        if (err = send_sig(sig,*p,0))
                                retval = err;
                return(retval);
        }
        if (pid < 0)
                return(kill_pg(-pid,sig,0));
        /* Normal kill */
        return(kill_proc(pid,sig,0));
}

/*
 * Determine if a process group is "orphaned", according to the POSIX
 * definition in 2.2.2.52.  Orphaned process groups are not to be affected
 * by terminal-generated stop signals.  Newly orphaned process groups are
 * to receive a SIGHUP and a SIGCONT.
 *
 * "I ask you, have you ever known what it is to be an orphan?"
 */
// The POSIX P1003.1 section 2.2.2.52 mentioned above is a description of the orphan process
// group. In both cases, when a process terminates, it may cause the process group to become
// "orphan." The connection between a process group and a parent outside its group depends
// on both the parent and its child processes. Therefore, if the last process outside the group
// that is connected to the parent process, or the immediate descendant of the last parent
// process, terminates, then the process group becomes an orphan process group. In either case,
// if the termination of the process causes the process group to become an orphaned group, all
// processes in the group are disconnected from their job control shell.
// The job control shell will no longer have any information about the existence of this process
// group. The process in the group that is in a stopped state will disappear forever. To solve
// this problem, a newly generated orphan process group containing a stop state process needs
// to receive a SIGHUP signal and a SIGCONT signal to indicate that they have disconnected from
// their session.
// The SIGHUP signal will cause members of the process group to be terminated unless they capture
// or ignore the SIGHUP signal. The SIGCONT signal will continue to run those processes that
// are not terminated by the SIGHUP signal. However, in most cases, if one of the processes
// in the group is in a stopped state, all processes in the group may be in a stopped state.
//
// Check if a process group is an orphan. Returns 0 if not; returns 1 if it is. The code loop
// scans the task array. If the task item is empty, or the process group id is different from
// the specified one, or the process is already in a zombie state, or the process's parent is
// an init process, then the scanned process is not a member of the group, or the request is
// not met, so skip over. Otherwise, the process is a member of the group and its parent is
// not the init process. At this time, if the group id of the parent is not equal to the group
// id pgrp, but the session of the parent is equal to the session of the process, it means that
// they belong to the same session. Therefore the specified pgrp group is certainly not an orphan
// group, otherwise...sighing.
int is_orphaned_pgrp(int pgrp)
{
        struct task_struct **p;

        for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
                if (!(*p) ||
                    ((*p)->pgrp != pgrp) ||
                    ((*p)->state == TASK_ZOMBIE) ||
                    ((*p)->p_pptr->pid == 1))
                        continue;
                if (((*p)->p_pptr->pgrp != pgrp) &&
                    ((*p)->p_pptr->session == (*p)->session))
                        return 0;
        }
        return(1);      /* (sighing) "Often!" */
}

// Check if the process group contains a job (process group) that is in a stopped state.
// Returns 1 if there is none; returns 0 if none. The search method is to scan the entire task
// array and check if any processes belonging to the group pgrp are in a stopped state.
static int has_stopped_jobs(int pgrp)
{
        struct task_struct ** p;

```

```
253         for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
254                 if ((*p)->pgrp != pgrp)
255                         continue;
256                 if ((*p)->state == TASK_STOPPED)
257                         return(1);
258         }
259         return(0);
260 }
261
    // Program exit handling function. Invoked by the syscall sys_exit() at line 365 below.
    // The function will process it according to the characteristics of the current process itself,
    // and set the current process state to TASK_ZOMBIE, and finally call the schedule() function
    // to execute other processes, and will not return.
262 volatile void do_exit(long code)
263 {
264         struct task_struct *p;
265         int i;
266
    // First free the memory page occupied by the current process code and data segments.
    // The first argument of the function free_page_tables() (the get_base() return value) indicates
    // the starting base address in the CPU linear address space, and the second (get_limit() return
    // value) indicates the byte length to be released. The current->ldt[1] in the get_base() macro
    // gives the location of the process code segment descriptor, and current->ldt[2] gives the
    // location of the data segment descriptor. 0x0f in get_limit() is the selector of the code
    // segment, and 0x17 is the selector of the data segment. That is, when the segment base address
    // is taken, the address of the segment descriptor is used as a parameter, and when the segment
    // limit (length) is taken, the segment selector is used as a parameter (since there is a dedecated
    // instruction LSL which can be used to get segment limit throgh a selector).
    // The free_page_tables() is located at the beginning of line 69 in mm/memory.c file;
    // The get_base() and get_limit() macros are located at line 265 in include/linux/sched.h.
267         free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
268         free_page_tables(get_base(current->ldt[2]),get_limit(0x17));

    // Then, all the files opened by the current process are closed, and the working directory pwd,
    // the root directory, the i-node of the executable file, and the library file are synchronized,
    // and the i-nodes are put back and are respectively blanked (released). Then set state of the
    // process to zombie state (TASK_ZOMBIE) and set process exit code.
269         for (i=0 ; i<NR_OPEN ; i++)
270                 if (current->filp[i])
271                         sys_close(i);
272         iput(current->pwd);
273         current->pwd = NULL;
274         iput(current->root);
275         current->root = NULL;
276         iput(current->executable);
277         current->executable = NULL;
278         iput(current->library);
279         current->library = NULL;
280         current->state = TASK_ZOMBIE;
281         current->exit_code = code;
282         /*
283          * Check to see if any process groups have become orphaned
284          * as a result of our exiting, and if they have any stopped
```

```
         * jobs, send them a SIGUP and then a SIGCONT.  (POSIX 3.2.2.2)
         *
         * Case i: Our father is in a different pgrp than we are
         * and we were the only connection outside, so our pgrp
         * is about to become orphaned.
         */
// POSIX 3.2.2.2 (1991 version) is a description of the exit() function. If the process group
// in which the parent is located is different from the current process, but all are in the
// same session, and the process group in which the current process is located is going to be
// an orphan, and the current process group contains job in stopped state, then two signals
// should be sent to the group: SIGHUP and SIGCONT.
        if ((current->p_pptr->pgrp != current->pgrp) &&
            (current->p_pptr->session == current->session) &&
            is_orphaned_pgrp(current->pgrp) &&
            has_stopped_jobs(current->pgrp)) {
                kill_pg(current->pgrp,SIGHUP,1);
                kill_pg(current->pgrp,SIGCONT,1);
        }
        /* Let father know we died */
        current->p_pptr->signal |= (1<<(SIGCHLD-1));

        /*
         * This loop does two things:
         *
         * A.  Make init inherit all the child processes
         * B.  Check to see if any process groups have become orphaned
         *      as a result of our exiting, and if they have any stopped
         *      jons, send them a SIGUP and then a SIGCONT.  (POSIX 3.2.2.2)
         */
// If the current process has child processes (whose p_cptr points to the most recently created
// child), then process 1 (init process) becomes the parent of all its child processes. If the
// child process is already in zombie state, the child terminating signal SIGCHLD is sent to
// the init process (parent).
        if (p = current->p_cptr) {
                while (1) {
                        p->p_pptr = task[1];
                        if (p->state == TASK_ZOMBIE)
                                task[1]->signal |= (1<<(SIGCHLD-1));
                        /*
                         * process group orphan check
                         * Case ii: Our child is in a different pgrp
                         * than we are, and it was the only connection
                         * outside, so the child pgrp is now orphaned.
                         */
// If the child is not in the same group as the current process but belongs to the same session,
// and the process group in which the current process is located is going to be an orphan, and
// this group has a job (process) in stopped state, then It is necessary to send two signals
// to this group: SIGHUP and SIGCONT. If the child process has sibling processes, continue to
// loop through these sibling processes.
                        if ((p->pgrp != current->pgrp) &&
                            (p->session == current->session) &&
                            is_orphaned_pgrp(p->pgrp) &&
                            has_stopped_jobs(p->pgrp)) {
                                kill_pg(p->pgrp,SIGHUP,1);
                                kill_pg(p->pgrp,SIGCONT,1);
                        }
                        if (p->p_osptr) {
                                p = p->p_osptr;
                                continue;
                        }
                        /*
                         * This is it; link everything into init's children
                         * and leave
                         */
// Through the above processing, all the sibling processes of the child process have been
// processed. At this point p points to the oldest sibling of the child process. So all these
// siblings are added to the doubly linked list header of the child process of the init process.
// After joining, the p_cptr of the init process points to the youngest child of the current
// process, and the oldest sibling child process p_osptr points to the youngest child process,
// while the youngest process's p_ysptr points to the oldest sibling subprocess . Finally, the
// p_cptr pointer of the current process is set to null and the loop is exited.
                        p->p_osptr = task[1]->p_cptr;
                        task[1]->p_cptr->p_ysptr = p;
                        task[1]->p_cptr = current->p_cptr;
                        current->p_cptr = 0;
                        break;
                }
        }
// If the current process is a session leader, and if it has a control terminal, it first sends
// a signal SIGHUP to the group using the control terminal, and then releases the terminal.
// Then scan the task array and empty (cancel) the terminal of the process in the session.
        if (current->leader) {
                struct task_struct **p;
                struct tty_struct *tty;

                if (current->tty >= 0) {
                        tty = TTY_TABLE(current->tty);
                        if (tty->pgrp>0)
                                kill_pg(tty->pgrp, SIGHUP, 1);
                        tty->pgrp = 0;
                        tty->session = 0;
                }
                for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                        if ((*p)->session == current->session)
                                (*p)->tty = -1;
        }
// If the current process used the coprocessor last time, then set it to NULL to discard the
// message. In addition, if debug symbol is defined, the audit process tree function is called.
// Finally, the scheduler is called to re-schedule the process to run, so that the parent process
// can handle other aftermath of the zombie process.
        if (last_task_used_math == current)
                last_task_used_math = NULL;
#ifdef DEBUG_PROC_TREE
        audit_ptree();
#endif
        schedule();
```

```
363 }
364
    // The system-call exit(), terminate the process. The parameter error_code is the exit status
    // information provided by the user program, only the low byte is valid. Shifting error_code
    // to the left by 8 bits is a requirement of the wait() or waitpid() function. The low byte
    // will be used to save the status of wait(). For example, if a process is in a suspended state
    // (TASK_STOPPED),its low byte is equal to 0x7f. See lines 13-19 of the sys/wait.h file. wait()
    // or waitpid() can use these macros to get the exit status code of the child process or the
    // reason (signal) of the child termination.
365 int sys_exit(int error_code)
366 {
367         do_exit((error_code&0xff)<<8);
368 }
369

    // The system-call waitpid(). Suspend the current process until the child specified by pid exits
    // (terminates) or receives a signal requesting termination of the process, or needs to call
    // a signal handler. If the child process pointed to by pid has already exited (which has become
    // a so-called zombie process), this syscall will return immediately. All resources used by
    // the child will be released.
    // If pid > 0, waiting for a child whose process id is equal to pid.
    // If pid = 0, waiting for a child whose group is equal to current process group.
    // If pid < -1, waiting for any child, its process group is equal to absolute value of pid.
    // If pid = -1, Indicates waiting for any child process.
    // If options = WUNTRACED, means that if child is stopped, it return immediately.
    // If options = WNOHANG, means that if no child exits or terminates,it reterns immediately.
    // If the return state pointer stat_addr is not empty, the state info is saved there.
    // parameters:
    // pid  - the process id; *stat_addr - pointer to the state info; options - waitpid option.
370 int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
371 {
372         int flag;               // selected child in ready or sleep state.
373         struct task_struct *p;
374         unsigned long oldblocked;
375
    // First verify that there is enough memory space to store the status information. Then reset
    // the flag. The child process sibling list is then scanned starting from the youngest child
    // of the current process.
376         verify_area(stat_addr,4);
377 repeat:
378         flag=0;
379         for (p = current->p_cptr ; p ; p = p->p_osptr) {
    // If the waiting child process number pid>0 and it is not equal to the pid of the scanned child
    // process p, it indicates that it is another child process of the current process, then skips
    // the process and then scans the next process. Otherwise, it means that the waiting child process
    // pid is found, so it jumps to line 390 to continue execution.
380                 if (pid>0) {
381                         if (p->pid != pid)
382                                 continue;
    // Otherwise, if you specify a pid=0 for the waiting process, it means that you are waiting
    // for any child processes whose process group id is equal to the current process group id.
    // If the process group id of the scanned process p is not equal to the group id of the current
    // process, it is skipped. Otherwise, it means that a child process whose process group id is
    // equal to the current process group id is found, and then jumps to line 390 to continue
    // execution.
383             } else if (!pid) {
384                     if (p->pgrp != current->pgrp)
385                             continue;
    // Otherwise, if the specified pid < -1, it means that any child whose process group id is equal
    // to the absolute value of pid are waiting. If the group id of the scanned process p is not
    // equal to the absolute value of the pid, it is skipped. Otherwise, it means that a child whose
    // process group id is equal to the absolute value of pid is found, and then jumps to line 390
    // to continue execution.
386             } else if (pid != -1) {
387                     if (p->pgrp != -pid)
388                             continue;
389             }
    // If the first three do not match the pid, it means that the current process is waiting for
    // any of its child processes (this time pid = -1).
    //
    // At this point, the selected process p is either its process id equal to the specified pid,
    // or any child process in the current process group, or a child whose process id is equal to
    // the absolute value of the pid, or any child process (pid is equal to -1). Next, it is processed
    // according to the state of this selected process p.
    //
    // When the process p is in a stopped state, if the WUNTRACED option is not set at this time,
    // it means that the program does not need to return immediately, or the exit code of the child
    // process is equal to 0, so the scanning continues to process other child processes. If WUNTRACED
    // is set and the child exit code is not 0, then the exit code is moved to the high byte, OR
    // the status message 0x7f, then placed in *stat_addr, and the child pid is returned immediately
    // after resetting child exit code. Here the return status of 0x7f makes the WIFSTOPPED() macro
    // true, see file include/sys/wait.h, line 14.
390             switch (p->state) {
391                     case TASK_STOPPED:
392                             if (!(options & WUNTRACED) ||
393                               !p->exit_code)
394                                     continue;
395                             put_fs_long((p->exit_code << 8) | 0x7f,
396                                     stat_addr);
397                             p->exit_code = 0;
398                             return p->pid;
    // If the child process p is in a dead state, it first accumulates the time it runs in the user
    // mode and the kernel state into the current process (parent process). Then take out the pid
    // and exit code of the child process, put the exit code into the return status position stat_addr
    // and release the child process. Finally, return the exit code and pid of the child process.
    // If the debug process tree symbol is defined, the process tree audit function is called.
399                     case TASK_ZOMBIE:
400                             current->cutime += p->utime;
401                             current->cstime += p->stime;
402                             flag = p->pid;
403                             put_fs_long(p->exit_code, stat_addr);
404                             release(p);
405 #ifdef DEBUG_PROC_TREE
406                             audit_ptree();
407 #endif
408                             return flag;
```

```
// If the state of this child p is neither stopped nor in zombie state, then set flag = 1.
// This Indicates that a child process that meets the requirements has been found, but it is
// in a running state or sleep state.
                        default:
                                flag=1;
                                continue;
                }
        }
// After the scan of the task array is finished, if the flag is set, it indicates that the child
// process that meets the wait condition is not in the exit or zombie state. At this time, if
// the WNOHANG option has been set, it will immediately return 0 and exit. Otherwise, the current
// process is placed in an interruptible wait state, and current process signal blocking bitmap
// is saved and modified to allow it to receive the SIGCHLD signal, then execute the scheduler.
        if (flag) {
                if (options & WNOHANG)
                        return 0;
                current->state=TASK_INTERRUPTIBLE;
                oldblocked = current->blocked;
                current->blocked &= ~(1<<(SIGCHLD-1));
                schedule();

// When the system starts executing this process again, if the process receives an unmasked
// signal other than SIGCHLD, it will return with the exit code "Restart System Call". Otherwise
// jump to the repeat label at the beginning of the function and repeat the processing steps.
                current->blocked = oldblocked;
                if (current->signal & ~(current->blocked | (1<<(SIGCHLD-1))))
                        return -ERESTARTSYS;
                else
                        goto repeat;
        }
// If flag = 0, it means that no subprocess that meets the requirements is found, then an error
// code is returned (the child process does not exist).
        return -ECHILD;
}

```

8.9 fork.c

8.9.1 Function Description

The fork() system-call is used to create child process. All processes in Linux are child processes of process 0 (task 0). The fork.c program includes a set of auxiliary processing functions for sys_fork() (starting at line 222 in kernel/sys_call.s). It gives two C functions used in the sys_fork() system-call: find_empty_process() and copy_process(). It also includes the process memory area validation and memory allocation functions verify_area() and copy_mem().

copy_process() is used to create and copy the code segment and data segment of the process and the environment. In the procedure of process replication, the work mainly involves the setting of information in the process data structure. The system first requests a page for the new process in the main memory area to store its task structure information, and copies all the contents of the current process task structure as a template for the new process task structure.

The code then modifies the contents of the copied task structure. First, the code sets the current process as the parent of the new process, clears the signal bitmap and resets the statistics for the new process. Then set registers in the new process task status segment (TSS) according to the current process environment. Since the new process return value should be 0, you need to set tss.eax = 0. The new process kernel state stack pointer tss.esp0 is set to the top of the memory page where the new process task structure is located, and the stack segment tss.ss0 is set to the kernel data segment selector. Tss.ldt is set to the index value of the LDT descriptor in the GDT. If the current process uses a coprocessor, you also need to save the full state of the coprocessor to the tss.i387 structure of the new process.

After that, the system sets the base address and limit of the new task code segment and data segment, and copies the page directory entry and page table entry of the current process paging management. If there is a file in the parent process that is open, the corresponding file in the child process is also open, so you need to increase the number of times the corresponding file is opened by one. The TSS and LDT descriptor entries for the new task are then set in the GDT, where the base address information points to tss and ldt in the new process task structure. Finally, set the new task to a runnable state and return a new process id to the current process.

Figure 8-13 is a position adjustment diagram for verifying the starting position and range in the memory verification function verify_area(). Because the memory write verification function write_verify() needs to operate in units of memory pages (4096 bytes), before calling write_verify(), it is necessary to adjust the starting position of the verification to the start position of the page, and correspondingly to the verification range.

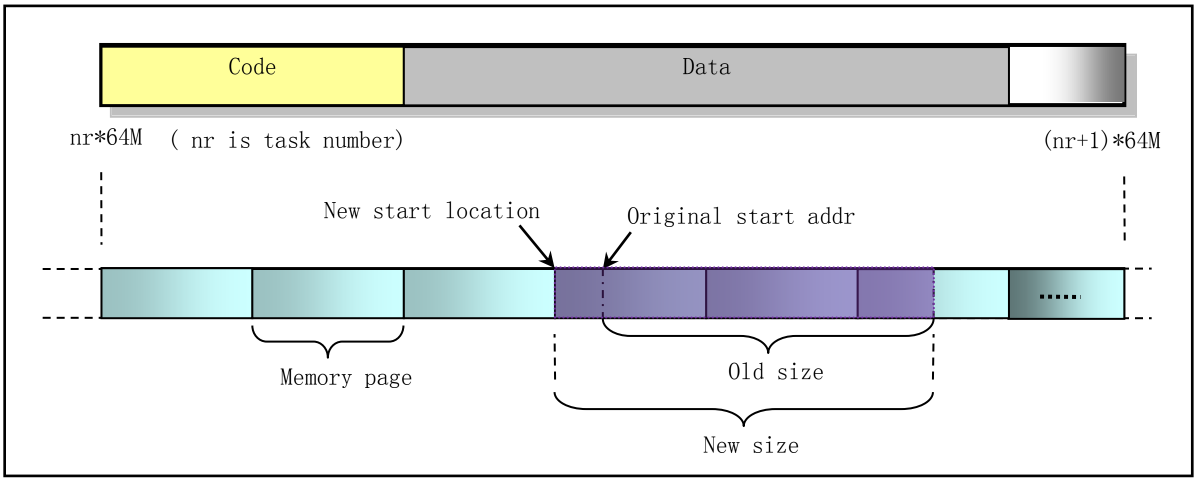


Figure 8-13 Memory verification range and starting position adjustment

The role of fork() is briefly described above based on the purpose of each function in the fork.c program. Here we will give a little more explanation on it. In general, fork() will first apply for a page of memory for the new process to copy the task data structure (also known as process control block, PCB) information of the parent process, and then modify the copied task data structure for the new process. These fields include resetting the registers of the TSS structure in the task structure by using the registers that is gradually pushed into stack when the system-call interrupt occurs (ie, the parameter of copy_process()), so that the state of the new process keeps the parent process state before entering the interrupt. The program then determines the starting position (nr * 64MB) in the linear address space for the new process. For the segmentation mechanism of the CPU, the code segment and data segment of Linux 0.12 are exactly the same in the linear address space. The page directory entry and page table entry for the parent process are then copied for the new process. For the Linux 0.12 kernel, all programs share a page directory table at the beginning of the physical memory, and the page table of the new process needs to apply for another memory page.

During the execution of fork(), the kernel does not immediately allocate code and data memory pages for new processes. The new process will work with the parent process to use the code and data memory pages already in the parent process. Only the memory pages that are accessed when one of the processes accesses the memory in write mode will be copied to the newly requested memory page before the write operation.

8.9.2 **Code Comments**

Program 8-8 linux/kernel/fork.c

```
/*
 *  linux/kernel/fork.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  'fork.c' contains the help-routines for the 'fork' system call
 * (see also system_call.s), and some misc functions ('verify_area').
 * Fork is rather simple, once you get the hang of it, but the memory
 * management can be a bitch. See 'mm/mm.c': 'copy_page_tables()'
 */
// <errno.h> Error number header file. Contains various error numbers in the system.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//      used functions of the kernel.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//      segment register operations.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//      descriptors/interrupt gates, etc. is defined.
#include <errno.h>

#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>
#include <asm/system.h>

 // Write page verification. If page is not writable, copy the page. See mm/memory.c, L274.
extern void write_verify(unsigned long address);

long last_pid=0;              // The latest process id, generated by get_empty_process().


 // The pre-write verification function for the process space.
 // For the 80386 CPU, the privilege level 0 code is executed without regard to whether the page
 // in user space is page protected.  Therefore, the data page protection flag in the user space
// does not work when the kernel code is executed, and the copy-on-write mechanism loses its
// effect. The verify_area() function is used to solve this problem. However, for the 80486
// or later CPU, there is a write protection flag WP (bit 16) in the control register CR0. The
// kernel can disable the privilege level 0 code to write data to the user space read-only page
// by setting the flag. Therefore, the 80486 and above CPUs can achieve the same purpose of
// this function by setting this flag.
//
// This function performs a pre-write detection operation for the range of logical addresses
// from addr to (addr + size). Since the detection is performed on a page-by-page basis, the
// program first needs to find the 'start' address of the page where 'addr' is located, and
// then 'start' plus the base address of the process data segment, so that this 'start' is changed
// into an address in the linear space of the CPU 4G. Finally, write_verify() is called cyclically
// to perform pre-write verification on the memory space of the specified size. If the page
// is read-only, a share check and copy page operation (copy-on-write) is performed.
void verify_area(void * addr,int size)
{
        unsigned long start;

// First, the start address 'start' is adjusted to the start position of the page, and the size
// of the verification area is adjusted accordingly. The start & 0xfff in the next sentence
// is used to get the offset in the page. The original verification range 'size' plus this offset
// is expanded to the range value starting from the beginning of the page. Therefore, it is
// also necessary to adjust the verification start position 'start' to the page boundary. See
// Figure 8-13 above.
        start = (unsigned long) addr;
        size += start & 0xfff;
        start &= 0xfffff000;                    // now start is logical address.

// Next, add the base address of the process data segment and turn 'start' into the address
// in the linear space of the system. Then loop to write page verification. If the page is not
// writable, copy the page (mm/memory.c, line 274).
        start += get_base(current->ldt[2]);     // include/linux/sched.h, line 277
        while (size>0) {
                size -= 4096;
                write_verify(start);
                start += 4096;
        }
}

// Copy memory page table.
// The parameter nr is the new task number; p is the new task data structure pointer. This function
// sets the code segment and data segment base address, limit, and copy the page table for the
// new task in the linear address space. Since the Linux system uses copy-on-write technology,
// only the new page directory entries and page table entries are set up for the new process,
// and the actual physical memory pages are not allocated for the new process. At this point,
// the new process shares all memory pages with its parent process. Returns 0 if successful,
// otherwise it returns the error code.
int copy_mem(int nr,struct task_struct * p)
{
        unsigned long old_data_base,new_data_base,data_limit;
        unsigned long old_code_base,new_code_base,code_limit;

// First, the limits in code and data segment descriptors in current process LDT is taken.
// 0x0f is the code segment selector; 0x17 is the data segment selector. Then take the base
// address of the code & data segments of the current process in the linear address space. Since
// the Linux 0.12 kernel does not support the separation of code and data segments, it is necessary
// to check whether the code segment and the data segment base address are the same, and the
// length of the data segment is required to be at least not less than the length of the code
// segment (see Figure 5-12), otherwise the kernel displays an error message and stops running.
// get_limit() and get_base() are defined on lines 277,279 in file include/linux/sched.h.
        code_limit=get_limit(0x0f);
        data_limit=get_limit(0x17);
        old_code_base = get_base(current->ldt[1]);
        old_data_base = get_base(current->ldt[2]);
        if (old_data_base != old_code_base)
                panic("We don't support separate I&D");
        if (data_limit < code_limit)
                panic("Bad data_limit");
// Then set the base address of the new process in the linear address space equal to
// (64MB * task no), and use this value to set the base address in the segment descriptor in
// the new process LDT. Then set the page directory entry and page table entry of the new process,
// that is, copy the page directory entry and page table entry of the current process (parent
// process). At this point, the child shares the memory page of the parent process. Normally,
// copy_page_tables() returns 0. Otherwise, it indicates an error, and the page entry that was
// just applied is released.
        new_data_base = new_code_base = nr * TASK_SIZE;
        p->start_code = new_code_base;
        set_base(p->ldt[1],new_code_base);
        set_base(p->ldt[2],new_data_base);
        if (copy_page_tables(old_data_base,new_data_base,data_limit)) {
                free_page_tables(new_data_base,data_limit);
                return -ENOMEM;
        }
        return 0;
}

/*
 *  Ok, this is the main fork-routine. It copies the system process
 * information (task[nr]) and sets up the necessary registers. It
 * also copies the data segment in it's entirety.
 */
// Copy process information.
// The parameters of this function start from the handler that enters the system-call interrupt
// INT 0x80, and until this function is called (sys_call.s line 231), These registers are
// gradually pushed into the kernel state stack of the process. The values (parameters) that
// are pushed onto the stack in sys_call.s file include:
// 1) The user stack ss, esp, eflags, and the return addresses cs, eip pushed when executing
//    the INT instruction;
// 2) ds, es, fs, and edx, ecx, ebx pushed on stack on lines 85-91 just after entering;
// 3) The return address pushed when the sys_fork() in sys_call_table is called on line 97
//    (represented by none);
// 4) On lines 226-230, gs, esi, edi, ebp, eax(nr) are pushed before calling copy_process().
// Among them, nr is the task array item index assigned by calling find_empty_process().
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
                long ebx,long ecx,long edx, long orig_eax,
                long fs,long es,long ds,
71                  long eip,long cs,long eflags,long esp,long ss)
72 {
73          struct task_struct *p;
74          int i;
75          struct file *f;
76
   // The code first allocates memory for the new task structure (if the allocation fails, it returns
   // an error code and exits). Then put the new task structure pointer into the nr item of the
   // task array. Where nr is task no, which is returned by the previous find_empty_process().
   // Then copy the contents of the task structure of the current process to the beginning of the
   // memory page p just applied.
77          p = (struct task_struct *) get_free_page();
78          if (!p)
79                  return -EAGAIN;
80          task[nr] = p;
81          *p = *current;  /* NOTE! this doesn't copy the supervisor stack */

   // Then some modifications are made to the content of the copied process structure as the task
   // structure of the new process. The state of the new process is first set to an uninterruptible
   // wait state to prevent the kernel from scheduling its execution. Then set the process ID pid
   // of the new process and initialize the process run time slice value equal to its priority
   // value (typically 15 ticks). Then reset the signal bitmap, the alarm timer, the session leader
   // flag, the running time statistics in the kernel and user mode, and the system time start_time
   // at which the process starts running.
82          p->state = TASK_UNINTERRUPTIBLE;
83          p->pid = last_pid;          // new pid obtained from find_empty_process()
84          p->counter = p->priority;   // run time slice value (number of ticks).
85          p->signal = 0;              // signal bitmap.
86          p->alarm = 0;               // alarm timer.
87          p->leader = 0;              /* process leadership doesn't inherit */
88          p->utime = p->stime = 0;    // user state and core state running time.
89          p->cutime = p->cstime = 0;  // childs user state and core state running time.
90          p->start_time = jiffies;    // the start time of the process (current time ticks).

   // Now modify the task status section TSS content (see the description after the program list).
   // Since the system allocates 1 page of memory to the task structure p, esp0 = (PAGE_SIZE +
   // (long) p) causes esp0 to point exactly to the top of the page. ss0:esp0 is used as a stack
   // for the program to execute in kernel mode.
   // In addition, as we already know in Chapter 5, each task has two segment descriptors in the
   // GDT table: one is the task's TSS segment descriptor and the other is the task's LDT table
   // segment descriptor. The statement on line 110 is to store the selector of the LDT segment
   // descriptor of this task into the TSS segment. When performing task switch, the CPU
   // automatically loads the LDT segment selector from the TSS into the LDTR register.
91          p->tss.back_link = 0;
92          p->tss.esp0 = PAGE_SIZE + (long) p;  // Task kernel state stack pointer.
93          p->tss.ss0 = 0x10;                   // selector for the kernel state stack.
94          p->tss.eip = eip;
95          p->tss.eflags = eflags;
96          p->tss.eax = 0;                      // This is why the new process will return 0.
97          p->tss.ecx = ecx;
98          p->tss.edx = edx;
99          p->tss.ebx = ebx;
100         p->tss.esp = esp;
```

```
        p->tss.ebp = ebp;
        p->tss.esi = esi;
        p->tss.edi = edi;
        p->tss.es = es & 0xffff;            // The segment register has only 16 bits.
        p->tss.cs = cs & 0xffff;
        p->tss.ss = ss & 0xffff;
        p->tss.ds = ds & 0xffff;
        p->tss.fs = fs & 0xffff;
        p->tss.gs = gs & 0xffff;
        p->tss.ldt = _LDT(nr);        // The selector for the task LDT descriptor (in GDT).
        p->tss.trace_bitmap = 0x80000000;    //(High 16 bits are valid).

// If the current task uses a coprocessor, its context is saved. The instruction CLTS is used
// to clear the task exchange flag TS in the control register CR0. The CPU sets this flag whenever
// a task switch occurs. This flag is used to manage the math coprocessor: if this flag is set,
// then each ESC instruction will be caught (Exception 7). If the coprocessor presence flag
// MP is also set, the WAIT instruction will also be captured. Therefore, if a task switch occurs
// after an ESC instruction begins execution, the contents of the coprocessor may need to be
// saved before executing the new ESC instruction. The capture handler will store the contents
// of the coprocessor and resets the TS flag. The instruction FNSAVE is used to save all states
// of the coprocessor to the memory area specified by the destination operand (tss.i387).
        if (last_task_used_math == current)
                __asm__("clts ; fnsave %0 ; frstor %0"::"m" (p->tss.i387));

// Next, the process page table is copied, that is, the base address and the limit in the new
// task code and data segment descriptors are set, and the page table is copied. If an error
// occurs (the return value is not 0), the corresponding entry in the task array is reset and
// the memory page allocated for the new task structure is released.
        if (copy_mem(nr,p)) {               // The return is not 0, indicating an error.
                task[nr] = NULL;
                free_page((long) p);
                return -EAGAIN;
        }
// Because the newly created child will share the open file with the parent process, if the
// file is opened in the parent, the number of times the corresponding file is opened needs
// to be increased by one. For the same reason, you need to increase the number of reference
// of the i nodes of the current process (parent process) by 1 for pwd, root, and executable.
        for (i=0; i<NR_OPEN;i++)
                if (f=p->filp[i])
                        f->f_count++;
        if (current->pwd)
                current->pwd->i_count++;
        if (current->root)
                current->root->i_count++;
        if (current->executable)
                current->executable->i_count++;
        if (current->library)
                current->library->i_count++;
// Subsequently, the new task TSS and LDT segment descriptors entries are set in the GDT. The
// limit of both segments is set to 104 bytes (see include/asm/system.h, lines 52-66). Then
// set the relationship list pointers between the processes, that is, insert the new process
// into the child process linked list of the current process. That is, the parent process of
// the new process is set to the current process, and the latest child process pointer p_cptr
// and the young sibling process pointer p_ysptr of the new process are set to be empty. Then
// let the new process's buddy process pointer p_osptr be set equal to the parent's latest child
// pointer. If the current process has other child processes, let the young sibling pointer
// p_yspter of the neighboring process point to the new process, and let child pointer of the
// current process point to this new process. Then set the new process to the ready state and
// finally return the new process id. set_tss_desc() and set_ldt_desc() are defined in file
// include/asm/system.h, 52-66. "gdt+(nr<<1)+FIRST_TSS_ENTRY" is the address of the TSS
// descriptor of the task nr in the global table. Since each task occupies 2 items in the GDT
// table, '(nr<<1)' should be included in the above formula. Note that the task register TR
// is automatically loaded by the CPU during task switching.
        set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
        set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
        p->p_pptr = current;                // parent pointer.
        p->p_cptr = 0;
        p->p_ysptr = 0;
        p->p_osptr = current->p_cptr;       // old sibling.
        if (p->p_osptr)                     // if old sibling exist, its young
                p->p_osptr->p_ysptr = p;    // sibling points to this new process.
        current->p_cptr = p;                // I am the current new child.
        p->state = TASK_RUNNING;        /* do this last, just in case */
        return last_pid;
}

// Obtain a unique process number last_pid for the new process. The function returns the task
// number (array item) in the task array.
// First get the new process id. If the global variable last_pid incremented by one is outside
// the representation range, re-use the pid number from 1. Then search for the pid just set
// in the task array to see if it is already in use. If it is, then jump to the beginning of
// the function to regain a pid number. Otherwise it means we have found a unique pid and it
// is last_pid. Then look for a free entry for the new task in the task array and return the
// item number. Note that last_pid is a global variable and does not need to be returned. In
// addidion, if 64 items in the task array have been fully occupied at this time, an error code
// is returned.
int find_empty_process(void)
{
        int i;

        repeat:
                if ((++last_pid)<0) last_pid=1;
                for(i=0 ; i<NR_TASKS ; i++)
                        if (task[i] && ((task[i]->pid == last_pid) ||
                                        (task[i]->pgrp == last_pid)))
                                goto repeat;
        for(i=1 ; i<NR_TASKS ; i++)                 // Task 0 is excluded.
                if (!task[i])
                        return i;
        return -EAGAIN;
}

```

8.9.3 **Information**

8.9.3.1 Task Status Segment (TSS)

Figure 8-14 below shows the contents of the task state segment (TSS). The TSS for each task is saved in the task data structure task_struct. Please refer to Chapter 4 for a detailed description of it.

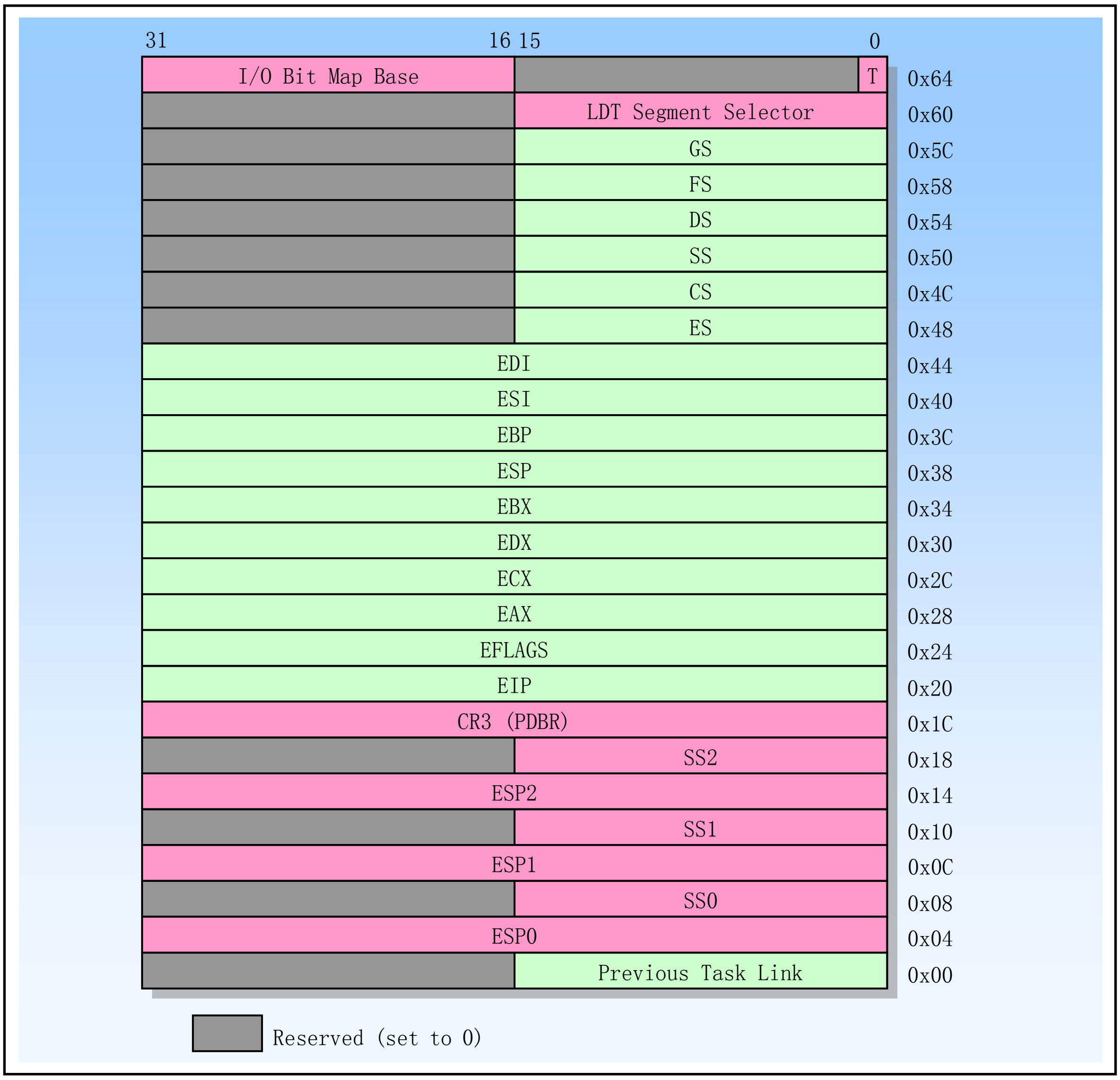


Figure 8-14 Information in the task status segment TSS.

All information required by the CPU management task is stored in a special type of segment, task state segment (TSS). The figure shows the TSS format for performing the 80386 task.

The fields in TSS can be divided into two categories: The first part is a set of information that is dynamically updated when the CPU performs a task switch. These fields are: general purpose registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI), segment registers (ES, CS, SS, DS, FS, GS), flag registers (EFLAGS), instruction pointers (EIP)), the selector of the previous TSS that performed the task (updated only when returning). The second type of field is a static set of information that the CPU will read but will not change. These fields are: the LDT selector for the task, the register containing the base address of the task page directory

(PDBR), the stack pointer for privilege level 0-2, the T bit that causes the CPU to generate a debug exception when the task is switched, I/ O-bit bitmap base address (the upper limit of the length is the upper limit of the length of the TSS, as described in the TSS descriptor).

The task status segment can be stored anywhere in the linear space. Similar to other types of segments, the TSS is also defined by descriptors. The TSS of the currently executing task is indicated by the task register (TR). The instructions LTR and STR are used to modify and read the selector in the task register (visible portion of the task register).

Each bit in the I/O bit map corresponds to one I/O port. For example, the bit of port 41 is the base address of the I/O bitmap +5, and the bit offset is 1. In protected mode, when an I/O instruction is encountered (IN, INS, OUT, and OUTS), the CPU first checks if the current privilege level is less than the IOPL of the flag register. If this condition is met, the I/O operation is executed. If not, the CPU will check the I/O bit map in the TSS. If the corresponding bit is set, a general protection exception will occur, otherwise the I/O operation will be performed.

If the I/O bitmap base address is set to be greater than or equal to the TSS segment limit length, it means that the TSS segment does not have an I/O permission bitmap, then all I/O instructions of the current privilege layer CPL>IOPL will result in exception protection. By default, the Linux 0.12 kernel sets the I/O bitmap base address to 0x8000, which is obviously larger than the TSS segment limit of 104 bytes, so there is no I/O permission bitmap in the Linux 0.12 kernel.

In Linux 0.12, SS0:ESP0 in the figure is used to store the stack pointer of the task running in kernel mode. SS1: ESP1 and SS2: ESP2 correspond to the stack pointers used when running privilege levels 1 and 2, respectively. These two privilege levels are not used in Linux. The stack pointer is stored in the SS:ESP register when the task is working in user mode. As can be seen from the above, each time the task enters the kernel state, the initial position of the kernel state stack pointer is unchanged, which is at the top position of the page where the task data structure is located.

8.10 sys.c

8.10.1 Function Description

The sys.c program contains many implementation functions for system-calls. Among them, if the function only has the return value -ENOSYS, it means that this version of the Linux kernel has not yet implemented this function, you can refer to the current kernel code to understand their implementation. For a description of all system-call functions, see the header file include/linux/sys.h.

The program contains a lot of functions related to process ID (pid), process group ID (pgrp or pgid), user ID (uid), user group ID (gid), actual user ID (ruid), valid user ID (euid), and session ID. (session) and other operation functions. The following is a brief description of these IDs.

A user has a user ID (uid) and a user group ID (gid). These two IDs are the IDs set for the user in the passwd file, and are often referred to as real user IDs (ruids) and real group IDs (rgids). In the i-node information of each file, the host user ID and group ID are saved, which indicate the file owner and the user group to which it belongs , and are mainly used for the authority discriminating operation when accessing or executing the file. In addition, in the task data structure of a process, three user IDs and group IDs are saved for different functions, as shown in Table 8–6.

Table 8-6 User ID and group ID associated with the process

Type	User ID	Group ID
Process	gid - User ID, indicating the user who owns the process.	gid - the group ID that indicates the user group that owns the process.
Efficient	euid - A efficient user ID indicating the access rights to the file.	egid - the efficient group ID. Indicate the permission to access the file.
Saved	suid - the saved user ID. When the set-user-ID flag of the executable file is set, the suid of the execution file is saved in the suid. Otherwise suid is equal to the euid of the process.	sgid - the saved group ID. When the set-group-ID flag of the execution file is set, the gid of the execution file is stored in the sgid. Otherwise sgid is equal to the process's egid.

The uid and gid of the process are the user ID and group ID of the process owner, that is, the real user ID (ruid) and the real group ID (rgid) of the process. Superusers can modify them using the functions set_uid() and set_gid(). The effective user ID and effective group ID are used for permission judgment when the process accesses the file.

The saved user ID (suid) and the saved group ID (sgid) are used by the process to access a file whose set-user-ID or set-group-ID flag is set. When executing a program, the euid of the process is usually the real user ID, and the egid is usually the real group ID. Therefore, the process can only access the effective user of the process, the files specified by the effective user group, or other files that are allowed to access. However, if the set-user-ID flag of a file is set, then the effective user ID of the process is set to the user ID of the file owner, so the process can access the restricted file with this flag set, and The user ID of the file owner is saved in the suid. Similarly, the set-group-ID flag of the file has a similar effect and is treated the same.

For example, if the owner of a program file is a superuser, but the program sets the set-user-ID flag, then when the program is run by a process, the effective user ID (euid) of the process will be set to the super user's ID (0). So this process has the privileges of the superuser. A practical example is the passwd command for Linux. This command is a program that sets the set-user-Id, thus allowing the user to modify their own password. Because the program needs to write the user's new password to the /etc/passwd file, and only the superuser has write access to the file, the passwd program needs to use the set-user-ID flag.

In addition, the process also has a process ID (pid) that identifies its own attribute, a process group ID (pgrp or pgid) of the owning process group, and a session ID (session) of the owning session. These three IDs are used to indicate the relationship between the process and the process, regardless of the user ID and the group ID.

8.10.2 Code comments

Program 8-9 linux/kernel/sys.c

```
/*
 *  linux/kernel/sys.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <errno.h> Error number header file. Contains various error numbers in the system.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
//     communication.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
//     commonly used functions of the kernel.
// <linux/config.h> Kernel configuration header file. Define keyboard language and hard disk
//     type (HD_TYPE) options.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//     segment register operations.
// <sys/times.h> Defines the running time structure tms and the times() function prototype in
//     the process.
// <sys/utsname.h> System name structure header file.
// <sys/param.h> Parameter file. Some hardware-related parameter values are given.
// <sys/resource.h> Resource file. Contains information on the limits and utilization of system
//     resources used by processes.
// <string.h> String header file. Defines some embedded functions about string operations.
#include <errno.h>

#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/kernel.h>
#include <linux/config.h>
#include <asm/segment.h>
#include <sys/times.h>
#include <sys/utsname.h>
#include <sys/param.h>
#include <sys/resource.h>
#include <string.h>

/*
 * The timezone where the local system is located.  Used as a default by some
 * programs who obtain this value by using gettimeofday.
 */
// The time zone structure 1st field (tz_minuteswest) represents the number of minutes west
// of GMT; the second field (tz_dsttime) is daylight saving time (DST) adjustment type. This
// structure is defined in include/sys/time.h.
struct timezone sys_tz = { 0, 0};

// Obtain the session ID of the process group according to the process group ID pgrp.
// This function is implemented in file kernel/exit.c, line 161.
extern int session_of_pgrp(int pgrp);

// Fetch the date and time (ftime - fetch time).
// System-calls with return value -ENOSYS indicates it has not been implemented.
int sys_ftime()
{
        return -ENOSYS;
}

int sys_break()
{
        return -ENOSYS;
}

```

```
// Used by the current process to debug the child process.
int sys_ptrace()
{
        return -ENOSYS;
}

// Change and print the terminal line settings.
int sys_stty()
{
        return -ENOSYS;
}

// Get the terminal line setting information.
int sys_gtty()
{
        return -ENOSYS;
}

// Rename the filename.
int sys_rename()
{
        return -ENOSYS;
}

int sys_prof()
{
        return -ENOSYS;
}

/*
 * This is done BSD-style, with no consideration of the saved gid, except
 * that if you set the effective gid, it sets the saved gid too.  This
 * makes it possible for a setgid program to completely drop its privileges,
 * which is often a useful assertion to make when you are doing a security
 * audit over a program.
 *
 * The general idea is that a program which uses just setregid() will be
 * 100% compatible with BSD.  A program which uses just setgid() will be
 * 100% compatible with POSIX w/ Saved ID's.
 */
// Set the real and/or effective group ID (gid) of the current task. If the task does not have
// superuser privileges, then only its real group ID and effective group ID can be swapped.
// If the task has superuser privileges, you can set the effective and real group IDs arbitrarily,
// and the saved gid (sgid) is set to a effective gid (egid). The real group ID (rgid) refers
// to the current gid of the process.
int sys_setregid(int rgid, int egid)
{
        if (rgid>0) {
                if ((current->gid == rgid) ||
                    suser())
                        current->gid = rgid;
                else
                        return(-EPERM);
        }
        if (egid>0) {
                if ((current->gid == egid) ||
                    (current->egid == egid) ||
                    suser()) {
                        current->egid = egid;
                        current->sgid = egid;
                } else
                        return(-EPERM);
        }
        return 0;
}

/*
 * setgid() is implemeneted like SysV w/ SAVED_IDS
 */
// Set the process group id (gid). If the task does not have superuser privileges, it can use
// setgid() to set its effective gid to its saved gid (sgid) or its real gid (rgid). If the
// task has superuser privileges, the rgid, egid, and sgid are all set to the gid specified
// by the parameter.
int sys_setgid(int gid)
{
        if (suser())
                current->gid = current->egid = current->sgid = gid;
        else if ((gid == current->gid) || (gid == current->sgid))
                current->egid = gid;
        else
                return -EPERM;
        return 0;
}

// Turn process billing on or off.
int sys_acct()
{
        return -ENOSYS;
}

// Map any physical memory to the virtual address space of the process.
int sys_phys()
{
        return -ENOSYS;
}

int sys_lock()
{
        return -ENOSYS;
}

int sys_mpx()
{
        return -ENOSYS;
}

```

```
int sys_ulimit()
{
        return -ENOSYS;
}

// Returns the time (in seconds) from January 1, 1970, 00:00:00 GMT.
// If parameter tloc is not null, then the time value is also stored there. Since the location
// pointed to by the parameter is in user space, you need to use the function put_fs_long()
// to store the time value in user space. When running in the kernel, the segment register fs
// is pointed to the current user data space by default. So the function can use the fs segment
// register to access values in user space.
int sys_time(long * tloc)
{
        int i;

        i = CURRENT_TIME;
        if (tloc) {
                verify_area(tloc,4);      // Verify mem capacity is sufficient (4 bytes).
                put_fs_long(i,(unsigned long *)tloc);
        }
        return i;
}

/*
 * Unprivileged users may change the real user id to the effective uid
 * or vice versa.  (BSD-style)
 *
 * When you set the effective uid, it sets the saved uid too.  This
 * makes it possible for a setuid program to completely drop its privileges,
 * which is often a useful assertion to make when you are doing a security
 * audit over a program.
 *
 * The general idea is that a program which uses just setreuid() will be
 * 100% compatible with BSD.  A program which uses just setuid() will be
 * 100% compatible with POSIX w/ Saved ID's.
 */
// Set the real and/or effective user id (uid) of the task. If the task does not have superuser
// privileges, then only its real uid (ruid) and effective uid (euid) can be swapped. If the
// task has superuser privileges, you can arbitrarily set effective and real user IDs. The saved
// uid (suid) is set to the same value as the euid.
int sys_setreuid(int ruid, int euid)
{
        int old_ruid = current->uid;

        if (ruid>0) {
                if ((current->euid==ruid) ||
                    (old_ruid == ruid) ||
                    suser())
                        current->uid = ruid;
                else
                        return(-EPERM);
        }
        if (euid>0) {
                if ((old_ruid == euid) ||
                    (current->euid == euid) ||
                    suser()) {
                        current->euid = euid;
                        current->suid = euid;
                } else {
                        current->uid = old_ruid;
                        return(-EPERM);
                }
        }
        return 0;
}

/*
 * setuid() is implemeneted like SysV w/ SAVED_IDS
 *
 * Note that SAVED_ID's is deficient in that a setuid root program
 * like sendmail, for example, cannot set its uid to be a normal
 * user and then switch back, because if you're root, setuid() sets
 * the saved uid too.  If you don't like this, blame the bright people
 * in the POSIX commmittee and/or USG.  Note that the BSD-style setreuid()
 * will allow a root program to temporarily drop privileges and be able to
 * regain them by swapping the real and effective uid.
 */
// Set the task user ID (uid). If the task does not have superuser privileges, it can use setuid()
// to set its effective uid (euid) to its saved uid (suid) or its real uid (ruid). If the task
// has superuser privileges, the ruid, euid, and suid will be set to the uid specified by the
// parameter.
int sys_setuid(int uid)
{
        if (suser())
                current->uid = current->euid = current->suid = uid;
        else if ((uid == current->uid) || (uid == current->suid))
                current->euid = uid;
        else
                return -EPERM;
        return(0);
}

// Set the system boot time. The parameter tptr is the time value (in seconds) that is counted
// from January 1, 1970, at 00:00:00 GMT.
// The calling process must have superuser privileges. Where HZ=100 is the operating frequency
// of the kernel system. Since the location of the parameter pointer is in user space, you need
// to use the function get_fs_long() to access the value. When running in the kernel, the segment
// register fs is pointed to the current user data space by default. So the function can use
// fs to access values in user space. The current time value provided by the function parameter
// minus the time second value (jiffies/HZ) that the system has been running is the boot time
// seconds.
int sys_stime(long * tptr)
{
        if (!suser())
                return -EPERM;
        startup_time = get_fs_long((unsigned long *)tptr) - jiffies/HZ;
        jiffies_offset = 0;
        return 0;
}

//// Get the current task runtime statistics.
// Returns the task runtime statistics of the tms structure at the user data space pointed to
// by tbuf. The tms structure includes the process user runtime, kernel runtime, child user
// runtime, and child kernel runtime. The return value of the function is the ticks that the
// system runs to the current time.
int sys_times(struct tms * tbuf)
{
        if (tbuf) {
                verify_area(tbuf,sizeof *tbuf);
                put_fs_long(current->utime,(unsigned long *)&tbuf->tms_utime);
                put_fs_long(current->stime,(unsigned long *)&tbuf->tms_stime);
                put_fs_long(current->cutime,(unsigned long *)&tbuf->tms_cutime);
                put_fs_long(current->cstime,(unsigned long *)&tbuf->tms_cstime);
        }
        return jiffies;
}

//// Set the program's end position in memory.
// When the value of the parameter end_data_seg is reasonable and the system does have enough
// memory and the process does not exceed its maximum data segment size, the function sets the
// value specified by end_data_seg at the end of the data segment. This value must be greater
// than the end of the code and less than 16KB of the end of the stack. The return value is
// the new end value of the data segment (if the return value is different from the required
// value, an error has occurred). This function is not directly called by the user, but is wrapped
// by the libc library function, and the return value is not the same.
int sys_brk(unsigned long end_data_seg)
{
// If the parameter is greater than the end of the code and is less than (stack - 16KB), set
// the new data segment end value.
        if (end_data_seg >= current->end_code &&
            end_data_seg < current->start_stack - 16384)
                current->brk = end_data_seg;
        return current->brk;               // Returns the current data segment end value.
}

/*
 * This needs some heave checking ...
 * I just haven't get the stomach for it. I also don't fully
 * understand sessions/pgrp etc. Let somebody who does explain it.
 *
 * OK, I think I have the protection semantics right.... this is really
 * only important on a multi-user system anyway, to make sure one user
 * can't send a signal to a process owned by another.  -TYT, 12/12/91
 */
//// Set the process group id of the specified process pid to pgid.
// The parameter pid is the process id. If the parameter pid is 0, then let this pid be equal
// to pid of the current process. The parameter pgid specifies the process group id. If it is
// 0, let it be equal to the process group id of the process pid. If the function is used to
// move a process from one process group to another, the two process groups must belong to the
```

```
    // same session. In this case, the parameter pgid specifies the existing process group ID to
    // join, and the session ID of the group must be the same as the process to be joined (L263).
245 int sys_setpgid(int pid, int pgid)
246 {
247         int i;
248 
    // If the parameter pid is 0, the pid is set to the pid of the current process. If the parameter
    // pgid is 0, then pgid is also the pid of the current process. If pgid is less than 0, an invalid
    // error code is returned.
249         if (!pid)
250                 pid = current->pid;
251         if (!pgid)
252                 pgid = current->pid;
253         if (pgid < 0)
254                 return -EINVAL;
    // Scan the task array for the task with the specified process pid. If the process with the
    // process ID is pid is found, and the parent process of the process is the current process
    // or the process is the current process, then if the task is already the session leader, an
    // error is returned. If the session id of the task is different from the current process, or
    // the specified process group id pgid is different from the pid, and the session id of the
    // pgid process group is different from the session id of the current process, an error is
    // returned. Otherwise, set the pgrp field of the found process to pgid and return 0. If the
    // process with the specified pid is not found, the return process doesn't have an error code.
255         for (i=0 ; i<NR_TASKS ; i++)
256                 if (task[i] && (task[i]->pid == pid) &&
257                     ((task[i]->p_pptr == current) ||
258                      (task[i] == current))) {
259                         if (task[i]->leader)
260                                 return -EPERM;
261                         if ((task[i]->session != current->session) ||
262                             ((pgid != pid) &&
263                              (session_of_pgrp(pgid) != current->session)))
264                                 return -EPERM;
265                         task[i]->pgrp = pgid;
266                         return 0;
267                 }
268         return -ESRCH;
269 }
270 
    // Returns the process group id of the current process. Equivalent to getpgid(0).
271 int sys_getpgrp(void)
272 {
273         return current->pgrp;
274 }
275 
    // Create a session (ie set its leader=1) and set its session id = its group id = its process
    // id. If the current process is already the session leader and is not a superuser, an error
    // is returned. Otherwise, set the current process to be new session leader (leader = 1). The
    // session and group id pgrp are set to equal to the process pid, and the current process has
    // no control terminal. The last system-call returns the session id.
276 int sys_setsid(void)
277 {
278         if (current->leader && !suser())
```

```
                return -EPERM;
        current->leader = 1;
        current->session = current->pgrp = current->pid;
        current->tty = -1;                // the current process has no control terminal.
        return current->pgrp;
}

/*
 * Supplementary group ID's
 */
//// Get the other auxiliary user group id of the current process.
// The groups[] array in the task structure holds multiple user group ids to which the process
// belongs. The array has a total of NGROUPS items. If the value of an item is NOGROUP (that
// is, -1), it means that all items are idle after the start of the item. Otherwise the user
// group id is saved in the array item.
// The parameter gidsetsize is the maximum number of user group ids that can be stored in the
// user cache, that is, the maximum number of items in the group list; the grouplist is the
// user space cache that stores these user group numbers.
int sys_getgroups(int gidsetsize, gid_t *grouplist)
{
        int     i;

// First verify that the user cache space pointed to by the grouplist is sufficient, and then
// obtain the user group id one by one from the groups[] array of the current process structure
// and copy it into the user cache. During the copying process, if the number of items in groups[]
// is greater than the number specified by the given parameter gidsetsize, it means that the
// cache given is too small to accommodate all the groups of the current process. The operation
// will return with an error code. If the copy operation is ok, the function will eventually
// return the number of copied user group ids.
        if (gidsetsize)
                verify_area(grouplist, sizeof(gid_t) * gidsetsize);

        for (i = 0; (i < NGROUPS) && (current->groups[i] != NOGROUP);
             i++, grouplist++) {
                if (gidsetsize) {
                        if (i >= gidsetsize)
                                return -EINVAL;
                        put_fs_word(current->groups[i], (short *) grouplist);
                }
        }
        return(i);                     // Returns the number of user group ids.
}

// Set the other secondary user group ids to which the current process belongs.
// The parameter gidsetsize is the number of user group ids to be set; the grouplist is the
// user space cache containing the user group ids.
int sys_setgroups(int gidsetsize, gid_t *grouplist)
{
        int     i;

// First check the validity of the permissions and parameters. Only the superuser can modify
// or set the secondary user group ids of the current process, and the number of items cannot
// exceed the capacity of the groups [NGROUPS] array. Then copy the user group id one by one
// from the user buffer to the array. A total of gidsetsize is copied. If the number of copies
// does not fill in groups[], fill in the next item with a value of -1 (NOGROUP). Finally, the
// function returns 0.
        if (!suser())
                return -EPERM;
        if (gidsetsize > NGROUPS)
                return -EINVAL;
        for (i = 0; i < gidsetsize; i++, grouplist++) {
                current->groups[i] = get_fs_word((unsigned short *) grouplist);
        }
        if (i < NGROUPS)
                current->groups[i] = NOGROUP;
        return 0;
}

// Check if the current process belongs to user group grp. Yes ret 1, otherwise returns 0.
int in_group_p(gid_t grp)
{
        int     i;

// If the effective group id (egid) of the current process is grp, the process belongs to the
// grp group, the function returns 1. Otherwise, it scans the process's secondary user group
// array for the grp group id. If so, the function also returns 1. If the item with the value
// NOGROUP is scanned, it means that the full valid item has been scanned and no matching group
// id is found, so the function returns 0.
        if (grp == current->egid)
                return 1;

        for (i = 0; i < NGROUPS; i++) {
                if (current->groups[i] == NOGROUP)
                        break;
                if (current->groups[i] == grp)
                        return 1;
        }
        return 0;
}

// The utsname structure contains some string fields that hold the name of the system. It contains
// 5 fields, which are: the name of the current operating system, the network node name (host
// name), the current operating system release level, the operating system version number, and
// the hardware type name that the system is running. This structure is defined in the
// include/sys/utsname.h file. Here they are set to their default values using the constant
// symbols in include/linux/config.h file. They are: "Linux", "(none)", "0", "0.12", "i386".
static struct utsname thisname = {
        UTS_SYSNAME, UTS_NODENAME, UTS_RELEASE, UTS_VERSION, UTS_MACHINE
};

// Get system name information.
int sys_uname(struct utsname * name)
{
        int i;

        if (!name) return -ERROR;
        verify_area(name,sizeof *name);
        for(i=0;i<sizeof *name;i++)
                put_fs_byte(((char *) &thisname)[i],i+(char *) name);
        return 0;
}

/*
 * Only sethostname; gethostname can be implemented by calling uname()
 */
// Set the system host name (network node name).
// The parameter name points to the buffer containing the host name string in the user data
// area; len is the host name string length.
int sys_sethostname(char *name, int len)
{
        int     i;

// The system hostname can only be set or modified by the superuser, and the hostname length
// cannot exceed the maximum length MAXHOSTNAMELEN.
        if (!suser())
                return -EPERM;
        if (len > MAXHOSTNAMELEN)
                return -EINVAL;
        for (i=0; i < len; i++) {
                if ((thisname.nodename[i] = get_fs_byte(name+i)) == 0)
                        break;
        }
// After the copy is completed, if the string provided by the user does not contain NULL
// characters, if the length of the copied hostname does not exceed MAXHOSTNAMELEN, a NULL
// is added after the host name string. If MAXHOSTNAMELEN characters have been filled, change
// the last character to NULL.
        if (thisname.nodename[i]) {
                thisname.nodename[i>MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] = 0;
        }
        return 0;
}

// Get the resource limits of the current process.
// An array rlim[RLIM_NLIMITS] is defined in task's structure to control the boundaries of the
// system's use of system resources. Each item in the array is an rlimit structure with two
// fields. One specifies the current limit (soft limit) of the specified resource, and the other
// indicates the system's maximum limit (hard limit) for the specified resource. Each item of
// the rlim[] array corresponds to the limit information of a resource for the current process.
// The Linux 0.12 system has a limit on six resources, namely RLIM_NLIMITS=6. Please refer to
// lines 41-46 in file include/sys/resource.h. The 'resource' specifies the name of the resource
// we are consulting. It is actually the index of the rlim[] array in the task structure. rlim
// is a user buffer pointer to the rlimit structure, which is used to store the obtained resource
// limit information.
int sys_getrlimit(int resource, struct rlimit *rlim)
{
// The resource being queried is actually the index value of the rlim[] array in the process
// task structure. The index value can of course not be greater than the maximum number of items
// in the array RLIM_NLIMITS. After verifying that the user buffer is sufficient, the resource
// structure information is copied into it, and returns 0.
```

```
        if (resource >= RLIM_NLIMITS)
                return -EINVAL;
        verify_area(rlim,sizeof *rlim);
        put_fs_long(current->rlim[resource].rlim_cur,        // Current (soft) limits.
                    (unsigned long *) rlim);
        put_fs_long(current->rlim[resource].rlim_max,        // System (hard) limits.
                    ((unsigned long *) rlim)+1);
        return 0;
}

// Sets resource limits for the current process.
// The parameter resource specifies the resource name for which we set the limit. It is actually
// the index of the rlim[] array in the task structure. The parameter rlim is a user buffer
// pointer to the rlimit structure for kernel to read new resource limits.
int sys_setrlimit(int resource, struct rlimit *rlim)
{
        struct rlimit new, *old;

// First determine the validity of the parameter resource (the task structure rlim[] index).
// Then let the rlimit structure pointer 'old' point to the current rlimit structure of the
// specified resource. The resource limit information provided by the user is then copied to
// the temporary rlimit structure 'new'. At this time, if the soft limits value or the hard
// limits value in the 'new' structure is greater than the original limits, and the current
// is not a superuser, the permission error is returned. Otherwise, it means that the information
// in 'new' is reasonable or the process is a super user, then the information specified in
// the original process is changed to the information in the 'new' structure, and 0 is returned.
        if (resource >= RLIM_NLIMITS)
                return -EINVAL;
        old = current->rlim + resource;            // old = current->rlim[resource]
        new.rlim_cur = get_fs_long((unsigned long *) rlim);
        new.rlim_max = get_fs_long(((unsigned long *) rlim)+1);
        if (((new.rlim_cur > old->rlim_max) ||
             (new.rlim_max > old->rlim_max)) &&
            !suser())
                return -EPERM;
        *old = new;
        return 0;
}

/*
 * It would make sense to put struct rusuage in the task_struct,
 * except that would make the task_struct be *really big*.  After
 * task_struct gets moved into malloc'ed memory, it would
 * make sense to do this.  It will make moving the rest of the information
 * a lot simpler!  (Which we're not doing right now because we're not
 * measuring them yet).
 */
// Get resource usage information of the specified process.
// This syscall provides the current process or its terminated or waiting child resource usage.
// If the parameter 'who' is equal to RUSAGE_SELF, the resource usage information of the current
// process is returned. If 'who' is RUSAGE_CHILDREN, returns the terminated or waiting child
// resource usage information of the current process. Symbolic constants RUSAGE_SELF and
// RUSAGE_CHILDREN and the rusage structure are all defined in file include/sys/resource.h.
```

```
int sys_getrusage(int who, struct rusage *ru)
{
        struct rusage r;
        unsigned long   *lp, *lpend, *dest;

// First check the validity of the process specified by the parameter 'who'. If 'who' is neither
// RUSAGE_SELF (specifying the current process) nor RUSAGE_CHILDREN (specifying the child),
// it is returned with an invalid parameter code. Otherwise, after verifying the user buffer
// area specified by the pointer ru, the temporary rusage structure area 'r' is cleared.
        if (who != RUSAGE_SELF && who != RUSAGE_CHILDREN)
                return -EINVAL;
        verify_area(ru, sizeof *ru);
        memset((char *) &r, 0, sizeof(r));      // at the end of include/strings.h

// If the parameter who is RUSAGE_SELF, the current process resource usage information is copied
// into the r structure. If the specified process who is RUSAGE_CHILDREN, the terminated or
// waiting child resource usage of the current process is copied to the temporary rusuage
// structure r. The macros CT_TO_SECS and CT_TO_USECS are used to convert the current system
// ticks into seconds and microseconds. They are defined in file include/linux/sched.h.
// jiffies_offset is the system parameter error adjustment.
        if (who == RUSAGE_SELF) {
                r.ru_utime.tv_sec = CT_TO_SECS(current->utime);
                r.ru_utime.tv_usec = CT_TO_USECS(current->utime);
                r.ru_stime.tv_sec = CT_TO_SECS(current->stime);
                r.ru_stime.tv_usec = CT_TO_USECS(current->stime);
        } else {
                r.ru_utime.tv_sec = CT_TO_SECS(current->cutime);
                r.ru_utime.tv_usec = CT_TO_USECS(current->cutime);
                r.ru_stime.tv_sec = CT_TO_SECS(current->cstime);
                r.ru_stime.tv_usec = CT_TO_USECS(current->cstime);
        }
// Then let the lp pointer point to the r structure, lpend to the end of the r structure, and
// the dest pointer to the ru structure in user space. Finally, copy the information in r into
// the user space ru and return 0.
        lp = (unsigned long *) &r;
        lpend = (unsigned long *) (&r+1);
        dest = (unsigned long *) ru;
        for (; lp < lpend; lp++, dest++)
                put_fs_long(*lp, dest);
        return(0);
}

// Get the current time of the system and return it in the specified format.
// The timeval structure contains two fields, seconds and microseconds (tv_sec and tv_usec).
// The timezone structure contains two fields, the number of minutes west of Greenwich Mean
// Time (tz_minuteswest) and the daylight saving time (dst) adjustment type (tz_dsttime). Both
// structures are defined in the include/sys/time.h file.
int sys_gettimeofday(struct timeval *tv, struct timezone *tz)
{
// If the timeval structure pointer is not empty, the current time (seconds and microseconds)
// is returned in the structure; if the pointer of the timezone structure in the given user
// data space is not empty, the structure is also returned. The startup_time in the code is
// the system boot time (seconds). The macros CT_TO_SECS and CT_TO_USECS are used to convert
```

```
    // the current systems ticks to be expressed in seconds and microseconds. They are defined in
    // the include/linux/sched.h file. Jiffies_offset is the system ticks error adjustment.
442         if (tv) {
443                 verify_area(tv, sizeof *tv);
444                 put_fs_long(startup_time + CT_TO_SECS(jiffies+jiffies_offset),
445                             (unsigned long *) tv);
446                 put_fs_long(CT_TO_USECS(jiffies+jiffies_offset),
447                             ((unsigned long *) tv)+1);
448         }
449         if (tz) {
450                 verify_area(tz, sizeof *tz);
451                 put_fs_long(sys_tz.tz_minuteswest, (unsigned long *) tz);
452                 put_fs_long(sys_tz.tz_dsttime, ((unsigned long *) tz)+1);
453         }
454         return 0;
455 }
456
457 /*
458  * The first time we set the timezone, we will warp the clock so that
459  * it is ticking GMT time instead of local time.  Presumably,
460  * if someone is setting the timezone then we are running in an
461  * environment where the programs understand about timezones.
462  * This should be done at boot time in the /etc/rc script, as
463  * soon as possible, so that the clock can be set right.  Otherwise,
464  * various programs will get confused when the clock gets warped.
465  */
    // Set the current time of the system.
    // The parameter tv is a pointer to the timeval structure in the user data area. tz is a pointer
    // to the timezone structure in the user data area. This operation requires superuser privileges.
    // If both are empty, nothing is done and the function returns 0.
466 int sys_settimeofday(struct timeval *tv, struct timezone *tz)
467 {
468         static int      firsttime = 1;
469         void            adjust_clock();
470
    // Superuser privileges are required to set the current time of the system. If the tz pointer
    // is not empty, set the system time zone information, that is, copy the user timezone structure
    // information to the sys_tz structure in the system (see line 24). If the system-call is called
    // for the first time and the parameter tv pointer is not empty, adjust the system clock value.
471         if (!suser())
472                 return -EPERM;
473         if (tz) {
474                 sys_tz.tz_minuteswest = get_fs_long((unsigned long *) tz);
475                 sys_tz.tz_dsttime = get_fs_long(((unsigned long *) tz)+1);
476                 if (firsttime) {
477                         firsttime = 0;
478                         if (!tv)
479                                 adjust_clock();
480                 }
481         }
    // If the timeval structure pointer tv of the parameter is not empty, the system clock is set
    // with the structure information. First, the system time expressed by the second value (sec)
    // plus the microsecond value (usec) is obtained from the position indicated by tv, and then
// the system startup time global variable startup_time is modified by the second value, and
// the system error value jiffies_offset is set by the microsecond value.
        if (tv) {
                int sec, usec;

                sec = get_fs_long((unsigned long *)tv);
                usec = get_fs_long(((unsigned long *)tv)+1);

                startup_time = sec - jiffies/HZ;
                jiffies_offset = usec * HZ / 1000000 - jiffies%HZ;
        }
        return 0;
}

/*
 * Adjust the time obtained from the CMOS to be GMT time instead of
 * local time.
 *
 * This is ugly, but preferable to the alternatives.  Otherwise we
 * would either need to write a program to do it in /etc/rc (and risk
 * confusion if the program gets run more than once; it would also be
 * hard to make the program warp the clock precisely n hours)  or
 * compile in the timezone information into the kernel.  Bad, bad....
 *
 * XXX Currently does not adjust for daylight savings time.  May not
 * need to do anything, depending on how smart (dumb?) the BIOS
 * is.  Blast it all.... the best thing to do not depend on the CMOS
 * clock at all, but get the time via NTP or timed if you're on a
 * network....                              - TYT, 1/1/92
 */
// Adjust the system startup time to the time based on GMT.
// The startup_time unit is seconds, so you need to multiply the time zone minute by 60.
void adjust_clock()
{
        startup_time += sys_tz.tz_minuteswest*60;
}

// Set the creation file attribute mask of the current process to mask & 0777 and return the
// original mask.
int sys_umask(int mask)
{
        int old = current->umask;

        current->umask = mask & 0777;
        return (old);
}

```

8.11 vsprintf.c

8.11.1 Function Description

The program mainly includes the vsprintf() function, which formats the parameters and outputs them to the buffer. Since this function is a standard function in the C library, there is basically no content related to the working principle of the kernel, so it can be skipped. In order to understand its application in the kernel, you can directly read the instructions of the function after the code. Please also refer to the C library function manual for how to use the vsprintf() function.

8.11.2 Code Comments

Program 8-10 linux/kernel/vsprintf.c

```
 1 /*
 2  *  linux/kernel/vsprintf.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 7 /* vsprintf.c -- Lars Wirzenius & Linus Torvalds. */
 8 /*
 9  * Wirzenius wrote this portably, Torvalds fucked it up :-)
10  */
   // Lars Wirzenius is a friend of Linus and worked in an office at Helsinki University. When
   // developing Linux in the summer of 1991, Linus was not very familiar with the C language at
   // the time, and would not be familiar with the variable parameter list function. So Lars
   // Wirzenius wrote this code for the kernel to display messages. He later (1998) admitted that
   // there was a bug in this code that was not discovered until 1994 and was corrected. This bug
   // is when the * is used as the output field width, the code forgets to increment the pointer
   // to skip the asterisk. This bug still exists in this code (line 130). His personal homepage
   // is http://liw.iki.fi/liw/
11
   // #include <stdarg.h> Standard parameter file. Define a list of variable parameters in the
   //     form of macros. It mainly describes a type (va_list) and three macros (va_start, va_arg
   //     and va_end) for the vsprintf, vprintf, vfprintf functions.
   // #include <string.h> A string header file. Mainly defines some embedded functions about
   //     string operations.
12 #include <stdarg.h>
13 #include <string.h>
14
15 /* we use this so that we can do without the ctype library */
16 #define is_digit(c)     ((c) >= '0' && (c) <= '9')      // check if it's a digital char.
17
   // Convert a string of characters to an integer. The input is a pointer to a numeric string
   // pointer and returns the result value. In addition, the pointer will move forward.
18 static int skip_atoi(const char **s)
19 {
20         int i=0;
21
22         while (is_digit(**s))
```

```
                i = i*10 + *((*s)++) - '0';
        return i;
}

// Here defines symbol constants for various conversion types.
#define ZEROPAD 1               /* pad with zero */
#define SIGN    2               /* unsigned/signed long */
#define PLUS    4               /* show plus */
#define SPACE   8               /* space if plus */
#define LEFT    16              /* left justified */
#define SPECIAL 32              /* 0x */
#define SMALL   64              /* use 'abcdef' instead of 'ABCDEF' */

// Division operation. Input: n is the dividend, base is the divisor; result: n is the quotient,
// and the function returns the remainder. See 3.3.2 for embedded assembly.
#define do_div(n,base) ({ \
int __res; \
__asm__("divl %4":"=a" (n),"=d" (__res):"0" (n),"1" (0),"r" (base)); \
__res; })

// Converts an integer to a string of the specified radix.
// Input: num - integer; base - radix; size - the length of the string;
// precision - the length of the number (precision); type - type options.
// Output: The converted string is stored in the buffer at the str pointer. The return value
// is a pointer to the end of the string after the number is converted to a string.
static char * number(char * str, int num, int base, int size, int precision
        ,int type)
{
        char c,sign,tmp[36];
        const char *digits="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        int i;

// If 'type' indicates a lowercase letter, a lowercase set of letters is defined. If 'type'
// indicates that you want to adjust left, the zero-fill flag in 'type' is masked. If the base
// is less than 2 or greater than 36, exit the program. That is, the program can only process
// numbers with a base between 2-32.
        if (type&SMALL) digits="0123456789abcdefghijklmnopqrstuvwxyz";
        if (type&LEFT) type &= ~ZEROPAD;
        if (base<2 || base>36)
                return 0;
// If 'type' indicates zero-filled, set variable c='0', otherwise set c equal space char.
// If 'type' indicates a signed number and the value num is less than 0, then negative sign
// is set and num is taken as an absolute value. Otherwise, if the 'type' indicates that it
// is a plus sign, then set sign=plus, otherwise if the 'type' has a space sign then sign =space,
// otherwise set sign to 0.
        c = (type & ZEROPAD) ? '0' : ' ' ;
        if (type&SIGN && num<0) {
                sign='-';
                num = -num;
        } else
                sign=(type&PLUS) ? '+' : ((type&SPACE) ? ' ' : 0);
// If signed, the width value is decremented by 1. If the type indicates a special conversion,
// then reduce the hexadecimal width by another 2 (for 0x) and for the octal width minus 1 (for
   // placing a zero before result).
57         if (sign) size--;
58         if (type&SPECIAL)
59                 if (base==16) size -= 2;
60                 else if (base==8) size--;
   // If num is 0, the temporary string = '0'; otherwise the num is converted to a character form
   // according to the given radix. If the number of digits is greater than the precision, the
   // precision is expanded to the number of digits. The width size minus the number of numeric
   // characters stored.
61         i=0;
62         if (num==0)
63                 tmp[i++]='0';
64         else while (num!=0)
65                 tmp[i++]=digits[do_div(num,base)];
66         if (i>precision) precision=i;
67         size -= precision;

   // From here on, the resulting conversion result is gradually formed and temporarily placed
   // in the string str. If there is no zero-fill and left-aligned flags in the 'type', Spaces
   // indicated by the remaining width is first filled in str. If the sign is needed, store the
   // sign symbol in it.
68         if (!(type&(ZEROPAD+LEFT)))
69                 while(size-->0)
70                         *str++ = ' ';
71         if (sign)
72                 *str++ = sign;
   // If the 'type' indicates a special conversion, a '0' is placed for the first location the
   // octal result; '0x' is stored for the hexadecimal value.
73         if (type&SPECIAL)
74                 if (base==8)
75                         *str++ = '0';
76                 else if (base==16) {
77                         *str++ = '0';
78                         *str++ = digits[33];   // 'X'或'x'
79                 }
   // If there is no left adjust flag in the 'type', the c ('0' or space) is stored in the remaining
   // width, see line 51.
80         if (!(type&LEFT))
81                 while(size-->0)
82                         *str++ = c;
   // At this time, i holds the number of digits of num. If the number of digits is less than the
   // precision, put (precision - i) '0' in str. Then the converted numeric characters are also
   // filled in str. A total of i.
83         while(i<precision--)
84                 *str++ = '0';
85         while(i-->0)
86                 *str++ = tmp[i];
   // If the width value is still greater than zero, it means that there is a left ajustment in
   // the 'type' flag. Then put spaces in the remaining width.
87         while(size-->0)
88                 *str++ = ' ';
89         return str;            // Returns the pointer to the end of the converted string.
90 }
```

```

// The following function sends the formatted output to the string buffer. The parameter fmt
// is the format string; args is a pointer to a parameter list; buf is the output string buffer.
int vsprintf(char *buf, const char *fmt, va_list args)
{
        int len;
        int i;
        char * str;             // Used to hold strings during the conversion.
        char *s;
        int *ip;

        int flags;              /* flags to number() */

        int field_width;        /* width of output field */
        int precision;          /* min. # of digits for integers; max
                                   number of chars for from string */
        int qualifier;          /* ’h’, ’l’, or ’L’ for integer fields */

// First, the character pointer is pointed to the buf, and then the format string is scanned,
// and each format conversion instruction is processed accordingly.
// The format conversion string starts with ’%’. Here we scan ’%’ from the fmt format string
// to find the beginning of the format conversion string. Normal characters that are not format
// directives are stored in str in order.
        for (str=buf ; *fmt ; ++fmt) {
                if (*fmt != ’%’) {
                        *str++ = *fmt;
                        continue;
                }

// Obtains the flag field in the format string and puts into the flags variable.
                /* process flags */
                flags = 0;
                repeat:
                        ++fmt;          /* this also skips first ’%’ */
                        switch (*fmt) {
                                case ’-’: flags |= LEFT; goto repeat;
                                case ’+’: flags |= PLUS; goto repeat;
                                case ’ ’: flags |= SPACE; goto repeat;
                                case ’#’: flags |= SPECIAL; goto repeat;
                                case ’’: flags |= ZEROPAD; goto repeat;
                                }

// Take the current parameter width field value into the field_width variable. If the width
// field is a numeric, it is directly taken as the width value. If the width field is the character
// ’*’, it means that the next specifies the width, so va_arg is called to take the width value.
// If the width value is less than 0, the negative number indicates that it has the flag field
// ’-’ (left-aligned), so it is necessary to add it to the flag variable and take the field
// width value as absolute value.
                /* get field width */
                field_width = -1;
                if (is_digit(*fmt))
                        field_width = skip_atoi(&fmt);
                else if (*fmt == ’*’) {
                        /* it's the next argument */     // bug here, should add "++fmt;"
                        field_width = va_arg(args, int);
                        if (field_width < 0) {
                                field_width = -field_width;
                                flags |= LEFT;
                        }
                }

// The following takes the precision field of the format and puts it into the precision variable.
// The flag for the start of the precision field is '.'. The processing is similar to the width
// field above. If the precision field is a number, it is taken directly as the precision value.
// If the precision field is the character '*', it means that the next parameter specifies the
// precision. So call va_arg to take the precision value. If the width value is less than 0,
// the field precision value is taken as 0.
                /* get the precision */
                precision = -1;
                if (*fmt == '.') {
                        ++fmt;
                        if (is_digit(*fmt))
                                precision = skip_atoi(&fmt);
                        else if (*fmt == '*') {
                                /* it's the next argument */   // should add ++fmt;
                                precision = va_arg(args, int);
                        }
                        if (precision < 0)
                                precision = 0;
                }

// This code analyzes the length modifier and stores it in the qualifer variable. For the meaning
// of h, l, L, see the description after the list.
                /* get the conversion qualifier */
                qualifier = -1;
                if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
                        qualifier = *fmt;
                        ++fmt;
                }

// The conversion format indicator is analyzed below.
// If the indicator is 'c', it means that the corresponding parameter should be a character.
// At this time, if the flag field indicates that it is not left-aligned, the field is preceded
// by a 'width - 1' spaces, and then the parameter character is placed. If the width field is
// still greater than 0, it means that it is left-aligned, then add a 'width - 1' spaces after
// the parameter character.
                switch (*fmt) {
                case 'c':
                        if (!(flags & LEFT))
                                while (--field_width > 0)
                                        *str++ = ' ';
                        *str++ = (unsigned char) va_arg(args, int);
                        while (--field_width > 0)
                                *str++ = ' ';
                        break;

```

```
// If the indicator is 's', it means that the corresponding parameter is a string. Then the
// length of the parameter string is taken first, and if it exceeds the precision field value,
// set the extended precision field equal to string length. If the flag indicates that it's
// not left-aligned, the field is preceded by a 'width-string length' spaces, and then the string
// is placed. If the width is still greater than 0, it means that it is left-aligned, then add
// 'width - string length' spaces after the string.
                case 's':
                        s = va_arg(args, char *);
                        len = strlen(s);
                        if (precision < 0)
                                precision = len;
                        else if (len > precision)
                                len = precision;

                        if (!(flags & LEFT))
                                while (len < field_width--)
                                        *str++ = ' ';
                        for (i = 0; i < len; ++i)
                                *str++ = *s++;
                        while (len < field_width--)
                                *str++ = ' ';
                        break;

// If the format character is 'o', it means that the corresponding parameter needs to be
// converted into a string of octal numbers. Call the number() function to handle it.
                case 'o':
                        str = number(str, va_arg(args, unsigned long), 8,
                                field_width, precision, flags);
                        break;

// If the format converter is 'p', it means that the corresponding parameter is a pointer type.
// At this time, if the parameter does not set the width field, the default width is 8, and
// you need to add zero. Then invoke the number() function for processing.
                case 'p':
                        if (field_width == -1) {
                                field_width = 8;
                                flags |= ZEROPAD;
                        }
                        str = number(str,
                                (unsigned long) va_arg(args, void *), 16,
                                field_width, precision, flags);
                        break;

// If the format converter is 'x' or 'X', it means that the corresponding parameter needs to
// be printed as a hexadecimal output. 'x' means lowercase letters.
                case 'x':
                        flags |= SMALL;
                case 'X':
                        str = number(str, va_arg(args, unsigned long), 16,
                                field_width, precision, flags);
                        break;

// If the format character is 'd', 'i' or 'u', then the parameter is an integer. 'd', 'i' stands
// for symbolic integers, so you need to add a signed flag. 'u' stands for an unsigned integer.
                case 'd':
                case 'i':
                        flags |= SIGN;
                case 'u':
                        str = number(str, va_arg(args, unsigned long), 10,
                                field_width, precision, flags);
                        break;

// If the format indicator is 'n', it means that the number of characters converted so far is
// saved to the position specified by the corresponding parameter pointer. First use va_arg()
// to get the pointer to the parameter, and then store the number of characters already converted
// into the position pointed to by the pointer.
                case 'n':
                        ip = va_arg(args, int *);
                        *ip = (str - buf);
                        break;

// If the format converter is not '%', a '%' is written directly into the output string. If
// there is still a character of the format converter, the character is also written directly
// into the output string, and the loop continues to process the format string. Otherwise, it
// means that it has been processed to the end of the format string, then exit the loop.
                default:
                        if (*fmt != '%')
                                *str++ = '%';
                        if (*fmt)
                                *str++ = *fmt;
                        else
                                --fmt;
                        break;
                }
        }
        *str = '\0';                  // add null to the end of the output string.
        return str-buf;               // return string length.
}

```

8.11.3 Information

8.11.3.1 Format string of vsprintf()

The vsprintf() function is one of the printf() series. These functions all produce formatted output: A format string fmt that determines the output format is received, and the parameters are formatted with the format string to produce a formatted output. The function declaration form is as follows:

```
    int vsprintf(char *buf, const char *fmt, va_list args)
```

The other functions in the printf() series declare a form similar to this. Printf will send the output directly to the standard output device stdout (the display / console), so there is no first parameter (buffer pointer) in the above declaration. cprintf also sends the output to the console. fprintf sends the output to a file, so the first argument will be a file handle. The printf with a 'v' character (for example, vfprintf) indicates that the arguments comes from the va_list args of the va_arg array. printf with a prefix 's' means that the format result is output to the string buffer buf (the buf should have enough space) and end with a null. The following describes in detail how to use the format string.

1. The format string

The format string in the printf family is used to control how functions are converted, formatted, and output their parameters. For each format, there must be a corresponding parameter, and too many parameters will be ignored. The format string contains two types of components, one is a simple character that will be copied directly into the output; the other is a conversion indicator string used to format the corresponding parameter.

2. The format indicator string

The format of the format indicator string is as follows:

```
    %[flags][width][.prec][|h|l|L][type]
```

Each conversion indication string needs to start with a percent sign (%). The meaning of each part is as follows:

- [flags] is an optional sequence of flag characters;
- [width] is an optional width indicator;
- [.prec] is an optional precision indicator;
- [h|l|L] Is an optional input length modifier;
- [type] is a conversion type character (or a conversion indicator).

● flags control output alignment, numeric symbols, decimal points, trailing zeros, binary, octal, or hexadecimal, see the notes in lines 27-33 above. The flag characters and their meanings are as follows:

'#' Indicates that the corresponding parameter needs to be converted to a "special form". For octal (o), the first character of the converted string must be a zero. For hexadecimal (x or X), the converted string must start with '0x' or '0X'. For e, E, f, F, g, and G, even if there are no decimal places, the conversion result will always have a decimal point. For g or G, the trailing zero will not be deleted.

'0' The conversion result should be zero attached. For d, i, o, u, x, X, e, E, f, g, and G, the left side of the conversion result will be filled with zeros instead of spaces. If the 0 and - flags are both present, the 0 flag will be ignored. For numerical conversion, if the precision field is given, the 0 flag is also ignored.

'-' The converted result will be left-adjusted (left) within the boundaries of the corresponding field. (The default is to make a right adjustment - right). The n conversion is an exception, and the conversion result will be filled with spaces on the right.

' ' A space should be reserved before a positive result resulting from a signed conversion.

'+' Indicates that a symbol (+ or -) is always required before a symbol conversion result. For the default case, only negative numbers use a negative sign.

● width specifies the output string width, which specifies the minimum width value of the field. If the result of the conversion is smaller than the specified width, then the left side (or the right side, if the right adjustment flag is given) needs to be filled with spaces or zeros (determined by the flags) and so on. In addition to using numbers to specify the width field, you can also use '*' to indicate that the width of the field is given by the next integer parameter. When the width of the conversion value is greater than the width specified by width, the small width value will not truncate the result under any circumstances. The field width is expanded to include the full result.

● precision is the number that describes the minimum number of output. For d, I, o, u, x, and X conversions, the precision value indicates the number of digits at least. For e, E, f, and F, this value indicates the number of digits that appear after the decimal point. For g or G, indicate the maximum number of significant digits. For s or S conversions, the precision value specifies the maximum number of characters in the output string.

● The length modifier indicator describes the output type form after the integer conversion. In the following description, the 'integer number conversion' represents d, i, o, u, x or X conversion.

'hh' Indicates that the subsequent integer conversion corresponds to a signed or unsigned character parameter.

'h' Indicates that the subsequent integer conversion corresponds to a signed integer or unsigned short integer parameter.

'l' Indicates that the subsequent integer conversion corresponds to a long integer or unsigned long integer argument.

'll' Indicates that the subsequent integer conversion corresponds to a long long integer or unsigned long long integer argument.

'L' Indicates that the e, E, f, F, g or G conversion result corresponds to a long double precision parameter.

● type is the format of the input parameter type and output that are accepted. The meaning of each conversion indicator is as follows:

'd,I' Integer parameters will be converted to signed integers. If there is precision, the minimum number of digits that need to be output is given. If the number of values being converted is small, it will be zeroed to the left. The default precision value is 1.

'o,u,x,X' Unsigned integers are converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x or X) representations. x means that lowercase letters (abcdef) are used to represent hexadecimal numbers, and X means uppercase letters (ABCDEF) for hexadecimal numbers. If there is a precision field, it means the minimum number of digits that need to be output. If the number of values being converted is small, it will be zeroed to the left. The default precision value is 1.

'e,E' These two conversion characters are used to round the arguments into a form of [-]d.ddde+dd. The number of digits after the decimal point is equal to the precision. If there is no precision field, take the default value of 6. If the precision is 0, no decimals appear. E means that the index is represented by a capital letter E. The index part is always represented by 2 digits. If the value is 0, then the index is 00.

'f,F' These two characters are used to round the arguments into a form of [-]ddd.ddd. The number of digits after the decimal point is equal to the precision. If there is no precision field, take the default value of 6. If the precision is 0, no decimals appear. If there is a decimal point, there will be at least one digit at the back.

'g,G' These two characters convert the argument to the format of f or e (in the case of G, the F or E format). The precision value specifies the number of integers. If there is no precision field, its default value is 6. If the precision is 0, it is treated as 1. If the index is less than -4 or greater than or equal to the precision when converting, the e format is used. The zeros after the decimal part will be deleted. The decimal point appears only if there is at least one decimal.

'c' Indicates that the argument will be converted to an unsigned character and the result of the conversion will be output.

's' Indicates that the input is required to point to a string, and the string is to end in null. If there is a precision field, only the number of characters required for precision is output, and the string does not have to end in null.

'p' Indicates that a hexadecimal number is output as a pointer.

'n' Used to save the number of characters converted so far to the position specified by the corresponding input pointer. The parameter is not converted.

'%' Indicates that a percent sign is output and no conversion is performed. That is, the entire conversion indication is now '%%'.

8.12 printk.c

8.12.1 Function Description

printk() is the print (display) function used by the kernel and has the same functionality as printf() in the C standard library. The reason for rewriting such a function is that you can't directly use the fs segment register dedicated to user mode in the kernel code, you need to save it first.

The reason why fs cannot be used directly is because in the actual screen display function tty_write(), the message to be displayed is taken from the data segment pointed to by the fs segment, that is, in the user program data segment. The message that needs to be displayed in the printk() function is in the kernel data segment, that is, in the kernel data segment pointed to by ds register when executed in the kernel code. Therefore, you need to temporarily use the fs segment register in the printk() function.

The printk() function first uses vsprintf() to format the parameters, and then calls tty_write() to print the information when the fs segment register is saved.

8.12.2 Code Comments

Program 8-11 linux/kernel/printk.c

```
/*
 *  linux/kernel/printk.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * When in kernel-mode, we cannot use printf, as fs is liable to
 * point to 'interesting' things. Make a printf with fs-saving, and
 * all is well.
 */
// <stdarg.h> Standard parameter header file. Define a list of variable parameters in the form
//     of macros. It mainly describes one type (va_list) and three macros (va_start,va_arg and
//     va_end) for the vsprintf, vprintf, and vfprintf functions.
// <stddef.h> The standard definition header file. NULL, offsetof(TYPE, MEMBER) is defined.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
#include <stdarg.h>
#include <stddef.h>

```

```
#include <linux/kernel.h>

static char buf[1024];                // Temporary buffer for display.

// vsprintf() is defined at line 92 in linux/kernel/vsprintf.c.
extern int vsprintf(char * buf, const char * fmt, va_list args);

// The display function used by the kernel.
int printk(const char *fmt, ...)
{
        va_list args;                 // is actually a character pointer type.
        int i;

// First run the parameter processing start function, then use the format string fmt to convert
// the parameter list args and output it to buf. The return value i is equal to the length of
// the output string. Then run the parameter processing end function. Finally, the console is
// called to display the function and return the number of characters displayed.
        va_start(args, fmt);
        i=vsprintf(buf,fmt,args);
        va_end(args);
        console_print(buf);           // chr_drv/console.c, line 995.
        return i;
}

```

8.13 panic.c

8.13.1 Function Description

The panic() function is used to display kernel error messages and put the system into an infinite dead loop. In many places in the kernel, this function is called if the kernel code has a serious error during execution. Calling the panic() function in many cases is a straightforward approach. This approach follows the UNIX "as concise as possible" principle.

8.13.2 Code Comments

Program 8-12 linux/kernel/panic.c

```
/*
 *  linux/kernel/panic.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * This function is used through-out the kernel (includeinh mm and fs)
 * to indicate a major problem.
 */
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
```

```
 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
 //     of the initial task 0, and some embedded assembly function macro statements about the
 //     descriptor parameter settings and acquisition.
#include <linux/kernel.h>
#include <linux/sched.h>

void sys_sync(void);    /* it's really int */   // fs/buffer.c, line 44.

// This function is used to display the major error messages that appear in the kernel, and
// run the file system synchronization function, then enter the endless loop - the system
// crashes. If the current process is task 0, it also indicates that the swapper task is in
// error and the file system synchronization function has not been run yet. The volatile keyword
// before the function name is used to tell the compiler gcc that the function will not return.
// This allows gcc to produce better code, and more importantly, use this keyword to avoid false
// warnings (uninitialized variables). This is equivalent to the current gcc attributes:
// 'void panic(const char *s) __attribute__ ((noreturn));'
volatile void panic(const char * s)
{
        printk("Kernel panic: %s\n\r",s);
        if (current == task[0])
                printk("In swapper task - not syncing\n\r");
        else
                sys_sync();
        for(;;);
}

```

8.14 Summary

This chapter mainly studies the 12 source files in the linux/kernel directory, and gives the implementation of some of the most important mechanisms in the kernel, including system calls, process scheduling, process replication, and process termination processing.

Beginning with the next chapter, we began to learn how the Linux kernel supports three block devices of hard disks, floppy disks, and memory virtual disks. First, the important data structures such as device request items and request queues used by the block device are described, and then the specific operation mode of the block device is described in detail. After that, the five source files in the block device directory are hacked one by one.

9 Block Device Driver

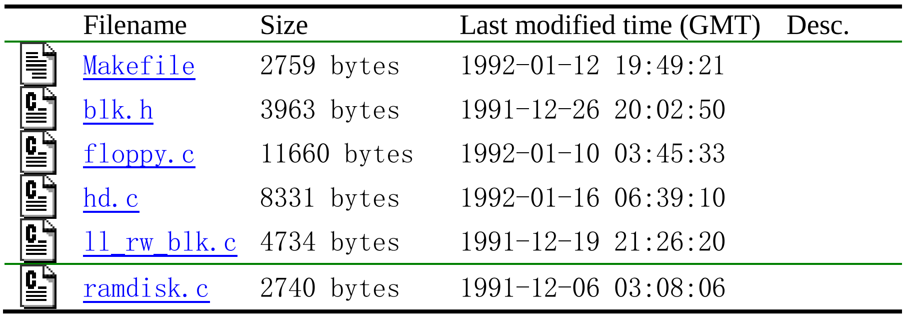
One of the main functions of the operating system is to communicate with peripheral I/O devices and to control these peripheral devices with a unified interface. All devices of the operating system can be roughly divided into two types: a block device and a character device. A block device is a device that can be addressed and accessed in units of fixed-size data blocks, such as hard disk devices and floppy disk devices. A character device is a device that operates on a character stream and cannot be addressed. For example, printer devices, network interface devices, and terminal devices. For ease of management and access, the operating system distinguishes these devices uniformly by device number. In the Linux 0.12 kernel, devices are divided into 7 classes, which have a total of 7 major device numbers (0 to 6). The devices in each type can be further differentiated based on the sub (sub, secondary) device numbers. The device types and associated devices for each device number are listed in Table 9–1. It can be seen from the table that some devices (memory devices) can be accessed as either block devices or as character devices. This chapter mainly discusses and describes the implementation principles and methods of block device drivers. A discussion of character devices is presented in the next chapter.

Table 9-1 The major device number in the Linux 0.12 kernel

Major No.	Name	Device type	Description
0	None	None	None
1	ram	Block/Char	Ram devices (virtual disk)
2	fd	Block	floppy device
3	hd	Block	harddisk device
4	ttyx	Char	device (virtual or serial terminal)
5	tty	Char	tty device
6	lp	Char	lp printer

The Linux 0.12 kernel mainly supports three types of block devices: hard disk, floppy disk and memory virtual disk. Since block devices are primarily related to file systems and caches, you can quickly take a look at the contents of the file system chapter before proceeding with this chapter. The source code files covered in this chapter are shown in List 9-1.

List 9-1 linux/kernel/blk_drv



	Filename	Size	Last modified time (GMT)	Desc.
	Makefile	2759 bytes	1992-01-12 19:49:21	
	blk.h	3963 bytes	1991-12-26 20:02:50	
	floppy.c	11660 bytes	1992-01-10 03:45:33	
	hd.c	8331 bytes	1992-01-16 06:39:10	
	ll_rw_blk.c	4734 bytes	1991-12-19 21:26:20	
	ramdisk.c	2740 bytes	1991-12-06 03:08:06	

The purpose of the program in this chapter can be divided into two categories, one is the driver corresponding to each device, such programs are: hard disk driver hd.c, floppy disk driver floppy.c, and Memory virtual disk driver ramdisk.c.

The other class includes only one program, which is used by the other program in the kernel to access the block device interface program ll_rw_blk.c. Another file is the block device-specific header file blk.h, which provides a uniform setup and the same device request start procedure for these three block devices to interact with the ll_rw_blk.c program.

9.1 Main Functions

Reading and writing data on hard disk and floppy block devices is performed by an interrupt handler. The amount of data read and written by the kernel each time is in units of one logical block (1024 bytes), while the block device controller accesses the block device in units of sectors (512 bytes). During processing, the kernel uses a read and write request entry wait queue to sequentially buffer the operation of reading and writing multiple logical blocks.

When a program needs to read a logical block on the hard disk, it will apply to the buffer management program, and the program process enters a sleep wait state. The buffer manager first looks in the buffer for whether it has been read before. If there is already in the buffer, the corresponding buffer block header pointer is directly returned to the program and the waiting process is woken up. If the required data block does not exist in the buffer, the buffer manager calls the low-level block read/write function ll_rw_block() in this chapter to issue a read data block operation request to the corresponding block device driver. The function creates a request structure item for this and inserts it into the request queue. In order to improve the efficiency of reading and writing the disk and reduce the distance the head moves, the kernel code uses the elevator algorithm to insert the request item into the request queue position where the head movement distance is the smallest.

If the request queue of the block device is empty at this time, it indicates that the block device is not busy at the moment. The kernel then immediately issues a read data command to the controller of the block device. When the controller reads the data into the specified buffer block, it will issue an interrupt request signal and call the corresponding read command post-processing function to process continuing reading of the sectors or end the request. For example, closing operation of the corresponding block device, setting flags about the buffer block data has been updated, and finally woken up the process waiting for the block data.

9.1.1 Block Device Requests and Request Queues

According to the above description, we know that the low-level read/write function ll_rw_block() is to establish a connection with various block devices through a request item and issue a read/write request operation. For various block devices, the kernel uses a block device table (array) blk_dev[] for management. Each block device occupies one item in the block device table. The structure of each block device item in the block device table is (see the file blk.h below):

```
struct blk_dev_struct {
        void (*request_fn)(void);                 // A function pointer of requests.
        struct request * current_request;         // current request structure pointer.
};
extern struct blk_dev_struct blk_dev[NR_BLK_DEV]; // Block device table (NR_BLK_DEV = 7).
```

The first field is a function pointer that is used to manipulate the request item of the corresponding block device. For example, for a hard disk drive, it is do_hd_request(), and for a floppy device it is do_floppy_request().The second field is the current request item structure pointer, which is used to indicate the request item currently being processed by the block device. All the request items in the table are set to NULL at the time of initialization.

The block device table will be set during the initialization function of each device in the init/main.c program. For ease of extension, Mr. Linus built the block device table into an array indexed by the major device number. In Linux 0.12, there are 7 major device numbers, as shown in Table 9–2. Among them, the main device numbers 1, 2, and 3 correspond to the block devices: a virtual disk, a floppy disk, and a hard disk. All other items in the block device array are set to NULL by default.

Table 9-2 The major device no and related operation functions

Main Dev No	Type	Description	Request function
0	None	None	NULL
1	Block/Char	ram, memory dev (ramdisk etc)	do_rd_request()
2	Block	fd, floppy disk device	do_fd_request()
3	Block	hd, hadrdware disk device	do_hd_request()
4	Char	ttyx device (virtual or serial terminal etc)	NULL
5	Char	tty device	NULL
6	Char	lp printing device	NULL

When the kernel issues a block device read or write or other operation request, the ll_rw_block() function will create a device request item using the operation function do_XX_request() according to the command specified in its parameter and the device no in the data buffer block header, and inserts it into the request queue using elevator algorithm. The 'XX' in the function name can be one of 'rd', 'fd' or 'hd', representing memory, floppy disk and hard disk block devices. The request item queue consists of items in the request item table (array). There are a total of 32 items. The data structure of each request item is as follows:

```
struct request {
        int dev;                                // The device no used (-1 means empty).
        int cmd;                                // Command (READ or WRITE).
        int errors;                             // The nr of errors during operation.
        unsigned long sector;                   // Starting sector. (1 block = 2 sectors)
        unsigned long nr_sectors;               // Read/write sector number.
        char * buffer;                          // Data buffer.
        struct task_struct * waiting;           // The place where the task waits for operation.
        struct buffer_head * bh;                // Buffer header (include/linux/fs.h, 68).
        struct request * next;                  // next request item.
};
extern struct request request[NR_REQUEST];      // Request item array (NR_REQUEST = 32).
```

The current request item pointer for each block device, together with the request link list for the device in the request array, constitutes the request queue for the device. Between the items, the next pointer field is used to form a linked list. Therefore, the block device item and the associated request queue form the structure shown in Figure 9-1. The main reason why the request item adopts the array and linked list structure is to satisfy two purposes: First, the array structure of the request item can be used to perform loop operations when searching for idle request blocks. The search access time complexity is constant, so the program can be written very concisely; secondly, in order to satisfy the elevator algorithm insertion request item operation, it is also necessary to adopt Linked list structure. As shown in Figure9-1, the hard disk device currently has four request items, and the floppy disk device has only one request item, and the virtual disk device does not currently have a read/write request item.

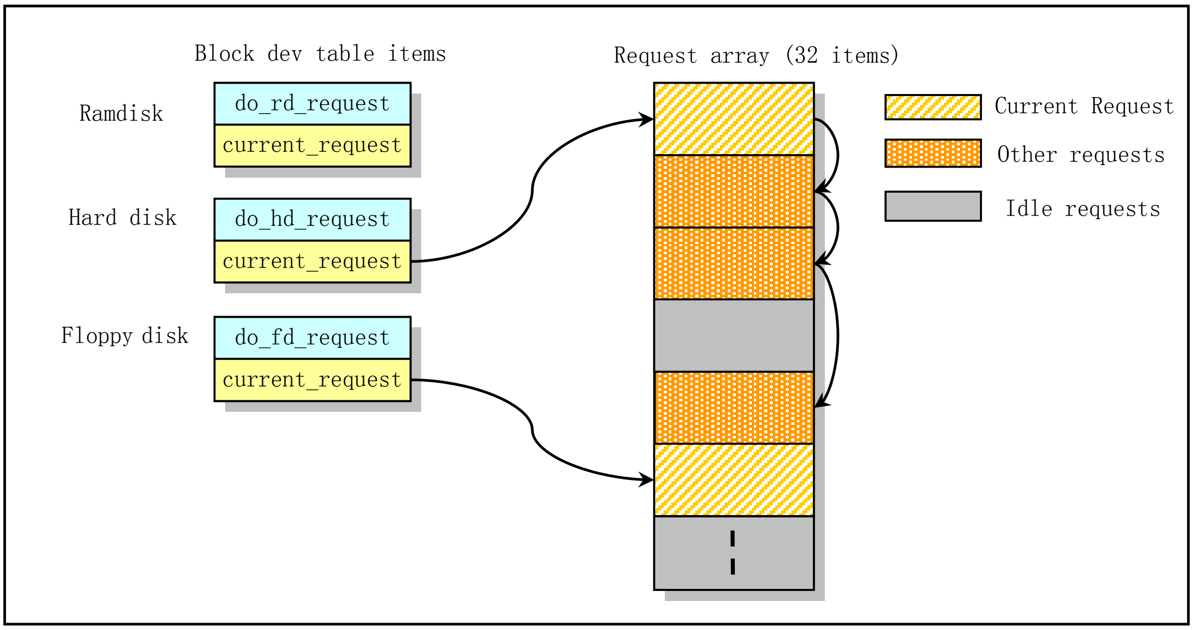


Figure 9-1 Device table entry and request item

For a currently idle block device, when the ll_rw_block() function establishes the first request item for it, the current request item pointer current_request directly points to the newly created request item, and immediately invokes the request function to start the block device read and write operations. When a block device already has a linked list of several request items, ll_rw_block() will use the elevator algorithm to insert the newly created request into the appropriate position of the linked list according to the principle of minimum head movement distance.

In addition, in order to satisfy the priority of the read operation, when searching for the array of request items for establishing a new request item, the search range of the free item for write operation is limited to the first 2/3 of the entire request array, and the remain 1/3 request items is specifically used for the read operation establishment request.

9.1.2 Block device access scheduling

Accessing data in blocks such as hard disks and floppy disks is a time consuming operation and affects system performance compared to accessing memory operations. Since the hard disk head seek operation (that is, moving the read/write head from one track to another designated track) takes a long time, it is necessary to sort the order of accessing the disk sectors before sending the operation command to the hard disk controller. That is, the order of each request item in the request linked list is sorted, so that all the disk sector blocks accessed by the request items are operated in order. In the Linux 0.12 kernel, the request items are sorted using the elevator algorithm. The principle of operation is similar to the movement of the elevator -- moving in one direction until the last "request" stop layer in that direction. Then perform the opposite direction of movement. For the disk, the head moves all the way to the center of the disc, or vice versa. See Figure 2-11 for the structure of the hard disk.

Therefore, the request items are not directly sent to the block device for processing in the received order, but the order of the request items needs to be optimized first. We usually refer to the relevant handler as the I/O scheduler. The I/O scheduler in Linux 0.12 only sorts the request items, and the current popular Linux kernel (such as 2.6.x) I/O scheduler also contains two access to adjacent disk sectors or Multiple request items are merged.

9.1.3 Block device operation method

When performing IO access operations between the system (kernel) and the hard disk, you need to consider the interaction between three objects. They are system, controller, and drive (such as hard drive or floppy drive), as shown in Figure 9-2. The system can send commands directly to the controller or wait for the controller to issue an interrupt request; after receiving the command, the controller will control the drive to perform related operations, read/write data or perform other operations. Therefore, we can regard the interrupt signal sent by the controller here as the synchronous operation signal between the three, and the operation steps are as follows:

- First, the system indicates the C function that the controller should invoke during the interrupt caused by the execution of the command, and then sends a read, write, reset or other operation command to the block device controller;
- When the controller completes the specified command, it will issue an interrupt request signal, which will cause the system to execute the interrupt processing of the block device, and call the specified C function to post-process the read/write or other commands after these commands are finished.

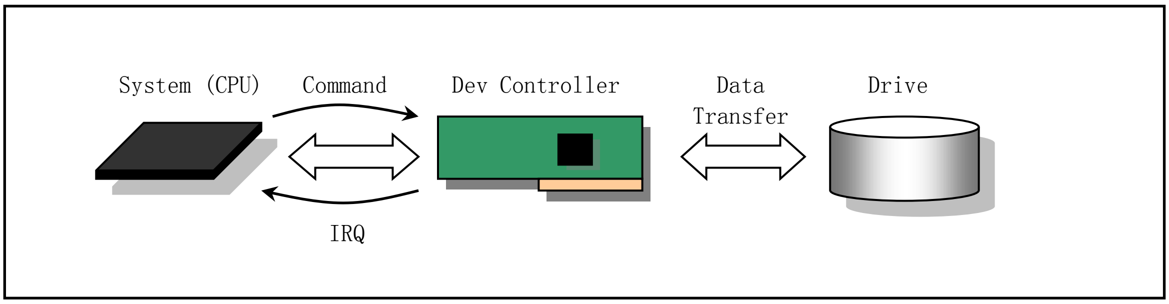


Figure 9-2 System, block device controllers, and drives

For disk write operations, the system needs to wait for the controller to give a response to allow data to be written to the controller after issuing a write command (using hd_out()), ie, waiting for the data request service flag DRQ of the controller status register to be set. Once DRQ is set, the system can send a sector of data to the controller buffer.

When all data is written to the drive (or an error occurs), the controller also generates an interrupt request signal to execute the pre-set C function (write_intr()) during interrupt processing. This function will check if there is still data to write. If so, the system will transfer the data of one sector to the controller buffer, and then wait for the interrupt generated by the controller to write the data to the drive again, and repeat this. If all the data has been written to the drive at this time, the C function performs the finishing work after the end of this write: Wake up the relevant process waiting for the request item data, wake up the process waiting for the request item, release the current request item and delete the request item from the linked list and release the locked buffer. Finally, the request item operation function is called to execute the next read/write disk request item (if any).

For the disk read operation, the system waits for the controller to generate an interrupt signal after transmitting a command including information such as the start position of the sector, the number of sectors, and the like to the controller. When the controller passes the specified sector data from the drive to its own buffer as required by the read command, an interrupt request is issued. This will execute the C function (read_intr()) that was previously set for the read operation. This function first puts the data of one sector in the controller buffer into the buffer of the system, and then decrements the number of sectors to be read. If there is still data to read (the decrement result value is not 0), continue to wait for the controller to send the next interrupt signal. If all the required sectors have been read into the system buffer at this time, the same end operation as the above write operation is performed.

For a virtual disk device, since its read and write operations do not involve a synchronous operation with an external device, there is no interrupt processing described above. The read and write operations of the current request item to the virtual device are completely implemented in do_rd_request().

One thing to be reminded is that after sending a read/write or other command to the hard disk or floppy disk controller, the function that sends the command does not wait for the execution of the issued command, but immediately returns to the program that called it. And finally return to the other program that invokes the block device function ll_rw_block () to wait for the completion of the block device IO. For example, the read block function bread() in the cache manager (fs/buffer.c line 267), after calling ll_rw_block(), it invokes the wait function wait_on_buffer() to let the process go to sleep immediately. The state is awakened in the end_request() function until the end of the associated block device IO.

9.2 blk.h

9.2.1 Function Description

This is the header file for block device parameters such as hard disks. Because it is only used for block devices, it is placed in the same place as the block device source file. The data structure request of the request item in the request queue is mainly defined; the elevator search algorithm is defined by the macro statement. For the virtual disk, floppy disk and hard disk, the corresponding constant values are defined according to their respective major device numbers.

9.2.2 Code annotation

Program 9-1 linux/kernel/blk_drv/blk.h

```
#ifndef _BLK_H
#define _BLK_H

#define NR_BLK_DEV      7               // nr of device types
/*
 * NR_REQUEST is the number of entries in the request-queue.
 * NOTE that writes may use only the low 2/3 of these: reads
 * take precedence.
 *
 * 32 seems to be a reasonable number: enough to get some benefit
 * from the elevator-mechanism, but not so much as to lock a lot of
 * buffers when they are in the queue. 64 seems to be too many (easily
 * long pauses in reading when heavy writing/syncing is going on)
 */
#define NR_REQUEST      32

/*
 * Ok, this is an expanded form so that we can use the same
 * request for paging requests when that is implemented. In
 * paging, 'bh' is NULL, and 'waiting' is used to wait for
 * read/write completion.
 */
// Below is the structure of the request item in the request queue. If the field dev = -1, it
// means that the item in the queue is not used. The field cmd can be a constant of READ(0)
// or WRITE(1) (defined in include/linux/fs.h). In addition, the kernel does not use the waiting
// pointer. Instead, the kernel uses the wait queue of the buffer block. Because waiting for
// a buffer block is equivalent to waiting for the completion of the request.
struct request {
        int dev;                        /* -1 if no request */  // device requested.
        int cmd;                        /* READ or WRITE */
        int errors;                      // count of error.
        unsigned long sector;            // start sector no.
        unsigned long nr_sectors;        // nr sectors needed.
        char * buffer;                   // data buffer.
        struct task_struct * waiting;    // waiting queue of tasks.
        struct buffer_head * bh;         // Buffer header (include/linux/fs.h, 73).
        struct request * next;           // points to the next request.
};

/*
 * This is used in the elevator algorithm: Note that
 * reads always go before writes. This is natural: reads
 * are much more time-critical than writes.
 */
// The values of the parameters s1 and s2 in the macro below are pointers to the request structure.
// The macro is used to determine the order of the two request item s1 and s2 in the request
// queue based on the information in them (command cmd (READ or WRITE), device number dev, and
// sector number sector being operated).  This order will be used as the order in which the
// request items are executed when accessing the block device. This macro will be used in the
// function add_request() (blk_drv/ll_rw_blk.c, line 96). This macro partially implements the
// I/O scheduling function, which implements the sorting function of the request item (the other
// is request item merging).
#define IN_ORDER(s1,s2) \
((s1)->cmd<(s2)->cmd || (s1)->cmd==(s2)->cmd && \
((s1)->dev < (s2)->dev || ((s1)->dev == (s2)->dev && \
(s1)->sector < (s2)->sector)))

// Block device structure.
struct blk_dev_struct {
        void (*request_fn)(void);                       // request handling function.
        struct request * current_request;               // current request item.
};

```

```
   // Block device table (array). Each block device occupies one item, a total of 7 items. The
   // index of the array is the major device number. The next is an array of request item structures,
   // a total of 32 items. wait_for_request is a process queue header waiting for an idle request.
50 extern struct blk_dev_struct blk_dev[NR_BLK_DEV];
51 extern struct request request[NR_REQUEST];
52 extern struct task_struct * wait_for_request;
53
   // An array of pointers to the total number of device data blocks. Each pointer entry points
   // to a total number of blocks array hd_sizes[] (blk_drv/hd.c, line 62) of the specified major
   // device. Each item of the array corresponds to the total number of data blocks owned by one
   // minor device (1 block size = 1 KB).
54 extern int * blk_size[NR_BLK_DEV];
55
   // In a block device driver (such as hd.c) that contains this header file, you must first define
   // the major device number of the device being processed. Thus, the correct macro definition
   // can be given to the driver in the following 63-90 lines.
56 #ifdef MAJOR_NR                               // current major number used.
57
58 /*
59  * Add entries as needed. Currently the only block devices
60  * supported are hard-disks and floppies.
61  */
62
   // If device is RAM disk, the following constants and macros are used.
63 #if (MAJOR_NR == 1)
64 /* ram disk */
65 #define DEVICE_NAME "ramdisk"
66 #define DEVICE_REQUEST do_rd_request          // request handler.
67 #define DEVICE_NR(device) ((device) & 7)      // Sub-device nr (0 - 7).
68 #define DEVICE_ON(device)                     // Turn on (ram disk doesn't need this).
69 #define DEVICE_OFF(device)
70
   // If device is floppy driver, the following constants and macros are used.
71 #elif (MAJOR_NR == 2)
72 /* floppy */
73 #define DEVICE_NAME "floppy"
74 #define DEVICE_INTR do_floppy                 // device interrupt handler.
75 #define DEVICE_REQUEST do_fd_request          // request handler.
76 #define DEVICE_NR(device) ((device) & 3)      // Sub-device nr (0 - 3).
77 #define DEVICE_ON(device) floppy_on(DEVICE_NR(device))
78 #define DEVICE_OFF(device) floppy_off(DEVICE_NR(device))
79
   // If device is hard disk, the following constants and macros are used.
80 #elif (MAJOR_NR == 3)
81 /* harddisk */
82 #define DEVICE_NAME "harddisk"
83 #define DEVICE_INTR do_hd                     // device interrupt handler.
84 #define DEVICE_TIMEOUT hd_timeout             // device timeout.
85 #define DEVICE_REQUEST do_hd_request          // request handler.
86 #define DEVICE_NR(device) (MINOR(device)/5)   // harddisk no (0,1).
87 #define DEVICE_ON(device)                     // the harddisk always running.
88 #define DEVICE_OFF(device)
89
```

```
#elif
/* unknown blk device */
#error "unknown blk device"

#endif

// For ease of programming, two macros are defined here: CURRENT is the current request item
// pointer, and CURRENT_DEV is the device number in the current request item CURRENT.
#define CURRENT (blk_dev[MAJOR_NR].current_request)
#define CURRENT_DEV DEVICE_NR(CURRENT->dev)

// If a device interrupt handling symbol is defined, it is declared as a function pointer and
// defaults to NULL.
#ifdef DEVICE_INTR
void (*DEVICE_INTR)(void) = NULL;
#endif
// If a device timeout symbol is defined, the same variable is defined with a value equal to
// 0, and the SET_INTR() macro is defined.
#ifdef DEVICE_TIMEOUT
int DEVICE_TIMEOUT = 0;
#define SET_INTR(x) (DEVICE_INTR = (x),DEVICE_TIMEOUT = 200)
#else
#define SET_INTR(x) (DEVICE_INTR = (x))
#endif
// Declare symbol DEVICE_REGUEST is a static function pointer with no arguments & no return.
static void (DEVICE_REQUEST)(void);

// Unlock the specified buffer block. If the specified buffer block bh is not locked, a warning
// message is displayed. Otherwise the buffer block is unlocked and the process waiting for
// the buffer block is woken up. This is an inline function, but is used as a "macro". The argument
// is the buffer block header pointer.
extern inline void unlock_buffer(struct buffer_head * bh)
{
        if (!bh->b_lock)
                printk(DEVICE_NAME ": free buffer being unlocked\n");
        bh->b_lock=0;
        wake_up(&bh->b_wait);
}

// End request processing "macro". The parameter uptodate is the update flag.
// First turn off the specified block device and then check if the read/write buffers are valid.
// If valid, the buffer data update flag is set according to the parameter value and the buffer
// is unlocked. If the parameter uptodate value is 0, it indicates that the operation of the
// request item has failed, so the related block device IO error message is displayed. Finally,
// the processes waiting for the request item and the processes waiting for the idle request
// item are awake, and the request item is released and deleted from the request list, and the
// current request item pointer is pointed to the next request.
extern inline void end_request(int uptodate)
{
        DEVICE_OFF(CURRENT->dev);                   // turn off the device.
        if (CURRENT->bh) {                          // the current request item pointer.
                CURRENT->bh->b_uptodate = uptodate; // set flag.
                unlock_buffer(CURRENT->bh);
        }
        if (!uptodate) {                                // updodate flag not set...
                printk(DEVICE_NAME " I/O error\n\r");
                printk("dev %04x, block %d\n\r",CURRENT->dev,
                        CURRENT->bh->b_blocknr);
        }
        wake_up(&CURRENT->waiting);             // Wake up process waiting for the request.
        wake_up(&wait_for_request);             // Wake up process waiting for idle request.
        CURRENT->dev = -1;                      // Release the request item.
        CURRENT = CURRENT->next;                // Point to the next request item.
}

// If the device timeout symbol DEVICE_TIMEOUT is defined, the CLEAR_DEVICE_TIMEOUT symbol is
// defined as "DEVICE_TIMEOUT = 0". Otherwise only define CLEAR_DEVICE_TIMEOUT.
#ifdef DEVICE_TIMEOUT
#define CLEAR_DEVICE_TIMEOUT DEVICE_TIMEOUT = 0;
#else
#define CLEAR_DEVICE_TIMEOUT
#endif

// If the device interrupt symbol DEVICE_INTR is defined, the CLEAR_DEVICE_INTR symbol is defined
// as "DEVICE_INTR = 0", otherwise it is defined as empty.
#ifdef DEVICE_INTR
#define CLEAR_DEVICE_INTR DEVICE_INTR = 0;
#else
#define CLEAR_DEVICE_INTR
#endif

// A macro that defines the initialization of a request item. Since the initialization of the
// request items at the beginning of the block device driver is similar, a uniform initialization
// macro is defined for them. This macro is used to make some validity judgments on the current
// request item. The work done is as follows:
// If the current request item of the device is empty (NULL), it means that the device has no
// request items yet to be processed. Then the work is skipped and the corresponding function
// is exited. Otherwise, if the major device number in the current request item is not equal
// to the major number defined by the driver, the request item queue is garbled, and the kernel
// displays an error message and stops. Otherwise, if the buffer block used in the request item
// is not locked, it also indicates that there is a problem with the kernel code, so an error
// message is displayed and the machine is stopped too.
#define INIT_REQUEST \
repeat: \
        if (!CURRENT) {\                                // no more requests need to processed.
                CLEAR_DEVICE_INTR \
                CLEAR_DEVICE_TIMEOUT \
                return; \
        } \
        if (MAJOR(CURRENT->dev) != MAJOR_NR) \          // major nr error.
                panic(DEVICE_NAME ": request list destroyed"); \
        if (CURRENT->bh) { \
                if (!CURRENT->bh->b_lock) \             // buffer error.
                        panic(DEVICE_NAME ": block not locked"); \
        }

```

```
#endif

#endif

```

9.3 hd.c

9.3.1 Function description

The hd.c program is the hard disk controller driver. It provides read and write operations to the hard disk controller and block devices, as well as hard disk initialization processing. All functions in the program can be divided into five categories according to their functions:

- Functions that initialize the hard disk and set the data structure information used by the hard disk, such as sys_setup() and hd_init();
- The function hd_out() that sends the command to the hard disk controller;
- The function do_hd_request() that handles the current request item of the hard disk;
- C functions called during hard disk interrupt handling, such as read_intr(), write_intr(), bad_rw_intr(), and recal_intr(). The do_hd_request() function will also be called in read_intr() and write_intr();
- The hard disk controller operates auxiliary functions such as controller_ready(), drive_busy(), win_result(), hd_out(), and reset_controler().

The sys_setup() function uses the information provided by the boot/setup.s to set the parameters of the hard drive included in the system. Then read the hard disk partition table and try to copy the root fs image on the boot disk to the memory virtual disk. If successful, the root file system in the virtual disk is loaded, otherwise the normal root file system loading operation is continued.

The hd_init() function is used to set the hard disk controller interrupt descriptor during kernel initialization and reset the hard disk controller interrupt mask to allow the hard disk controller to send an interrupt request signal.

Hd_out() is the hard disk controller operation command send function. This function takes a C function pointer parameter that is called during an interrupt. Before sending a command to the controller, it first uses this parameter to preset the function pointer (do_hd, such as read_intr()) that is called during the interrupt. It then sends the command parameter block to the hard disk controller (ports 0x1f0 to 0x1f7) in the prescribed manner. The function returns immediately and does not wait for the hard disk controller to execute the read/write commands. In addition to the controller diagnostics (WIN_DIAGNOSE) and the establishment of the drive parameters (WIN_SPECIFY), the hard disk controller will send an interrupt request signal to the CPU after receiving any other commands and executing the commands. This causes the system to perform the hard disk interrupt processing (in system_call.s, line 235).

do_hd_request() is the operation function of the hard disk request item. The operation process is as follows:

(1) First determine if the current request item exists. If the current request pointer is empty, it indicates that the current hard disk block device has no pending request items, so the program is immediately exited. This is the statement executed in the macro INIT_REQUEST. Otherwise continue processing the current request item.

(2) verifying the reasonableness of the device number specified in the current request item and the

requested disk start sector number;

(3) calculating the disk track number, head number and cylinder number of the request data according to the information provided by the current request item;

(4) If the reset flag has been set, the hard disk recalibration flag is also set, and the hard disk is reset, and the "create drive parameter" command (WIN_SPECIFY) is resent to the controller. This command does not cause a hard disk interrupt;

(5) If the recalibration flag is set, send the hard disk recalibration command (WIN_RESTORE) to the controller, and pre-set the C function (recal_intr()) that needs to be executed in the interrupt caused by the command, and drop out. The main function of the recal_intr() is to re-execute this function when the controller terminates the command and raises an interrupt.

(6) If the current request item specifies a write operation, first set the C function to write_intr() to send a command parameter block of the write operation to the controller. The controller's status register is queried cyclically to determine if the request service flag (DRQ) is set. If the flag is set, it indicates that the controller has "agreed" to receive the data, and then the data in the buffer pointed to by the request item is written into the data buffer of the controller. If the flag is still not set after the loop query times out, the operation failed. The bad_rw_intr() function is then called, and it is determined whether to abandon the current request item or to set a reset flag according to the number of occurrences of the error to continue reprocessing the current request item.

(7) If the current request item is a read operation, set the C function to read_intr(), and send a read operation command to the controller.

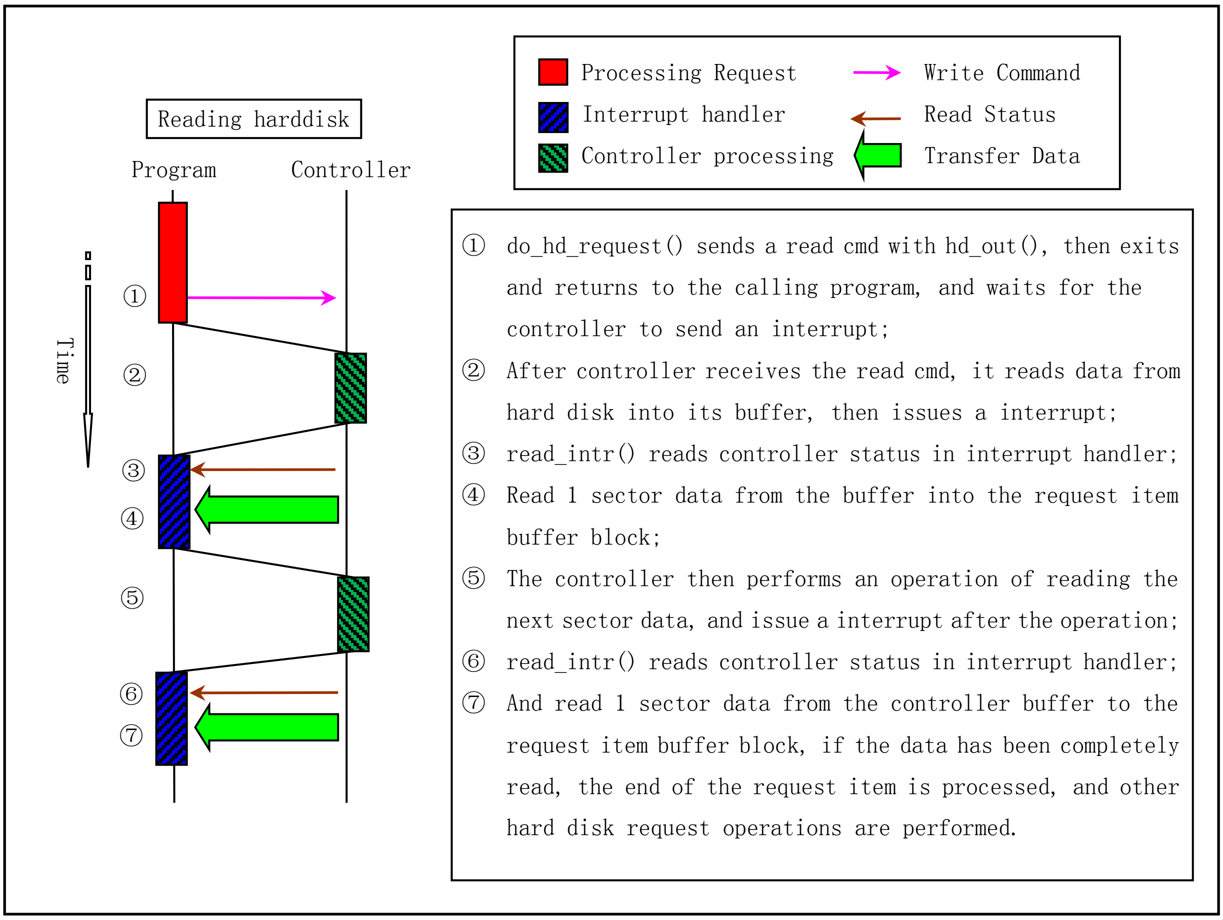
write_intr() is a C function that is set to be called during the interrupt when the current request item is a write operation. When the controller completes the write command, it will immediately send an interrupt request signal to the CPU, so the function will be called immediately after the controller write operation is completed.

The function first calls the win_result(), which reads the controller's status register to determine if an error has occurred. If an error occurs during the write operation, bad_rw_intr() is called, and it is determined whether to abort to continue processing the current request according to the number of errors occurring in processing the current request, or whether a reset flag needs to be set to continue to reprocess the current request item. If no error occurs, it is determined whether all the data required has been written to the disk according to the total number of sectors to be written indicated in the current request item. If there is still data to write to the disk, use the port_write() function to copy the data of one sector to the controller buffer. If the data has all been written, the end_request() is called to handle the end of the current request: Wake up the process waiting for the completion of the request, wake up the process waiting for the idle request item (if any), set the buffer data updated flag by the current request item, and release the current request (remove the item from the block device list) . Finally, continue to call the do_hd_request() function to continue processing other request items on the hard disk device.

read_intr() is a C function that is set to be called during an interrupt when the current request item is a read operation. After the controller reads the specified sector data from the hard disk drive into its own buffer, it immediately sends an interrupt request signal. The main purpose of this function is to copy the data in the controller to the buffer specified by the current request.

The same as when write_intr() starts, the function first calls the win_result() to read the controller's status register to determine if an error has occurred. If an error occurs while reading the disk, the same processing as write_intr() is performed. If no errors have occurred, use the port_read() function to copy the data of one sector from the controller buffer to the buffer specified by the request. Then, according to the total number of sectors to be read indicated in the current request item, it is checked whether all the data has been read. If there is still data to read, exit to wait for the next interrupt to arrive. If the data has been obtained, the end_request() function is called to process the end of the current request item: wake up the processes waiting for the current request item to complete, wake up the processes waiting for the idle request item (if any), set the buffer data updated flag, release the current request item (remove the item from the block device list). Finally, continue to call the do_hd_request() to continue processing other request items on the hard disk device.

In order to more clearly see the processing of the hard disk read and write operations, we can show the execution timing relationship between these functions, the interrupt handling and the hard disk controller as shown in Figure 9-3 and Figure 9-4.



① do_hd_request() sends a read cmd with hd_out(), then exits and returns to the calling program, and waits for the controller to send an interrupt;

② After controller receives the read cmd, it reads data from hard disk into its buffer, then issues a interrupt;

③ read_intr() reads controller status in interrupt handler;

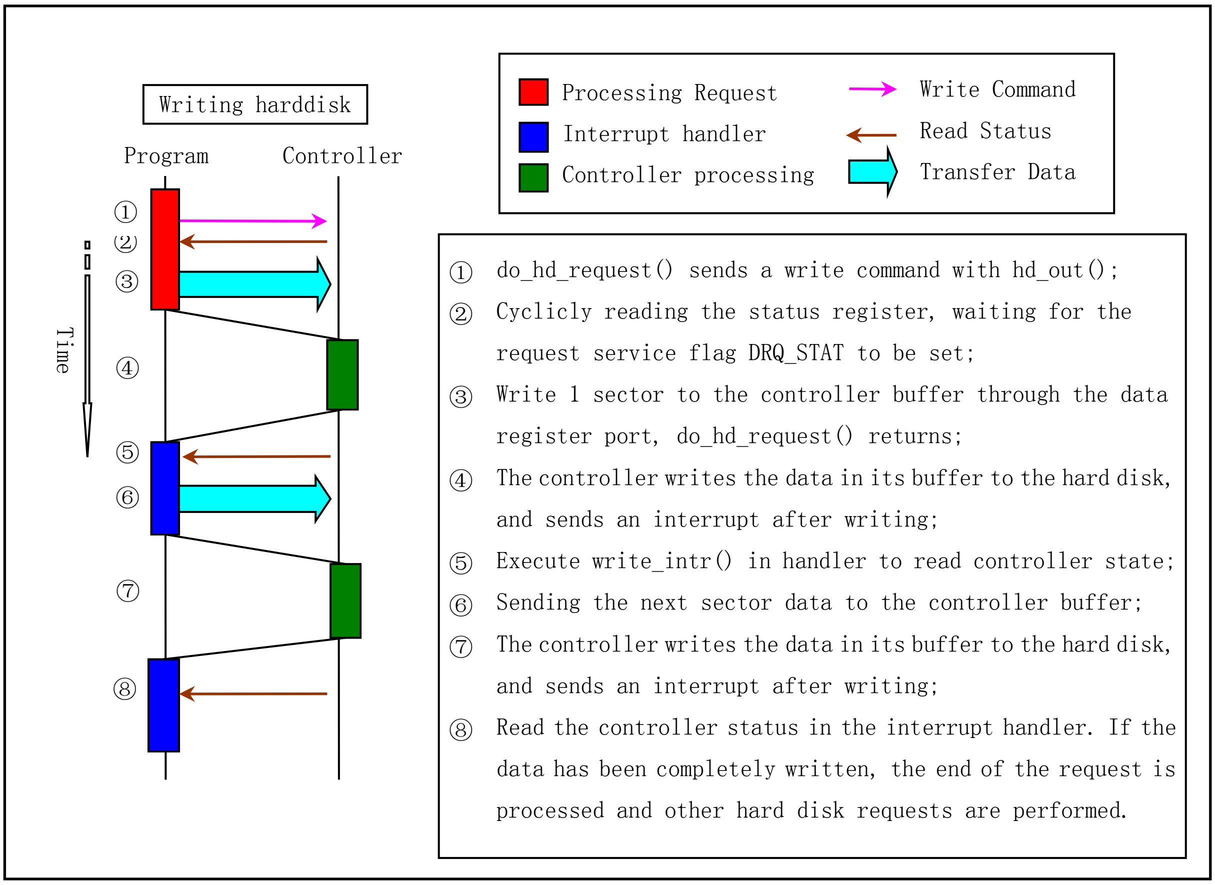
④ Read 1 sector data from the buffer into the request item buffer block;

⑤ The controller then performs an operation of reading the next sector data, and issue a interrupt after the operation;

⑥ read_intr() reads controller status in interrupt handler;

⑦ And read 1 sector data from the controller buffer to the request item buffer block, if the data has been completely read, the end of the request item is processed, and other hard disk request operations are performed.

Figure 9-3 Timing relationship of hard disk read data operations



① do_hd_request() sends a write command with hd_out();

② Cyclicly reading the status register, waiting for the request service flag DRQ_STAT to be set;

③ Write 1 sector to the controller buffer through the data register port, do_hd_request() returns;

④ The controller writes the data in its buffer to the hard disk, and sends an interrupt after writing;

⑤ Execute write_intr() in handler to read controller state;

⑥ Sending the next sector data to the controller buffer;

⑦ The controller writes the data in its buffer to the hard disk, and sends an interrupt after writing;

⑧ Read the controller status in the interrupt handler. If the data has been completely written, the end of the request is processed and other hard disk requests are performed.

Figure 9-4 Timing relationship of hard disk write data operations

As can be seen from the above analysis, the four most important functions in this program are hd_out(), do_hd_request(), read_intr(), and write_intr(). Understand the role of these four functions and understand the operation process of the hard disk drive.

It is worth noting again that after using hd_out() to send a read/write or other command to the hard disk controller, the hd_out() function does not wait for the execution of the issued command, but immediately returns to the program that called it, for example do_hd_request().The do_hd_request() function will immediately return to the function that called it (add_request()), and finally return to the other program that calls the block device read/write function ll_rw_block() (for example, the bread() function in fs/buffer.c), waiting for the completion of the block device IO.

9.3.2 Code annotation

Program 9-2 linux/kernel/blk_drv/hd.c

```
/*
 *  linux/kernel/hd.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * This is the low-level hd interrupt support. It traverses the
 * request-list, using interrupts to jump between functions. As
 * all the functions are called within interrupts, we may not
```

```
11  * sleep. Special care is recommended.
12  *
13  *  modified by Drew Eckhardt to check nr of hd's from the CMOS.
14  */
15
  // <linux/config.h> Kernel configuration header file. Define keyboard language and hard disk
  //      type (HD_TYPE) options.
  // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
  //      of the initial task 0, and some embedded assembly function macro statements about the
  //      descriptor parameter settings and acquisition.
  // <linux/fs.h> File system header file. Define the file table structure (file, buffer_head,
  //      m_inode, etc.).
  // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
  //      used functions of the kernel.
  // <linux/hdreg.h> Hard disk parameter header file. Define access to the hard disk register port,
  //      status code, partition table and other information.
  // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
  //      descriptors/interrupt gates, etc. is defined.
  // <asm/io.h> Io header file. Defines the function that operates on the io port in the form of
  //      a macro's embedded assembler.
  // <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
  //      segment register operations.
16 #include <linux/config.h>
17 #include <linux/sched.h>
18 #include <linux/fs.h>
19 #include <linux/kernel.h>
20 #include <linux/hdreg.h>
21 #include <asm/system.h>
22 #include <asm/io.h>
23 #include <asm/segment.h>
24
   // Define the hard disk major number symbol constant. In the driver, the major device number
   // must be defined before the blk.h file is included. This symbol is used in the blk.h file
   // to determine some other symbol constants and macros associated with it.
25 #define MAJOR_NR 3          // harddisk major no is 3.
26 #include "blk.h"
27
   // Read the CMOS parameter macro function. This macro reads the hard drive information in CMOS.
   // outb_p, inb_p are port input and output macros defined in include/asm/io.h. It is exactly
   // the same macro used to read CMOS clock information in init/main.c.
28 #define CMOS_READ(addr) ({ \
29 outb_p(0x80|addr,0x70); \           // 0x70 is write port, 0x80|addr is CMOS addr.
30 inb_p(0x71); \                      // 0x71 is read port.
31 })
32
33 /* Max read/write errors/sector */
34 #define MAX_ERRORS      7
35 #define MAX_HD          2           // nr of hard disks supported by the system.
36
   // Recalibration. The recalibration function (line 311) called in the hard disk interrupt
   // handler during reset operation.
37 static void recal_intr(void);
   // A function that handles hard disk read and write failures.
   // End the processing of this request item, or set the reset flag to perform the hard disk
   // controller reset operation and then try again (242 lines).
38 static void bad_rw_intr(void);
39
   // Recalibration flag. When this flag is set, recal_intr() is called in the program to move
   // the head to the 0 cylinder.
40 static int recalibrate = 0;
   // Reset flag. This flag is set when a read or write error occurs and the associated reset function
   // is called to reset the hard disk and controller.
41 static int reset = 0;
42
43 /*
44  *  This struct defines the HD's and their types.
45  */
   // Harddisk information struct.
   // The fields are: the nr of heads, the nr of sectors per track, the nr of cylinders, the
   // pre-compensation cylinder nr before writing, the cylinder landing nr of the head, and the
   // control byte. See the instructions following the program list for their meaning.
46 struct hd_i_struct {
47         int head,sect,cyl,wpcom,lzone,ctl;
48         };

   // If the symbol constant HD_TYPE has been defined in the include/linux/config.h file, the
   // parameter defined therein is taken as the data in the hard disk information array hd_info[].
   // Otherwise, the default value is set to 0 first, and will be reset in the setup() function.
49 #ifdef HD_TYPE
50 struct hd_i_struct hd_info[] = { HD_TYPE };
51 #define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct)))      // count hd nr.
52 #else
53 struct hd_i_struct hd_info[] = { {0,0,0,0,0,0},{0,0,0,0,0,0} };
54 static int NR_HD = 0;
55 #endif
56
   // Define the hard disk partition structure. Give the physical start sector number and the total
   // number of partition sectors for each partition from the 0 track of the hard disk. Items in
   // multiples of 5 (such as hd[0] and hd[5], etc.) represent parameters of the entire hard disk.
57 static struct hd_struct {
58         long start_sect;                   // The partition start sector in the hard disk.
59         long nr_sects;                     // The total nr of sectors in the partition.
60 } hd[5*MAX_HD]={{0,0},};
61
   // An array of the total number of data blocks in each partition of the hard disk.
62 static int hd_sizes[5*MAX_HD] = {0, };
63
   // Read port inline assembly macros. Read the port, read nr word, and save it in buf.
64 #define port_read(port,buf,nr) \
65 __asm__("cld;rep;insw"::"d" (port),"D" (buf),"c" (nr):"cx","di")
66
   // Write port inline macros. Write port, write nr words, take data from buf.
67 #define port_write(port,buf,nr) \
68 __asm__("cld;rep;outsw"::"d" (port),"S" (buf),"c" (nr):"cx","si")
69
```

```
70 extern void hd_interrupt(void);         // Harddisk interrupt handler (sys_call.s, line 235).
71 extern void rd_load(void);              // The ram disk load function (ramdisk.c, line 71).
72
73 /* This may be used only once, enforced by 'static int callable' */
   // System setup syscall function.
   // The function parameter BIOS is set by the init subroutine in the initialization program
   // init/main.c to point to the hard disk parameter table. The hard disk parameter table structure
   // contains the contents of two hard disk parameter tables (32 bytes in total), which are copied
   // from the memory 0x90080. The information at 0x90080 is obtained by the setup.s program using
   // the ROM BIOS function. For a description of the hard disk parameter table, see Table 6-4
   // in Section 6.3.3. The main purpose of this function is to read the CMOS hard disk information,
   // used to set the hard disk partition structure hd, and try to load the RAM virtual disk and
   // the root file system.
74 int sys_setup(void * BIOS)
75 {
76         static int callable = 1;                 // used to limit invocation only once.
77         int i,drive;
78         unsigned char cmos_disks;
79         struct partition *p;
80         struct buffer_head * bh;
81
   // First set the callable flag so that this function can only be called once. Then set the hard
   // disk information array hd_info[]. If the symbol HD_TYPE is already defined in the
   // include/linux/config.h file, it means that the hd_info[] array has been set on line 49 above.
   // Otherwise, you need to read the hard disk parameter table at memory 0x90080. The setup.s
   // program will store one or two hard disk parameter tables in the memory here.
82         if (!callable)
83                 return -1;
84         callable = 0;
85 #ifndef HD_TYPE                                   // Read if HD_TYPE is not defined.
86         for (drive=0 ; drive<2 ; drive++) {
87                 hd_info[drive].cyl = *(unsigned short *) BIOS;        // cylinders.
88                 hd_info[drive].head = *(unsigned char *) (2+BIOS);    // headers.
89                 hd_info[drive].wpcom = *(unsigned short *) (5+BIOS); // write pre-com.
90                 hd_info[drive].ctl = *(unsigned char *) (8+BIOS);     // control byte.
91                 hd_info[drive].lzone = *(unsigned short *) (12+BIOS);// landing zone.
92                 hd_info[drive].sect = *(unsigned char *) (14+BIOS);   // sectors/track.
93                 BIOS += 16;                // Each hd parameter table is 16 bytes long.
94         }
   // When the setup.s program takes the BIOS hard disk parameter table information, if there is
   // only one hard disk in the system, the 16 bytes corresponding to the second hard disk will
   // be cleared. Therefore, if you check whether the number of the second hard disk cylinder is
   // 0, you can know if there is a second hard disk.
95         if (hd_info[1].cyl)
96                 NR_HD=2;                   // The nr of hard disks is set to 2.
97         else
98                 NR_HD=1;
99 #endif

   // At this point, the hard disk information array hd_info[] has been set, and the number of
   // hard disks NR_HD is determined. Now set up the hard disk partition structure array hd[]. Items
   // 0 and 5 of the array represent the overall parameters of the two hard disks, while items
   // 1-4 and 6-9 represent the parameters of the four partitions of the two hard disks,
   // respectively.Only two items (items 0 and 5) indicating the overall information of the hard
   // disk are set here.
           for (i=0 ; i<NR_HD ; i++) {
                   hd[i*5].start_sect = 0;                         // starting sector number.
                   hd[i*5].nr_sects = hd_info[i].head*
                                   hd_info[i].sect*hd_info[i].cyl;  // total nr of sectors.
           }

           /*
                   We querry CMOS about hard disks : it could be that
                   we have a SCSI/ESDI/etc controller that is BIOS
                   compatable with ST-506, and thus showing up in our
                   BIOS table, but not register compatable, and therefore
                   not present in CMOS.

                   Furthurmore, we will assume that our ST-506 drives
                   <if any> are the primary drives in the system, and
                   the ones reflected as drive 1 or 2.

                   The first drive is stored in the high nibble of CMOS
                   byte 0x12, the second in the low nibble.  This will be
                   either a 4 bit drive type or 0xf indicating use byte 0x19
                   for an 8 bit type, drive 1, 0x1a for drive 2 in CMOS.

                   Needless to say, a non-zero value means we have
                   an AT controller hard disk for that drive.


           */

   // According to the above principle, here is to check whether the hard disk is compatible with
   // the AT controller. See Section 7.1.3 for a description of CMOS information. Here, the hard
   // disk type byte is read from the CMOS offset address 0x12. If the low nibble value (the second
   // hard disk type) is not 0, it means that the system has two hard disks, otherwise the system
   // has only one hard disk. If the value read from 0x12 is 0, it means there is no AT compatible
   // hard disk in the system.
           if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
                   if (cmos_disks & 0x0f)
                           NR_HD = 2;
                   else
                           NR_HD = 1;
           else
                   NR_HD = 0;
   // If NR_HD = 0, the two hard disks are not compatible with the AT controller, and the data
   // structures of the two hard disks are all cleared. If NR_HD = 1, the parameter of the second
   // hard disk is cleared.
           for (i = NR_HD ; i < 2 ; i++) {
                   hd[i*5].start_sect = 0;
                   hd[i*5].nr_sects = 0;
           }
   // Ok, so far we have clearly determined the number of hard disks NR_HD contained in the system.
   // Now let's read the partition table information in the first sector of each hard disk, which
   // is used to set the information of each partition of the hard disk in the partition structure
   // array hd[].  First, read the first data block of the hard disk (fs/buffer.c, line 267) by
   // using the read block function bread().The first parameter (0x300, 0x305) is the device number
   // of the two hard disks, and the second parameter (0) is the block number to be read.  If the
   // read operation is successful, the data will be stored in the data area of the buffer block
   // bh.  If the buffer block head pointer bh is 0, it means that the read operation failed, the
   // error message is displayed and the machine stops. Otherwise, we judge the validity of the
   // data according to whether the last two bytes of the first sector are 0xAA55, so that it can
   // be known whether the partition table at the beginning of the offset 0x1BE in the sector is
   // valid. If it is valid, the hard disk partition table information is stored in the hard disk
   // partition structure array hd[]. Finally release the bh buffer.
           for (drive=0 ; drive<NR_HD ; drive++) {
                   if (!(bh = bread(0x300 + drive*5,0))) {        // 0x300、0x305 is dev no.
                           printk("Unable to read partition table of drive %d\n\r",
                                   drive);
                           panic("");
                   }
                   if (bh->b_data[510] != 0x55 || (unsigned char)
                       bh->b_data[511] != 0xAA) {                 // check flag 0xAA55
                           printk("Bad partition table on drive %d\n\r",drive);
                           panic("");
                   }
                   p = 0x1BE + (void *)bh->b_data;    // partition table is located at 0x1BE.
                   for (i=1;i<5;i++,p++) {
                           hd[i+5*drive].start_sect = p->start_sect;
                           hd[i+5*drive].nr_sects = p->nr_sects;
                   }
                   brelse(bh);                        // Release the buffer.
           }
   // Now count the total number of data blocks in each partition and save it in the hard disk
   // partition total data block array hd_sizes[]. Then let the block device item (3) of the block
   // size array point to this array.
           for (i=0 ; i<5*MAX_HD ; i++)
                   hd_sizes[i] = hd[i].nr_sects>>1 ;
           blk_size[MAJOR_NR] = hd_sizes;             // MAJOR_NR = 3

   // Now we have finally completed the task of setting the hard disk partition structure array
   // hd[]. If a hard disk does exist and its information has been read into its partition table,
   // the ok message is displayed. Then try to load the root fs image (blk_drv/ramdisk.c, line
   // 71) into the memory ram disk. That is, if the system is provided with a virtual ram disk,
   // it is determined whether the boot disk also contains the image data of the root fs. If there
   // is (the boot disk is called the integrated disk at this time), try to load and store the
   // image into the ram disk, and then modify the root fs device number ROOT_DEV to the device
   // number of the ram disk. The switching device is then initialized. Finally install the root
   // file system.
           if (NR_HD)
                   printk("Partition table%s ok.\n\r",(NR_HD>1)?"s":"");
           rd_load();                           // blk_drv/ramdisk.c, line 71.
           init_swapping();                     // mm/swap.c, line 199.
           mount_root();                        // fs/super.c, line 241.
           return (0);
   }

   //// Check and cycle to wait for the hard disk controller to be ready.
```

```
    // Read the hard disk controller status register port HD_STATUS (0x1f7), loop to detect if the
    // drive ready bit (bit 6) is set, and if the controller busy bit (bit 7) is reset. If the return
    // value retries is 0, it means that the time waiting for the controller to idle has timed out
    // and an error occurs; if the return value is not 0, the controller returns to the idle state
    // during the waiting (loop) time period, OK!
    // In fact, we only need to check if the status register busy bit (bit 7) is 1 to determine
    // if the controller is busy. Whether the drive is ready (ie, bit 6 is 1) is independent of
    // the state of the controller. So we can rewrite the line 172 statement as:
    //      while (--retries && (inb_p(HD_STATUS)&0x80));
    // In addition, since the current PC speed is very fast, we can increase the number of waiting
    // cycles, for example, 10 times more!
168 static int controller_ready(void)
169 {
170         int retries = 100000;
171 
172         while (--retries && (inb_p(HD_STATUS)&0xc0)!=0x40);
173         return (retries);
174 }
175 
    // Checks the status of hard disk after executing a command (win stands for Winchester hd)
    // Read the status of the command execution result in the status register. Returning 0 means
    // normal; 1 means error. If the execution command is wrong, you need to read the error register
    // HD_ERROR (0x1f1).
176 static int win_result(void)
177 {
178         int i=inb_p(HD_STATUS);                // get status.
179 
180         if ((i & (BUSY_STAT | READY_STAT | WRERR_STAT | SEEK_STAT | ERR_STAT))
181                 == (READY_STAT | SEEK_STAT))
182                 return(0); /* ok */
183         if (i&1) i=inb(HD_ERROR);              // if ERR_STAT is set, read HD_ERROR.
184         return (1);
185 }
186 
    //// Send command block to the hard disk controller.
    // Params: drive - harddisk no (0-1); nsect - nr of read/write sectors;
    // sect - starting sector; head - head number; cyl - cylinder number;
    // cmd - command code (see controller cmd list); intr_addr() - C function pointer.
    // After the hard disk controller is ready, the function sets the global function pointer variable
    // do_hd to point to the C handler that will be called in the hard disk interrupt handler, and
    // then sends the hard disk control byte and the 7-byte parameter command block. The harddisk
    // interrupt handler is located at line235 in file kernel/sys_call.s.
    // Line 191 defines a register variable __res. This variable will be saved in a register for
    // quick access. If you want to specify a register (such as eax), you can write the sentence
    // as "register char __res asm("ax");".
187 static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
188                 unsigned int head,unsigned int cyl,unsigned int cmd,
189                 void (*intr_addr)(void))
190 {
191         register int port asm("dx");            // define a register variable.
192 
    // First check the validity of the parameters. If the drive number is greater than 1 (only 0,
    // 1 is valid) or the head number is greater than 15, the program does not support, stop. Otherwise
// check and loop waiting for the drive to be ready. If it is not ready after waiting for a
// while, it indicates that the hard disk controller is faulty and also stops.
        if (drive>1 || head>15)
                panic("Trying to write bad sector");
        if (!controller_ready())
                panic("HD controller not ready");

// Next, we set the C function pointer do_hd that will be called when the hard disk interrupt
// occurs (the function pointer is defined between lines 56-109 in file blk.h, please pay special
// attention to lines 83-100). Then send a control byte to the hard disk controller cmd port
// (0x3f6) to establish the control mode of the specified drive. This control byte is the ctl
// field in the array of hard disk information structures. After that, a 7-byte parameter command
// block is sent to controller port 0x1f1-0x1f7.
        SET_INTR(intr_addr);                    // do_hd = intr_addr
        outb_p(hd_info[drive].ctl,HD_CMD);      // out control byte.
        port=HD_DATA;                           // (0x1f0)
        outb_p(hd_info[drive].wpcom>>2,++port); // Param: write pre-comp.(divided by 4)
        outb_p(nsect,++port);                   // Param: total nr of r/w sectors.
        outb_p(sect,++port);                    // Param: starting sector.
        outb_p(cyl,++port);                     // Param: The cylinder nr low 8 bits.
        outb_p(cyl>>8,++port);                  // Param: The cylinder nr high 8 bits.
        outb_p(0xA0|(drive<<4)|head,++port);    // Param: drive no + head no.
        outb(cmd,++port);                       // Param: Hard disk command.
}

//// Wait for the drive to be ready.
// This function loops to wait for the main status register busy flag to be reset. If the ready
// or seek finished flag is set, the hard disk is ready and if it is successful, it returns
// 0. Returns 1 if it is still busy after a while.
static int drive_busy(void)
{
        unsigned int i;
        unsigned char c;

// The master status register HD_STATUS of the controller is cyclically read, waiting for the
// ready flag bit to be set, and the busy bit is reset. Then detect the busy bit,ready bit,
// and the seek end bit. If only the ready or seek end flag is set, it indicates that the hard
// disk is ready and returns 0. Otherwise, it means that the timeout has expired at the end
// of the loop, so the warning message is displayed and 1 is returned.
        for (i = 0; i < 50000; i++) {
                c = inb_p(HD_STATUS);                     // get main status byte.
                c &= (BUSY_STAT | READY_STAT | SEEK_STAT);
                if (c == (READY_STAT | SEEK_STAT))
                        return 0;
        }
        printk("HD controller times out\n\r");
        return(1);
}

//// Diagnostic reset of the hard disk controller.
// The enable reset (4) control byte is first sent to the control register port (0x3f6), and
// then waits for a period of time for the controller to perform a reset operation. The normal
// control byte (allow retry, re-read) is then sent to the port , and wait for the hard disk
```

```
    // is ready. If you wait for the hard disk ready timeout, a busy warning message is displayed.
    // Then read the contents of the error register. If it is not equal to 1 (1 means no error),
    // the hard disk controller reset failure message is displayed.
224 static void reset_controller(void)
225 {
226         int     i;
227
228         outb(4,HD_CMD);                        //  4 is reset byte.
229         for(i = 0; i < 1000; i++) nop();       // wait a while.
230         outb(hd_info[0].ctl & 0x0f ,HD_CMD);   // normal control byte(allow retry, reread).
231         if (drive_busy())
232                 printk("HD-controller still busy\n\r");
233         if ((i = inb(HD_ERROR)) != 1)
234                 printk("HD-controller reset failed: %02x\n\r",i);
235 }
236
    //// Hard disk reset operation.
    // First reset the hard disk controller and then send the hard disk controller command "Create
    // Drive Parameters". This function is called again in the hard disk interrupt handler caused
    // by this command. At this point, the function will determine whether to perform error processing
    // or continue to perform the request item processing based on the result of executing the
    // command.
237 static void reset_hd(void)
238 {
239         static int i;
240
    // If the reset flag is set, the reset hard disk controller operation is performed after the
    // reset flag is cleared. Then send the "Create Drive Parameters" command to the controller
    // for the i-th hard disk. When the controller executes the command, it will issue a hard disk
    // interrupt signal. At this point, this function will be executed again by the interrupt handler.
    // Since the reset flag has been reset at this time, the statement starting at line 246 will
    // be executed first to determine whether the command execution is ok. If an error still occurs,
    // the bad_rw_intr() function is called to count the number of errors and determine if the reset
    // flag is set again based on the error times. If the reset flag is set again, it will jump
    // to the label repeat to re-execute this function. If the reset operation is ok, the "Create
    // Drive Parameter" command is sent for the next hard disk, and the same processing as described
    // above is performed. If all hard disks in the system have executed the sent command normally,
    // the do_hd_request() function is called again to start processing the request item.
241 repeat:
242         if (reset) {
243                 reset = 0;
244                 i = -1;                         // initialize the current hd number.
245                 reset_controller();
246         } else if (win_result()) {
247                 bad_rw_intr();
248                 if (reset)
249                         goto repeat;
250         }
251         i++;                                    // handling next hd.
252         if (i < NR_HD) {
253                 hd_out(i,hd_info[i].sect,hd_info[i].sect,hd_info[i].head-1,
254                         hd_info[i].cyl,WIN_SPECIFY,&reset_hd);
255         } else
```

```
                do_hd_request();                 // request item processing.
}

//// The default function called by an unexpected hard disk interrupt.
// This function is the default C function called in the hard disk interrupt handler when an
// unexpected hard disk interrupt occurs. This function is invoked when the called function
// pointer is NULL. See kernel/sys_call.s, line 256. The function sets the reset flag after
// displaying the warning message, and then continues to invoke the request item function
// go_hd_request() and performs a reset processing operation therein.
void unexpected_hd_interrupt(void)
{
        printk("Unexpected HD interrupt\n\r");
        reset = 1;
        do_hd_request();
}

//// The processing function for handling hard disk read/write failure.
// If the number of errors in the read sector operation is greater than or equal to 7, the current
// request item is terminated and the process waiting for the request is awake, and the
// corresponding buffer update flag is reset, indicating that the data is not updated. If the
// number of errors in reading/writing a sector operation has been greater than 3 times, it
// is required to perform a controller reset operation (set reset flag).
static void bad_rw_intr(void)
{
        if (++CURRENT->errors >= MAX_ERRORS)
                end_request(0);
        if (CURRENT->errors > MAX_ERRORS/2)
                reset = 1;
}

//// The read sector function called in the interrupt.
// This function will be called during the hard disk interrupt that is triggered at the end
// of the hard disk read command. The controller generates an interrupt request signal after
// the read command is executed and triggers the execution of the interrupt handler. At this
// point, the C function pointer do_hd in interrupt handler has pointed to read_intr(), so the
// function will be executed after a read sector operation is completed(or an error).
static void read_intr(void)
{
// This function first determines if the read command operation is in error. If the controller
// is still busy after the command is finished, or the command execution error occurs, the hard
// disk operation failure problem is handled, and then the hard disk is again requested to perform
// reset processing and continue to execute other request items, and then returns.
//
// In the function bad_rw_intr(), each read operation error will accumulate the number of errors
// in the current request item. If the number of errors is less than half of the maximum number
// of allowed, the hard disk reset operation will be performed first, and then the request
// processing will be executed. If the number of errors has been greater than or equal to the
// maximum number of allowed errors MAX_ERRORS (7 times), the processing of this request item
// is ended, and the next request item in the queue is processed.
// In function do_hd_request(), it is determined whether the reset, ecalibration, etc. need
// to be performed according to the specific flag status at that time, and then the next request
// is continued or processed.
        if (win_result()) {                  // if there is error...
277                 bad_rw_intr();               // r/w failure handling.
278                 do_hd_request();             // continue handling request items.
279                 return;
280         }
    // If there is no error in the read operation, the data of one sector is read from the data
    // register port into the buffer of the request, and the sector number to be read is decremented.
    // If it is not equal to 0 after decrementing, it means that there is data to be read in this
    // request, so the interrupt C function pointer sets to point to ead_intr() again and return
    // directly, waiting for the hard disk to interrupt again after reading another sector data.
    // Note 1: Number 256 in line 281 refers to the memory word, which is 512 bytes.
    // Note 2: The statement on line 262 again sets the do_hd pointer to read_intr() because the
    // hard disk interrupt handler will set the function pointer to NULL each time do_hd is called.
See lines 251-253 in kernel/sys_call.s file.
281         port_read(HD_DATA,CURRENT->buffer,256);     // put in buffer.
282         CURRENT->errors = 0;                    // Clear the number of errors.
283         CURRENT->buffer += 512;                 // adjust the buffer pointer.
284         CURRENT->sector++;                      // increase sectors read.
285         if (--CURRENT->nr_sectors) {            // if there is still data to read
286                 SET_INTR(&read_intr);           // points to read_intr() again.
287                 return;
288         }
    // Execution here, indicating that all sector data of this request item has been read. The
    // end_request() function is then called to handle the end of the request. Finally, call
    // do_hd_request() again to process other hard disk requests.
289         end_request(1);                         // set data updated flag.
290         do_hd_request();                        // Perform other request operations.
291 }
292
    //// The write sector function called in the interrupt.
    // This function will be called during the hard disk interrupt that is triggered at the end
    // of the hard disk write command. This function is handled similarly to read_intr(). After
    // the write command is executed, a hard disk interrupt signal is generated and the hard disk
    // interrupt handler is executed. At this point, the C function pointer do_hd called in the
    // interrupt handler has already pointed to write_intr(), so the function will be executed after
    // a write sector operation is completed (or an error).
293 static void write_intr(void)
294 {
    // This function first determines if the write command operation is in error. If the controller
    // is still busy after the command is finished, or the command execution error occurs, the hard
    // disk operation failure problem is handled, and then the hard disk is again requested to perform
    // reset processing and continue to execute other request items, and then returns.
295         if (win_result()) {                    // If controller returns error messages,
296                 bad_rw_intr();                 // r/w error handling.
297                 do_hd_request();               // continue handling request items.
298                 return;
299         }
    // At this point, it indicates that the write one sector operation is successful, so the number
    // of sectors to be written is decremented by one. If it is not 0, it means that there are still
    // sectors to write, so the current request start sector number is +1, and the request data
    // buffer pointer is adjusted to point to the next piece of data to be written. Then reset the
    // C function pointer do_hd in the hard disk interrupt handler to point to this function. The
    // 512-byte data is then written to the controller data port, and the function returns to wait
    // for the interrupt generated after the operation is completed.
```

```
        if (--CURRENT->nr_sectors) {            // If there are still sectors to write...
                CURRENT->sector++;
                CURRENT->buffer += 512;
                SET_INTR(&write_intr);          // do_hd points write_intr() again.
                port_write(HD_DATA,CURRENT->buffer,256);
                return;
        }
// If all the sector data of this request item has been written, the end_request() function
// is called to process the end of the request item. Finally, call do_hd_request() again to
// process other requests.
        end_request(1);                        // set data updated flag.
        do_hd_request();                       // Perform other request operations.
}

//// The hard disk recalibration (reset) function called in the interrupt.
// If the hard disk controller returns an error message, the function first performs a read/write
// failure process, and then requests the hard disk to perform a corresponding (reset) process.
// In function do_hd_request(), it is determined whether the reset, ecalibration, etc. need
// to be performed according to the specific flag status at that time, and then the next request
// is continued or processed.
static void recal_intr(void)
{
        if (win_result())                     // if error invoke bad_rw_intr()
                bad_rw_intr();
        do_hd_request();
}

// Hard disk timeout processing function.
// This function will be called in do_timer() (kernel/sched.c, line 340). After sending a command
// to the hard disk controller, if the controller has not issued any interrupt signal after
// hd_timeout ticks, the operation of the controller (or hard disk) has timed out. At this point
// do_timer () will call this function to set the reset flag, and call do_hd_request () to perform
// the reset process. If the hard disk controller issues a hard disk interrupt request signal
// within a predetermined time (200 ticks) and starts executing the hard disk interrupt handler,
// the ht_timeout value is set to 0 in the handler. At this point do_timer() will skip this
// function.
void hd_times_out(void)
{
// If there is currently no request item to process (the request item pointer is NULL), there
// is no timeout and it can be returned directly. Otherwise, the warning message is displayed
// first, and then it is judged whether the number of errors occurring during the execution
// of the current request item is greater than the value MAX_ERRORS (7). If yes, the processing
// of this request item is terminated in a failure form (data updated flag is not set), then
// the C function pointer do_hd is set to NULL, and the reset flag is set. The reset operation
// is then performed in the request item handler do_hd_request().
        if (!CURRENT)
                return;
        printk("HD timeout");
        if (++CURRENT->errors >= MAX_ERRORS)
                end_request(0);
        SET_INTR(NULL);                       // let do_hd = NULL, time_out = 200
        reset = 1;
        do_hd_request();
```

```
328 }
329
    //// Perform hard disk read/write request operations.
    // The function first calculates the cylinder number on the hard disk, the sector number in
    // the current track, the head number, etc. according to the device number and the starting
    // sector number information in the current request item, and then according to the command
    // in the request item (READ/ WRITE) sends a read or write command to the hard disk.  If the
    // controller reset flag or the recalibration flag has been set, then a reset or recalibration
    // operation will be performed first.
    // If the request item is the first one of the block device (the device is originally idle),
    // the current request item pointer will directly point to the request item (see ll_rw_blk.c,
    // line 84), and the function will be called to execute the read/write operation. Otherwise,
    // in the handling of hard disk interrupt caused by the completion of a read/write, if there
    // is still a request item to be processed, this function will also be called during the interrupt
    // handling.  See kernel/sys_call.s, line 235.
330 void do_hd_request(void)
331 {
332         int i,r;
333         unsigned int block,dev;
334         unsigned int sec,head,cyl;
335         unsigned int nsect;
336
    // The function first checks the validity of the request item. Exit if there are no requests
    // in the request queue (see blk.h, line 148). Then take the sub-device number in the device
    // number and the starting sector in the current request item. The sub-device number corresponds
    // to each partition on the hard disk. If the sub-device number does not exist or the starting
    // sector is greater than the partition sector number -2, the request is terminated and jumps
    // to the label repeat (defined in macro INIT_REQUEST, line 149 in blk.h). Since one block of
    // data (2 sectors, ie 1024 bytes) is required to be read/written at a time, the requested sector
    // number cannot be greater than the last penultimate sector number in the partition. Then,
    // by adding the starting sector number of the partition corresponding to the sub-device number,
    // the block to be read/written is mapped to the absolute sector number of the entire hard disk.
    // The sub-device number is divided by 5 to get the corresponding hard disk number.  For the
    // description of the hard disk device number, see Table 6-1 in Section 6.2.3.
337         INIT_REQUEST;
338         dev = MINOR(CURRENT->dev);
339         block = CURRENT->sector;                          // starting sector.
340         if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
341                 end_request(0);
342                 goto repeat;                              // in blk.h, line 149.
343         }
344         block += hd[dev].start_sect;
345         dev /= 5;                                  // dev is now the hd no (0,or 1).

    // Then, based on the obtained absolute sector number 'block' and hard disk number 'dev',
    // we can calculate the sector number (sec), the cylinder number (cyl), and the head number
    // (head) in the hard disk. The following inline assembly code is used to calculate these data
    // according to the number of sectors per track and the number of heads in the hard disk
    // information structure. The calculation method is:
    // The statement on line 346 indicates that EAX is the sector number 'block' and "0" in EDX.
    // The DIVL instruction divides the sector number EDX:EAX by the number of sectors per track
    // (hd_info[dev].sect), the resulting quotient is in EAX, and the remainder is in EDX. Among
    // them, EAX is the total number of tracks to the specified position (all head faces), and EDX
    // is the sector number on the current track.
    // The statement on line 348 indicates that EAX is the calculated total number of tracks, and
    // EDX is set to zero. The DIVL instruction divides the total number of tracks EDX:EAX by the
    // total number of heads (hd_info[dev].head). The integer division value obtained in EAX is
    // the cylinder number (cyl), and the remainder obtained in EDX is the head number (head).
346         __asm__("divl %4":"=a" (block),"=d" (sec):"0" (block),"1" (0),
347                 "r" (hd_info[dev].sect));
348         __asm__("divl %4":"=a" (cyl),"=d" (head):"0" (block),"1" (0),
349                 "r" (hd_info[dev].head));
350         sec++;                                  // current track sector number is adjusted.
351         nsect = CURRENT->nr_sectors;            // The number of sectors to read/write.

    // Now we have the cylinder number (cyl) corresponding to the starting sector 'block' to be
    // read/written, the sector number (sec) on the current track, the head number (head), and the
    // total sector number (nsect) to be read or written. . Then we can send I/O operation commands
    // to the hard disk controller based on this information. But before sending, we also need to
    // see if there are flags to reset the controller state and recalibrate the hard disk. It is
    // usually necessary to recalibrate the hard disk head position after the reset operation. If
    // these flags have been set, it may indicate some problems with the previous hard disk operation,
    // or it is now the first hard disk read and write operation of the system, so we need to reset
    // the hard disk or controller and recalibrate the hard disk.
    //
    // If the reset flag is set at this time, a reset operation is required. Then reset the hard
    // disk and controller, and set the flag that hard disk needs to be recalibrated, and return.
    // The function reset_hd() will first send a reset (recalibration) command to the hard disk
    // controller and then send the command "Create Drive Parameter".
352         if (reset) {
353                 recalibrate = 1;                // need to recalibrate.
354                 reset_hd();
355                 return;
356         }
    // If the recalibrate flag is set at this time, the flag is first reset and then a recalibration
    // command is sent to the controller. This command performs a track seek operation to move the
    // head from anywhere to the 0 cylinder.
357         if (recalibrate) {
358                 recalibrate = 0;
359                 hd_out(dev,hd_info[CURRENT_DEV].sect,0,0,0,
360                         WIN_RESTORE,&recal_intr);
361                 return;
362         }
    // If none of the above two flags are set, then we can start sending real data read/write
    // operations to the hard disk controller. If the current request is a write sector operation,
    // a write command is sent, and then the status register information is cyclically read and
    // it is determined whether the request service flag DRQ_STAT is set. DRQ_STAT is the request
    // service bit of the hard disk status register, indicating that the drive is ready to transfer
    // one word or one byte of data between the host and the data port. Exit the loop if the request
    // service DRQ is set. If it is not set after the end of the loop, it means that the request
    // to write the hard disk command failed, so jump to deal with the problem or continue to execute
    // the next hard disk request. Otherwise, we can write 1 sector of data to the hard disk controller
    // data register port HD_DATA.
363         if (CURRENT->cmd == WRITE) {
364                 hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
365                 for(i=0 ; i<10000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
```

```
                        /* nothing */ ;
                if (!r) {
                        bad_rw_intr();
                        goto repeat;               //  label is in blk.h, line 149.
                }
                port_write(HD_DATA,CURRENT->buffer,256);

    // If the current request is to read hard disk data, a read sector command is sent to the
    // controller. Stop if the command is invalid.
        } else if (CURRENT->cmd == READ) {
                hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
        } else
                panic("unknown hd-command");
}

    // Hard disk system initialization.
    // The function sets the hard disk interrupt descriptor and enable the controller to send
    // interrupt request signal. It sets the request handler of the hard disk to do_hd_request(),
    // and then sets the hard disk interrupt gate descriptor. hd_interrupt is the address of its
    // interrupt handler (kernel/sys_call.s, line 235).  The hard disk interrupt is INT 0x2E (46),
    // which corresponds to the interrupt request signal IRQ14 of the 8259A chip. The mask bit of
    // INT2 on the master chip 8259A is then reset, allowing interrupt request signal to be issued
    // from slave chip. Then reset the interrupt mask bit of the hard disk (on the slave),allowing
    // the hard disk controller to send an interrupt request signal. The macro set_intr_gate() of
    // the interrupt gate descriptor in IDT is in include/asm/system.h.
void hd_init(void)
{
        blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;     // do_hd_request()。
        set_intr_gate(0x2E,&hd_interrupt);                 // set interrupt handler
        outb_p(inb_p(0x21)&0xfb,0x21);                 // reset mask bit 2 on master chip.
        outb(inb_p(0xA1)&0xbf,0xA1);                   // reset mask bit 6 on slave chip.
}

```

9.3.3 Information

9.3.3.1 AT Hard Disk Interface Register

The programming register port description for the AT hard disk controller is shown in Table 9–3. Also see the description in the include/linux/hdreg.h header file.

Table 9-3 AT hard disk controller register port and its use

Port	Name	Read operation	Write operation
0x1f0	HD_DATA	Data Register - Sector data (read, write and format)	
0x1f1	HD_ERROR,HD_PRECOMP	Error Register (HD_ERROR)	Write Precomp Register (HD_PRECOMP)
0x1f2	HD_NSECTOR	Total Sectors Register - Total number of sectors (read, write, check and format)	
0x1f3	HD_SECTOR	Sector Number Register - Start sector number (read, write and check)	
0x1f4	HD_LCYL	Cylinder Number Register - Cylinder number low byte (read,write,check and format)	
0x1f5	HD_HCYL	Cylinder Number Register - Cylinder number high byte (read,write,check and format)	
0x1f6	HD_CURRENT	Drive/Head Register - Drive/Head No.(101dhhhh, d=Drvie No, h=Head No)	
0x1f7	HD_STATUS,HD_COMMAND	Main Status Register (HD_STATUS)	Command Register (HD_COMMAND)

0x3f6	HD_CMD	---	Harddisk Control Register (HD_CMD)
0x3f7		Digital Input Register (1.2MB Floopy)	---

The port registers are described in detail below.

◆Data register (HD_DATA, 0x1f0)

This is a pair of 16-bit high speed PIO data transmitters for sector read, write and track formatting operations. The CPU writes to the hard disk through the data register or reads data of one sector from the hard disk, that is, repeats reading/writing cx=256 words using the command 'rep outsw' or 'rep insw'.

◆Error register (read) / write precompensation register (write) (HD_ERROR, 0x1f1)

When reading, this register holds an 8-bit error status. However, the data in this register is valid only when Bit 0 of the main status register (HD_STATUS, 0x1f7) is set. The meaning when executing the controller diagnostic command is different from that of other commands. See Table 9–4.

This register acts as a write pre-compensation register during a write operation. It records the write pre-compensation starting cylinder number. Corresponds to a word at 0x05 of the basic parameter table of the hard disk, which is divided by 4 and output. But most of the current hard drives ignore this parameter. For a description of the hard disk parameter table, see Table 6-4 in Section 6.3.3.

What is write pre-compensation?

Early hard disks have a fixed number of sectors per track, and since each sector has a fixed 512 bytes, the physical track length occupied by each sector is shorter as it gets closer to the center of the disk. The ability to cause magnetic media to store data is reduced. Therefore, for the hard disk head, it is necessary to take certain measures to put the data of one sector into a relatively small sector at a relatively high density. The common method used is the Write Precompensation technique. That is, starting from the edge of the disc to a position near a track (cylinder) in the center of the disc, the write current in the head is adjusted in some way.

The specific adjustment method is as follows: the representation of the binary data 0, 1 on the disk is recorded by a magnetic recording and encoding method (for example, FM, MFM, etc.). If the adjacent recording bits are magnetized and flipped twice, magnetic field overlap may occur. Therefore, the peak value of the corresponding electrical waveform will drift when the data is read at this time. If the recording density is increased, the degree of peak drift is increased, and sometimes the data bits cannot be separated and recognized, resulting in read data errors. The way to overcome this problem is to use pre-write compensation or post-read compensation techniques. Pre-write compensation means that the pulse compensation is written in advance in the opposite direction to the peak drift of the readout before the write data is sent to the driver. If the peak value of the signal drifts forward when read, the signal is delayed to be written; if the signal drifts backward when read, the signal is written in advance. Thus, when reading, the position of the peak can be close to the normal position.

Table 9-4 Hard disk controller error register

Value	Dianostic command	Other commands
0x01	No error	Data flag missing
0x02	Controller error	Track 0 error
0x03	Sector buffer error	
0x04	ECC part error	Command abandon
0x05	Control processor error	
0x10		ID not found
0x40		ECC error
0x80		Bad sector

◆Sector Number Register (HD_NSECTOR, 0x1f2)

This register holds the number of sectors specified by the read, write, verify, and format commands. When used for multi-sector operation, the register is automatically decremented by one each time a sector operation is completed, until it is zero. If the initial value is 0, it means that the maximum number of sectors is 256.

◆Begin Sector No Register (HD_SECTOR, 0x1f3)

This register holds the sector start number specified by the read, write, and verify operation commands. In multi-sector operation, the starting sector number is stored, and the operation is automatically incremented by 1 for each sector operation.

◆Cylinder number register (HD_LCYL, HD_HCYL, 0x1f4, 0x1f5)

The two cylinder registers respectively store the lower 8 bits and the upper 2 bits of the cylinder number.

◆Driver/head register (HD_CURRENT, 0x1f6)

This register holds the drive and head numbers specified by the read, write, verify, seek, and format commands. Its bit format is 101dhhhh. Where 101 indicates that the ECC check code is used and 512 bytes per sector; d indicates the selected drive (0 or 1); hhhh indicates the selected head, as shown in Table 9–5.

Table 9-5 Drive/head register

Bit	Name	Description
0	HS0	Head number lowest bit
1	HS1	
2	HS2	
3	HS3	Head number highest bit
4	DRV	Select drive, 0 - Select drive 0; 1 - Select drive 1
5	Reserved	Always 1
6	Reserved	Always 0
7	Reserved	Always 1

◆Main Status Register (Read) / Command Register (Write) (HD_STATUS/HD_COMMAND, 0x1f7)

At the time of reading, it corresponds to an 8-bit main status register. Reflects the operating state of the hard disk controller before and after executing the command. The meaning of each bit in this register is shown in Table 9–6.

Table 9-6 8-bit main status register

Bit	Name	Mask	Description
0	ERR_STAT	0x01	Command execution error. When this bit is set, the previous command ends with an error. At this point, the bits in the error register and status register contain some information that caused the error.
1	INDEX_STAT	0x02	Received the index. This bit is set when an index flag is encountered while the disk is spinning.
2	ECC_STAT	0x04	ECC checksum error. This bit is set when a recoverable data error is encountered and has been corrected. This situation does not interrupt a multi-sector read operation.
3	DRQ_STAT	0x08	Data request service. When this bit is set, it indicates that the drive is ready to transfer one word or one byte of data between the host and the data port.
4	SEEK_STAT	0x10	The drive seek ends. When this bit is set, it indicates that the seek operation has been

			completed and the head has stopped on the specified track. This bit does not change when an error occurs. This bit will again indicate the completion status of the current seek after the host has read the status register.
5	WRERR_STAT	0x20	Drive failure (write error). This bit does not change when an error occurs. This bit will again indicate the error status of the current write operation only after the host has read the status register.
6	READY_STAT	0x40	The drive is ready. Indicates that the drive is ready to receive commands. This bit does not change when an error occurs. This bit will again indicate the current drive ready state after the host has read the status register. At power-on, this bit should be reset until the drive speed is normal and the command can be received.
7	BUSY_STAT	0x80	The controller is busy. This bit is set when the drive is operating by the controller of the drive. At this point, the host cannot send a command block. A read of any of the command registers will return the value of the status register. This bit will be set under the following conditions: (*) It is within 400 nanoseconds after the machine reset signal RESET becomes negative or the SRST of the device control register is set. The set state of this bit is required to be no longer than 30 seconds after a machine reset. (*) The host is within 400 nanoseconds of writing commands such as recalibration, read, read buffer, initialization of drive parameters, and execution of diagnostics. (*) Within 5 microseconds of 512 bytes of data transferred during a write operation, write buffer, or format track command.

When a write operation is performed, the port corresponds to a command register. It accepts hard disk control commands from the CPU. There are 8 commands, as shown in Table 9–7. The last column is used to describe the actions taken by the controller after the end of the corresponding command (causing an interrupt or doing nothing).

Table 9-7 AT hard disk controller command list

Command Name		Command Code Byte		Default value	Command End Form
		High 4 bits	D3 D2 D1 D0		
WIN_RESTORE	Drive Recalibration (Reset)	0x1	R R R R	0x10	Interrupt
WIN_READ	Read Sector	0x2	0 0 L T	0x20	Interrupt
WIN_WRITE	Write Sector	0x3	0 0 L T	0x30	Interrupt
WIN_VERIFY	Sector Check	0x4	0 0 0 T	0x40	Interrupt
WIN_FORMAT	Format track	0x5	0 0 0 0	0x50	Interrupt
WIN_INIT	Controller Initialization	0x6	0 0 0 0	0x60	Interrupt
WIN_SEEK	Seek Track	0x7	R R R R	0x70	Interrupt
WIN_DIAGNOSE	Controller Dianostic	0x9	0 0 0 0	0x90	Interrupt or Idle
WIN_SPECIFY	Build Drive Parameters	0x9	0 0 0 1	0x91	Interrupt

The lower 4 bits of the command code byte in the table are additional parameters, which means:

R is the step rate. When R=0, the step rate is 35us; R=1 is 0.5ms, which is incremented by this amount. The default R=0 in the program.

L is the data mode. L = 0 indicates that the read/write sector is 512 bytes; L = 1 indicates that the read/write sector is 512 plus 4 bytes of ECC code. The default value in the program is L=0.

T is the retry mode. T=0 indicates that retry is allowed; T=1 prohibits retry. Take T=0 in the kernel program.

These commands are described in detail below.

(1) 0x1X -- (WIN_RESTORE), drive recalibrate command

This command moves the read/write head from any position on the disk to the 0 cylinder. When the command is received, the drive sets the BUSY_STAT flag and issues a 0 cylinder seek command. The driver then waits for the end of the seek operation, then updates the state, resets the BUSY_STAT flag, and generates an interrupt.

(2) 0x20 -- (WIN_READ) retryable read sector; 0x21 -- no retry read sector.

The read sector command can read from 1 to 256 sectors starting from the specified sector. If the sector count in the specified command block (see Table 9-9) is 0, it means that 256 sectors are read. When the drive accepts the command, the BUSY_STAT flag will be set and execution of the command will begin. For a single sector read operation, if the track position of the head is not correct, the driver implicitly performs a seek operation. Once the head is on the correct track, the drive head is positioned to the corresponding ID field in the track address field.

For the no retry sector read command, if the specified ID field cannot be read correctly before the two index pulses occur, the drive will give an error message that the ID is not found in the error register. For a retryable read sector command, the drive will retry multiple times when it encounters a problem in the read ID field. The number of retries is set by the drive manufacturer.

If the drive correctly reads the ID field, it needs to identify the Data Address Mark in the specified number of bytes, otherwise it will report an error that the data address mark did not find. Once the head finds the data address mark, the drive reads the data in the data field into the sector buffer. If an error occurs, the drive sets the error bit, sets DRQ_STAT, and generates an interrupt. Regardless of whether an error occurs, the drive always sets DRQ_STAT after reading the sector. After the command is completed, the command block register will contain the cylinder number, head number and sector number of the last sector read.

For multi-sector read operations, each time the drive is ready to send a sector of data to the host, DRQ_STAT is set, the BUSY_STAT flag is cleared, and an interrupt is generated. When the sector data transfer ends, the drive resets the DRQ_STAT and BUSY_STAT flags, but sets the BUSY_STAT flag after the last sector transfer is complete. At the end of the command, the command block register will contain the cylinder number, head number and sector number of the last sector read.

If an uncorrectable error occurs in a multi-sector read operation, the read operation will terminate at the sector where the error occurred. Similarly, the command block register will contain the cylinder number, head number and sector number of the error sector. Regardless of whether the error can be corrected, the drive puts the data into the sector buffer.

(3) 0x30 -- (WIN_WRITE) retryable write sector; 0x31 -- no retry write sector.

The Write Sector command can write from 1 to 256 sectors starting from the specified sector. If the sector count in the specified command block (see Table 9-9) is 0, it means that 256 sectors are to be written. When the drive accepts the command, it sets DRQ_STAT and waits for the sector buffer to be filled with data. There is no interruption when starting to add data to the sector buffer for the first time. Once the data is full, the drive resets the DRQ, sets the BUSY_STAT flag, and begins executing the command.

For operations that write data for one sector, the drive sets DRQ_STAT when it receives the command and waits for the host to fill the sector buffer. Once the data has been transferred, the drive sets BUSY_STAT and resets DRQ_STAT. As with the read sector operation, if the track position of the head is incorrect, the drive implicitly performs a seek operation. Once the head is on the correct track, the drive head is positioned to the correspondingID field in the track address field.

If the ID field is correctly read, the data in the sector buffer, including the ECC byte, is written to disk. When the drive has processed a sector, the BUSY_STAT flag is cleared and an interrupt is generated. At this point the host can read the status register. At the end of the command, the command block register will contain the cylinder number, head number and sector number of the last sector written.

During a multi-sector write operation, in addition to the operation of the first sector, when the drive is ready to receive data for one sector from the host, DRQ_STAT is set, the BUSY_STAT flag is cleared, and an interrupt is generated. Once a sector has been transferred, the drive resets the DRQ and sets the BUSY flag. When the last sector is written to disk, the drive clears the BUSY_STAT flag and generates an interrupt (at this point DRQ_STAT has been reset). At the end of the write command, the command block register will contain the cylinder number, head number and sector number of the last sector written.

If an error occurs in a multi-sector write operation, the write operation will terminate at the sector where the error occurred. Similarly, the command block register will contain the cylinder number, head number and sector number of the error sector.

(4) 0x40 -- (WIN_VERIFY) retryable sector read verification; 0x41 -- no retry sector verification.

The execution of this command is the same as the read sector operation, but this command does not cause the drive to set DRQ_STAT and does not transfer data to the host. When the read verification command is received, the drive sets the BUSY_STAT flag. When the specified sector is verified, the drive resets the BUSY_STAT flag and generates an interrupt. At the end of the command, the command block register will contain the cylinder number, head number and sector number of the last verified sector.

If an error occurs in the multi-sector verification operation, the verification operation will terminate at the sector where the error occurred. Similarly, the command block register will contain the cylinder number, head number and sector number of the error sector.

(5) 0x50 -- (WIN_FORMAT) Format the track command.

The track address is specified in the sector count register. When the drive accepts the command, it sets the DRQ_STAT bit and then waits for the host to fill the sector buffer. When the buffer is full, the drive clears DRQ_STAT, sets the BUSY_STAT flag, and begins execution of the command.

(6) 0x60 -- (WIN_INIT) controller initialization.

(7) 0x7X -- (WIN_SEEK) seek operation.

The seek operation command moves the selected head in the command block register to the specified track. When the host issues a seek command, the drive sets the BUSY flag and generates an interrupt. The drive does not set SEEK_STAT (DSC - seek completion) until the seek operation is completed. The seek operation may not have completed before the drive generates an interrupt. If the host issues a new command to the drive while the seek operation is in progress, then BUSY_STAT will remain set until the end of the seek. Then the drive starts executing the new command.

(8) 0x90 -- (WIN_DIAGNOSE) drive diagnostic command.

This command performs the diagnostic test process implemented inside the drive. Driver 0 sets the BUSY_STAT bit within 400 ns of the command.

If the system contains a second drive, drive 1, then both drives perform diagnostic operations. Drive 0 will wait for drive 1 to perform a diagnostic operation for 5 seconds. If drive 1 diagnostics fails, drive 0 appends 0x80 to its diagnostic status. If the host detects that the diagnostic operation of the drive 1 has failed while reading the state of the drive 0, it sets the drive select bit (bit 4) of the drive/head register (0x1f6) and then reads the state of the drive 1. If the drive 1 passes the diagnostic test or the drive 1 does not exist, the drive 0 directly loads its own diagnostic status into the error register. If drive 1 does not exist, then drive 0 only reports its own diagnostic results and generates an interrupt after resetting the BUSY_STAT bit.

(9) 0x91 -- (WIN_SPECIFY) Create a drive parameter command.

This command is used to allow the host to set the head swap and sector count loop values for multi-sector operations. When the command is received, the drive sets the BUSY_STAT bit and generates an interrupt. This command uses only the values of the two registers. One is the sector count register, which is used to specify the number of sectors; the other is the driver/head register, which is used to specify the number of heads, and the drive select bit (bit 4) is set according to the specifically selected driver.

This command does not verify the selected sector count value and number of heads. If these values are invalid, the drive will not report an error until another command uses these values and invalidates an access error.

◆Hard disk control register (write) (HD_CMD, 0x3f6)

This register is write-only and is used to store the hard disk control byte and control the reset operation. Its definition is the same as the byte description at shift 0x08 of the hard disk basic parameter table, as shown in Table 9–8.

Table 9-8 The meaning of the hard disk control byte

Offset	Bit	Control Byte Description (drive step selection)
0x08	0	Not used
	1	Reserved (0) (Close IRQ)
	2	Allow reset
	3	Set if the number of heads is greater than 8
	4	Not used (0)
	5	If there is a manufacturer's bad area map at the cylinders +1, set 1
	6	Prohibit ECC retry
	7	Prohibit access retry

9.3.3.2 AT hard disk controller programming

When operating the hard disk controller, you need to send parameters and commands at the same time. The command format is shown in Table 9–9. First, you need to send a 6-byte parameter, and finally issue a 1-byte command code. No matter what command, you need to completely output the 7-byte command block, and write to port 0x1f1 -- 0x1f7 in turn. Once the command block register is loaded, the command begins execution.

Table 9-9 Hard disk controller command format

Port	Description
0x1f1	Write pre-compensation start cylinder number
0x1f2	Number of sectors

0x1f3	Starting sector number
0x1f4	Cylinder number low byte
0x1f5	Cylinder number high byte
0x1f6	Drive number / head number
0x1f7	Command code

First, the CPU outputs a control byte to the control register port (HD_CMD, 0x3f6) to establish a corresponding hard disk control mode. After the mode is established, parameters and commands can be sent in the above order. The steps are:

1. Detect controller idle state: The CPU reads the main status register. If bit 7 (BUSY_STAT) is 0, the controller is idle. If the controller is always busy within the specified time, it is judged as a timeout error. See the controller_ready() function on line 168 in hd.c.
2. Check if the drive is ready: The CPU determines if the main status register bit 6 (READY_STAT) is 1 to see if the drive is ready. If 1, the CPU can output parameters and commands. See the drive_busy() function on line 209 in hd.c.
3. Output command block: Outputs parameters and commands to the corresponding ports in sequence. See the hd_out() function starting with line 187 in hd.c.
4. CPU waits for interrupt generation: After the command is executed, the hard disk controller will generate an interrupt request signal (IRQ14 - corresponding int46) or set the controller state to idle, indicating the end of the operation or the request for sector transfer (multi-sector read/write). The function called in the program hd.c during the interrupt processing is shown in the code 237--293. There are 5 functions corresponding to 5 cases: hard disk reset, unexpected interrupt, bad read/write interrupt, read interrupt and write interrupt.
5. Detection operation result: The CPU reads the main status register again. If bit 0 is equal to 0, the command execution is successful, otherwise it fails. If it fails, you can further query the error register (HD_ERROR) to get the error code. See the win_result() function on line 176 of hd.c.

9.3.3.3 硬盘分区表

If the PC starts the operating system from the hard disk, the ROM BIOS program reads the first sector into the memory 0x7c00 after executing the machine self-test diagnostic program, and gives execution control to the code on the sector to continue executing. This particular sector is referred to as the Master Boot Record (MBR), and its content structure is shown in Table 9-10.

Table 9-10 The structure of the hard disk master boot sector MBR

Offset	Name	Size (Byte)	Description
0x000	MBR Code	446	Boot program code and data.
0x1BE	Partition table entry 1	16	The first partition table entry, a total of 16 bytes.
0x1CE	Partition table entry 2	16	The second partition table entry, 16 bytes.
0x1DE	Partition table entry 3	16	The third partition table entry, 16 bytes.
0x1EE	Partition table entry 4	16	The fourth partition table entry, 16 bytes.
0x1FE	Boot flag	2	Valid boot sector flags: 0x55, 0xAA

In addition to the initial 446-byte boot executable code, the MBR also contains a hard disk partition table with a total of four entries. The partition table is stored at the 0x1BE--0x1FD offset position of the 1st sector of the 0 cylinder 0 of the hard disk. In order to enable multiple operating systems to share hard disk resources, the hard disk can logically divide all sectors into 1--4 partitions. The sector numbers between each partition are contiguous. Each entry in the partition table has 16 bytes and is used to describe the characteristics of a partition. The size of the partition, the cylinder number, the track number and the sector number of the starting and ending are stored, as shown in Table 9–11.

Table 9-11 Hard disk partition table entry structure

Offset	Name	Size	Description
0x00	boot_ind	1 byte	Boot index. Only one partition of the 4 partitions can be booted at a time. 0x00 - Do not boot from this partition; 0x80 - Boot from this partition.
0x01	head	1 byte	Partition start head number. The head number ranges from 0 to 255.
0x02	sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the beginning of the partition.
0x03	cyl	1 byte	The lower 8 bits of the cylinder number at the starting of the partition.
0x04	sys_ind	1 byte	Partition type. 0x0b - DOS; 0x80 - Old Minix; 0x83 - Linux . . .
0x05	end_head	1 byte	The head number at the end of the partition. It ranges from 0 to 255.
0x06	end_sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the end of the partition .
0x07	end_cyl	1 byte	The lower 8 bits of the cylinder number at the end of the partition.
0x08-0x0b	start_sect	4 byte	The physical sector number at the beginning of the partition. It counts from 0 in the order of the sector number of the entire hard disk.
0x0c-0x0f	nr_sects	4 byte	The number of sectors occupied by the partition.

The fields 'head', 'sector', and 'cyl' in the table represent the head number, sector number, and cylinder number at the beginning of a partition, respectively. The range of the 'head' number ranges from 0 to 255. The lower 6 bits in the 'sector' byte field represent the sector number counted in the current cylinder. The sector number count range is 1-63. The upper 2 bits of the 'sector' field form a 10-bit cylinder number with the 'cyl' field, which ranges from 0 to -1023. Similarly, the 'end_head', 'end_sector', and 'end_cyl' fields in the table indicate the head number, sector number, and cylinder number at the end of the partition, respectively. Therefore, if we use C to indicate the cylinder number, H for the head number, and S for the sector number, the starting CHS value of the partition can be expressed as:

```
H = head;
S = sector & 0x3f;
C = (sector & 0xc0) << 2) + cyl;
```

The 'start_sect' field in the table is the 4-byte partition start physical sector number. It represents the sector number of the entire hard disk compiled from 0. The encoding method is: starting from CHS for 0 cylinder, 0 head and 1 sector (0, 0, 1), first encode the sectors in the current cylinder in order, and then encode the head from 0 to the maximum head number. Finally, the cylinder is counted. If the total number of heads of a hard disk is MAX_HD and the total number of sectors per track is MAX_SECT, then the physical sector number phy_sector of the hard disk corresponding to a CHS value is:

```
phy_sector = (C * MAX_HEAD + H) * MAX_SECT + S - 1
```

The first sector of the hard disk (0 cylinder 0 header 1 sector) has the same purpose as the first sector (boot sector) on the floppy disk except for one partition table. Only its code will move itself from 0x7c00 to 0x6000 during execution to free up space at 0x7c00, and then find out which active partition is based on the information in the partition table. Then load the first sector of the active partition to 0x7c00 to execute. A partition from which cylinder, head and sector of the hard disk are recorded in the partition table. Therefore, it is possible to know from the partition table where the first sector of an active partition (ie, the boot sector of the partition) is on the hard disk.

9.3.3.4 Relationship between absolute sector and current cylinder, sector, head.

Assume that the number of sectors per track of the hard disk is track_secs, the total number of heads is dev_heads, and the total number of tracks is tracks. For the specified sequential sector number sector, the corresponding current cylinder number is cyl, the sector number on the current track is sec, and the current head number is head. Then, if you want to convert from the specified sequential sector number to the corresponding current cylinder number, the current track sector number, and the current head number, you can use the following steps:

- sector / track_secs = The quotient is tracks, and the remainder is sec;
- tracks / dev_heads = the quotient is cyl, and the remainder is head;
- On the current track, the sector number starts from 1, so you need to increase sec by 1.

If you want to convert the specified current cyl, sec, and head to the sequential sector number from the hard disk, the process is exactly the opposite. The conversion formula is exactly the same as given above, namely:

```
sector = (cyl * dev_heads + head) * track_secs + sec -1
```

9.4 ll_rw_blk.c

9.4.1 Function Description

This program is mainly used to perform low-level block device read/write operations. It is the interface program between all block devices (hard drive, floppy drive and virtual ram disk) in this chapter and other parts of the system. By calling the program's read/write function ll_rw_block(), other programs in the system can asynchronously read and write data from the block device. The main purpose of this function is to create block device read and write request items for other programs and insert them into the specified block device request queue. The actual read and write operations are done by the request handling function request_fn() of the device. For hard disk operations, the function is do_hd_request(); for floppy operations, the function is do_fd_request(); for virtual disks it is do_rd_request().

If ll_rw_block() builds a request item for a block device and determines that the device is idle by checking that the current request item pointer of the block device is NULL, the newly created request item is set as the current request, and the request_fn() is directly invoked. Otherwise, the elevator request algorithm will be used to insert the newly created request item into the request linked list queue of the device for processing. When request_fn() ends processing, the request item is removed from the linked list.

Since request_fn() completes the processing of a request item, it will call request_fn() itself again through the interrupt callback C function (mainly read_intr() and write_intr()) to process the remaining request items in the linked list. Therefore, as long as there are unprocessed request items in the linked list (or called queues), they will be processed one after another until the linked list of request items of the device is empty. When the linked list of request items is empty, request_fn() will no longer send commands to the drive controller, but will exit immediately. Therefore, the loop call to the request_fn() function ends, as shown in Figure 9-5.

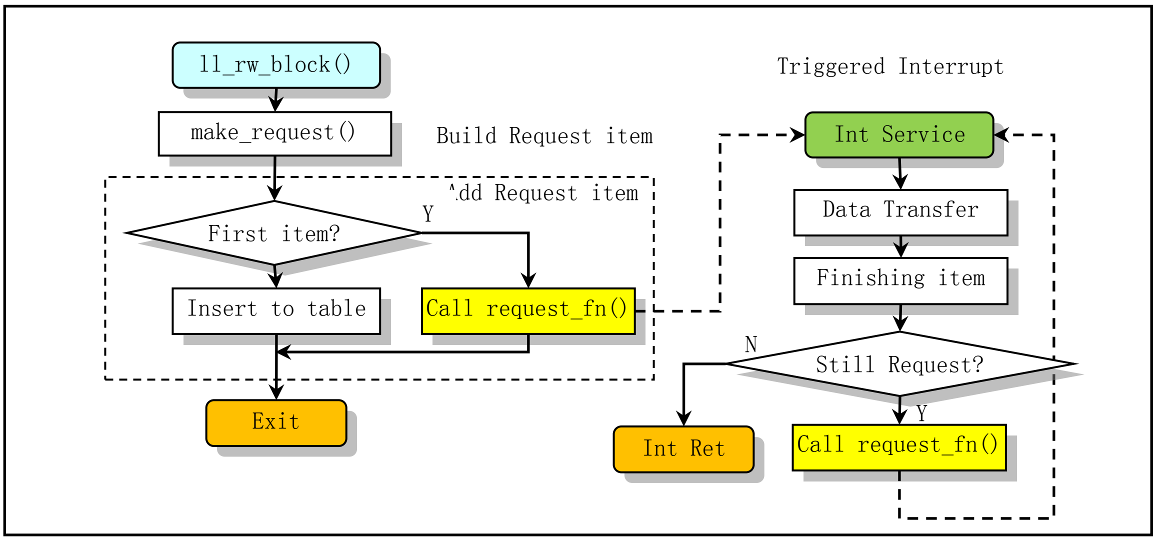


Figure 9-5 ll_rw_block call sequence

For the virtual disk device, since its read and write operations do not involve the above-mentioned synchronous operation with the external hardware device, there is no interrupt processing described above. The read and write operations of the current request item to the virtual device are completely implemented in do_rd_request().

9.4.2 Code Annotation

Program 9-3 linux/kernel/blk_drv/ll_rw_blk.c

```
/*
 *  linux/kernel/blk_dev/ll_rw.c
 *
 * (C) 1991 Linus Torvalds
 */

/*
 * This handles all read/write requests to block devices
 */
// <errno.h> Error number header file. Contains various error numbers in the system.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//     descriptors/interrupt gates, etc. is defined.
#include <errno.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/system.h>

#include "blk.h"               // request structure, linked list queue.

/*
 * The request-struct contains all necessary data
 * to load a nr of sectors into memory
 */
// request item array queue. NR_REQUEST = 32
struct request request[NR_REQUEST];

/*
 * used to wait on when there are no free requests
 */
struct task_struct * wait_for_request = NULL;

/* blk_dev_struct is:
 *      do_request-address
 *      next-request
 */
// An array of block devices. This array uses the major device number as the index. The actual
// content will be filled in at the initialization of each device driver. For example, when
// the hard disk driver is initialized (hd.c, line 378), the first statement is used to set
// the contents of blk_dev[3]. See the file blk_drv/blk.h, line 45, line 50.
struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
        { NULL, NULL },         /* no_dev */
        { NULL, NULL },         /* dev mem */
        { NULL, NULL },         /* dev fd */
        { NULL, NULL },         /* dev hd */
        { NULL, NULL },         /* dev ttyx */
        { NULL, NULL },         /* dev tty */
        { NULL, NULL }          /* dev lp */
};

/*
 * blk_size contains the size of all block-devices:
 *
 * blk_size[MAJOR][MINOR]
 *
 * if (!blk_size[MAJOR]) then no minor size checking is done.
 */
// An array of pointers to the total number of device data blocks. Each pointer item points
// to an array of the total number of blocks for the specified major device number. The total
// number of blocks in the array corresponds to the total number of data blocks owned by a
// sub-device determined by the sub-device number (1 block size = 1 KB).
int * blk_size[NR_BLK_DEV] = { NULL, NULL, };     // nr of block devices, NR_BLK_DEV = 7.

```

```
// Locks the specified buffer block.
// If the specified buffer block has been locked by another task, sleep itself (uninterruptedly
// waiting) until the task of performing the unlock explicitly wakes up the task.
static inline void lock_buffer(struct buffer_head * bh)
{
        cli();                          // disable int.
        while (bh->b_lock)              // sleeps if locked, until buffer is unlocked.
                sleep_on(&bh->b_wait);
        bh->b_lock=1;                   // locked the buffer immediately.
        sti();                          // enable int.
}

// Unlock the locked buffer.
// This function is identical to the function of the same name in the blk.h file, but the
// implementation in blk.h is used as a macro.
static inline void unlock_buffer(struct buffer_head * bh)
{
        if (!bh->b_lock)                // if not locked...
                printk("ll_rw_block.c: buffer not locked\n\r");
        bh->b_lock = 0;                 // reset lock flag.
        wake_up(&bh->b_wait);           // wake up the task waiting for this buffer.
}

/*
 * add-request adds a request to the linked list.
 * It disables interrupts so that it can muck with the
 * request-lists in peace.
 *
 * Note that swapping requests always go before other requests,
 * and are done in the order they appear.
 */
//// Add a request item to the linked list.
// The parameter dev is a pointer to the specified block device structure (blk.h, line 45),
// which has a request function pointer and a current request item pointer; req is a request
// item structure pointer with the content set.
// This function adds the already set request item req to the linked list of request of the
// specified device. If the device's current request pointer is NULL, then req can be set to
// the current request item and the device request item handler can be called immediately.
// Otherwise, the req request item is inserted into the linked list of the request item.
static void add_request(struct blk_dev_struct * dev, struct request * req)
{
        struct request * tmp;

// First, the pointers and flags of the request items provided by the parameters are further
// set.  The next request item pointer in the request is set to NULL. Disable the interrupt
// and clear the request-related buffer dirty flag.
        req->next = NULL;
        cli();
        if (req->bh)
                req->bh->b_dirt = 0;        // clear the buffer dirty flag.
// Then check if the specified device has current request item, that is, check if the device
// is busy. If the current request item (current_request) field of the specified device dev
// is NULL, it means that the device has no request item at present, this time is the first
// request item, and is the only one. Therefore, the block device current request pointer can
// be directly pointed to the request item, and the request function of the corresponding device
// is immediately executed.
        if (!(tmp = dev->current_request)) {
                dev->current_request = req;
                sti();                          // Enable int.
                (dev->request_fn)();            // runs request function, ie.do_hd_request().
                return;
        }
// If the device currently has current request item in process, the elevator algorithm is first
// used to search for the best insertion location, and then the request item is inserted into
// the request list. During the search course, if it is determined that the buffer block pointer
// to be inserted is NULL, that is, there is no buffer block, then an item needs to be found,
// which already has a buffer block available. Therefore, if the free entry buffer block header
// pointer at the current insertion position (after tmp) is not empty, this position is selected.
// Then exit the loop and insert the request item here. Finally enable the interrupt and exit
// the function. The role of the elevator algorithm is to minimize the movement distance of
// the disk head, thereby reducing hard disk access time.
//
// The following statement in the loop is used to compare the request item referred to by req
// with the existing request item in the request queue, and find out the correct position order
// in which the req is inserted into the queue. Then break the loop and insert req into the
// correct position of the queue.
        for ( ; tmp->next ; tmp=tmp->next) {
                if (!req->bh)
                        if (tmp->next->bh)
                                break;
                        else
                                continue;
                if ((IN_ORDER(tmp,req) ||                // blk.h, line 40.
                    !IN_ORDER(tmp,tmp->next)) &&
                    IN_ORDER(req,tmp->next))
                        break;
        }
        req->next=tmp->next;
        tmp->next=req;
        sti();
}

//// Create a request and insert it into the request queue.
// The parameter major is the major device number; rw is the specified command; bh is the buffer
// header pointer for storing data.
static void make_request(int major,int rw, struct buffer_head * bh)
{
        struct request * req;
        int rw_ahead;

/* WRITEA/READA is special case - it is not really needed, so if the */
/* buffer is locked, we just forget about it, else it's a normal read */
// Here the suffix 'A' character after 'READ' and 'WRITE' represents the word Ahead, indicating
// the pre-read/write block of data. This function first does some processing on the case of
// the READA/WRITEA command. These two commands discard the read/write request for the case
// where the specified buffer is in use and has been locked. Otherwise, it operates as a normal
```

```
    // READ/WRITE command. In addition, if the command given by the parameter is neither READ nor
    // WRITE, it means that the kernel program is faulty, then displays an error message and stops
    // the kernel. Note that the flag rw_ahead has been set here for the parameter whether it is
    // a prefetch/write command before modifying the command.
113         if (rw_ahead = (rw == READA || rw == WRITEA)) {
114                 if (bh->b_lock)
115                         return;
116                 if (rw == READA)
117                         rw = READ;
118                 else
119                         rw = WRITE;
120         }
121         if (rw!=READ && rw!=WRITE)
122                 panic("Bad block dev command, must be R/W/RA/WA");
123         lock_buffer(bh);
124         if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
125                 unlock_buffer(bh);
126                 return;
127         }
128 repeat:
129 /* we don't allow the write-requests to fill up the queue completely:
130  * we want some room for reads: they take precedence. The last third
131  * of the requests are only for reads.
132  */
    // Ok, now we have to generate and add read/write request items for this function. First we
    // need to find a free item (slot) in the request array to store the new request item. The search
    // process begins at the end of the request array. According to the above requirements, for
    // the read command request, we start the search directly from the end of the queue, and for
    // the write request, we can only fill in the empty entry from the queue 2/3 to the head of
    // the queue. So we start searching from the back. When the device field dev of the request
    // structure request is -1, it means that the item is idle. If none of the items are free, that
    // is, the request item array has been searched over the header, then check whether the request
    // is read/write in advance (READA or WRITEA), and if so, discard the request operation.
    // Otherwise, let the request operation sleep first (to wait for the queue to free up the empty
    // item), and then search the request queue after a while.
133         if (rw == READ)
134                 req = request+NR_REQUEST;           // for read, search from the end.
135         else
136                 req = request+((NR_REQUEST*2)/3);  // for write, search from 2/3 backward.
137 /* find an empty request */
138         while (--req >= request)
139                 if (req->dev<0)
140                         break;
141 /* if none found, sleep on new requests: check for rw_ahead */
142         if (req < request) {                       // no free item ...
143                 if (rw_ahead) {                    // exit if it's read/write ahead.
144                         unlock_buffer(bh);
145                         return;
146                 }
147                 sleep_on(&wait_for_request);
148                 goto repeat;                       //  line 128.
149         }
150 /* fill up the request-info, and add it to the queue */
```

```
// OK, we have found an idle request. So after we set the new request, we call add_request()
// to add it to the request queue and exit immediately. See blk_drv/blk.h for the request
// structure, line 23. Where req->sector is the starting sector number of the read/write
// operation, and req->buffer is the buffer in which the request item stores data.
        req->dev = bh->b_dev;             // device no.
        req->cmd = rw;                    // command (READ/WRITE).
        req->errors=0;                    // error count.
        req->sector = bh->b_blocknr<<1;   // start sector,(1block = 2 sectors).
        req->nr_sectors = 2;              // number of sectors read/written.
        req->buffer = bh->b_data;         // data buffer location.
        req->waiting = NULL;              // waiting list for the operation.
        req->bh = bh;                     // buffer header.
        req->next = NULL;                 // points to next request.
        add_request(major+blk_dev,req);   // add_request(blk_dev[major],req).
}

//// Low-level page read and write function (Low-Level Read Write Page).
// The block device data is accessed in units of pages (4K), that is, 8 sectors are read/written
// each time. See the ll_rw_blk() function below.
void ll_rw_page(int rw, int dev, int page, char * buffer)
{
        struct request * req;
        unsigned int major = MAJOR(dev);

// First check the legality of the function parameters. If the device's major number does not
// exist or the device's request handling function does not exist, an error message is displayed
// and returned. If the command given by the parameter is neither READ nor WRITE, it means that
// the kernel program is faulty, displays an error message and stops the kernel.
        if (major >= NR_BLK_DEV || !(blk_dev[major].request_fn)) {
                printk("Trying to read nonexistent block-device\n\r");
                return;
        }
        if (rw!=READ && rw!=WRITE)
                panic("Bad block dev command, must be R/W");
// After the parameter check is complete, we now need to create a request for this operation.
// First we need to find a free item (slot) in the request array to hold the new request item.
// The search action begins at the end of the request array. So we started searching from the
// back. When the device field dev of the request structure is less than 0, it means that the
// item is idle. If none of the items are idle, let the request operation sleep first (to wait
// for the idle item), and then search the request queue after a while.
repeat:
        req = request+NR_REQUEST;                 // points to the end.
        while (--req >= request)
                if (req->dev<0)
                        break;
        if (req < request) {
                sleep_on(&wait_for_request);      //  sleep at wait_for_request.
                goto repeat;
        }
/* fill up the request-info, and add it to the queue */
// OK, we have found an idle request. So we set up the new request item, put the current process
// into an uninterruptible sleep state, then call add_request() to add it to the request queue,
// and then directly call the scheduler to let the current process sleep, waiting page read
// from the switching device. Here, instead of exiting the function directly like the
// make_request(), schedule() is called here. This is because the make_request() only reads
// 2 sectors, but here it takes 8 sectors to read/write to the switching device, which takes
// a long time. So the current process definitely needs to wait and sleep. So let the process
// go to sleep directly, saving the need to perform these judgments elsewhere in the program.
        req->dev = dev;                         // device no.
        req->cmd = rw;                          // command.
        req->errors = 0;                        // error count.
        req->sector = page<<3;                  // starting sector.
        req->nr_sectors = 8;                    // nr of sectors read/written.
        req->buffer = buffer;                   // data buffer.
        req->waiting = current;                 // waiting queue.
        req->bh = NULL;                         // buffer header.
        req->next = NULL;                       // points to next request.
        current->state = TASK_UNINTERRUPTIBLE;
        add_request(major+blk_dev,req);         // add to request queue.
        schedule();
}

//// Low-level data block read and write functions (Low Level Read Write Block).
// This function is an interface between the block device driver and the rest of the system.
// Usually called in the fs/buffer.c program. Its main purpose is to create block device read
// and write request items and insert them into the specified block device request queue. The
// actual read and write operations are done by the device's request_fn() function, and for
// hard disk, the function is do_hd_request(); for floppy the function is do_fd_request(); for
// virtual disks it is do_rd_request(). In addition, before calling this function, the caller
// needs to first save the information of the read/write block device in the buffer block header
// structure, such as the device number and block number. Parameters: rw - is the command READ,
// READA, WRITE or WRITEA; bh - data buffer block header pointer.
void ll_rw_block(int rw, struct buffer_head * bh)
{
        unsigned int major;             // major device no ( 3 for hard disk).

// If the device's major number does not exist or the device's request action function does
// not exist, an error message is displayed and returned.  Otherwise create a request and insert
// into the request queue.
        if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
        !(blk_dev[major].request_fn)) {
                printk("Trying to read nonexistent block-device\n\r");
                return;
        }
        make_request(major,rw,bh);
}

//// The block device initialization function, called by the initialization program main.c.
// Initializes the request array and sets all request items as free items (dev = -1). There
// are 32 items (NR_REQUEST = 32).
void blk_dev_init(void)
{
        int i;

        for (i=0 ; i<NR_REQUEST ; i++) {
                request[i].dev = -1;
                request[i].next = NULL;
        }
}

```

9.5 ramdisk.c

9.5.1 Function

This file is a virtual Ram Disk driver created by Theodore Ts'o. A ram disk device is a way to use physical memory to simulate the actual disk storage data. Its purpose is mainly to improve the speed of reading and writing of "disk" data. In addition to taking up some valuable memory resources, the main disadvantage is that once the system is shut down or crashes, all the data in the virtual disk will disappear. Therefore, the virtual disk usually only stores some common tool programs or temporary data such as system commands, rather than important input documents.

When the symbol RAMDISK is defined in the linux/Makefile, the kernel initializer draws a memory area of a specified size in memory for the virtual disk data. The virtual ram disk capacity is equal to the value of RAMDISK (KB). If RAMDISK=512, the virtual disk size is 512KB. The specific location of the virtual disk in physical memory is determined during the kernel initialization phase (init/main.c, line 150), which is located between the kernel cache and the main memory area. If the running machine contains 16MB of physical memory, the kernel code will set the virtual ram disk area at the beginning of 4MB of memory. At this time, the memory allocation is shown in Figure 9-6.

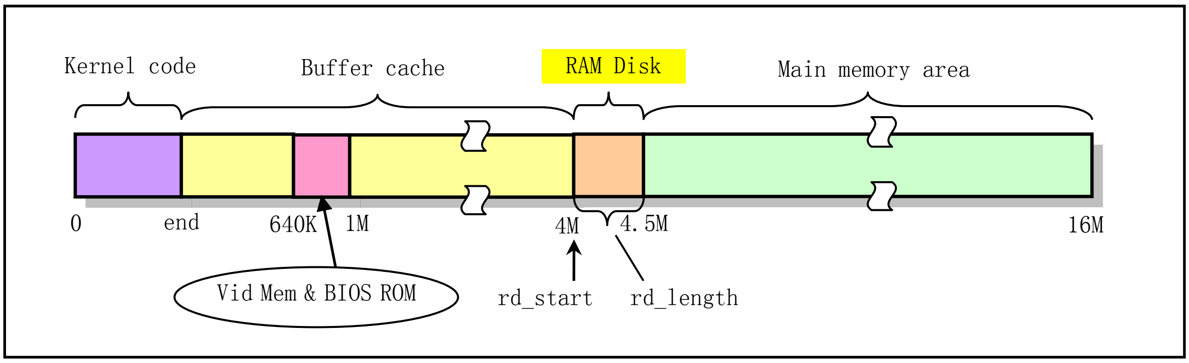


Figure 9-6 The specific location of the ram disk in the 16MB memory system

The read and write access operations to the ram disk are in principle the same as those for ordinary disks, and they need to be operated according to the access mode of the block device. Since the implementation does not involve synchronization with an external controller or device, its implementation is relatively simple. For data transfer between the system and the device, it is only necessary to perform an in-memory block copy operation.

The ramdisk.c file contains 3 functions. The rd_init() function is called by the init/main.c program during system initialization to determine the specific location and size of the virtual disk in physical memory; do_rd_request() is the request function of the virtual disk device to implement virtual disk data access operation; rd_load() is a virtual disk root file system load function. During the system initialization phase, the rd_load() function is used to attempt to load a root file system into the virtual disk from the disk block location specified on the boot disk. In this function, this starting boot disk block location is set to 256. Of course, you can also modify this value according to your specific requirements, as long as the disk capacity specified by this value can accommodate the kernel image file. In this way, a "two-in-one" disk composed of a kernel boot image file (Bootimage) plus a root file system image file (Rootiamge) can boot a Linux system just like a DOS system disk. The experimental creation of this type of combination disk (integrated disk) can be found in Chapter 17.

Before the root file system image is loaded from disk using the normal method, the system first executes the rd_load() function, trying to read the root file system super block from block 257 of the disk. If successful, the root file image file is read into the memory virtual disk, and the root file system device flag ROOT_DEV is set to the virtual disk device (0x0101). Otherwise, rd_load() is exited, and the system continues to load the root file system from other devices in the normal way. The operation process of loading the root file system to the ram disk is shown in Figure 9-7.

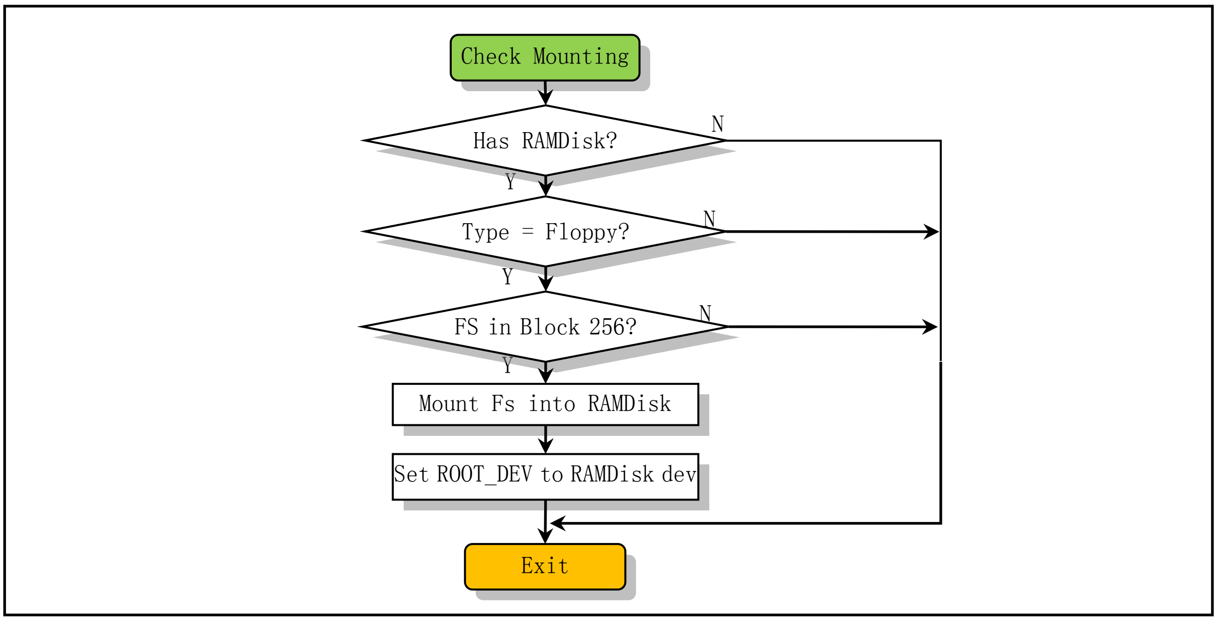


Figure 9-7 Flowchart for loading the root file system to the ram disk

If the symbol RAMDISK and its size are defined in the linux/Makefile configuration file when compiling the Linux 0.12 kernel source code, then after booting and initializing the RAMDISK area, it will first try to check the location at the 256th disk block on the disk, Is there a root file system? The detection method is to determine whether there is a valid file system super block in the 257th disk block. If so, the file system is loaded into the RAMDISK area in memory and used as the root file system. So we can use a boot disk that integrates the root file system to boot the system to the shell command prompt. If a valid root file system is not stored at the specified disk block location (256th disk block) on the boot disk, the kernel will prompt to insert the root file system disk. After the user presses the Enter key to confirm, the kernel will read the root file system on a separate disk into the virtual disk area for execution.

On a 1.44MB kernel boot boot disk, put a basic root file system at the beginning of the 256th block on the disk, you can combine to form an integrated disk, the layout is shown in Figure 9-8.

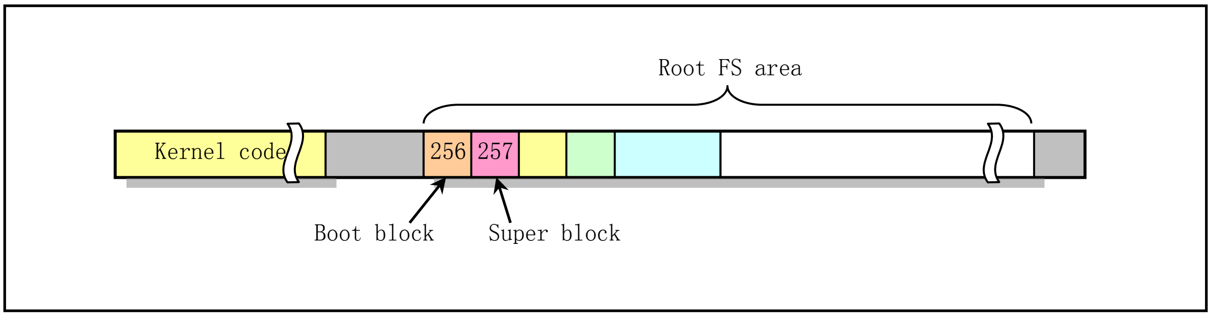


Figure 9-8 Data block layout on the integrated disk

9.5.2 Code Annotation

Program 9-4 linux/kernel/blk_drv/ramdisk.c

```
/*
 *  linux/kernel/blk_drv/ramdisk.c
 *
 *  Written by Theodore Ts'o, 12/2/91
 */
// Theodore Ts'o (Ted Ts'o) is a famous figure in the Linux community. The popularity of Linux
// in the world also has his great contribution. As early as the Linux operating system came
// out, he provided a maillist service for the development of Linux with great enthusiasm, and
// established a Linux ftp server site (tsx-11.mit.edu) in North America. One of his biggest
// contributions to Linux was to propose and implement the ext2 file system. This file system
// has become the de facto file system standard in the Linux world. Recently, he introduced
// the ext3 and ext4 file systems, which greatly improved the stability, recoverability and
// access efficiency of the file system. To pay tribute to him, the Linux Journal's 97th issue
// (May 2002) interviewed him and used him as a cover person. Now he is a staff engineer working
// at Google, where he still work on file system and storage. His homepage:thunk.org/tytso/

// <string.h> String header file. Defines some embedded functions about string operations.
// <linux/config.h> Kernel configuration header file. Define keyboard language and hard disk
//     type (HD_TYPE) options.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/fs.h> File system header file. Define the file table structure (file, buffer_head,
//     m_inode, etc.).
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//     descriptors/interrupt gates, etc. is defined.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//     segment register operations.
// <asm/memory.h> Memory copy header file. Contains memcpy() embedded assembly macro functions.
#include <string.h>

#include <linux/config.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/kernel.h>
```

```
13 #include <asm/system.h>
14 #include <asm/segment.h>
15 #include <asm/memory.h>
16
   // Define the RAM disk major number symbol constant. The major device number must be defined
   // in the driver before the blk.h file is included, as this symbolic constant value is used
   // in the blk.h file to determine a range of other constant symbols and macros.
17 #define MAJOR_NR 1
18 #include "blk.h"
19
   // The starting position of the virtual disk in memory, which is determined in the initialization
   // function rd_init() on line 52. See kernel initialization program init/main.c, line 151.
20 char    *rd_start;                  // the starting address of the ram disk in memory.
21 int     rd_length = 0;              // memory size (in bytes) occupied by the ram disk.
22
   // The ram disk current request operation function.
   // The structure of this function is similar to the do_hd_request() of the hard disk driver
   // (see hd.c, line 330). After the low-level block device interface function ll_rw_block()
   // establishes the request item of the ram disk (rd) and adds it to the linked list of rd, this
   // function is called to process the current request item of rd. The function first calculates
   // the starting addr of the memory corresponding to the ram disk of the starting sector specified
   // in the current request item and the bytes length len corresponding to the required number
   // of sectors, and then operates according to the command in the request item. If the command
   // is WRITE, the data in the buffer pointed to by the request is directly copied to the memory
   // location addr. If it is a read operation, the reverse is true. After the data is copied,
   // you can directly call end_request() to end the request. Then jump to the beginning of the
   // function and then process the next request item. Exit if there is no request left.
23 void do_rd_request(void)
24 {
25         int     len;
26         char    *addr;
27
   // First check the legality of the request item, and exit if there is no request item (see blk.h,
   // line 148). Then calculate the address addr corresponding to the starting sector in the virtual
   // disk and the occupied memory byte length len. The following sentence is used to obtain the
   // memory start position and memory length corresponding to the starting sector in the request
   // item, where 'sector << 9 represents sector * 512', which is converted into bytes. CURRENT
   // is defined as '(blk_dev[MAJOR_NR].current_request)' in blk.h.
28         INIT_REQUEST;
29         addr = rd_start + (CURRENT->sector << 9);
30         len = CURRENT->nr_sectors << 9;
   // If the minor-device number in the current request is not 1 or the corresponding memory start
   // position is greater than the end of the ram disk, the request item is ended, and the jump
   // to the repeat is performed to process the next virtual disk request item. The label 'repeat'
   // is defined in the macro INIT_REQUEST, see line 149 of the blk.h file.
31         if ((MINOR(CURRENT->dev) != 1) || (addr+len > rd_start+rd_length)) {
32                 end_request(0);
33                 goto repeat;
34         }
   // Then perform the actual read and write operations. If it is a write command (WRITE), the
   // contents of the buffer in the request are copied to the address 'addr', and the length is
   // 'len' bytes. If it is a read command (READ), the memory content started by 'addr' is copied
   // into the request item buffer, and the length is 'len' bytes. Otherwise it will print that
   // the command does not exist and crashes.
35         if (CURRENT-> cmd == WRITE) {
36                 (void ) memcpy(addr,
37                               CURRENT->buffer,
38                               len);
39         } else if (CURRENT->cmd == READ) {
40                 (void) memcpy(CURRENT->buffer,
41                               addr,
42                               len);
43         } else
44                 panic("unknown ramdisk-command");
   // Then, after the request item is successfully processed, the update flag is set, and the next
   // request item of the device is processed.
45         end_request(1);
46         goto repeat;
47 }
48
49 /*
50  * Returns amount of memory which needs to be reserved.
51  */
   // Virtual ram disk initialization function.
   // The function first sets the request handler pointer of the virtual disk to point to
   // do_rd_request(), then determines the starting address, byte length of the virtual disk in
   // physical memory, and clears the entire virtual extent. Finally return the length of the ram
   // disk. When the RAMDISK value is set to non-zero in the linux/Makefile, it means that the
   // virtual ram disk device will be created in the system. In this case, the kernel initialization
   // process calls this function (init/main.c, line 151). The second parameter 'length' is assigned
   // to RAMDISK * 1024 in bytes.
52 long rd_init(long mem_start, int length)
53 {
54         int     i;
55         char    *cp;
56
57         blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;    // do_rd_request()
58         rd_start = (char *) mem_start;                    // 4MB for the 16MB machine.
59         rd_length = length;                               // size of the ram disk.
60         cp = rd_start;
61         for (i=0; i < length; i++)                        // cleared.
62                 *cp++ = '\0';
63         return(length);
64 }
65
66 /*
67  * If the root device is the ram disk, try to load it.
68  * In order to do this, the root device is originally set to the
69  * floppy, and we later change it to be ram disk.
70  */
   //// Try to load the root file system into the ram disk.
   // This function will be called in the kernel setup function setup() (hd.c, line 162). The
   // variable 'block=256' on line 75 indicates that the root file system image file is located
   // at the beginning of the 256th disk block on the boot disk. (1 disk block = 1024 bytes).
71 void rd_load(void)
72 {
        struct buffer_head *bh;         // cache buffer head pointer.
        struct super_block      s;
        int             block = 256;    /* Start at block 256 */
        int             i = 1;
        int             nblocks;        // The amountof file system disk blocks.
        char            *cp;            /* Move pointer */

// First check the validity and integrity of the ram disk. If the length of the ramdisk is zero,
// then exit. Otherwise, the size of the ramdisk and the starting position of the memory are
// displayed. If the root file device is not a floppy device at this time, it also exits.
        if (!rd_length)
                return;
        printk("Ram disk: %d bytes, starting at 0x%x\n", rd_length,
                (int) rd_start);
        if (MAJOR(ROOT_DEV) != 2)
                return;
// Then read the basic parameters of the root file system, that is, read the floppy disk blocks
// 256+1, 256 and 256+2. Here block+1 refers to the super block of the root file system in the
// virtual disk. The function breada() is used to read the specified data block from the floppy
// disk, mark the block that still needs to be read, and then return the buffer pointer containing
// the data block (fs/buffer.c, line 322). Then copy the disk superblock in the buffer to the
// s variable (d_super_block is the superblock structure) and release the buffer. Then we begin
// to check the validity of the super block. If the fs magic number in the super block is
// incorrect, it means that the loaded data block is not the MINIX file system, so it exits.
// See the File System chapter for the structure of the MINIX superblock.
        bh = breada(ROOT_DEV,block+1,block,block+2,-1);
        if (!bh) {
                printk("Disk error while looking for ramdisk!\n");
                return;
        }
        *((struct d_super_block *) &s) = *((struct d_super_block *) bh->b_data);
        brelse(bh);
        if (s.s_magic != SUPER_MAGIC)
                /* No ram disk image present, assume normal floppy boot */
                return;
// Then we try to read the entire root file system into the memory virtual disk extent. For
// a file system, the amount of logical blocks (or number of zones) is stored in the s_nzones
// field of the superblock structure. The number of data blocks contained in a logical block
// is specified by the field s_log_zone_size. Therefore, the total number of data blocks nblocks
// in a file system is equal to (the amount of logical blocks * 2^ (the power of each block)),
// that is, nblocks = (s_nzones * 2^s_log_zone_size). If the amount of data blocks in the file
// system is greater than the number of blocks that the ram disk can hold, the load operation
// cannot be performed, but only the error message is displayed and returned.
        nblocks = s.s_nzones << s.s_log_zone_size;
        if (nblocks > (rd_length >> BLOCK_SIZE_BITS)) {
                printk("Ram disk image too big!  (%d blocks, %d avail)\n",
                        nblocks, rd_length >> BLOCK_SIZE_BITS);
                return;
        }
// Otherwise, if the virtual disk can hold the total number of file system blocks, we display
// the load block information and let 'cp' point to the beginning of the virtual disk in memory.
// Then start the loop operation to load the root file system image file on the floppy disk
// to the ram disk. In the course of operation, if the number of disk blocks that need to be
// loaded at one time is greater than 2, we use the advanced read-ahead function breada(),
// otherwise we use the bread() function for single-block reading. If an I/O error occurs during
// the disk reading process, you can only abandon the loading process and return. The read disk
// block is copied from the cache to the corresponding location of the memory virtual disk using
// the memcpy() function, and the number of loaded blocks is displayed. The octal number '\010'
// in the display string indicates that a tab is displayed.
        printk("Loading %d bytes into ram disk... 0000k",
                nblocks << BLOCK_SIZE_BITS);
        cp = rd_start;
        while (nblocks) {
                if (nblocks > 2)                             // need read ahead ?
                        bh = breada(ROOT_DEV, block, block+1, block+2, -1);
                else                                         // read one block each time.
                        bh = bread(ROOT_DEV, block);
                if (!bh) {
                        printk("I/O error on block %d, aborting load\n",
                                block);
                        return;
                }
                (void) memcpy(cp, bh->b_data, BLOCK_SIZE);   // copy to location cp.
                brelse(bh);
                printk("\010\010\010\010\010%4dk",i);        // nr of blocks loaded.
                cp += BLOCK_SIZE;                            // next disk block.
                block++;
                nblocks--;
                i++;
        }
// When the entire root file system starting from the 256 disk block in the boot disk is loaded,
// we display "done", and the current root fs device number is changed to the virtual disk
// device number 0x0101, and finally returned.
        printk("\010\010\010\010\010done \n");
        ROOT_DEV=0x0101;
}

```

9.6 floppy.c

9.6.1 Function description

This program is a floppy disk controller driver. Like other block device drivers, the program also uses the request item operation function (do_fd_request() for the floppy disk drive) to perform read and write operations on the floppy disk. The main difference from the hard disk driver is that the floppy disk driver uses more timing functions and operations.

Considering that the floppy disk drive does not normally rotate when it is not working, we need to wait for the motor of the drive to start and reach the normal operating speed before the actual floppy disk can be read or written. Compared to the speed of the computer, this period of time is very long, usually takes about 0.5 seconds. In addition, when reading and writing to a disk is completed, we also need to stop the drive motor to reduce the head's friction on the disk surface. But we can't stop it after the disk is finished, because it may need to be read and written right away. Therefore, after a drive has not been operated, it is necessary to idle the drive motor for a period of time to wait for possible read and write operations. If the drive does not operate for a long time, the program stops it from rotating. The time to maintain the rotation can be set to about 3 seconds. Furthermore, when a disk read/write operation fails, or some other condition causes a drive's motor to not be turned off, we also need to have the system automatically turn it off after a certain amount of time. The Linux kernel sets this delay value to 100 seconds.

It can be seen that many delay (timer) operations are used when operating the floppy disk drive, so more timing processing functions are involved in the driver. There are also several functions that are closely related to the timer are placed in kernel/sched.c (lines 215-281). This is the biggest difference between a floppy disk driver and a hard disk driver, and it is also a reason why a floppy disk driver is more complicated than a hard disk driver.

Although the program is more complicated, the working principle of floppy disk read/write operations is the same as other block devices. The program also uses the request item and the linked list structure of the request to handle all read/write operations to the floppy disk, so the request item function do_fd_request() is still one of the important functions in the program. This function can be expanded as a main line while reading. In addition, the programming and operation of the floppy disk controller is complicated, involving many controller execution states and flags. Therefore, you need to refer to the instructions behind the program and the header file include/linux/fdreg.h. This header file defines all floppy controller parameter constants and explains the meaning of these constants.

9.6.2 Code Annotation

Program 9-5 linux/kernel/blk_drv/floppy.c

```
/*
 *  linux/kernel/floppy.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 02.12.91 - Changed to static variables to indicate need for reset
 * and recalibrate. This makes some things easier (output_byte reset
 * checking etc), and means less interrupt jumping in case of errors,
 * so the code is hopefully easier to understand.
 */

/*
 * This file is certainly a mess. I've tried my best to get it working,
 * but I don't like programming floppies, and I have only one anyway.
 * Urgel. I should check for more errors, and do more graceful error
 * recovery. Seems there are problems with several drives. I've tried to
 * correct them. No promises.
 */

/*
 * As with hd.c, all routines within this file can (and will) be called
 * by interrupts, so extreme caution is needed. A hardware interrupt
 * handler may not sleep, or a kernel panic will happen. Thus I cannot
 * call "floppy-on" directly, but have to set a special timer interrupt
 * etc.
 *
 * Also, I'm not certain this works on more than 1 floppy. Bugs may
 * abund.
 */

  // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
  //     data of the initial task 0, and some embedded assembly function macro statements
  //     about the descriptor parameter settings and acquisition.
  // <linux/fs.h> File system header file. Define the file table structure (file,
  //     buffer_head, m_inode, etc.).
  // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
  //     commonly used functions of the kernel.
  // <linux/fdreg.h> Floppy disk file. Contains some definitions of floppy disk controller
  //     parameters.
  // <asm/system.h> System header file. An embedded assembly macro that defines or
  //     modifies descriptors/interrupt gates, etc. is defined.
  // <asm/io.h> Io header file. Defines the function that operates on the io port in the
  //     form of a macro's embedded assembler.
  // <asm/segment.h> Segment operation header file. An embedded assembly function is defined
  //     for segment register operations.
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/fdreg.h>
#include <asm/system.h>
#include <asm/io.h>
#include <asm/segment.h>

  // Define the floppy drive major device number symbol. In the driver, the major device number
  // must be defined before the blk.h file is included, because this symbolic constant is used
  // in the blk.h file to determine some other related symbol constants and macros.
#define MAJOR_NR 2                   // floppy drive major number.
#include "blk.h"                     // block dev header file. requests, queues are defined.

static int recalibrate = 0;          // flag: recalibrate head position (return to zero).
static int reset = 0;                // flag: reset operation needed.
static int seek = 0;                 // flag: perform a seek operation.

  // The current digital output register (DOR), defined in kernel/sched.c, line 223, the default
  // value is 0x0C. This variable contains important flags from the floppy drive operation,
  // including select floppy drive, control motor start, start reset floppy disk controller, and
  // enable/disable DMA and interrupt requests. See the description of the DOR register after
  // the program listing.
extern unsigned char current_DOR;

  // Byte direct output (inline assembly macro). Output the value 'val' to the port.
#define immoutb_p(val,port) \
__asm__("outb %0,%1\n\tjmp 1f\n1:\tjmp 1f\n1:"::"a" ((char) (val)),"i" (port))

  // These two macros are defined to calculate the device number of the floppy drive. The parameter
  // x is the minor device number. minor device number = TYPE*4 + DRIVE. The calculation method
  // is shown after the program listing.
```

```
#define TYPE(x) ((x)>>2)           // Type of floppy drive (2--1.2Mb, 7--1.44Mb).
#define DRIVE(x) ((x)&0x03)        // The floppy drive number (0--3 corresponds to A--D).
/*
 * Note that MAX_ERRORS=8 doesn't imply that we retry every bad read
 * max 8 times - some types of errors increase the errorcount by 2,
 * so we might actually retry only 5-6 times before giving up.
 */
#define MAX_ERRORS 8

/*
 * globals used by 'result()'
 */
// See the include/linux/fdreg.h header file for the meaning of the bits in these status bytes.
// See also the instructions at the end of the program list.
#define MAX_REPLIES 7                                // FDC returns up to 7 bytes of results.
static unsigned char reply_buffer[MAX_REPLIES];  // used to store the response results.
#define ST0 (reply_buffer[0])                        // result status byte 0.
#define ST1 (reply_buffer[1])                        // result status byte 1.
#define ST2 (reply_buffer[2])                        // result status byte 2.
#define ST3 (reply_buffer[3])                        // result status byte 3.

/*
 * This struct defines the different floppy types. Unlike minix
 * linux doesn't have a "search for right type"-type, as the code
 * for that is convoluted and weird. I've got enough problems with
 * this driver as it is.
 *
 * The 'stretch' tells if the tracks need to be boubled for some
 * types (ie 360kB diskette in 1.2MB drive etc). Others should
 * be self-explanatory.
 */
// Define the floppy drvie data structure. The floppy disk parameters are:
// size           number of sectors;
// sect           sectors per track;
// head           number of heads;
// track          number of tracks;
// stretch        flag, if the tracks need to be handled specifically;
// gap            Sector gap length (bytes);
// rate           Data transfer rate;
// spec1          Parameters (high 4-bit step rate, low 4-bit head unloading time).
static struct floppy_struct {
        unsigned int size, sect, head, track, stretch;
        unsigned char gap,rate,spec1;
} floppy_type[] = {
        {    0, 0,0, 0,0,0x00,0x00,0x00 },      /* no testing */
        {  720, 9,2,40,0,0x2A,0x02,0xDF },      /* 360kB PC diskettes */
        { 2400,15,2,80,0,0x1B,0x00,0xDF },      /* 1.2 MB AT-diskettes */
        {  720, 9,2,40,1,0x2A,0x02,0xDF },      /* 360kB in 720kB drive */
        { 1440, 9,2,80,0,0x2A,0x02,0xDF },      /* 3.5" 720kB diskette */
        {  720, 9,2,40,1,0x23,0x01,0xDF },      /* 360kB in 1.2MB drive */
        { 1440, 9,2,80,0,0x23,0x01,0xDF },      /* 720kB in 1.2MB drive */
        { 2880,18,2,80,0,0x1B,0x00,0xCF },      /* 1.44MB diskette */
};
```

```
95
96 /*
97  * Rate is 0 for 500kb/s, 2 for 300kbps, 1 for 250kbps
98  * Spec1 is 0xSH, where S is stepping rate (F=1ms, E=2ms, D=3ms etc),
99  * H is head unload time (1=16ms, 2=32ms, etc)
100  *
101  * Spec2 is (HLD<<1 | ND), where HLD is head load time (1=2ms, 2=4 ms etc)
102  * and ND is set means no DMA. Hardcoded to 6 (HLD=6ms, use DMA).
103  */
104
    // floppy_interrupt is the floppy drive interrupt handler label in the kernel/sys_call.s
    // program. It will be used in the floppy disk initialization function floppy_init() (line 469)
    // to initialize the interrupt trap gate descriptor.
105 extern void floppy_interrupt(void);
    // This is the temporary floppy buffer defined at line 132 of boot/head.s. If the buffer of
    // the request item is somewhere above 1MB in memory, you need to set the DMA buffer at the
    // temporary buffer area. Because the 8237A chip can only be addressed within the 1MB address
    // range.
106 extern char tmp_floppy_area[1024];
107
108 /*
109  * These are global variables, as that's the easiest way to give
110  * information to interrupts. They are the data used for the current
111  * request.
112  */
    // These so-called "global variables" refer to the variables used by the C functions called
    // in the floppy disk interrupt handler. Of course, these C functions are all within the program.
113 static int cur_spec1 = -1;                         // current spec1.
114 static int cur_rate = -1;
115 static struct floppy_struct * floppy = floppy_type;  // floppy point to the floppy_type[].
116 static unsigned char current_drive = 0;
117 static unsigned char sector = 0;
118 static unsigned char head = 0;
119 static unsigned char track = 0;
120 static unsigned char seek_track = 0;
121 static unsigned char current_track = 255;
122 static unsigned char command = 0;                  // read/write command.
123 unsigned char selected = 0;                        // drive selected flag.
124 struct task_struct * wait_on_floppy_select = NULL;   // wait floppy queue.
125
    //// Deselect a floppy drive.
    // If the floppy drive nr specified by the function parameter is not currently selected, a warning
    // message is displayed. Then reset the floppy drive selected flag and wake up the task waiting
    // to select the floppy drive. The lower 2 bits of the Digital Output Register (DOR) are used
    // to specify the selected floppy drive (0-3 to A-D).
126 void floppy_deselect(unsigned int nr)
127 {
128         if (nr != (current_DOR & 3))
129                 printk("floppy_deselect: drive not selected\n\r");
130         selected = 0;
131         wake_up(&wait_on_floppy_select);
132 }
133
```

```
/*
 * floppy-change is never called from an interrupt, so we can relax a bit
 * here, sleep etc. Note that floppy-on tries to set current_DOR to point
 * to the desired drive, but it will probably not survive the sleep if
 * several floppies are used at the same time: thus the loop.
 */
//// Check the floppy disk replacement in the specified floppy drive.
// The parameter 'nr' is the floppy drive number. Returns 1 if the floppy disk is replaced,
// otherwise returns 0. The function first selects the specified floppy drive 'nr' and then
// tests the controller's digital input register (DIR) to determine if the floppy disk in the
// drive has been replaced. This function is called by the check_disk_change() in the program
// fs/buffer.c (line 119).
int floppy_change(unsigned int nr)
{
// First, let the floppy disk in the floppy drive spin up and reach the normal working speed.
// This takes a certain amount of time. The method is to use the floppy timer function
// do_floppy_timer () (kernel / sched.c, line 264) for a certain delay processing. The
// floppy_on() function (sched.c, line 251) is used to determine if the delay has expired
// (mon_timer[nr]==0?). If not, let the current process continue to sleep. If the delay has
// expired, do_floppy_timer() will wake up the current process.
repeat:
        floppy_on(nr);               // Start and wait for the specified floppy drive nr.
// After the floppy disk is started (rotated), let's check to see if the currently selected
// floppy drive is drive 'nr' specified by the function parameter. If the currently selected
// floppy drive is not the specified floppy drive nr and other floppy drives have been selected,
// then the current task is put into an uninterruptible wait state, waiting for other floppy
// drives to be deselected, see floppy_deselect() above. If no other floppy drive is currently
// selected, or if the other floppy drive is deselected and the current floppy drive is still
// not the specified floppy drive nr when the current task is woken up, then jump to the beginning
// of the function and re-circulate.
        while ((current_DOR & 3) != nr && selected)
                sleep_on(&wait_on_floppy_select);
        if ((current_DOR & 3) != nr)
                goto repeat;
// Now the floppy controller has selected the floppy drive 'nr' we specified. The value of the
// digital input register DIR is then taken. If its highest bit (bit 7) is set, it means the
// floppy disk has been replaced, then the motor can be turned off and 1 will exit. Otherwise,
// the motor is turned off and 0 is exited, indicating that the disk has not been replaced.
        if (inb(FD_DIR) & 0x80) {
                floppy_off(nr);
                return 1;
        }
        floppy_off(nr);
        return 0;
}

////Copy 1024 bytes of data from the memory address 'from' to the address 'to'.
#define copy_buffer(from,to) \
__asm__("cld ; rep ; movsl" \
        ::"c" (BLOCK_SIZE/4),"S" ((long)(from)),"D" ((long)(to)) \
        :"cx","di","si")

//// Setup (initialize) the floppy disk DMA channel.
```

```
// Data access operations on floppy disk are performed using DMA. Therefore, it is necessary
// to set the channel 2 dedicated to floppy drive on the DMA chip before each data transmission.
// See the information after the program list for the DMA programming method.
static void setup_DMA(void)
{
        long addr = (long) CURRENT->buffer;      // current request buffer address.

// First check the location of the buffer for the request item. If the buffer is somewhere above
// 1MB in memory, you need to set the DMA buffer in the temporary buffer area (tmp_floppy_area).
// Because the 8237A chip can only be addressed within the 1MB address range. If it is a write
// disk command, you also need to copy the data from the request item buffer to the temporary
// area.
        cli();
        if (addr >= 0x100000) {
                addr = (long) tmp_floppy_area;
                if (command == FD_WRITE)
                        copy_buffer(CURRENT->buffer,tmp_floppy_area);
        }
// Next we start to set up DMA channel 2, but you need to mask the channel before you start
// setting up. The single channel mask register port is 10. Bits 0-1 specify the DMA channel
// (0--3), bits 2:1 indicate masking, and 0 indicates that the request is allowed. The mode
// word is then written to DMA controller ports 12 and 11 (read is 0x46 and write is 0x4A).
// Write the buffer address 'addr' and the number of bytes to be transferred 0x3ff (0--1023).
// Finally, the mask of DMA channel 2 is reset, and the DREQ signal requested by DMA2 is opened.
/* mask DMA 2 */
        immoutb_p(4|2,10);              // port 10
/* output command byte. I don't know why, but everyone (minix, */
/* sanches & canton) output this twice, first to 12 then to 11 */
// The following inline assembly code writes the mode word to the "clear sequence trigger" port
// 12 and mode register port 11 of the DMA controller (0x46 when the disk is read and 0x4A when
// the disk is written).
// Since the address and count registers of each channel are 16 bits, it is necessary to operate
// in two steps when setting them, one low byte and one high byte. Which byte is actually written
// is determined by the state of the trigger. When the trigger is 0, the low byte is accessed;
// when the trigger is 1, the high byte is accessed. The state of the trigger changes once per
// visit. Writing to port 12 sets the flip-flop to a 0 state, so that the setting of the 16-bit
// register starts from the low byte.
        __asm__("outb %%al,$12\n\tjmp 1f\n1:\tjmp 1f\n1:\t"
        "outb %%al,$11\n\tjmp 1f\n1:\tjmp 1f\n1:"::
        "a" ((char) ((command == FD_READ)?DMA_READ:DMA_WRITE)));
/* 8 low bits of addr */
// Write the base/current address register (port 4) to DMA channel 2.
        immoutb_p(addr,4);
        addr >>= 8;
/* bits 8-15 of addr */
        immoutb_p(addr,4);
        addr >>= 8;
/* bits 16-19 of addr */
// The DMA can only be addressed in 1MB of memory, and its upper 16-19 bits are placed in the
// page register (port 0x81).
        immoutb_p(addr,0x81);
/* low 8 bits of count-1 (1024-1=0x3ff) */
// Write the base/current byte counter value (port 5) to DMA channel 2.
```

```
        immoutb_p(0xff,5);
/* high 8 bits of count-1 */
// A total of 1024 bytes (two sectors) are transmitted at a time.
        immoutb_p(3,5);
/* activate DMA 2 */
        immoutb_p(0|2,10);
        sti();
}

//// Output a byte command or parameter to the floppy drive controller.
// Before sending a byte to the controller, the controller needs to be in a ready state, and
// the data transfer direction must be set from CPU to FDC, so the function needs to read the
// controller state information first. The loop query method is used here for proper delay.
// If an error occurs, the reset flag is set.
static void output_byte(char byte)
{
        int counter;
        unsigned char status;

// First, the state of the main state controller FD_STATUS (0x3f4) is cyclically read. If the
// read status is STATUS_READY and the direction bit STATUS_DIR = 0 (CPU ◊ FDC), the specified
// byte is output to the data port.
        if (reset)
                return;
        for(counter = 0 ; counter < 10000 ; counter++) {
                status = inb_p(FD_STATUS) & (STATUS_READY | STATUS_DIR);
                if (status == STATUS_READY) {
                        outb(byte,FD_DATA);
                        return;
                }
        }
// If it cannot be sent after the end of the cycle of 10,000 times, the reset flag is set and
// an error message is printed.
        reset = 1;
        printk("Unable to send byte to FDC\n\r");
}

//// Read the execution result information of the FDC.
// The result message is up to 7 bytes and is stored in the array reply_buffer[].Returns the
// number of result bytes read in. If the return value = -1, it indicates an error. The program
// is handled in a similar way to the above function.
static int result(void)
{
        int i = 0, counter, status;

// If the reset flag is set, exit immediately to perform the reset operation in the subsequent
// program. Otherwise, the state of the main state controller FD_STATUS (0x3f4) is cyclically
// read. If the read controller status is READY, indicating that no data is available, then
// the number of bytes i read is returned. If the controller status is: The direction flag is
// set (CPU <- FDC), ready, busy, it indicates that data is readable. The result data in the
// controller is then read into the response result array. The maximum number of bytes read
// is MAX_REPLIES(7).
        if (reset)
```

```
                return -1;
        for (counter = 0 ; counter < 10000 ; counter++) {
                status = inb_p(FD_STATUS)&(STATUS_DIR|STATUS_READY|STATUS_BUSY);
                if (status == STATUS_READY)
                        return i;
                if (status == (STATUS_DIR|STATUS_READY|STATUS_BUSY)) {
                        if (i >= MAX_REPLIES)
                                break;
                        reply_buffer[i++] = inb_p(FD_DATA);
                }
        }
// If it cannot be read after the end of the cycle of 10,000 times, the reset flag is set and
// an error message is printed.
        reset = 1;
        printk("Getstatus times out\n\r");
        return -1;
}

//// Floppy disk read/write error handling function.
// This function determines the further action that needs to be taken based on the number of
// floppy disk read and write errors. If the number of currently processed request errors is
// greater than the specified maximum number of errors, MAX_ERRORS (8 times), no further
// operational attempts are made for the current request. If the number of read/write errors
// has exceeded MAX_ERRORS/2, then the floppy drive needs to be reset, so the reset flag is
// set. Otherwise, if the number of errors is less than half of the maximum value, then only
// the head position needs to be recalibrated, so the recalibration flag is set. The actual
// reset and recalibration process will be performed in subsequent programs.
static void bad_flp_intr(void)
{
// First increase the number of errors in the current request item by one. If the current request
// item has more errors than the maximum allowed, the current floppy drive is deselected and
// the request is terminated (the buffer contents are not updated).
        CURRENT->errors++;
        if (CURRENT->errors > MAX_ERRORS) {
                floppy_deselect(current_drive);
                end_request(0);
        }
// If the number of errors in the current request item is greater than half of the maximum number
// of allowed errors, set the reset flag to reset the floppy drive afterwards, and try again.
// Otherwise, the floppy drive needs to be recalibrated and try again.
        if (CURRENT->errors > MAX_ERRORS/2)
                reset = 1;
        else
                recalibrate = 1;
}

/*
 * Ok, this interrupt is called after a DMA read/write has succeeded,
 * so we check the results, and copy any buffers.
 */
//// The floppy disk read/write function called in the interrupt.
// This function is called during the interrupt handling process that is initiated after the
// floppy drive controller operation ends. The function first reads the status information of
// the operation result, and accordingly determines whether the operation has a problem and
// handles it accordingly. If the read/write operation is successful, then if the request is
// a read operation and its buffer is in memory above 1MB, then the data needs to be copied
// from the floppy temporary buffer to the buffer of the request.
static void rw_interrupt(void)
{
// First read the result information of the FDC execution. If the number of returned result
// bytes is not equal to 7, or there is an error flag in status byte 0, 1, or 2, if a write
// protection error occurs, an error message is displayed, the current drive is released, and
// the current request is terminated. Otherwise, the error counting is performed, and then the
// floppy request item operation is continued. See the fdreg.h file for the meaning of the
// following states.
// ( 0xf8 = ST0_INTR | ST0_SE | ST0_ECE | ST0_NR )
// ( 0xbf = ST1_EOC | ST1_CRC | ST1_OR | ST1_ND | ST1_WP | ST1_MAM ), should be 0xb7
// ( 0x73 = ST2_CM | ST2_CRC | ST2_WC | ST2_BC | ST2_MAM )
        if (result() != 7 || (ST0 & 0xf8) || (ST1 & 0xbf) || (ST2 & 0x73)) {
                if (ST1 & 0x02) {         // 0x02 = ST1_WP - Write Protected。
                        printk("Drive %d is write protected\n\r",current_drive);
                        floppy_deselect(current_drive);
                        end_request(0);
                } else
                        bad_flp_intr();
                do_fd_request();
                return;
        }
// If the buffer of the current request item is above the 1MB address, the content of the floppy
// disk read operation is still placed in the temporary buffer and needs to be copied into the
// buffer of the current request item. Finally release the current floppy drive (deselected)
// and execute the current request item end processing: wake up the process waiting for the
// request item, wake up the process waiting for the idle request item (if any), delete the
// request item from the linked list of requests . Then continue to perform other floppy request
// operations.
        if (command == FD_READ && (unsigned long)(CURRENT->buffer) >= 0x100000)
                copy_buffer(tmp_floppy_area,CURRENT->buffer);
        floppy_deselect(current_drive);
        end_request(1);
        do_fd_request();
}

//// Set DMA channel 2 and output commands and parameters to the floppy disk controller
// (1-byte command + 0~7 byte parameter).
// If the reset flag is not set, then a floppy disk interrupt will be generated and the floppy
// disk interrupt handler will be executed after the function exits and the floppy disk controller
// performs the corresponding read/write operation.
inline void setup_rw_floppy(void)
{
        setup_DMA();                // Initialize the floppy disk DMA channel.
        do_floppy = rw_interrupt;   // set function called in the int.
        output_byte(command);       // send command.
        output_byte(head<<2 | current_drive);  // param: head no + drive no.
        output_byte(track);         // param: track no.
        output_byte(head);          // param: head no.
        output_byte(sector);        // param: start sector no.
        output_byte(2);            /* sector size = 512 */
        output_byte(floppy->sect); // param: sectors per track.
        output_byte(floppy->gap);  // param: gap between sectors.
        output_byte(0xFF);         /* sector size (0xff when n!=0 ?) */
// If any of the above output_byte() operations fail, the reset flag is set. The reset processing
// code in do_fd_request() is executed immediately.
        if (reset)
                do_fd_request();
}

/*
 * This is the routine called after every seek (or recalibrate) interrupt
 * from the floppy controller. Note that the "unexpected interrupt" routine
 * also does a recalibrate, but doesn't come here.
 */
//// The C function called during the interrupt process after the seek operations.
// First, the detection interrupt status command is sent, and the status information ST0 and
// the track information of the head are obtained.  If an error occurs, the error count detection
// process is executed or the floppy operation request item is canceled. Otherwise, set the
// current track variable according to the status information, then call the function
// setup_rw_floppy() to set the DMA and output the read/write commands and parameters.
static void seek_interrupt(void)
{
// The check interrupt status command is sent first to obtain the result of the seek operation
// execution. This command takes no arguments. The returned result is two bytes: ST0 and the
// current track number of the head. Then read the result information of the FDC execution.
// If the number of returned result bytes is not equal to 2, or ST0 is not the end of the seek,
// or the track on which the head is located (ST1) is not equal to the set track, an error has
// occurred. Then, the error counting is processed, and then the execution of the floppy disk
// request item or the execution of the reset processing is continued. Note that the sense
// interrupt status command (FD_SENSEI) should return 2 result bytes, that is the result() return
// value should equal 2.
/* sense drive status */
        output_byte(FD_SENSEI);
        if (result() != 2 || (ST0 & 0xF8) != 0x20 || ST1 != seek_track) {
                bad_flp_intr();
                do_fd_request();
                return;
        }
// If the seek operation is successful, the floppy disk operation of the current request is
// continued, that is, the command and parameters are sent to the floppy disk controller.
        current_track = ST1;       // set current track.
        setup_rw_floppy();         // set DMA, output floppy commands and parameters.
}

/*
 * This routine is called when everything should be correctly set up
 * for the transfer (ie floppy motor is on and the correct floppy is
 * selected).
 */
//// Read/write data transfer function.
static void transfer(void)
{
    // First check if the current drive parameter is the one of the specified drive. If not, send
    // the set drive parameter command and the corresponding parameters (param1: high 4-bit step
    // rate, low 4-bit head unloading time; param2: head loading time). Then it is judged whether
    // the current data transmission rate is consistent with the specified drive, and if not, the
    // rate of the specified floppy drive is sent to the data transmission rate control register
    // (FD_DCR).
312         if (cur_spec1 != floppy->spec1) {             // check the current parameters.
313                 cur_spec1 = floppy->spec1;
314                 output_byte(FD_SPECIFY);              // send set disk parameters command.
315                 output_byte(cur_spec1);               /* hut etc */
316                 output_byte(6);                       /* Head load time =6ms, DMA */
317         }
318         if (cur_rate != floppy->rate)                 // check current rate.
319                 outb_p(cur_rate = floppy->rate,FD_DCR);
    // If any of the above output_byte() operations fail, the reset flag will be set. So here we
    // need to check the reset flag. If the reset is actually set, the reset processing code in
    // do_fd_request() is executed immediately.
320         if (reset) {
321                 do_fd_request();
322                 return;
323         }
    // If the seek flag is zero at this time (ie no seek is required), the DMA is set and the
    // corresponding operation command and parameters are sent to the floppy disk controller and
    // returned. Otherwise, the seek processing is performed, so the function called in the floppy
    // interrupt is first set to the seek track function. If the starting track number is not equal
    // to zero, the head seek command and parameters are sent. The parameter used is the global
    // variable value set on line 112--121. If the starting track number seek_track is 0, a
    // recalibration command is executed to return the head to the zero track.
324         if (!seek) {
325                 setup_rw_floppy();                      // Send command & parameter block.
326                 return;
327         }
328         do_floppy = seek_interrupt;                     // set invoked function.
329         if (seek_track) {                               // start track.
330                 output_byte(FD_SEEK);                   // send seek command.
331                 output_byte(head<<2 | current_drive);// param: head + current drive.
332                 output_byte(seek_track);                // param: track no.
333         } else {
334                 output_byte(FD_RECALIBRATE);            // send recalibrate command.
335                 output_byte(head<<2 | current_drive);// param: head + current drive.
336         }
    // Similarly, if any of the above output_byte() operations fail, the reset flag will be set
    // and the reset processing code in do_fd_request() is executed immediately.
337         if (reset)
338                 do_fd_request();
339 }
340
341 /*
342  * Special case - used after a unexpected interrupt (or reset)
343  */
    //// The floppy drive recalibration function called in the interrupt.
    // The check interrupt status command (no parameter) is sent first. If the return result
    // indicates an error, the reset flag is set, otherwise the recalibration flag is cleared. Then
// execute the floppy disk request item processing function to perform the corresponding
// operation. Note that the sense interrupt status command (FD_SENSEI) will return 2 result
// bytes, that is the result() return value should equal 2.
static void recal_interrupt(void)
{
        output_byte(FD_SENSEI);                    // send sense interrupt status cmd.
        if (result()!=2 || (ST0 & 0xE0) == 0x60)   // reset if there are errors
                reset = 1;
        else
                recalibrate = 0;
        do_fd_request();
}

//// The unexpected interrupt handling function called in the floppy interrupt.
// The check interrupt status command (no parameter) is sent first. If the return result
// indicates an error, the reset flag is set, otherwise the recalibration flag is set.
void unexpected_floppy_interrupt(void)
{
        output_byte(FD_SENSEI);                  // send sense interrupt status cmd.
        if (result()!=2 || (ST0 & 0xE0) == 0x60) // reset if there are errors.
                reset = 1;
        else
                recalibrate = 1;
}

//// The floppy disk recalibration function.
// The recalibration flag is first reset and a recalibration command and its parameters are
// sent to the floppy disk controller (FDC). When the controller executes the recalibration
// command, it calls the recal_interrupt() function in the floppy disk interrupt it raises.
static void recalibrate_floppy(void)
{
        recalibrate = 0;
        current_track = 0;
        do_floppy = recal_interrupt;             // point to recal function.
        output_byte(FD_RECALIBRATE);             // cmd: recalibrate.
        output_byte(head<<2 | current_drive);    // param: head no + drive no.
// Similarly, if any of the above output_byte() operations fail, the reset flag will be set
// and the reset processing code in do_fd_request() is executed immediately.
        if (reset)
                do_fd_request();
}

//// The floppy disk controller FDC reset handling function called in the interrupt.
// First send the sense interrupt status command (no parameters) and read the returned result
// byte. Then send the set floppy drive parameter command and its related parameters, and finally
// call the request processing function do_fd_request() again to perform the request item or
// error processing operation.
static void reset_interrupt(void)
{
        output_byte(FD_SENSEI);                  // send sense interrupt status cmd.
        (void) result();
        output_byte(FD_SPECIFY);                 // send drive param setting cmd.
        output_byte(cur_spec1);                  /* hut etc */
        output_byte(6);                         /* Head load time =6ms, DMA */
        do_fd_request();
}

/*
 * reset is done by pulling bit 2 of DOR low for a while.
 */
//// Reset the floppy disk controller.
// The function first sets the parameters and flags, clears the reset flag, and then sets the
// floppy variables cur_spec1 and cur_rate to invalid. These two parameters need to be reset
// after the reset operation. Then set the recalibration flag and set the C function
// reset_interrupt() called in the floppy disk interrupt triggered by the FDC after the reset
// operation. Finally, the digital output register (DOR) bit 2 is set to 0 for a while to perform
// a reset operation on the floppy drive. Bit 2 of DOR is the start/reset floppy drive bit.
static void reset_floppy(void)
{
        int i;

        reset = 0;
        cur_spec1 = -1;                     // invalidated.
        cur_rate = -1;
        recalibrate = 1;                    // set recalibration flag.
        printk("Reset-floppy called\n\r");
        cli();
        do_floppy = reset_interrupt;        // point reset function.
        outb_p(current_DOR & ~0x04,FD_DOR); // do reset command.
        for (i=0 ; i<100 ; i++)             // delay for a while.
                __asm__("nop");
        outb(current_DOR,FD_DOR);           // enable controller again.
        sti();
}

//// The floppy timer function called in the timer interrupt.
// The do_fd_request() function adds a delay timer to the current request in order to wait for
// the specified floppy motor to spin up to reach the normal operating speed before performing
// the operation requested by a request. This is the function that is called when the timer
// expires. It first checks the digital output register (DOR) to select the currently specified
// drive and then calls the read-write transfer function transfer().
static void floppy_on_interrupt(void)         // floppy_on() interrupt。
{
/* We cannot do a floppy-select, as that might sleep. We just force it */
// If the current drive is different from the DOR, you will need to reset the DOR to the currently
// specified drive. After outputting the current DOR value to the DOR register, the timer is
// used to delay 2 ticks to allow the command to be executed, and then the floppy disk read/write
// transfer function transfer() is called. If the current drive matches the DOR, then the floppy
// disk read and write transfer function can be called directly.
        selected = 1;                        // set drive selected flag.
        if (current_drive != (current_DOR & 3)) {
                current_DOR &= 0xFC;         // clear selected drive.
                current_DOR |= current_drive; // set current drive.
                outb(current_DOR,FD_DOR);    // send current DOR.
                add_timer(2,&transfer);      // add timer and related function.
        } else
```

```
415                 transfer();
416 }
417
    //// Floppy disk read/write request item processing function.
    // This is the main function in the floppy driver. Its main uses are: (1) Processing the case
    // where the reset flag or the re-correction flag is set; (2) Obtaining the parameter block
    // of the floppy drive by the request item using the device number in the request item; (3)
    // Starting the floppy disk read/write operation by using the kernel timer.
418 void do_fd_request(void)
419 {
420         unsigned int block;
421
    // First check if there is a reset flag or a recalibration flag. If one of them exists, the
    // function returns immediately after processing the relevant flag.
422         seek = 0;                               // reset seek flag.
423         if (reset) {
424                 reset_floppy();
425                 return;
426         }
427         if (recalibrate) {
428                 recalibrate_floppy();
429                 return;
430         }
    // The important aspects of this function start here. First use the INIT_REQUEST macro in the
    // blk.h file to check the validity of the request item, and exit if there is no request. Then
    // use the device number in the request item to get the parameter block of the specified floppy
    // drive. This parameter block will be used below to set the global variable parameter block
    // used by the floppy disk operation (see lines 112 - 122). The floppy disk type
    // (MINOR(CURRENT->dev)>>2) in the request item number is used as the index value of the disk
    // type array floppy_type[] to get the parameter block of the specified floppy drive.
431         INIT_REQUEST;
432         floppy = (MINOR(CURRENT->dev)>>2) + floppy_type;

    // The following code begins to set the global variable parameter value on lines 112-122. If
    // the current drive 'current_drive' is not the drive specified in the request item, the flag
    // seek is set to indicate that the drive needs to perform seek processing before performing
    // the read/write operation. Then set the current drive to the drive specified in the request.
433         if (current_drive != CURRENT_DEV)    // the drive specified in the request.
434                 seek = 1;
435         current_drive = CURRENT_DEV;

    // Next, start setting the read/write start sector block. Since each read and write is in block
    // units (1 block is 2 sectors), the starting sector needs to be at least 2 sectors smaller
    // than the total number of sectors of the disk. Otherwise, the request item parameter is invalid,
    // and the floppy disk request item is terminated to execute the next request item.  Then
    // calculate: the sector number, head number, track number, and seek track number (for
    // floppy drives to read discs of different formats).
436         block = CURRENT->sector;
437         if (block+2 > floppy->size) {
438                 end_request(0);
439                 goto repeat;
440         }
441         sector = block % floppy->sect;    // the sector number on the track.
```

```
442         block /= floppy->sect;           // track number.
443         head = block % floppy->head;     // head no.
444         track = block / floppy->head;    // track no.
445         seek_track = track << floppy->stretch; // seek track number related to drive type.

    // Then see if we still need to perform the seek operation first. If the seek number is different
    // from the track number of the current head, a seek operation is required, and the seek flag
    // is required. Finally we set up the floppy command to be executed.
446         if (seek_track != current_track)
447                 seek = 1;
448         sector++;                        // sectors count from 1.
449         if (CURRENT->cmd == READ)
450                 command = FD_READ;
451         else if (CURRENT->cmd == WRITE)
452                 command = FD_WRITE;
453         else
454                 panic("do_fd_request: unknown command");
    // After setting all the global variable values on lines 112-122, we can start the request item
    // operation. Here, the operation is started using a timer.  Because it is necessary to start
    // the drive motor first and reach the normal running speed, the floppy drive can be read and
    // written, which takes a certain amount of time. So here ticks_to_floppy_on() is used to
    // calculate the start delay time, and then use this delay to set a timer. The function
    // floppy_on_interrupt() is called when the time expires.
455         add_timer(ticks_to_floppy_on(current_drive),&floppy_on_interrupt);
456 }
457
    // The total number of data blocks contained in various types of floppy disk.
458 static int floppy_sizes[] ={                  // initial data for array blk_size[].
459            0,    0,    0,    0,
460          360,  360 , 360,  360,
461         1200, 1200, 1200, 1200,
462          360,  360,  360,  360,
463          720,  720,  720,  720,
464          360,  360,  360,  360,
465          720,  720,  720,  720,
466         1440, 1440, 1440, 1440
467 };
468
    //// Floppy disk system initialization function.
    // Set the processing function do_fd_request() of the floppy device request and set the floppy
    // disk interrupt gate (int 0x26, corresponding to the hardware interrupt request signal IRQ6).
    // The masking of the interrupt signal is then reset to allow the floppy disk controller FDC
    // to send an interrupt request signal. The setting macro set_trap_gate() of the trap gate
    // descriptor in the interrupt descriptor table IDT is defined in the header file
    // include/asm/system.h.
469 void floppy_init(void)
470 {
    // Set the floppy disk interrupt gate descriptor. floppy_interrupt is its interrupt handler,
    // see kernel/sys_call.s, line 267. The interrupt number is int 0x26 (38), corresponding to
    // the 8259A chip interrupt request signal IRQ6.
471         blk_size[MAJOR_NR] = floppy_sizes;
472         blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;   // = do_fd_request()。
473         set_trap_gate(0x26,&floppy_interrupt);
```

```
        outb(inb_p(0x21)&~0x40,0x21);                   // reset floppy int mask bit.
}

```

9.6.3 Information

9.6.3.1 Device number of the floppy disk drive

In Linux, the floppy drive's major number is 2, and the minor device number is determined by the floppy drive type and the floppy drive sequence number, which is:

```
    FD Minor No.  = TYPE * 4 + DRIVE
```

Among them, DRIVE is 0-3, corresponding to floppy drive A, B, C or D; TYPE is the type of floppy drive, for example, 2 means 1.2M floppy drive, 7 means 1.44M floppy drive, as shown in Table 9-12. That is, it is the index value of the floppy type array (floppy_type[]) defined in floppy.c, line 85.

Table 9-12 Floppy drive type

Type	Description
0	Not used.
1	360KB PC Floppy drive.
2	1.2MB AT Floppy drive.
3	360kB Floppy disk used in the 720kB drive.
4	3.5" 720kB Floppy drive.
5	360kB Floppy disk used in the 1.2MB drive.
6	720kB Floppy disk used in the 1.2MB drive.
7	1.44MB Floppy drive.

For example, type 7 indicates a 1.44MB drive, drive number 0 indicates an A drive, because 7*4 + 0 = 28, so (2, 28) refers to the 1.44M drive A, the device number is 0x021C, and the corresponding device file name is /dev/fd0 or /dev/PS0. Similarly, type 2 represents a 1.22MB drive, then 2*4 + 0 = 8, so (2,8) refers to the 1.2M drive A, the device number is 0x0208, and the corresponding device file name is /dev/at0.

9.6.3.2 Floppy Drive Controller

Since it is necessary to select a floppy drive, wait for the motor to reach a certain speed before reading and writing the floppy disk, and when the data block is transmitted, it needs to be realized by means of the DMA controller, it is cumbersome to program the floppy disk controller (FDC). When programming the FDC, it usually needs to access 4 ports, corresponding to one or more registers on the floppy disk controller. For the 1.2M floppy disk controller there are some of the ports shown in Table 9-13.

Table 9-13 Floppy disk controller ports

I/O port	Name	Reed/Write	Register Name
0x3f2	FD_DOR	Write only	Digital Output Register (DOR)
0x3f4	FD_STATUS	Read only	Main Status Register (STATUS)

0x3f5	FD_DATA	Read only	Result Register(RESULT)
		Write only	Data Register(DATA)
0x3f7	FD_DIR	Read only	Digital Input Register (DIR)
	FD_DCR	Write only	Drive Control Register (DCR)(Transfer Rate)

The digital output register (DOR) port is an 8-bit register that controls the driver motor turn-on, drive select, start/reset FDC, and enable/disable DMA and interrupt requests. The meaning of each bit of this register is shown in Table 9-14.

Table 9-14 Digital output register definition

Bit	Name	Description
7	MOT_EN3	Motor control for drive D: 1- Start motor; 0-Stop motor.
6	MOT_EN2	Motor control for drive C: 1- Start motor; 0-Stop motor.
5	MOT_EN1	Motor control for drive B: 1- Start motor; 0-Stop motor.
4	MOT_EN0	Motor control for drive A: 1- Start motor; 0-Stop motor.
3	DMA_INT	DMA and IRQ channel: 1-Enabled; 0-Disabled.
2	RESET	Controller reset: 1-Controller enabled; 0-Reset controller.
1	DRV_SEL1	00-11 is used to select floppy drive A-D respectively.
0	DRV_SEL0	

The FDC's Main Status Register (MSR) is also an 8-bit register that reflects the basic state of the floppy disk controller FDC and floppy disk drive FDD operation. Typically, the status bits of the main status register are read before the CPU sends a command to the FDC or before the FDC obtains the result of the operation to determine if the current FDC data register is ready and to determine the direction of data transfer. The definition of each bit of the MSR is shown in Table 9-17.

Table 9-15 The definition of each bit of the MSR

Bit	Name	Description
7	RQM	Data port ready: FDC data register is ready.
6	DIO	Tranfer direction: 1- FDC -> CPU; 0 - CPU -> FDC
5	NDM	Non DMA mode: 1 - Controller not in DMA; 0 - DMA mode
4	CB	Controller busy: FDC is busy in executing a command.
3	DDB	Drive D is busy.
2	DCB	Drive C is busy.
1	DBB	Drive B is busy.
0	DAB	Drive A is busy.

The data port of the FDC corresponds to multiple registers: a write-only command and parameter register and a read-only result register. Only one register can appear on data port 0x3f5 at a time. When accessing a write-only register, the direction bit DIO of the main status register must be 0 (CPU -> FDC), and vice versa when accessing a read-only register. When accessing the write-only register to send commands and parameters, the command is 1 byte and the related parameter is 1-8 bytes. When accessing the read-only register to read the result, the result is only read after the FDC is not busy, and usually the result data has a maximum of 7 bytes.

For the data input register (DIR), only bit 7 (D7) is valid for the floppy disk, which is used to indicate the disk replacement status, and the remaining seven bits are used for the hard disk controller interface.

The write-only disk control register (DCR) is used to select the data transfer rate that the disc uses on different types of drives. Only the lower 2 bits (D1D0) are used, 00 means 500 kbps, 01 means 300 kbps, and 10 means 250 kbps.

In the Linux 0.12 kernel, the data transfer between the driver and the disk in the floppy drive is implemented by the DMA controller. Therefore, before performing read and write operations, you need to initialize the DMA controller first and program the floppy drive controller. For a 386-compatible PC, the floppy drive controller uses the hardware interrupt request signal IRQ6 (corresponding to interrupt descriptor 0x26) and uses channel 2 of the DMA controller. See the following sections for details on DMA control processing.

9.6.3.3 Floppy disk controller command

The floppy disk controller can accept a total of 15 commands, each of which goes through three phases: the command phase, the execution phase, and the result phase.

The command phase is that the CPU sends command byte and parameter bytes to the FDC. The first byte is always the command byte (command code). This is followed by parameter of 0-8 bytes. These parameters are usually the drive number, head number, track number, sector number, and total number of read/write sectors.

The execution phase is the operation specified by the FDC command. During the execution phase, the CPU does not intervene on the FDC. Generally, the FDC issues an interrupt request to let the CPU know the end of the command execution. If the FDC command sent by the CPU is to transfer data, the FDC can operate in an interrupt mode or a DMA mode. Each time the interrupt mode is transmitted by 1 byte, the DMA mode can transfer a large amount of data at a time. Under the management of the DMA controller, data is transferred between the FDC and the memory until all data has been transferred. At this time, the DMA controller notifies the FDC of the transmission byte count termination signal, and finally the FDC issues an interrupt request signal to inform the CPU that the execution phase is over.

The result phase is that the CPU reads the FDC data register (result register) return value to obtain the result of the FDC command execution. The result data returned is 0--7 bytes in length. For commands that do not return result data, we should send a detection interrupt status command to the FDC to get the status of the operation.

Since only six of the 15 commands are used in the Linux 0.12 floppy driver, only the commands used are described here.

1. Recalibration command (FD_RECALIBRATE)

This command is used to return the head to track 0. Usually used to recalibrate the head when the floppy disk operation is in error. The command code is 0x07, and the parameter is the specified drive letter (0-3).

The command has no result phase, and the program needs to obtain the execution result by executing "detect interrupt status" command. The format of this command is shown in Table 9-16.

Table 9-16 Format of recalibration command (FD_RECALIBRATE)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	0	0	0	0	0	1	1	1	Recalibration command code: 0x07
Parameter	1	0	0	0	0	0	0	US1	US2	Disk drive number.
Execution										The head moves to 0 track
Result		None.								You need to use the command to get the

			execution result.

2. Head seek command (FD_SEEK)

This command moves the head of the selected drive to the specified track. The first parameter specifies the drive number and head number, bits 0-1 are the drive number, bit 2 is the head number, and other bits are useless. The second parameter specifies the track number.

The command has no result phase, and the program needs to obtain the execution result by executing "detect interrupt status" command. The format of this command is shown in Table 9-17.

Table 9-17 Format of head seek command (FD_SEEK)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	0	0	0	0	1	1	1	1	Head seek command code: 0x0F
Parameter	1	0	0	0	0	0	HD	US1	US2	Head number, Drive number.
	2	C								Track number.
Execution										The head moves to the specified track.
Result		None.								You need to use the command to get the execution result.

3. Read sector data command (FD_READ)

This command is used to read the sector starting from the specified location on the disk and transfer it to the system memory buffer via DMA controller. Whenever a sector is read, parameter 4 (R) is automatically incremented by one to continue reading the next sector until the DMA controller sends a transmission count termination signal to the floppy disk controller. This command usually begins after the head seek command is executed and the head is already on the specified track. The format of the command is shown in Table 9-18.

In the returned result, the track number C and the sector number R are the positions at which the current head is located. Since the starting sector number R is automatically incremented by one after reading one sector, the R value in the result is the next unread sector number. If the last sector on a track (ie EOT) is read, the track number is also incremented by one and the R value is reset to one.

Table 9-18 Format of read sector data command (FD_READ)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	MT	MF	SK	0	0	1	1	0	Read sector command code: 0xE6 (MT=MF=SK=1)
Parameters	1	0	0	0	0	0	0	US1	US2	Drive number
	2	C								Track number (Cylinder address, 0 to 255)
	3	H								Head number (Head address, 0 or 1)
	4	R								Start sector number (Sector address)
	5	N								Sector size code (N=0..7: 128,256,512,,16KB)
	6	EOT								Final sector number. of the track (End of Track)
	7	GPL								Length of gap between sectors (3)
	8	DTL								The number of bytes in sector, when N=0
Execution										Data is transferred from disk to system
Result	1	ST0								Status byte 0

	2	ST1	Status byte 1
	3	ST2	Status byte 2
	4	C	Track number
	5	H	Header number
	6	R	Sector number
	7	N	Sector size code (0..7: 128,256,512,...,16KB)

Among them, the meanings of MT, MF and SK are:

- MT represents multi-track operation. MT = 1 indicates that two heads are allowed to operate continuously on the same track.
- MF indicates the recording method. MF=1 means to select the MFM recording mode, otherwise it is the FM recording mode.
- SK indicates whether to skip the sector with the delete flag. SK=1 means skip.

The meanings of the returned three status bytes ST0, ST1, and ST2 are shown in Table 9–19, Table 9–20, and Table 9–21, respectively.

Table 9-19 Status byte 0 (ST0)

Bit	Name	Description
7	ST0_INTR	Reason for interrupt. 00 - Normal termination of command; 01 - Abnormal termination of command; 10 - Invalid command; 11 - Abnormal termination caused by Polling.
6		
5	ST0_SE	Seek End. The controller completed a SEEK or RECALIBRATE command, or a READ/WRITE with implied seek command.
4	ST0_ECE	Equip. Check Error. Recalibration track 0 error.
3	ST0_NR	Not Ready. Floppy disk drive not ready.
2	ST0_HA	Head address. The current head number when interrupt occurs.
1	ST0_DS	Drive Select. Drive number when interrupt occurs. 00 - 11 corresponds to drive 0 - 3 rescpectively
0		

Table 9-20 Status byte 1 (ST1)

Bit	Name	Description
7	ST1_EOC	End of Cylinder. Tried to access a sector beyond the final sector of the track.
6		Unused. This bit is always 0.
5	ST1_CRC	The controller detected a CRC error in ID field or the Data field of a sector.
4	ST1_OR	Over Run. Data transfer timeout, DMA controller failure
3		Unused (0).
2	ST1_ND	No Data. The specified sector was not found.
1	ST1_WP	Write Protect.
0	ST1_MAM	Missing Address Mask. Sector ID address mark not found.

Table 9-21 Status byte 2 (ST2)

Bit	Name	Description

7		Unused (0).
6	ST2_CM	Control Mark. When SK=0, the read data encounters the delete flag.
5	ST2_CRC	CRC error. The sector data field CRC check error.
4	ST2_WC	Wrong Cylinder. The track number C of the sector ID info does not match.
3	ST2_SEH	Scan Equal Hit. The scanning conditions meet the requirements.
2	ST2_SNS	Scan Not Satisfied: Scanning conditions do not meet the requirements
1	ST2_BC	Bad Cylinder. The track C = 0xFF in the sector ID info, the track is bad.
0	ST2_MAM	Missing Address Mask. The sector ID data address mark not found.

4. Write sector data command (FD_WRITE)

This command is used to write data from the memory buffer to disk. In the DMA transfer mode, the floppy drive controller serially writes the data in the memory to the specified sector of the disk. Each time a sector is written, the starting sector number is automatically incremented by one and continues to write one sector until the floppy drive controller receives the count termination signal from the DMA controller. The format of the command is shown in Table 9-22. The abbreviated name has the same meaning as in the read command.

Table 9-22 Format of write sector data command (FD_WRITE)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	MT	MF	0	0	0	1	0	1	Write data command code: 0xC5 (MT=MF=1)
Parameters	1	0	0	0	0	0	0	US1	US2	Floppy drive number.
	2	C								Track number.
	3	H								Head number.
	4	R								Start sector number
	5	N								Sector size code.
	6	EOT								Final sector number. of the track
	7	GPL								Length of gap between sectors (3)
	8	DTL								The number of bytes in sector, when N=0
Execution										Data is transferred from the system to the disk
Result	1	ST0								Status byte 0
	2	ST1								Status byte 1
	3	ST2								Status byte 2
	4	C								Track number
	5	H								Head number
	6	R								Sector number
	7	N								Sector size code.

5. Check interrupt status command (FD_SENSEI)

After sending this command, the floppy controller will immediately return the normal results 1 and 2 (ie, state ST0 and the track number PCN where the head is located). They are the result states after the controller executes the previous command. An interrupt signal is usually sent to the CPU after the execution of a command. For interrupts caused by read/write sectors, read/write tracks, read/write delete flags, read ID field, format and scan commands, and commands in non-DMA transfer mode, the cause of the interrupt can be known directly based on the flag of the main status register. For the interrupt caused by the drive's ready signal change, seek and recalibration (head return to zero), because there is no return result, you need to use this command to read the status information after the controller executes the command. The format of this command is shown in Table 9-23.

Table 9-23 Format of check interrupt status command (FD_SENSEI)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	0	0	0	0	1	0	0	0	Detection interrupt status cmd code: 0x08
Execution										
Result	1	ST0								Status byte 0
	2	C								Track number of the current head.

6. Set drive parameter command (FD_SPECIFY)

This command is used to set the three initial timer values and the selected transmission mode inside the floppy disk controller, that is, the drive motor step rate (SRT), head loading/unloading (HLT/HUT) time, and whether to use DMA mode for transmission, are sent to the floppy drive controller. The format of this command is shown in Table 9-24. The time unit is the value when the data transmission rate is 500 KB/S. In addition, in the Linux 0.12 kernel, the parameter byte 1 of the command phase is the spec1 described in the original comment on lines 96-103 in floppy.c file; the parameter byte 2 is the spec2. From the original comment and the program statement on line 316, spec2 is fixedly set to a value of 6 (ie, HLT = 3, ND = 0), indicating that the head load time is 6 milliseconds, and the DMA mode is used.

Table 9-24 Format of setting drive parameter command (FD_SPECIFY)

Phase	Seq	D7	D6	D5	D4	D3	D2	D1	D0	Description
Command	0	0	0	0	0	0	0	1	1	Set parameter command code: 0x03
Parameters	1	SRT (Unit: 1ms)				HUT (Unit: 16ms)				Motor step rate, head unloading time
	2	HLT (Unit: 2ms)							ND	Head load time, non-DMA mode
Execution										Set the controller, no interrupt issued.
Result		None.								None.

9.6.3.4 Floppy disk controller programming

In a PC, the floppy disk controller generally uses a compatible chip of NEC PD765 or Intel 8287A, such as Intel's 82078. Since the driver of the floppy disk is relatively complicated, the programming method of the floppy disk controller composed of such a chip is described in detail below. Typical disk operations include not only sending commands and waiting for controllers to return results. The control of a floppy disk drive is a low-level operation that requires the program to interfere with its execution at different stages.

1. Interaction between command and result phases

Before the above disk operation commands and parameters are sent to the floppy disk controller, the controller's main status register (MSR) must be queried first to know the ready state of the drive and the data transfer direction. The floppy disk driver uses an output_byte(byte) function to do this. The equivalent block diagram of this function is shown in Figure 9-9.

This function is looped until the data port ready flag RQM of the main status register is 1, and the direction flag DIO is 0 (CPU -> FDC), at which point the controller is ready to accept command and parameter bytes. The loop statement starts from the timeout counting function to cope with the case where the controller does not respond. In this driver, the number of loops is set to 10000 times. The choice of the number of loops needs to be careful to avoid the program making an incorrect timeout. In the Linux kernel version 0.1x to 0.9x, it is often encountered that the number of cycles needs to be adjusted, because the PCs used at that time have different speeds (16MHz - 40MHz), so the actual delay caused by the loop will be very different. This can be seen from the discussion of many articles in the early Linux mailing list. To completely solve this problem, it is best to use the system hardware clock to generate a fixed frequency delay value.

For the result phase of reading the result bytes of the controller, it is also necessary to take the same operation method as the send command, except that the data transfer direction flag request is set (FDC -> CPU). The corresponding function in this program is result(), which stores the result status bytes into the reply_buffer[] byte array.

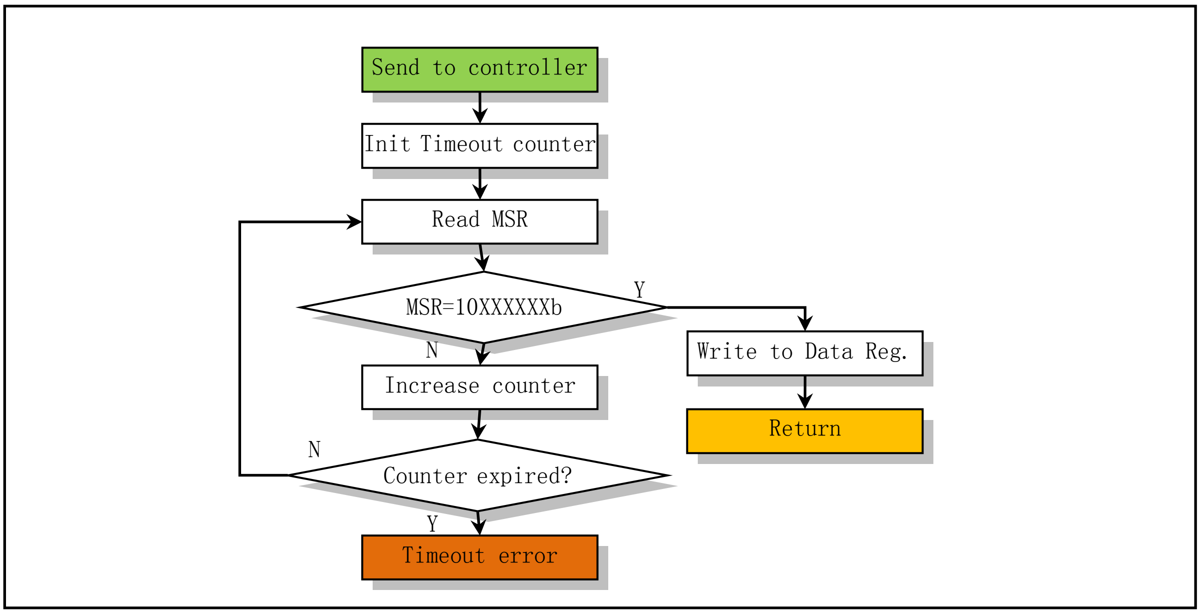


Figure 9-9 Send command or parameter bytes to the floppy controller

2. Floppy disk controller initialization

Initializing the floppy disk controller involves configuring the appropriate parameters for the drive after the controller is reset. The controller reset operation is to set the FDC reset flag (bit 2 of DOR) to 0 (reset) and then 1. After the FDC reset, the value set by the "Specify Drive Parameters" command SPECIFY is no longer valid and needs to be re-established. In the floppy.c program, the reset operation is included in the function reset_floppy() and the interrupt handler C function reset_interrupt(). The previous function is used to modify bit 2 of the DOR register to reset the controller. The latter function is used to re-establish the driver parameters in the controller using the SPECIFY command after the controller is reset. During the data transfer preparation phase, if it is detected that the current drive parameters in the FDC are different from the actual disk specifications, it will be additionally reset at the beginning of the transfer function transfer().

After the controller is reset, the specified transfer rate should also be sent to the digital control register DCR to reinitialize the data transfer rate. If the machine performs a reset operation (such as a warm boot), the data transfer rate will become the default value of 250Kpbs. However, the reset operation issued to the controller through the digital output register DOR does not affect the data transfer rate.

3. Drive recalibration and head seek

Driver recalibration (FD_RECALIBRATE) and head seek (FD_SEEK) are two head positioning commands. The recalibration command moves the head to zero track, and the head seek command moves the head to the specified track. These two head positioning commands are different from typical read/write commands because they have no result phase. Once one of these two commands is issued, the controller will immediately return to the Ready state in the Main Status Register (MSR) and perform head positioning operations in the background. When the positioning operation is completed, the controller generates an interrupt request service. At this point, you should send a "Detect Interrupt Status" command to end the interrupt and read the status after the positioning operation. Since the drive and motor enable signals are directly controlled by the digital output register (DOR), if the drive or motor has not been started, the operation of writing DOR must be performed before the positioning command is issued. A related flow chart is shown in Figure 9-10.

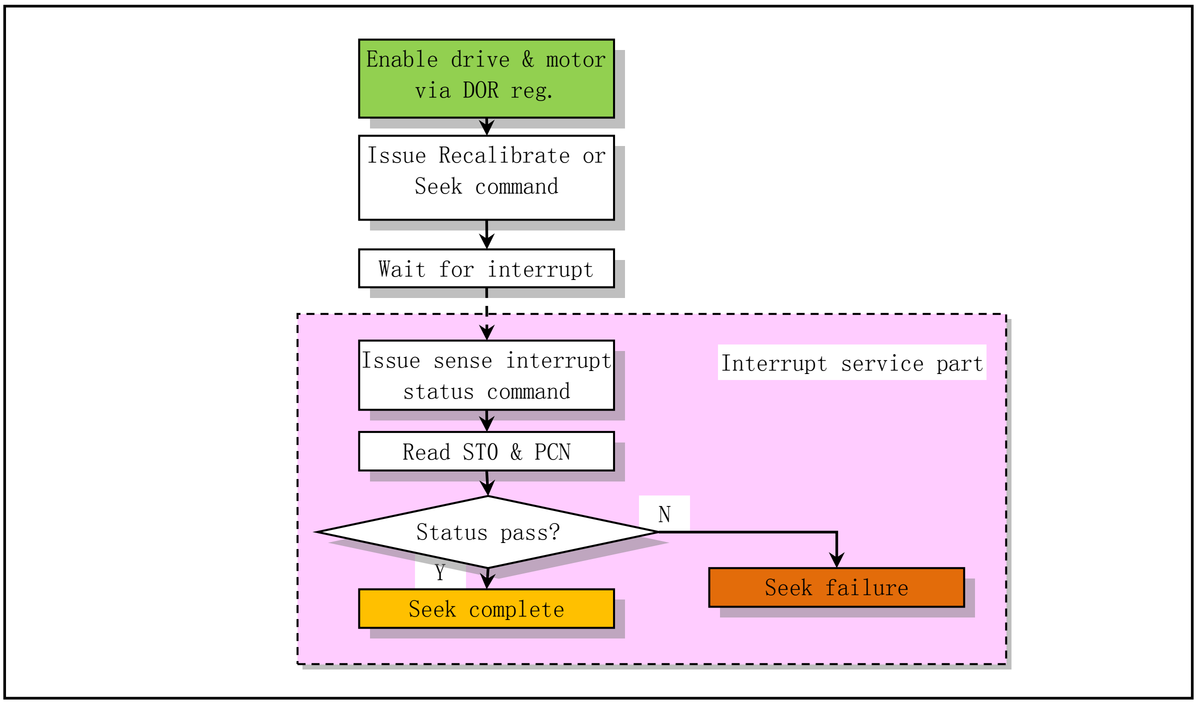

Enable drive & motor via DOR reg.
Issue Recalibrate or Seek command
Wait for interrupt
Issue sense interrupt status command
Interrupt service part
Read ST0 & PCN
Status pass?
N
Y
Seek complete
Seek failure

Figure 9-10 Recalibration and seek operation

4. Data read/write operations

Data read or write operations take a few steps to complete. First the drive motor needs to be turned on and the head is positioned on the correct track, then the DMA controller is initialized and finally a data read or write command is sent. In addition, you need to determine the processing plan when an error occurs. A typical operational flow chart is shown in Figure 9-11.

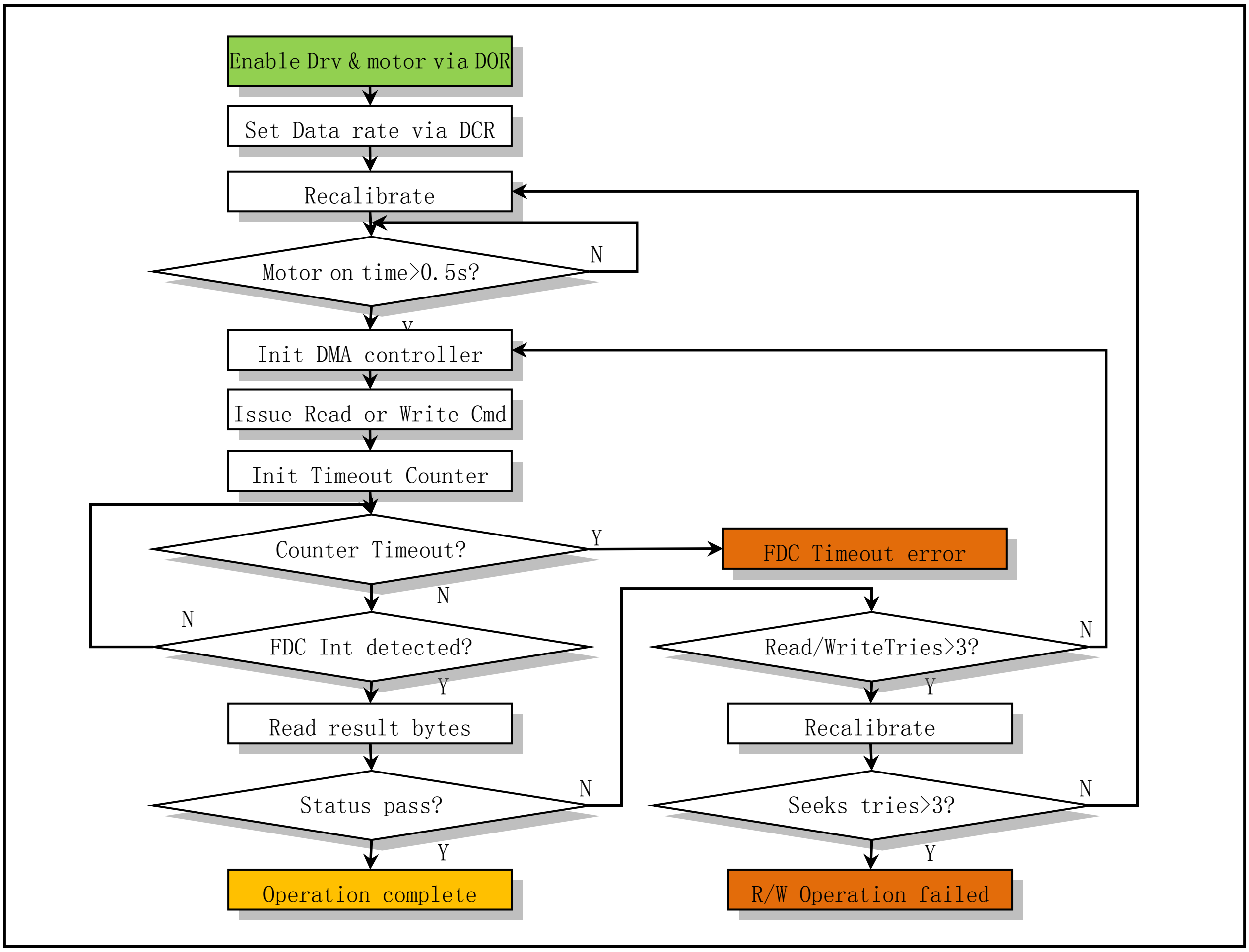


Figure 9-11 Data read/write operation flow diagram

The disk drive's motor must first reach normal operating speed before the disk can start transfering data. For most 3½-inch floppy drives, this boot time takes approximately 300ms, while the 5¼-inch floppy drive takes approximately 500ms. This startup delay time was set to 500ms in the floppy.c program.

After the motor is started, it is necessary to use the digital control register DCR to set the data transfer rate that matches the current floppy disk medium.

If the implicit seek mode is not enabled, then the seek command FD_SEEK needs to be sent to position the head on the correct track. After the seek operation is completed, the head also takes a period of loading time. For most drives, this delay takes at least 15ms. When the implicit seek mode is used, the head load time (HLT) specified by the "Specify Drive Parameters" command can be used to determine the minimum head arrival time. For example, in the case where the data transmission rate is 500 Kbps, if HLT = 8, the effective head in-position time is 16 ms. Of course, if the head is already in place on the correct track, it will not take this time.

Then the DMA controller is initialized and the read and write commands are executed. Usually, after the data transfer is completed, the DMA controller will issue a termination count (TC) signal, at which point the floppy disk controller will complete the current data transfer and issue an interrupt request signal indicating that the operation has reached the result stage. If an error occurs during operation or the last sector number equals the last sector of the track (EOT), the floppy disk controller will immediately enter the result phase.

According to the above flow diagram, if an error is found after reading the result status byte, the data read or write operation command is restarted by reinitializing the DMA controller. A continuous error usually indicates that the seek operation did not cause the head to reach the specified track, and the recalibration of the head should be repeated multiple times and the seek operation should be performed again. If there is still an error after that, the controller will report the read or write operation failure to the driver.

5. Disk format operations

Although the Linux 0.12 kernel does not implement the format operation of the floppy disk, as a reference, the disk formatting operation is briefly described here. The disk formatting operation involves positioning the heads on each track and creating a fixed format field for composing the data field.

After the motor has started and the correct data transfer rate is set, the head will return to zero track. At this point, the disk needs to reach a normal and stable operating speed within a 500ms delay time.

The identification field (ID field) established on the disk during the formatting operation is provided by the DMA controller during the execution phase. The DMA controller is initialized to provide values for the track (C), head (H), sector number (R), and sector bytes for each sector identification field. For example, for a disk with 9 sectors per track, each sector size is 2 (512 bytes). If the track 7 is formatted with head 1, the DMA controller should be programmed to transmit 36 bytes (9 sectors x 4 bytes per sector), the data fields should be: 7,1,1,2,7,1,2,2,7,1,3,2,...,7, 1,9,2. Because the data provided by the floppy disk controller is recorded directly on the disk as an identification field during the execution of the format command, the content of the data can be arbitrary. So some people use this feature to prevent protected disk copying.

After each head on one track has performed the formatting operation, it is necessary to perform a seek operation to advance the head to the next track and repeat the formatting operation. Because the Formatted Track command does not contain an implicit seek operation, the seek command SEEK must be used. Similarly, the head in-position time discussed above also needs to be set after each seek.

9.6.3.5 DMA Controller Programming

The primary purpose of the Direct Memory Access (DMA) controller is to enhance the system's data transfer performance by allowing external devices to transfer data directly to the memory. Usually it is implemented by the Intel 8237A chip or its compatible chip on the machine. By programming the DMA controller, the data transfer between the peripheral and the memory can be performed independently of the CPU. Therefore, the CPU can do other things during data transfer. The working process of the DMA controller to transfer data is as follows:

1．Initialization of the DMA controller.

The program initializes it through the DMA controller port, which includes: (1) sending control commands to the DMA controller; (2) sending memory start address for transfer; (3) sending data length. The command sent indicates whether the DMA channel used for the transfer, whether the memory is transferred to the peripheral (write) or the peripheral data is transferred to the memory, whether it is a single-byte transfer or a bulk (block) transfer. For PCs, the floppy disk controller is designated to use DMA channel 2. In the Linux 0.12 kernel, the floppy disk driver uses a single-byte transfer mode. Since the Intel 8237A chip has only 16 address pins (eight of which are used in conjunction with the data lines), only 64KB of memory space can be addressed. In order to allow it to access 1MB of address space, the PC uses a LS670 chip as a DMA page register, and divides 1MB of memory into 16 pages for operation, as shown in Table 9–25. Therefore, the transferred memory start address needs to be converted into the DMA page value and the offset address in the page, and the data length of each transmission cannot exceed 64 KB.

From this we can see that the transfer buffer we set in memory must be within 1MB address space. However, if the actual data buffer (such as the user buffer) is outside the 1MB space, we need to setup a temporary transfer buffer in the memory 1MB address area for use by the DMA, and copy the transfered data between the temporary buffer and the actual user buffer. This is exactly the way the Linux 0.12 kernel is used, see the function setup_DMA() (program floppy.c, line 106 and line 161).

Table 9-25 DRAM page corresponding memory address range

DMA page	Address range (64KB)
0x00	0x00000 - 0x0FFFF
0x01	0x10000 - 0x1FFFF
0x02	0x20000 - 0x2FFFF
0x03	0x30000 - 0x3FFFF
0x04	0x40000 - 0x4FFFF
0x05	0x50000 - 0x5FFFF
0x06	0x60000 - 0x6FFFF
0x07	0x70000 - 0x7FFFF
0x08	0x80000 - 0x8FFFF
0x09	0x90000 - 0x9FFFF
0x0A	0xA0000 - 0xAFFFF
0x0B	0xB0000 - 0xBFFFF
0x0C	0xC0000 - 0xCFFFF
0x0D	0xD0000 - 0xDFFFF
0x0E	0xE0000 - 0xEFFFF
0x0F	0xF0000 - 0xFFFFF

2． Data transmission

After the initialization is completed, the mask register of the DMA controller is modified, and DMA channel 2 is enabled, so that the DMA controller starts data transmission.

3． End of transmission

When all the data to be transferred is transferred, the DMA controller will generate an "End of Process" (EOP) signal to the floppy controller. At this point, the floppy disk controller can perform the end operation: turn off the drive motor and send an interrupt request signal to the CPU.

In the PC/AT machine, the DMA controller has 8 independent channels available, of which the last 4 channels are 16 bits. The floppy disk controller is designated to use DMA channel 2. You must set it first before using it. This involves operations on three ports: the page register port, the (offset) address register port, and the data count register port. Since the DMA register is 8-bit and the address and count values are 16 bits, they need to be sent twice. The low byte is sent first, then the high byte is sent. The port address corresponding to each channel is shown in Table 9–26.

Table 9-26 Page, address, and count register ports used by each DMA channel

DMA Channel	Page register	Base address register	Word count register
0	0x87	0x00	0x01
1	0x83	0x02	0x03
2	0x81	0x04	0x05
3	0x82	0x06	0x07

4	0x8F	0xC0	0xC2
5	0x8B	0xC4	0xC6
6	0x89	0xC8	0xCA
7	0x8A	0xCC	0xCE

For normal DMA applications, there are five common registers that control the operation and state of the DMA controller. These are the Command Register, the Request Register, the Single Mask Register, the Mode Register, and the Clear Pre/Post Pointer Trigger, as shown in Table 9–27. The Linux 0.12 kernel mainly uses three shaded register ports (0x0A, 0x0B, 0x0C) in the table.

Table 9-27 registers commonly used in DMA programming

Register Name	Read/Write	Port address	
		Channel 0 - 3	Channel 4 - 7
Status register/Command register	Read/Write	0x08	0xD0
Request register	Write	0x09	0xD2
Single Channel mask register	Write	0x0A	0xD4
Mode register	Write	0x0B	0xD6
Clear First/Last Flip-Flop	Write	0x0C	0xD8
Temporary register/Mater Clear	Read/Write	0x0D	0xDA
Clear Mask register	Write	0x0E	0xDC
Full Mask register	Write	0x0F	0xDE

Command Register -- The command register is used to specify the operational requirements of the DMA controller chip and set the overall state of the DMA controller. Usually it does not need to change after boot initialization. In the Linux 0.12 kernel, the floppy driver directly uses the ROM BIOS settings after booting. For reference, the meaning of the bits of the command register is listed here, as shown in Table 9–28. Note that when reading the same port, you will get the information of the DMA controller status register.

Table 9-28 DMA command register format

Bit	Description
7	DMA response peripheral signal DACK: 0-DACK active low; 1-DACK active high.
6	Peripheral request DMA signal DREQ: 0-DREQ active low; 1-DREQ active high.
5	Write mode selection: 0-select late write; 1-select extended write; X-if bit 3=1.
4	DMA channel priority mode: 0-fixed priority; 1-rotating priority.
3	DMA cycle selection: 0 - normal timing cycle (5); 1- compression timing cycle (3); X - if bit 0 = 1.
2	Start DMA controller: 0 - enable controller; 1 - disables the controller.
1	Channel 0 address hold: 0 - disable channel 0 address hold; 1 - allow channel 0 address to hold; X - if bit 0 = 0.
0	Memory transfer mode: 0 - disable memory to memory; 1 - enable memory to memory.

Request Register -- The 8237A can respond to requests for DMA service which are initiated by software as well as by a DREQ. The request register is used to record the request service signal DREQ of the peripheral to the channel, one bit for each channel. When DREQ is active, it corresponds to position 1, which is set to 0 when the DMA controller responds to it. If the DMA request signal DREQ pin is not used, the DMA controller's service can also be requested by directly setting the request bit of the corresponding channel by programming. In the PC, the floppy disk controller has a direct request signal DREQ connection to channel 2 of the DMA controller, so there is no need to operate the register in the Linux kernel. For reference, the byte format of the request channel service is listed here, as shown in Table 9-29.

Table 9-29 The meaning of each bit of the DMA request register

Bit	Description
7 -3	Not used.
2	Set flag. 0 - Request bit is set; 1 - Request bit is reset (set to 0).
1	Channel selection. 00-11 selects channel 0-3.
0	

Mask Register -- The port of the single mask register is 0x0A (0xD4 for 16-bit channels). A channel is masked, meaning that the DMA request signal DREQ issued by the peripheral using the channel does not get the response from the DMA controller, therefore, the DMA controller cannot be operated on the channel. Each channel has associated with it a mask bit which can be set to disable the incoming DREQ. The meaning of each bit of this register is shown in Table 9–30.

Table 9-30 The meaning of each bit of the DMA single mask register

Bit	Description
7 -3	Not used.
2	mask flag. 1 - set mask bit; 0 - Clear mask bit.
1	Channel selection. 00-11 selects channel 0-3.
0	

Mode Register -- The mode register is used to specify how a DMA channel operates. The meaning of each bit of this register is shown in Table 9-31. In the Linux 0.12 kernel, two setting modes, read disk (0x46) and write disk (0x4A), are used. According to Table 9-31, use DMA channel 2, read disk (write memory) transfer, disable autoinitialization, select address increment and use single byte mode; 0x4A means set mode: use DMA channel 2, write disk (read memory) transfer, disable autoinitialization, select address increment and use single byte mode.

Table 9-31 The meaning of each bit of the DMA mode register

Bit	Description
7	Select the transfer mode: 00-request mode; 01-single byte mode; 10-block byte mode; 11-cascade mode.
6	
5	Address method. 0 - address increment; 1 - address decrement.
4	Autoinitialization. 0-Autoinitialization disable; 1-Autoinitialization enable.
3	Transmission type: 00-verify transfer; 01-write memory transfer; 10-read memory transfer; 11- illegal; XX - if bits 6-7=11.
2	
1	Channel selection. 00-11 selects channel 0-3 respectively.
0	

Since the channel address and count registers can read and write 16-bit data, you need to perform two write operations, one low byte and one high byte at a time. Which byte is actually written is determined by the state of the software command clear first/last flip-flop. This command must be executed prior to writing or reading new address or word count information to the 8237A. This initializes the flip-flop to a known state so that subsequent accesses to register contents by the CPU will address upper and lower bytes in the correct sequence. Port 0x0C is used to initialize the byte-order flip-flop to the default state before reading or writing the address or count information in the DMA controller. When the clear byte first/last flip-flop is 0, the low byte is accessed; when the it is 1, the high byte is accessed. The flip-flop changes once per visit. The 0x0C port can be written to set the clear first/last flip-flop to the 0 state.

When using the DMA controller, it usually needs to follow a certain step. The following uses the DMA controller method to describe the DMA's brief programming steps:

1. Turn off the interrupt to eliminate any interference;
2. Modify the mask register (port 0x0A) to mask the DMA channel that needs to be used. For the floppy disk driver is channel 2;
3. Write to the 0x0C port and set "clear first/last flip-flop" to the default state;
4. Write mode register (port 0x0B) to set the operation mode word of the specified channel;
5. Write the address register (port 0x04) to set the offset address in the memory page used by the DMA. Write the low byte first, then write the high byte;
6. Write the page register (port 0x81) to set the memory page used by the DMA;
7. Write the count register (port 0x05), set the number of bytes for DMA transfer, which should be the transfer length -1. We also need to write once for the high and low bytes. In this book, the length of the floppy disk driver required by the DMA controller is 1024 bytes, so the length of the write DMA controller should be 1023 (ie 0x3FF);
8. Modify the mask register (port 0x0A) again to enable the DMA channel;
9. Finally, enable the interrupt to allow the floppy controller to issue an interrupt request to the system after the transfer is complete.

9.7 Summary

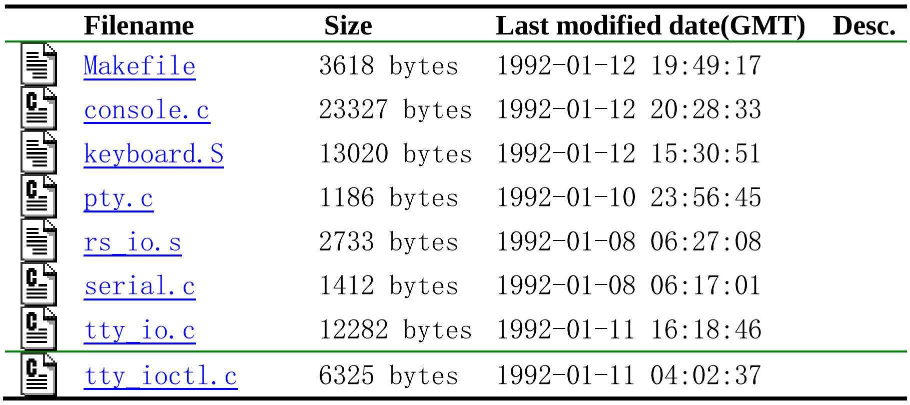
This chapter firstly introduces the main data structures such as request items and request queues used by the block device driver, and then introduces the general working method of the block device from the relationship between the system processor, the device controller and the specific block device. It is worth reminding again that the access to the block device data uses the interrupt handler method (except the virtual memory disk), after the device issues a read and write command, it can directly return to the calling program, and enter the non-interruptible sleep waiting queue, waiting for the block device operation to complete. In addition, the access of the upper layer program to the block device data is realized by a unified low-level block device read/write function ll_rw_block().

In the next chapter, let's learn about another different type of device: character devices, such as terminal console devices. It is one of the interactive device we communicate directly with the system.

10 Character Device Driver

In the Linux 0.12 kernel, character devices mainly include control terminal devices and serial terminal devices. The code in this chapter is used to operate the input and output of these devices. For the basic working principle of the terminal driver, we can refer to the book "The Design of the UNIX Operating System" by Mr. Maurice J. Bach or other related books. This chapter contains a total of seven source code files, as shown in Listing 10-1, two of which are programmed in assembly language. These files are located in the kernel/chr_drv/ directory.

List 10-1 linux/kernel/chr_drv



Filename	Size	Last modified date(GMT)	Desc.
Makefile	3618 bytes	1992-01-12 19:49:17	
console.c	23327 bytes	1992-01-12 20:28:33	
keyboard.S	13020 bytes	1992-01-12 15:30:51	
pty.c	1186 bytes	1992-01-10 23:56:45	
rs_io.s	2733 bytes	1992-01-08 06:27:08	
serial.c	1412 bytes	1992-01-08 06:17:01	
tty_io.c	12282 bytes	1992-01-11 16:18:46	
tty_ioctl.c	6325 bytes	1992-01-11 04:02:37	

10.1 Main Functions

The programs in this chapter can be divided into three parts. The first part is about the RS-232 serial line driver, including the programs rs_io.s and serial.c; The second part is about the console driver, which includes the keyboard interrupt driver keyboard.S and the console display driver console.c; The third part is the interface part between the terminal driver and the upper layer program, including the terminal input and output program tty_io.c and the terminal control program tty_ioctl.c. Below we first outline the basic principles of terminal control driver implementation, and then explain these basic functions in three parts. After that, each source file will be explained and annotated in detail.

10.1.1 Basic Principles of Terminal Drivers

The terminal driver is used to control the terminal device, transfer data between the terminal device and the process, and perform certain processing on the transmitted data. The raw data typed by the user on the keyboard is transferred to a receiving process after being processed by the terminal program. The data sent by the process to the terminal is displayed on the terminal screen after being processed by the terminal program, or transmitted to the remote terminal through the serial communication line. The terminal operating mode can be divided into two types according to the manner in which the terminal program treats input or output data. One is the canonical mode, in which the data passing through the terminal program will be transformed and then sent out. For example, the TAB character is expanded to 8 space characters, and the typed backspace is used to control the deletion of the previously typed characters. The processing functions used are generally referred to as line disciplines or line discipline modules. The other is a non-canonical mode or raw mode. In this mode, the line discipline program transfers data only between the terminal and the process without performing a canonical mode conversion on the data.

In the terminal driver, the code can be divided into a direct driver of the character device and an interface program directly connected to the upper layer according to their relationship with the device and the position in the execution flow. We can use Figure 10-1 to illustrate this control relationship.

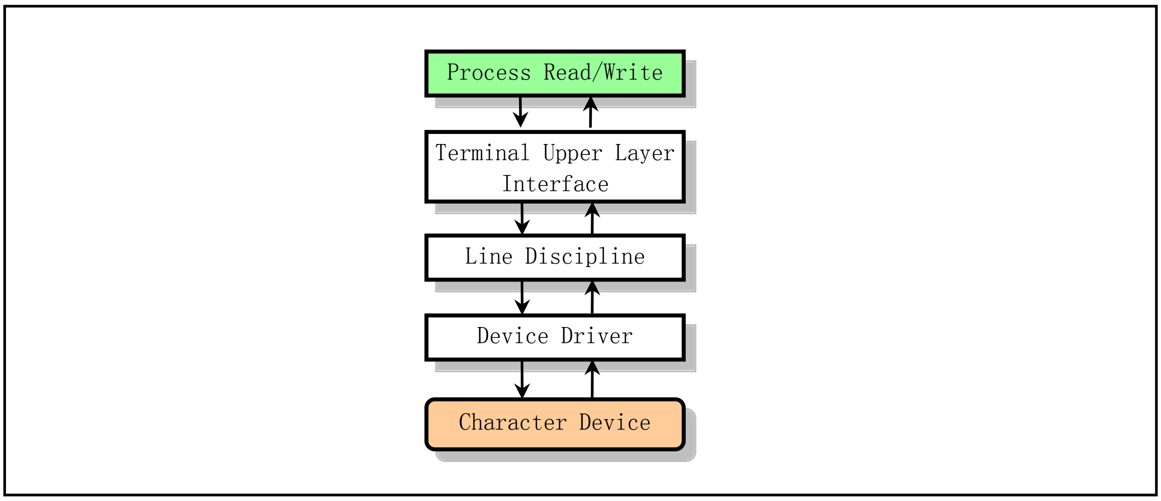


Figure 10-1 Terminal driver control flow

10.1.2 Types of Terminal Devices Supported by Linux

A terminal is a character type device that has many types. We usually use tty to refer to various types of terminal devices. tty is the abbreviation of Teletype, which is the earliest terminal device produced by Teletype, which looks like a teletypewriter. In the Linux 0.1x system device file directory /dev/, it usually contains the following terminal device files:

```
crw-rw-rw-   1 root     tty        5,   0 Jul 30  1992 tty        // Control terminal
crw--w--w-   1 root     tty        4,   0 Jul 30  1992 tty0       // virtual terminal alias.
crw--w--w-   1 root     tty        4,   1 Jul 30  1992 console    // Console
crw--w--w-   1 root     other      4,   1 Jul 30  1992 tty1       // virtual terminal 1
crw--w--w-   1 root     tty        4,   2 Jul 30  1992 tty2
crw--w--w-   1 root     tty        4,   3 Jul 30  1992 tty3
crw--w--w-   1 root     tty        4,   4 Jul 30  1992 tty4
crw--w--w-   1 root     tty        4,   5 Jul 30  1992 tty5
crw--w--w-   1 root     tty        4,   6 Jul 30  1992 tty6
crw--w--w-   1 root     tty        4,   7 Jul 30  1992 tty7
crw--w--w-   1 root     tty        4,   8 Jul 30  1992 tty8
crw-rw-rw-   1 root     tty        4,  64 Jul 30  1992 ttys1     // Serial port terminal 1
crw-rw-rw-   1 root     tty        4,  65 Jul 30  1992 ttys2
crw--w--w-   1 root     tty        4, 128 Jul 30  1992 ptyp0     // primary pseudo terminal.
crw--w--w-   1 root     tty        4, 129 Jul 30  1992 ptyp1
crw--w--w-   1 root     tty        4, 130 Jul 30  1992 ptyp2
crw--w--w-   1 root     tty        4, 131 Jul 30  1992 ptyp3
crw--w--w-   1 root     tty        4, 192 Jul 30  1992 ttyp0     // slave pty
```

```
crw--w--w-   1 root     tty        4, 193 Jul 30  1992 ttyp1
crw--w--w-   1 root     tty        4, 194 Jul 30  1992 ttyp2
crw--w--w-   1 root     tty        4, 195 Jul 30  1992 ttyp3
```

These terminal device files can be divided into the following types:

1. Serial port terminal (/dev/ttySn)

A serial port terminal is a terminal device that is connected using a computer serial port. The computer treats each serial port as a character device. For some time these serial port devices are often referred to as terminal devices because its primary use at that time was to connect to a terminal. The device file names corresponding to these serial ports are /dev/ttyS0, /dev/ttyS1, etc., and the device numbers are (4, 64), (4, 65), etc., respectively corresponding to COM1 and COM2 under the DOS system. To send data to a port, you can redirect the standard output to these special file names on the command line. For example, typing "echo test > /dev/ttyS1" at the command prompt will send the word "test" to the device connected to the ttyS1 port.

2. Pseudo terminal (/dev/ptyp, /dev/ttyp)

A pseudo terminal (or Pseudo - TTY, abbreviated as PTY) is a device that functions like a general terminal, but the device is not related to any terminal hardware. The pseudo terminal device is used to provide a terminal-like interface for other programs, and is mainly used to provide a terminal interface for the network server and the login shell program when logging in to the host through the network, or to provide a terminal style interface for the terminal program running in the X Window window. Of course, we can also use the pseudo terminal to establish a data read and write channel between any two programs that use the terminal interface. In order to provide terminal style interfaces for two applications or processes, the pseudo terminals are paired, and one is called a master terminal or a pseudo terminal master, and the other is called a slave terminal or a pseudo terminal slave. For paired pseudo-terminal logical devices like ptyp1 and ttyp1, ptyp1 is the master or control terminal, and ttyp1 is the slave. Data written to any of the pseudo terminals is received directly by the paired pseudo terminal through the kernel. For example, for the master device /dev/ptyp3 and the slave device /dev/ttyp3, if a program treats ttyp3 as a serial port device, its read/write operations on that port are reflected in the corresponding logical terminal device ptyp3, and ptyp3 is another logical device used by programs for read and write operations. In this way, two programs can communicate with each other through this logical device, and one of the programs using the slave device ttyp3 considers that it is communicating with a serial port. This is much like a pipeline operation between pairs of logical devices.

For a pseudo-terminal slave device, any program designed to use a serial port device can use the logic device, but for a program using the master device, it is specifically designed to use the pseudo-terminal master device. For example, if someone connects to your computer using a telnet program on the Internet, the telnet program may start to connect to the pseudo terminal master device ptyp2, and a getty program should run on the corresponding ttyp2 port. When telnet retrieves a character from the far end, the character is passed to the getty program via ptyp2 and ttyp2, and the getty program sends the "login:" string information to the network via the ttyp2, ptyp2, and telnet programs. In this way, the login program and the telnet program can communicate through the "pseudo terminal". By using the appropriate software, we can connect two or more pseudo terminal devices to the same physical port.

Previous Linux systems only had a maximum of 16 pairs of ttyp (ttyp0-ttypf) device file names, but nowadays Linux systems usually use the "prm-pty master" naming scheme, such as /dev/ptm3. Its corresponding end is automatically created as /dev/pts/3, so that a pty pseudo terminal can be dynamically provided when needed. The directory /dev/pts on current Linux systems is a file system of type devpts.

Although the "file" /dev/pts/3 appears to be one of the device file systems, it is actually a different file system.

3. Control terminal (/dev/tty)

The character device file /dev/tty is an alias for the Controlling Terminal. Its major device number is 5 and the minor device number is 0. If the current process has a control terminal, /dev/tty is the device file of the current process control terminal. We can use the command "ps -ax" to see which control terminal the process is connected to. For the login shell, /dev/tty is the terminal we use, and its device number is (5,0). We can use the command "tty" to see which actual terminal device it corresponds to. In fact /dev/tty is somewhat similar to a link to an actual terminal device.

If an terminal user executes a program but does not want the control terminal (such as a background server program), the process can try to open the /dev/tty file first. If the opening is successful, the process has a control terminal. At this point we can use the ioctl() call with the TIOCNOTTY (Terminal IO Control NO TTY) parameter to abandon the control terminal.

4. Console (/dev/ttyn, /dev/console)

In a Linux system, computer monitor is often referred to as console terminal or console. It emulates a VT200 or Linux type terminal (TERM=Linux) and has some character device files associated with it: tty0, tty1, tty2, and so on. When we log in on the console, we are using tty1. In addition, using Alt+[F1—F6], we can switch to tty2, tty3, etc. Tty1 – tty6 is called a virtual terminal, and tty0 is an alias for the currently used virtual terminal. Information generated by the Linux system is sent to tty0, so no matter which virtual terminal is currently being used, system information is sent to our screen.

You can log in to different virtual terminals, so you can have several different sessions at the same time. However, only the system or superuser root can write to /dev/tty0, and sometimes /dev/console will also connect to the terminal device. However, in Linux 0.12 systems, /dev/console is usually connected to the first virtual terminal tty1.

5. Other types of terminals

Today's Linux systems also have many other types of terminal device special files for many different character devices. For example, the /dev/ttyIn terminal device for ISDN devices. I won't go into details here.

10.1.3Terminal data structures

Each terminal device has a tty_struct data structure, which is mainly used to store information such as the current parameter settings of the terminal device, the foreground process group ID and the character IO buffer queue. This structure is defined in the include/linux/tty.h file and its structure is as follows:

```
struct tty_struct {
         struct termios termios;                  // Terminal io properties and control chars.
         int pgrp;                                // The process group to which it belongs.
         int stopped;
         void (*write)(struct tty_struct * tty);  // pointer to tty write function.
         struct tty_queue read_q;                 // tty read queue.
         struct tty_queue write_q;                // tty write queue.
         struct tty_queue secondary;              // tty auxiliary queue (or canonical queue).
};
extern struct tty_struct tty_table[];             // tty structure array.
```

The Linux kernel uses the array tty_table[] to store information about each terminal device in the system.

Each array item is a data structure tty_struct that corresponds to a terminal device in the system. The Linux 0.12 kernel supports a total of three terminal devices, one for the console device and the other two for the serial terminal devices that use the two serial ports on the system.

The termios structure is used to store the io attribute of the corresponding terminal device. A detailed description of the structure is described below. pgrp is the process group ID, which indicates the process group in the foreground in a session, that is, the process group that currently owns the terminal device. pgrp is mainly used for job control operations of processes. stopped is a flag indicating whether the corresponding terminal device has been discontinued. The function pointer *write() is the output processing function of the terminal device. The function pointer *write() is the output processing function of the terminal device. For the console terminal, it is responsible for driving the display hardware, displaying characters and other information on the screen, and for the serial terminal connected through the system serial port, it is responsible for outputting Characters are sent to the serial port.

The data processed by the terminal is stored in the character buffer queues (or character lists) of the three tty_queue structures. The structure is shown below:

```
struct tty_queue {
        unsigned long data;                  // Current data statistics.
                                             // Serial port address for serial terminals.
        unsigned long head;                  // The header of the data in the buffer.
        unsigned long tail;                  // The data tail pointer.
        struct task_struct * proc_list;      // Processes waiting for this buffer queue.
        char buf[1024];                      // The buffer of the queue.
};
```

The buffer length of each tty character queue is 1K bytes. The read buffer queue read_q is used to temporarily store the original character sequence input from the keyboard or the serial terminal; the write buffer queue write_q is used to store data written to the console display or the serial terminal; The auxiliary queue secondary is used to store the data taken from the read_q and processed (filtered) by the canonical mode program (based on the ICANON flag), and the data also referred to as cooked mode data. This is the canonical input data after the canonical program converts the special characters in the original data, such as the backspace character, and it is read and used by the program in the character line unit. The upper terminal read function tty_read() is used to read the characters in the secondary queue.

When reading in the data typed by the user, the interrupt handler is only responsible for putting the original character data into the input buffer queue, and the C function (copy_to_cooked()) called during the interrupt processing handles the character conversion. For example, when a process writes data to a terminal, the terminal driver calls the canonical mode function copy_to_cooked(), writes all the data in the user buffer to the buffer queue, and sends the data to the terminal for display. When a key is pressed on the terminal, the triggered keyboard interrupt handler puts the character corresponding to the key scan code into the read queue read_q, and calls the canonical mode handler to process the characters in read_q and put them into the auxiliary queue secondary. At the same time, if the echo flag (L_ECHO) of the terminal device is set, the character is also placed in the write queue write_q, and the terminal write function is called to display the character on the screen. Usually, the echo flag is set except for typing a password or other special requirements. We can change these flag values by modifying the information in the termios structure of the terminal.

The above tty_struct structure also includes a termios structure defined in the include/termios.h header file, the contents of which are as follows:

```
struct termios {
        unsigned long c_iflag;          /* input mode flags */
        unsigned long c_oflag;          /* output mode flags */
        unsigned long c_cflag;          /* control mode flags */
        unsigned long c_lflag;          /* local mode flags */
        unsigned char c_line;           /* line discipline */        // or speed
        unsigned char c_cc[NCCS];       /* control characters */
};
```

Where c_iflag is the input mode flag set. The Linux 0.12 kernel implements all 11 input flags defined by POSIX.1, as described in the termios.h header file. The terminal device driver uses these flags to control how the characters entered by the terminal are transformed (filtered). For example, do you need to convert the entered newline character (NL) into a carriage return (CR), whether you need to convert the input uppercase characters to lowercase characters (because some terminal devices can only input uppercase characters before). In the Linux 0.12 kernel, the associated handler is copy_to_cooked() in the tty_io.c file. See also lines 86 - 99 of the include/termios.h file.

c_oflag is the set of output mode flags. The terminal device driver uses these flags to control how characters are output to the terminal, primarily in the tty_write() function of tty_io.c. See lines 102-132 of the termios.h file.

c_cflag is a control mode flag set, which is mainly used to define serial terminal transmission characteristics, including baud rate, number of character bits, and number of stop bits. See lines 135 -- 166 in the termios.h file.

c_lflag is a local mode flag set, mainly used to control the driver's interaction with the user. For example, if you need to echo (Echo) characters, whether you need to display the deleted characters directly on the screen, and whether you need to make the control characters typed on the terminal generate signals. These operations are mainly used in the copy_to_cooked() function and tty_read().For example, if the ICANON flag is set, it indicates that the terminal is in the canonical mode input state, otherwise the terminal is in the non-canonical mode. If the ISIG flag is set, it means that the system needs to generate a corresponding signal when receiving the control characters INTR, QUIT, and SUSP issued by the terminal. See lines 169 -- 183 in the termios.h file.

The above four types of flag sets are all unsigned long, and each bit can represent a flag, so each flag set can have up to 32 input flags. All of these flags and their meanings can be found in the termios.h header file.

The c_cc[NCCS] array contains all the special characters that can be modified by the terminal. Where NCCS (=17) is the length value of the array. For example, you can modifying the break character (^C) to be generated by other keys. The default value of the terminal's c_cc[] array is defined in the include/linux/tty.h file. The array item symbol names are defined when the program references the items in the array. These names begin with the letter V, such as VINTR, VMIN. See lines 67-83 of the termios.h file.

Therefore, using the system call ioctl or using the correlation function (tcsetattr()), we can change the terminal's setting parameters by modifying the information in the termios structure. The canonical mode function operates on these setup parameters. For example, the control terminal should echo the typed characters, set the baud rate of the serial terminal transmission, clear the read buffer queue, and write the buffer queue.

When the user modifies the terminal parameters and resets the canonical mode flag, the terminal can be set to work in the raw mode. At this point, the canonical mode handler will transfer the data entered by the user to the user intact, and the carriage return is treated as a normal character. Therefore, when the user uses the system-call read, some decision-making scheme should be made to determine when the system-call read is completed and returned. This will be determined by the VTIME and VMIN control characters in the terminal termios structure. These two are the timeout timing values for the read operation. VMIN indicates the minimum number of characters to read in order to satisfy the read operation; VTIME is a read operation wait timing value.

We can use the command stty to view the settings of the flags in the current terminal device termios structure. Typing the stty command at the Linux 0.1x system command line prompt displays the following information:

```
[/root]# stty
---------Characters----------
INTR:   '^C'  QUIT:   '^\'  ERASE: '^H'  KILL:   '^U'  EOF:    '^D'
TIME:    0    MIN:     1    SWTC:   '^@'  START: '^Q'  STOP:   '^S'
SUSP:   '^Z'  EOL:    '^@'  EOL2:   '^@'  LNEXT: '^V'
DISCARD:  '^O'  REPRINT:  '^R'  RWERASE:  '^W'
----------Control Flags---------
-CSTOPB  CREAD -PARENB -PARODD  HUPCL -CLOCAL -CRTSCTS
Baud rate: 9600 Bits: CS8
----------Input Flags----------
-IGNBRK -BRKINT -IGNPAR -PARMRK -INPCK -ISTRIP  -INLCR -IGNCR
 ICRNL  -IUCLC   IXON   -IXANY   IXOFF -IMAXBEL
---------Output Flags---------
 OPOST -OLCUC  ONLCR -OCRNL -ONOCR -ONLRET -OFILL -OFDEL
Delay modes: CR0 NL0 TAB0 BS0 FF0 VT0
-----------Local Flags---------
 ISIG  ICANON -XCASE  ECHO -ECHOE -ECHOK -ECHONL -NOFLSH
-TOSTOP  ECHOCTL  ECHOPRT  ECHOKE -FLUSHO -PENDIN -IEXTEN
rows 0 cols 0
```

Among them, the minus sign indicates that there is no setting. Also for current Linux systems, you need to type 'stty -a' to display all of this information, and the display format is different.

The above main data structures used by the terminal program and the relationship between them can be seen in Figure 10-2.

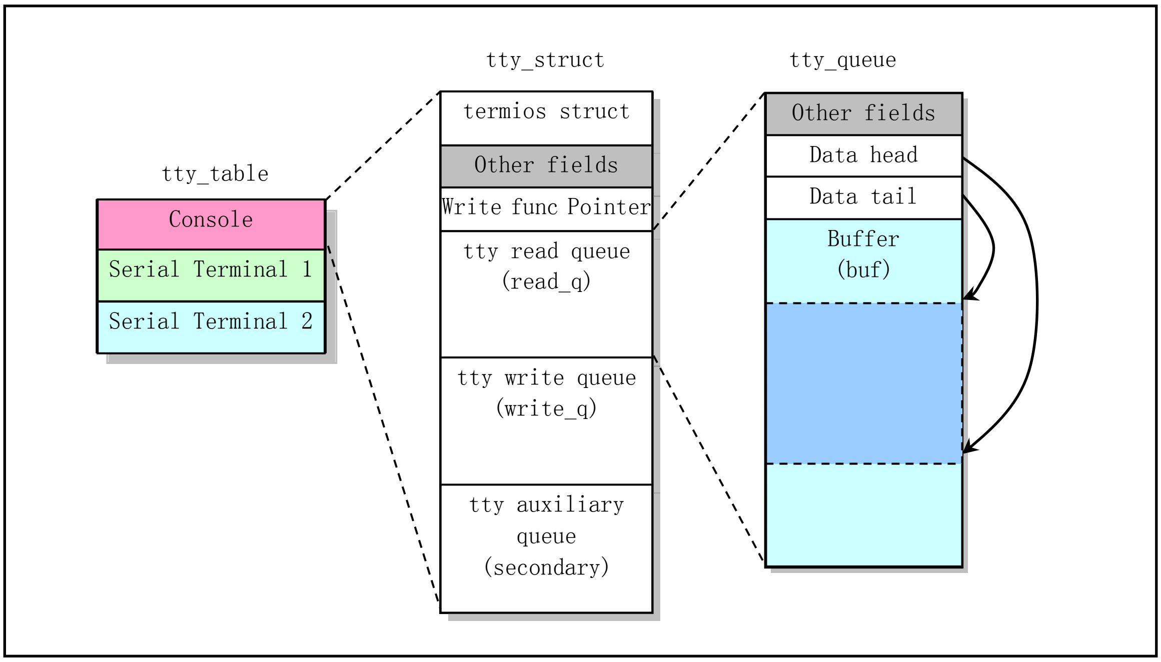


Figure 10-2 Data structures in the terminal program

10.1.4 **Canonical mode and non-canonical mode**

As can be seen from the foregoing, according to the way in which the terminal program processes the input or output data, we can divide the terminal working mode into a canonical mode and a non-canonical mode or a raw mode. For the canonical mode, the data passing through the terminal device will be modified and transformed according to relevant rules or standards, and then sent out. The transformation function used is often referred to as a line discipline or a line procedure module. For the non-canonical mode, the line discipline program only transfers data between the terminal and the process, and does not perform the transformation processing of the data. Below we detail the processing rules used in these two modes, as well as the implementation in the kernel code.

10.1.4.1 Canonical mode

When the ICANON flag in c_lflag is set, the terminal input data is processed in accordance with the canonical mode. The input characters are now assembled into lines, and the process reads as a line of characters. When a line of characters is entered, the terminal driver will return immediately. The delimiters for the line are NL, EOL, EOL2, and EOF. In addition to the last EOF (end of file) will be deleted by the handler, the remaining four characters will be returned to the caller as the last character of a line.

In canonical mode, the following characters entered by the terminal will be processed: ERASE, KILL, EOF, EOL, REPRINT, WERASE, and EOL2.

ERASE is a backspace. In canonical mode, when the copy_to_cooked() function encounters this character, the last character entered in the buffer queue is deleted. If the last character in the queue is the character of the previous line (for example, NL), no processing is done. This character is then ignored and not placed in the buffer queue.

KILL is a line delete character. It deletes the last line of characters in the queue. This character is then ignored.

EOF is the end of file. This character and the end-of-line characters EOL and EOL2 will be treated as carriage returns in the copy_to_cooked() function. The character encountered in the read operation function will return immediately. EOF characters are not placed in the queue but ignored.

REPRINT and WERASE are characters recognized in extended canonical mode. REPRINT will cause all unread inputs to be output. WERASE is used to remove words (skip whitespace characters). In Linux 0.12, the program ignores the recognition and processing of these two characters.

10.1.4.2 Non-Canonical mode

If the ICANON flag in the c_iflag set is in reset state, the terminal program operates in the non-canonical mode. At this time, the terminal program does not process the above characters, but treats them as ordinary characters, and the input data does not have the concept of a line. When the terminal program returns to the read process is determined by the values of MIN and TIME (VMIN, VTIME). These two variables are variables in the c_cc[] array. By modifying them you can change how the process reads characters in non-canonical mode.

MIN indicates the minimum number of characters to read for a read operation; TIME specifies the timeout value for which characters are waiting to be read (the unit of measure is 1/10th of a second). According to their values, they can be described in four cases.

1. MIN>0，TIME>0

   At this time TIME is a character interval timeout timing value, which will take effect after receiving the first character. If a MIN character is received before the timeout, the read operation returns immediately. If a timeout occurs before the MIN characters are received, the read operation returns the number of characters that have been received. At least one character can be returned at this time. Therefore, if the secondary is empty before receiving a character, the read process will be blocked (sleep).

2. MIN>0，TIME=0

   At this time, the read operation returns only when MIN characters are received. Otherwise, wait indefinitely (blocking).

3. MIN=0，TIME>0

   At this time TIME is a read operation timeout timing value. When a character is received or has timed out, the read operation returns immediately. If it is timed back, the read operation returns 0 characters.

4. MIN=0，TIME=0

   With this setting, if there is data in the queue that can be read, the read operation reads the number of characters required. Otherwise, it returns 0 characters immediately.

In the above four cases, MIN only indicates the minimum number of characters read. If the process requires more characters than MIN, then the current requirements of the process may be met as long as there are characters in the queue. For the read operation of the terminal device, see the tty_read() function in the program tty_io.c.

10.1.5 Console Terminal and Serial Terminal Devices

Two types of terminals can be used in the Linux 0.12 system, one is the console terminal on the host, and the other is the serial hardware terminal device. The console terminal is operated and managed by the keyboard interrupt handler program keyboard.s and the display control program console.c in the kernel. It receives the display characters or control messages passed by the upper tty_io.c program and controls the display of characters on the host screen. At the same time, the console (host) transfers the code generated by the keyboard to the tty_io.c program via keyboard.s. The serial terminal device is connected to the serial port of the computer through a line, and directly exchanges information with tty_io.c through the serial program rs_io.s in the kernel.

The two programs, keyboard.s and console.c, are actually much like emulation programs that use a monitor and keyboard to emulate a hardware terminal device in a Linux system host. Just because it is on the host, we call this simulated terminal environment a console terminal, or directly called a console. The functions implemented by these two programs are equivalent to the role of the terminal processing program in the ROM of a serial terminal device (except for the communication part), and also like a terminal emulation software on a normal PC. So, although the two programs are in the kernel, we can still look at them independently. The main difference between this "simulated terminal" and the ordinary hardware terminal device is that there is no need to communicate the driver through the serial line. Therefore, the keyboard.s and console.c programs must be able to simulate all the hardware processing functions of an actual terminal device (such as DEC's VT100 terminal), that is, all processing functions except the communication part in the terminal device firmware. See Figure 10-3 for the differences in processing structure and similarities between the console terminal and the serial terminal device. Therefore, if we have a certain understanding of the working principle of general hardware terminal equipment or terminal emulation program, then reading these two programs will not encounter any difficulties.

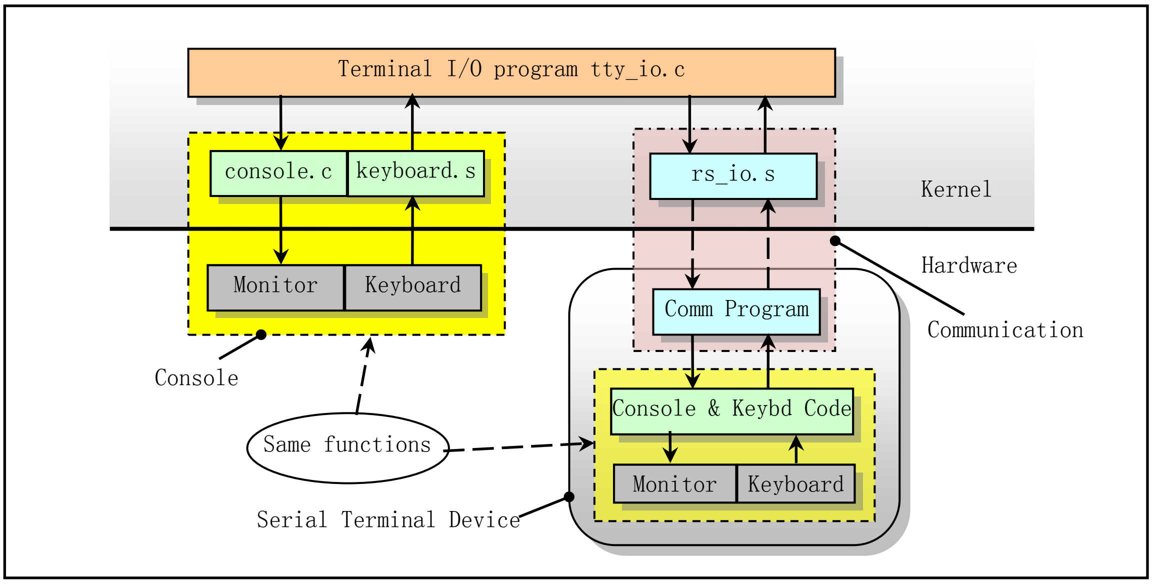


Figure 10-3 Console terminal and serial terminal device diagram

10.1.5.1 Console Driver

In the Linux 0.12 kernel, the terminal console driver involves the keyboard.S and console.c programs. keyboard.S is used to process the characters typed by the user, put them into the read buffer queue read_q, and call the copy_to_cooked() function to read the characters in read_q, and then convert them into the auxiliary buffer queue secondary. The console.c program implements the output display processing of the code received by the console terminal.

For example, when the user types a character on the keyboard, it causes a keyboard interrupt response (interrupt request signal IRQ1, corresponding to interrupt number INT 33) to execute the handler. At this point, the keyboard interrupt handler reads the corresponding keyboard scan code from the keyboard controller, and then translates the corresponding character according to the used keyboard scan code mapping table into the tty read queue read_q. Then call the C function do_tty_interrupt() of the interrupt handler, which directly calls the canonical mode function copy_to_cooked() to filter the character and put it into the tty auxiliary queue 'secondary', and put the character into the tty write queue write_q. Then call the write console function con_write() for console output (display) processing. At this time, if the echo attribute of the terminal is set, the character will be displayed on the screen. The do_tty_interrupt() and copy_to_cooked() functions are implemented in tty_io.c. The entire operation process is shown in Figure 10-4.

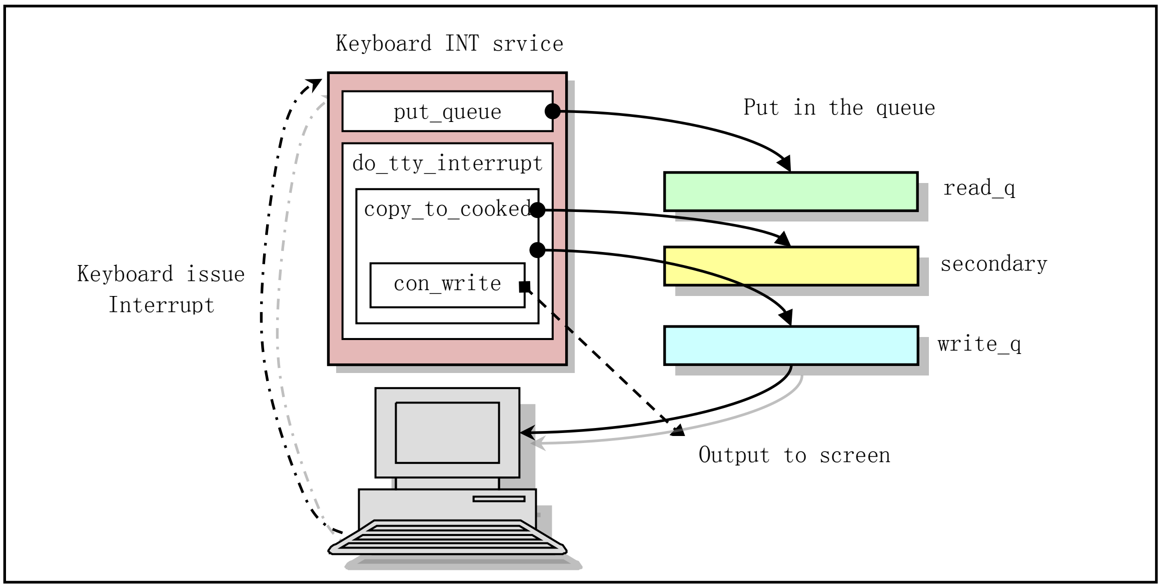


Figure 10-4 Console keyboard interrupt service

For the process to perform tty write operations, the terminal driver handles the characters one by one. When the write buffer queue write_q is not full, it takes a character from the user buffer and processes it and puts it in write_q. When the user data is all placed in the write_q queue or the write_q is full at this time, the write function specified in the terminal structure tty_struct is called, and the data in the write_q buffer queue is output to the console screen. For console terminals, the write function is con_write(), which is implemented in the console.c program.

Therefore, in general, the keyboard interrupt handler keyboard.S of the console terminal is mainly used to put the characters typed by the user into the read_q buffer queue. Its on-screen display handler, console.c, is used to retrieve characters from the write_q queue and display them on the screen. The relationship between all three character buffer queues and the above functions or files can be clearly shown in Figure 10-5.

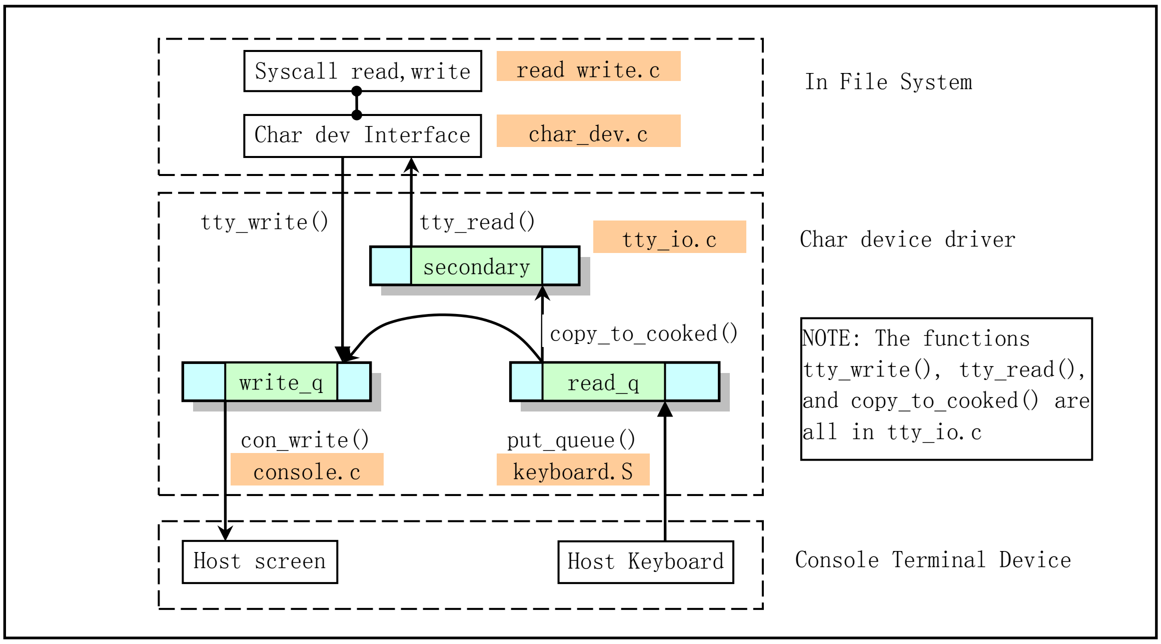


Figure 10-5 Relationships between console character queues and functions

10.1.5.2 Serial Terminal Driver

The programs that handle serial terminal operations are serial.c and rs_io.s. The serial.c program is responsible for initializing the serial port and enabling the serial interrupt transmit character operation by unmasking the transmit hold register empty interrupt. The rs_io.s program is a serial interrupt handler. It is mainly processed according to the four reasons for causing serial interrupts.

The serial interrupts caused by the system are: (1) due to changes in the modem state; (2) due to changes in line status; (3) due to received characters; (4) due to the requirement to send characters (The flag of transmitter holding register empty is set). The processing of the first two cases that cause an interrupt is reset by reading the corresponding status register. For the case of receiving a character, the program first needs to put the character into the read buffer queue read_q, and then call the copy_to_cooked() function to convert it into a canonical mode character in character line units and place it in the auxiliary queue 'secondary'. For the case where a character needs to be sent, the program first takes out a character from the location pointed by a tail pointer of the write buffer queue write_q and sends it out, and then checks whether the write buffer queue is empty, and if there are still have characters, it performs transmission operation cyclically.

For terminals that are accessed through the system serial port, in addition to processing similar to the console, input/output handling operations for serial communication are required. The reading of data is put into the read queue read_q by the serial interrupt handler, and then the same operation as the console terminal is performed.

For example, for a terminal connected to serial port 1, the typed characters will first be transmitted to the host over the serial line, causing host serial port 1 to issue an interrupt request. At this point, the serial port interrupt handler puts the character into the tty read queue read_q of serial terminal 1, and then calls the C function do_tty_interrupt() of the interrupt handler. The function will directly call the row rule function copy_to_cooked() to filter the characters and put them into the tty auxiliary queue secondary. At the same time, the character is placed in the tty write queue write_q, and the function rs_write() of the serial terminal 1 is called. The function, in turn, sends the character back to the serial terminal, and if the echo attribute of the terminal is set, the character is displayed on the screen of the serial terminal.

When the process needs to write data to a serial terminal, the operation process is similar to that of the write terminal, except that the write function in the tty_struct data structure of the terminal is the serial terminal write function rs_write().This function unmasks the "Transmit Holding Register Empty Enable Interrupt", causing a serial interrupt to occur when the transmit holding register is empty. The serial interrupt process takes a character from the write_q write buffer queue and puts it into the transmit holding register for character transmission according to the cause of the interrupt. The operation process also sends a character for one interrupt, and finally, when the write_q is empty, the transmit hold register empty enable interrupt bit is masked again, thereby preventing such an interrupt from being generated again.

The serial terminal write function rs_write() is implemented in the serial.c program, and the serial interrupt program is implemented in rs_io.s. See Figure 10-6 for the relationship between the three buffer queues of serial terminals and functions.

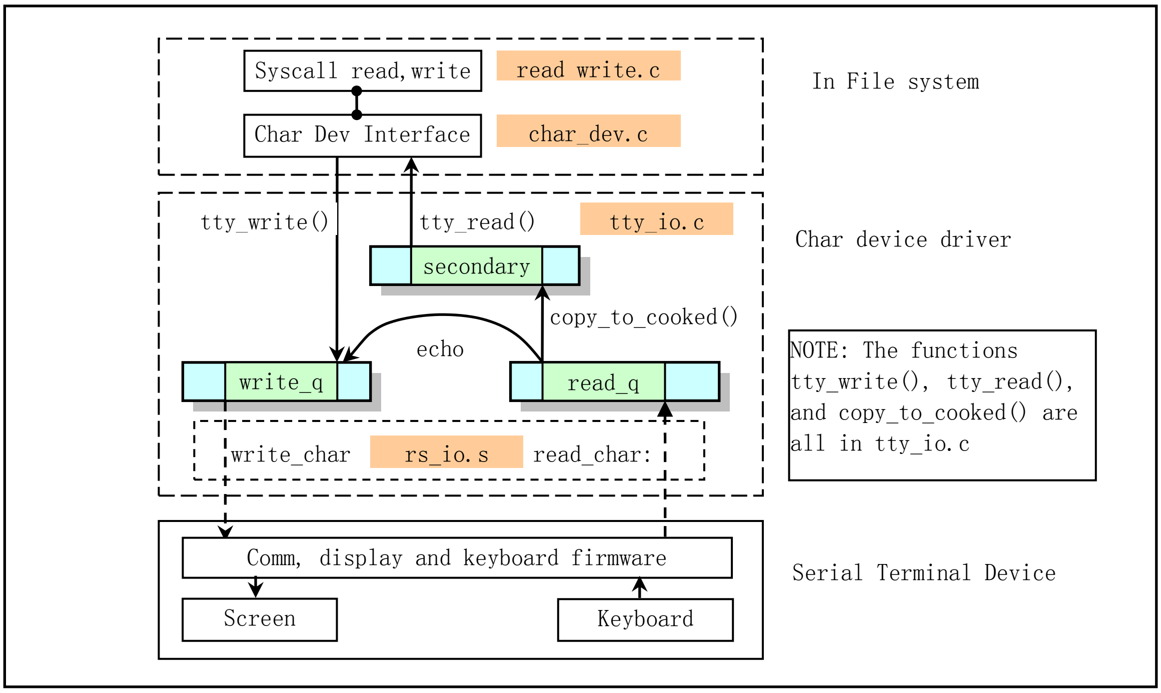


Figure 10-6 The relationship between character queues and functions of serial terminal devices

As can be seen from Figure 10-6, the main difference between the serial terminal and the console is that the serial terminal replaces the console operation display and keyboard program console.c and keyboard.S with the program rs_io.s, and the rest processing method is exactly the same.

10.1.6 Terminal Driver Interface

Typically, users interact with devices through the file system. Each device has a file name and, accordingly, an index node (i node) in the file system. But the file type in the i-node is the device type to distinguish it from other regular files. So users can access the device directly using file system calls. The terminal driver also provides a call interface function to the file system for this purpose. The interface between the terminal driver and other programs in the system is implemented using the generic functions in the tty_io.c file, which implements the read terminal function tty_read() and the write terminal function tty_write(), and the input line rule function copy_to_cooked().In addition, in the tty_ioctl.c program, an input/output control function (or system-call) tty_ioctl() that modifies terminal parameters is implemented. The setting parameters of the terminal are placed in the termios structure in the terminal data structure. There are many parameters and complex, please refer to the description in the include/termios.h file.

For different terminal devices, there can be different canonical mode programs to match. However, there is only one canonical mode function in Linux 0.12, so the row rule field 'c_line' in the termios structure does not work and is set to 0.

10.2 keyboard.S

10.2.1 Function descriptions

The keyboard.S program mainly includes a keyboard interrupt handler, and the related C function called is do_tty_interrupt().The program first sets the state flag variable 'mode' to be used later in the program according to the state of the special keys on the keyboard (for example, Alt, Shift, Ctrl, and Caps). Then, according to the key scan code that causes the keyboard interrupt, the corresponding scan code processing subroutine that has been arranged into the jump table is used to put the character associated with the scan code into the read character queue (read_q). Next, call the C function do_tty_interrupt() (tty_io.c, line 397), which contains only one call to the canonical function copy_to_cooked().The main purpose of this canonical mode function is to put the characters in the read_q read buffer queue into the normal mode queue (secondary queue) after proper processing, and if the corresponding terminal device sets the echo flag, then the program will also be place the character directly into the write queue (write_q), and the write tty function will be called to display the characters on the terminal screen.

For the scan code of the AT keyboard, when the key is pressed, the scan code of the key is sent, but when the key is released, two bytes will be sent, the first one is 0xf0, and the second is the same scan code as when the key pressed. For backward compatibility, the PC designer converted the scan code from the AT keyboard to the scan code of the old PC/XT standard keyboard, so the program only needs to process the PC/XT scan code. For a description of the keyboard scan code, see the description in section 10.3 of the program listing. Here is also a reminder that in keyboard-related terms, 'make' means that the key is pressed; 'break' means that the key is released.

In addition, the file name of this program is different from other gas assembly language programs, and its suffix is in uppercase '.S'. Using such a suffix allows the assembler to use the GNU C compiler's preprocessor CPP, which means that many C language directives can be used in your assembly language program. For example, "#include", "#if", etc., refer to the specific usage of the relevant code in the program.

10.2.2 **Code Annotation**

Program 10-1 linux/kernel/chr_drv/keyboard.S

```
/*
 *  linux/kernel/keyboard.S
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      Thanks to Alfred Leung for US keyboard patches
 *              Wolfgang Thiel for German keyboard patches
 *              Marc Corsini for the French keyboard
 */

/* KBD_FINNISH for Finnish keyboards
 * KBD_US for US-type
 * KBD_GR for German keyboards
 * KBD_FR for Frech keyboard
 */
// Define the type of keyboard used for later selection of character mapping code table.
#define KBD_FINNISH            // Linus is a Finnish man ☺

.text
// Declared as a global variable, used to set the keyboard interrupt descriptor at init.
.globl _keyboard_interrupt     // same as keyboard_interrupt().
```

```

/*
 * these are for the keyboard read functions
 */
// Size is the keyboard buffer queue length in bytes.
size    = 1024          /* must be a power of two ! And MUST be the same
                           as in tty_io.c !!!! */

// The following is the offset of each field in the keyboard buffer queue data structure tty_queue
// (include/linux/tty.h, line 22).
head = 4                // Offset of the head field in the tty_queue structure.
tail = 8                // Offset of the tail field.
proc_list = 12          // the wait process field offset (wait for this buffer).
buf = 16                // offset of the buffer field in the tty_queue structure.

// The three flag bytes (mode, leds, e0) are used in this program. The meaning of each bit in
// each byte flag is as follows:
// (1) 'mode' is the press status flag of the special keys. Used to indicate the state of the
// case conversion keys (caps), alternate keys (alt), control keys (ctrl), and shift keys.
//     bit 7 - caps key pressed;       bit 3 - right ctrl key pressed;
//     bit 6 - caps key status;        bit 2 - left ctrl key pressed;
//     bit 5 - right alt key pressed;  bit 1 - right shift key pressed;
//     bit 4 - left alt key pressed;   bit 0 - left shift key pressed.
// (2) 'leds' is a status flag for indicating the keyboard indicator, that is, the status of
// the LED tube indicating the num-lock, caps-lock, and scroll-lock.
//     bit 7-3 all zeros, not used.;   bit 1 num-lock (Initial setting 1);
//     bit 2 caps-lock;                bit 0 scroll-lock.
// (3) 'e0' This flag is set when the scan code is 0xe0 or 0xe1. It indicates that it is followed
// by a one or two character scan code. Usually, if the scan code 0xe0 is received, it means
// that there is still one character following it; if the scan code 0xe1 is received, it means
// that it is followed by 2 characters. See the description after the program list.
//     bit 1 = 1 Received the 0xe1;    bit 0 = 1 Received the 0xe0.
mode:   .byte 0         /* caps, alt, ctrl and shift mode */
leds:   .byte 2         /* num-lock, caps, scroll-lock mode (nom-lock on) */
e0:     .byte 0

/*
 *  con_int is the real interrupt routine that reads the
 *  keyboard scan-code and converts it into the appropriate
 *  ascii character(s).
 */                             // Note: The original comment above is obsolete.
//// Keyboard interrupt handler entry point.
// When the keyboard controller receives a key pressed operation of the user, it sends a interrupt
// request signal IRQ1 to the interrupt controller. This keyboard interrupt handler is executed
// when the CPU responds to the request. The interrupt handler reads the key scan code from
// the keyboard controller port (0x60) and calls the corresponding scan code subroutine for
// processing.
// First, read the scan code of the current button from port 0x60, and then determine whether
// the scan code is 0xe0 or 0xe1. If so, it immediately responds to the keyboard controller
// and sends an End of Interrupt (EOI) signal to the interrupt controller to allow the keyboard
// controller to continue generating interrupt signals, allowing us to receive subsequent
// characters. If the two special scan codes are not received, we invoke the corresponding key
// processing subroutine in the key jump table key_table according to the scan code value, and
```

```
   // put the character corresponding to the scan code into the read character buffer queue read_q.
   // Then, after responding to the keyboard controller and sending the EOI signal, the function
   // do_tty_interrupt() is called (actually calling copy_to_cooked()) to process the characters
   // in read_q and place them in the secondary auxiliary queue.
42 _keyboard_interrupt:
43         pushl %eax
44         pushl %ebx
45         pushl %ecx
46         pushl %edx
47         push %ds
48         push %es
49         movl $0x10,%eax         // Set ds and es to kernel data segment.
50         mov %ax,%ds
51         mov %ax,%es
52         movl _blankinterval,%eax
53         movl %eax,_blankcount   // preset black screen time count (blankinterval).
54         xorl %eax,%eax          /* %eax is scan code */
55         inb $0x60,%al           // Read scan code to al.
56         cmpb $0xe0,%al          // jump to set_e0 if scan code is 0xe0
57         je set_e0
58         cmpb $0xe1,%al          // jump to set_e1 if scan code is 0xe1
59         je set_e1
60         call key_table(,%eax,4) // key handler: key_table + eax*4 (see line 513).
61         movb $0,e0              // Reset e0 flag after return.

   // The following code (lines 62-72) performs a hardware reset operation on the PC standard
   // keyboard circuit using the 8255A. Port 0x61 is the address of the 8255A output port B. Bit
   // 7 (PB7) of this port is used to disable and enable handling of keyboard data. This program
   // is used to respond to the received scan code by first disabling the keyboard and then
   // immediately re-enabling the keyboard to work.
62 e0_e1:  inb $0x61,%al           // Get port B state, PB7 is used to enable/disable keyboard.
63         jmp 1f                  // delay for awhile.
64 1:      jmp 1f
65 1:      orb $0x80,%al           // set bit7 of al.
66         jmp 1f
67 1:      jmp 1f
68 1:      outb %al,$0x61          // set port B bit7 to disable keyboard.
69         jmp 1f
70 1:      jmp 1f
71 1:      andb $0x7F,%al          // reset bit7 of al.
72         outb %al,$0x61          // reset port B bit7 to enable keyboard.
73         movb $0x20,%al          // send EOI signal to 8259A chip.
74         outb %al,$0x20
75         pushl $0                // The console tty = 0, and pushed as a parameter.
76         call _do_tty_interrupt  // converted to canonical mode
77         addl $4,%esp            // Discard the parameters of the stack.
78         pop %es
79         pop %ds
80         popl %edx
81         popl %ecx
82         popl %ebx
83         popl %eax
84         iret
```

```
85 set_e0: movb $1,e0              // When 0xe0 is received, bit0 of e0 flag is set.
86         jmp e0_e1
87 set_e1: movb $2,e0              // When 0xe1 is received, bit1 of e0 flag is set.
88         jmp e0_e1
89
90 /*
91  * This routine fills the buffer with max 8 bytes, taken from
92  * %ebx:%eax. (%ebx is high). The bytes are written in the
93  * order %al,%ah,%eal,%eah,%bl,%bh ... until %eax is zero.
94  */
   // First, take the console's read buffer queue read_q address from the buffer queue address
   // table table_list (tty_io.c, line 81), then copy the characters in the al register to the
   // read queue header pointer and move the head pointer forward by 1 byte. If the head pointer
   // moves out of the end of the read buffer, let it wrap around to the beginning of the buffer,
   // and then see if the buffer queue is full at this time, that is, compare the queue head pointer
   // with the tail pointer (equal means full). If it is full, discard the remaining characters
   // that may be in ebx:eax. If the buffer is not full, transfer the data in ebx:eax to the right
   // by 8 bits (ie move the ah value to al, bl -> ◊ah, bh -> bl), and then repeat the above process
   // for al. Until all characters have been processed, the current head pointer value is saved,
   // and then check if there is a process waiting for the read queue, and if so, wake up.
95 put_queue:
96         pushl %ecx
97         pushl %edx                   // Get read buffer queue pointer.
98         movl _table_list,%edx         # read-queue for console
99         movl head(%edx),%ecx         // Take the queue head pointer -> ecx.
100 1:     movb %al,buf(%edx,%ecx)      // put char in al into the head pointer position.
101        incl %ecx                    // moved forward by 1 byte.
102        andl $size-1,%ecx            // Adjust the head pointer.
103        cmpl tail(%edx),%ecx          # buffer full - discard everything
104        je 3f
105        shrdl $8,%ebx,%eax           // Move ebx right 8 bits into eax, ebx is unchanged.
106        je 2f                        // Are there still characters? if no then jump.
107        shrl $8,%ebx                 // Move ebx value to the right by 8 bits
108        jmp 1b
109 2:     movl %ecx,head(%edx)         // If all chars in queue, the head pointer is saved.
110        movl proc_list(%edx),%ecx    // Waiting process pointer for this queue?
111        testl %ecx,%ecx              // Check if there are processes waiting for this queue.
112        je 3f                        // No, then jump (to line 114).
113        movl $0,(%ecx)               // Yes, the process is awake.
114 3:     popl %edx
115        popl %ecx
116        ret
117
   // From here on, it is the subroutine of each key corresponding to the pointer in the jump table
   // key_table, and called from the statement in line 60 above. The jump table key_table starts
   // at line 513.
   // The following code sets the corresponding bit in the mode flag according to the scan code
   // of ctrl or alt. If the 0xe0 scan code is received before the scan code (the e0 flag is set),
   // it means that the ctrl or alt key on the right side of the keyboard is pressed, and the bit
   // in the mode flag is set correspondingly to ctrl or alt.
118 ctrl:  movb $0x04,%al               // 0x04 (bit2), the left ctrl key in mode.
119        jmp 1f
120 alt:   movb $0x10,%al               // 0x10 (bit4), the left alt key in mode.
```

```
121 1:      cmpb $0,e0                 // e0 set? (key pressed is right ctrl/alt key)?
122         je 2f                      // no, jump (to line 124).
123         addb %al,%al               // yes, change to right-click flag (bit3 or bit5).
124 2:      orb %al,mode               // Set the corresponding bit in the mode flag.
125         ret
```

// This code handles the scan code when the ctrl or alt key is released, and resets the
// corresponding bit in the mode flag. In the operation, it is necessary to judge whether it
// is the ctrl or alt key on the right side of the keyboard according to whether the e0 flag
// is set.

```
126 unctrl: movb $0x04,%al             // bit2 is for the left ctrl key in mode.
127         jmp 1f
128 unalt:  movb $0x10,%al             // 0x10 is the bit 4 for the left alt key in mode.
129 1:      cmpb $0,e0                 // e0 set? (Is the right ctrl/alt key released?)?
130         je 2f                      // no, jump (to line 132)
131         addb %al,%al               // yes, then change to reset flag bit (bit3 or bit5).
132 2:      notb %al                   // Reset corresponding bit in mode flag.
133         andb %al,mode
134         ret
135
```

// This code handles the scan codes when the left or right shift keys are pressed or released
// to set and reset the corresponding bits in mode.

```
136 lshift:
137         orb $0x01,mode             // left shift key is pressed, set mode bit0.
138         ret
139 unlshift:
140         andb $0xfe,mode            // left shift key is released, reset mode bit 0.
141         ret
142 rshift:
143         orb $0x02,mode             // right shift key is pressed, set mode bit1.
144         ret
145 unrshift:
146         andb $0xfd,mode            // right shift key is released, reset mode bit1.
147         ret
148
```

// This code handles caps scan code. It is known by mode bit7 that the caps key is currently in the pressed state. If yes, return, otherwise flip the bit 6 (caps key pressed) of mode flag and the caps-lock bit (bit 2) of the leds flag, and set the caps key pressed flag bit (bit 7).

```
149 caps:   testb $0x80,mode           // bit7 set ? (caps pressed?)
150         jne 1f                     // yes, jump to line 169.
151         xorb $4,leds               // flip caps-lock(bit2) in leds flag.
152         xorb $0x40,mode            // flip caps state (bit6) in mode flag.
153         orb $0x80,mode             // set bit7 (caps pressed) in mode flag.
```

// This code turns the LED indicator on or off according to the leds flag.

```
154 set_leds:
155         call kb_wait               // Wait for controller input buffer to be empty.
156         movb $0xed,%al             /* set leds command */
157         outb %al,$0x60             // Send kbd command 0xed to port 0x60.
158         call kb_wait
159         movb leds,%al              // get leds flag as parameter.
160         outb %al,$0x60
161         ret
162 uncaps: andb $0x7f,mode            // caps released, then reset bit7 in mode flag.
163         ret
```

```
164 scroll:
165         testb $0x03,mode            // ctrl key pressed too?
166         je 1f                       // no, jump (to line 169).
167         call _show_mem              // yes, display memory info (mm/memory.c, line 457)
168         jmp 2f
169 1:      call _show_state            // no, show process state info (kernel/sched.c, 45)
170 2:      xorb $1,leds                // flip bit0 of leds if scroll key pressed.
171         jmp set_leds                // turn on or off LED indicator based on leds flag.
172 num:    xorb $2,leds                // flip bit1 of leds flag if num key pressed.
173         jmp set_leds                // turn on or off LED indicator based on leds flag.
174
175 /*
176  *  curosr-key/numeric keypad cursor keys are handled here.
177  *  checking for numeric keypad etc.
178  */
    // The following code first determines if the scan code is issued by the right numeric keypad
    // buttons on the keyboard. If not, exit the this subrotine. If the last key Del(0x53) on the
    // numeric keypad is pressed, then it is determined whether or not the "Ctrl-Alt-Del" key
    // combination is pressed. If the combination key is pressed, jump to the system reboot code.
    // Please refer to the XT keyboard scan code table given in section 10.2.3 after this program
    // list, or browse the first set of keyboard scan code table in the appendix. As can be seen
    // from the table, the scan code range of the keys on the numeric keypad are [0x47 - 0x53],
    // and when they are used as direction keys, they will have the preamble code e0.
179 cursor:
180         subb $0x47,%al          // If scan code not in range [0x47 - 0x53], it ret (198).
181         jb 1f                   // That is, scan code is not generated from numeric keypad.
182         cmpb $12,%al            // (0x53 - 0x47 = 12)
183         ja 1f
184         jne cur2                /* check for ctrl-alt-del */
    // If it is equal to 12, it indicates that the 'del' key has been pressed, so it continues to
    // check whether 'ctrl' and 'alt' are also pressed simultaneously.
185         testb $0x0c,mode        // ctrl key pressed? jump if not.
186         je cur2
187         testb $0x30,mode        // alt key pressed?
188         jne reboot              // yes, then jump to reboot system (line 594).

    // Next, the code determines whether the numeric keypad keys are used as direction (page up/dn ,
    // insert, delete, etc.) keys. If the pressed button is indeed on the keypad, and if the leading
    // scan code e0 is received at this time, it means that the button on the keypad is used as
    // the direction key. Then jump to handle the cursor movement or insert/delete button. If e0
    // is not set, first check if the num-lock LED is on. If it is not lit, also perform cursor
    // movement processing. However, if the num-lock light is on (indicating that the keypad is
    // used as a numeric key) and the shift key is also pressed, then we will treat the keypad key
    // at this time as a cursor movement operation.
189 cur2:   cmpb $0x01,e0           /* e0 forces cursor movement */  // e0 flag set ?
190         je cur                  // jump to cur if e0 is set.
191         testb $0x02,leds        /* not num-lock forces cursor */  // test leds flag.
192         je cur                  // if num's LED is on, do cursor movement operation.
193         testb $0x03,mode        /* shift forces cursor */
194         jne cur                 // if shift is pressed, also do cursor movement operation.

    // Finally, the numeric keypad keys are used as numeric keys. Then, the small number table
    // num_table is queried according to the scan code, and the numeric characters corresponding
    // to the keys are taken out and placed in the buffer queue. Since the number of characters
    // placed in the character queue at a time requires <=8, it is necessary to clear ebx before
    // jumping to put_queue to execute. See the comment on line 95.
195         xorl %ebx,%ebx
196         movb num_table(%eax),%al      // obtain the digital char with eax as index.
197         jmp put_queue                 // parameter: 1-8 chars in ebx:eax.
198 1:      ret
199
    // This code handles cursor movement or insert/delete buttons. Here, the representative
    // character of the corresponding key in the cursor character table is first obtained. If the
    // character is <='9' (5, 6, 2 or 3), this means that it is one of a Page Up, Page Dn, Insert,
    // or Delete key, and the character '~' is need to be added to the function character sequence.
    // However, this kernel does not recognize and process them. The code then moves the contents
    // of ax into the eax high word, putting 'esc[' into ax, and with the characters in the high
    // word of eax to form a moving sequence. Finally, the character sequence is placed in the
    // character queue.
200 cur:    movb cur_table(%eax),%al  // get cursor char to al.
201         cmpb $'9,%al              // if <='9' (5,6,2 or 3) it's page up/dn or Ins/Del key
202         ja ok_cur                 // it needs to add char '~'.
203         movb $'~,%ah
204 ok_cur: shll $16,%eax             // move content of ax to high word of eax.
205         movw $0x5b1b,%ax          // put 'esc [' in ax
206         xorl %ebx,%ebx
207         jmp put_queue
208
209 #if defined(KBD_FR)
210 num_table:
211         .ascii "789 456 1230."      // ASCII code associated to keys on numeric keypad.
212 #else
213 num_table:
214         .ascii "789 456 1230,"
215 #endif
216 cur_table:
217         .ascii "HA5 DGC YB623"      // associated to the arrow, Ins, Del keys on keypad.
218
219 /*
220  * this routine handles function keys
221  */
    // The subroutine converts the function key scan code into an escape character sequence and
    // stores it in the read queue. The range of function keys to be checked by the code is F1--F12.
    // The scan codes of the function keys F1--F10 are 0x3B-0x44, and the scan codes of F11 and
    // F12 are 0x57, 0x58. First, F1--F12 is first converted into the corresponding sequence number
    // 0--11, and then the function key table func_table is queried to obtain the corresponding
    // escape character sequence and placed in the character queue. If the Alt key is pressed while
    // a function key is pressed, only the switching operation of the control terminal is performed.
222 func:
223         subb $0x3B,%al              // F1 scan code is 0x3B
224         jb end_func                 // ret if not a function key
225         cmpb $9,%al                 // F1--F10 ?
226         jbe ok_func                 // jump if yes
227         subb $18,%al                // key F11, F12 ?
228         cmpb $10,%al                // key F11 ?
229         jb end_func                 // ret if order no less than F11's.
```

```
        cmpb $11,%al                // key F12 ?
        ja end_func                 // ret if order no great than F12's.
ok_func:
        testb $0x10,mode            // left alt pressed ?
        jne alt_func                // jump to change console if yes.
        cmpl $4,%ecx                /* check that there is enough room */
        jl end_func                 // ret if not enough space (4 bytes).
        movl func_table(,%eax,4),%eax       // get escape character sequence of the key.
        xorl %ebx,%ebx
        jmp put_queue
// The following code handles the alt + Fn key combination to change the virtual control
// terminal. At this time, eax is the function key index or order number (F1 -- 0), corresponding
// to the virtual control terminal number.
alt_func:
        pushl %eax                  // push the index number as virtual console number.
        call _change_console        // chr_dev/tty_io.c, line 87.
        popl %eax                   // discard the parameter.
end_func:
        ret

/*
 * function keys send F1:'esc [ [ A' F2:'esc [ [ B' etc.
 */
func_table:
        .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
        .long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
        .long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b

// Scan code - ASCII character mapping table.
// The scan code of the corresponding key is mapped to ASCII characters according to the keyboard
// type defined previously (FINNISH, US, GERMEN, FRANCH).
#if     defined(KBD_FINNISH)
key_map:
        .byte 0,27               // for scan code 0x00, 0x01;
        .ascii "1234567890+'"    // for scan code 0x02,...0x0c,0x0d. Bellow is similar.
        .byte 127,9
        .ascii "qwertyuiop}"
        .byte 0,13,0
        .ascii "asdfghjkl|{"
        .byte 0,0
        .ascii "'zxcvbnm,.-"
        .byte 0,'*,0,32          /* 36-39 */
        .fill 16,1,0             /* 3A-49 */
        .byte '-,0,0,0,'+        /* 4A-4E */
        .byte 0,0,0,0,0,0,0      /* 4F-55 */
        .byte '<
        .fill 10,1,0

shift_map:                       // mapping table when shift is pressed at the same time.
        .byte 0,27
        .ascii "!\"#$%&/()=?`"
        .byte 127,9
        .ascii "QWERTYUIOP]^"
        .byte 13,0
        .ascii "ASDFGHJKL\\["
        .byte 0,0
        .ascii "*ZXCVBNM;:_"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '>
        .fill 10,1,0

alt_map:                        // mapping table when alt key is pressed at the same time.
        .byte 0,0
        .ascii "\0@\0$\0\0{[]}\\\0"
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte '~,13,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0,0,0           /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte 0,0,0,0,0         /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '|
        .fill 10,1,0

#elif defined(KBD_US)           // mapping table for us keyboard.

key_map:
        .byte 0,27
        .ascii "1234567890-="
        .byte 127,9
        .ascii "qwertyuiop[]"
        .byte 13,0
        .ascii "asdfghjkl;'"
        .byte '`,0
        .ascii "\\zxcvbnm,./"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '<
        .fill 10,1,0


shift_map:
        .byte 0,27
        .ascii "!@#$%^&*()_+"
        .byte 127,9
        .ascii "QWERTYUIOP{}"
        .byte 13,0
        .ascii "ASDFGHJKL:\""
        .byte '~,0
        .ascii "|ZXCVBNM<>?"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '>
        .fill 10,1,0

alt_map:
        .byte 0,0
        .ascii "\0@\0$\0\0{[]}\\\0"
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte '~,13,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0,0,0           /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte 0,0,0,0,0         /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '|
        .fill 10,1,0

#elif defined(KBD_GR)           // mapping table for german keyboard.

key_map:
        .byte 0,27
        .ascii "1234567890\\'"
        .byte 127,9
        .ascii "qwertzuiop@+"
        .byte 13,0
        .ascii "asdfghjkl[]^"
        .byte 0,'#
        .ascii "yxcvbnm,.-"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '<
        .fill 10,1,0


shift_map:
        .byte 0,27
        .ascii "!\"#$%&/()=?`"
        .byte 127,9
        .ascii "QWERTZUIOP\\*"
        .byte 13,0
        .ascii "ASDFGHJKL{}~"
        .byte 0,''
        .ascii "YXCVBNM;:_"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '>
        .fill 10,1,0

alt_map:
        .byte 0,0
        .ascii "\0@\0$\0\0{[]}\\\0"
        .byte 0,0
        .byte '@,0,0,0,0,0,0,0,0,0,0
        .byte '~,13,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0,0,0           /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte 0,0,0,0,0         /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '|
        .fill 10,1,0


#elif defined(KBD_FR)           // mapping table for France keyboard.

key_map:
        .byte 0,27
        .ascii "&{\"'(-}_/@)="
        .byte 127,9
        .ascii "azertyuiop^$"
        .byte 13,0
        .ascii "qsdfghjklm|"
        .byte '`,0,42           /* coin sup gauche, don't know, [*|mu] */
        .ascii "wxcvbn,;:!"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '<
        .fill 10,1,0

shift_map:
        .byte 0,27
        .ascii "1234567890]+"
        .byte 127,9
        .ascii "AZERTYUIOP<>"
        .byte 13,0
        .ascii "QSDFGHJKLM%"
        .byte '~,0,'#
        .ascii "WXCVBN?./\\"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
```

```
436         .byte '-,0,0,0,'+       /* 4A-4E */
437         .byte 0,0,0,0,0,0,0     /* 4F-55 */
438         .byte '>
439         .fill 10,1,0
440
441 alt_map:
442         .byte 0,0
443         .ascii "\0~#{[|`\\^@]}"
444         .byte 0,0
445         .byte '@,0,0,0,0,0,0,0,0,0,0
446         .byte '~,13,0
447         .byte 0,0,0,0,0,0,0,0,0,0,0
448         .byte 0,0
449         .byte 0,0,0,0,0,0,0,0,0,0,0
450         .byte 0,0,0,0           /* 36-39 */
451         .fill 16,1,0            /* 3A-49 */
452         .byte 0,0,0,0,0         /* 4A-4E */
453         .byte 0,0,0,0,0,0,0     /* 4F-55 */
454         .byte '|
455         .fill 10,1,0
456
457 #else
458 #error "KBD-type not defined"
459 #endif
460 /*
461  * do_self handles "normal" keys, ie keys that don't change meaning
462  * and which have just one character returns.
463  */
    // The code first selects a corresponding character mapping table (alt_map, shift_map or key_map)
    // according to the mode flag, and then searches the mapping table according to the scan code
    // of the button to obtain the corresponding character (ASCII code). Then, according to whether
    // the current character is pressed with the ctrl or alt button at the same time and the character
    // ASCII code value, a certain conversion is performed. Finally, the converted result characters
    // are stored in the read buffer queue.
    // The following code first selects one of the alt_map, shift_map, or key_map mapping tables
    // based on the mode flag.
464 do_self:
465         lea alt_map,%ebx            // alt_map table address -> ebx (use alt_map table).
466         testb $0x20,mode            /* alt-gr */  // right alt key pressed ?
467         jne 1f                      // jump to label 1 if yes.
468         lea shift_map,%ebx          // otherwise, use shift_map table.
469         testb $0x03,mode            // any shift key pressed ?
470         jne 1f                      // jump to label 1 if yes.
471         lea key_map,%ebx            // otherwise, use key_map table.

    // Now you have chosen the mapping table to use. Next, the corresponding character in the mapping
    // table is obtained according to the scan code value. If there is no corresponding character
    // in the mapping table, it returns (turns to none, line 495).
472 1:      movb (%ebx,%eax),%al        // get the corresponding char from mapping table.
473         orb %al,%al                 // is it a non-zero ASCII code char ?
474         je none                     // jump to none and return if the code is zero.

    // If the ctrl key is pressed or the caps key is locked and the character is in the range 'a'--'}'
    // [0x61--0x7D], it is converted to the corresponding character in range of [0x41--0x5D]
    // by decrementing 0x20. Otherwise jump to label 2 to continue execution.
475         testb $0x4c,mode              /* ctrl or caps */   // ctrl pressed or caps lock ?
476         je 2f                         // jump to label 2 if not.
477         cmpb $'a,%al                  // check to see if the char is in range of 'a'-- '}'
478         jb 2f                         // jum to label 2 if not.
479         cmpb $'},%al
480         ja 2f
481         subb $32,%al                  // The char ASCII code value is decremented by 0x20.

    // If the ctrl key has been pressed and the character is within the range '@'--'_' [0x40--0x5F],
    // it is subtracted from 0x40 and converted to a value range [0x00--0x1F]. The characters in
    // this range are control characters. This means that characters in the range ctrl + ['@'--'_']
    // can produce corresponding control characters in the range [0x00-0x1F]. For example, pressing
    // ctrl + 'M' will generate the corresponding carriage return control character.
482 2:      testb $0x0c,mode              /* ctrl */   // ctrl key pressed ?
483         je 3f                         // jump forward to label 3 if not.
484         cmpb $64,%al                  // check to see if char is in range [0x40--0x5F].
485         jb 3f                         // jump to label 3 if not.
486         cmpb $64+32,%al
487         jae 3f
488         subb $64,%al                  // convert to control char [0x00--0x1f].

    // If the left alt key is pressed simultaneously, bit 7 of the character is set. That is,
    // characters in the extended character set whose value is greater than 0x7f can be generated
    // at this time.
489 3:      testb $0x10,mode              /* left alt */  // left alt pressed ?
490         je 4f                         // jump to label 4 if not
491         orb $0x80,%al                 // set bit 7 of the char.

    // Finally, we put the character in al into the read buffer queue.
492 4:      andl $0xff,%eax               // only one character
493         xorl %ebx,%ebx
494         call put_queue
495 none:   ret
496
497 /*
498  * minus has a routine of it's own, as a 'E0h' before
499  * the scan code for minus means that the numeric keypad
500  * slash was pushed.
501  */
    // Note that for Finnish and German keyboards, scan code 0x35 corresponds to the '-' key.
    // See lines 264 and 365.
502 minus:  cmpb $1,e0                    // e0 flag is set to 0x01 ? (e0 received ?)
503         jne do_self                   // jump to normal processing of char if no.
504         movl $'/,%eax                 // otherwise replace '-' with '/'
505         xorl %ebx,%ebx
506         jmp put_queue
507
508 /*
509  * This table decides which routine to call when a scan-code has been
510  * gotten. Most routines just call do_self, or none, depending if
511  * they are make or break.
```

```
 */      // Note that 'make' means a key is pressed, and 'break' means released.
key_table:
        .long none,do_self,do_self,do_self      /* 00-03 s0 esc 1 2 */
        .long do_self,do_self,do_self,do_self   /* 04-07 3 4 5 6 */
        .long do_self,do_self,do_self,do_self   /* 08-0B 7 8 9 0 */
        .long do_self,do_self,do_self,do_self   /* 0C-0F + ' bs tab */
        .long do_self,do_self,do_self,do_self   /* 10-13 q w e r */
        .long do_self,do_self,do_self,do_self   /* 14-17 t y u i */
        .long do_self,do_self,do_self,do_self   /* 18-1B o p } ^ */
        .long do_self,ctrl,do_self,do_self      /* 1C-1F enter ctrl a s */
        .long do_self,do_self,do_self,do_self   /* 20-23 d f g h */
        .long do_self,do_self,do_self,do_self   /* 24-27 j k l | */
        .long do_self,do_self,lshift,do_self    /* 28-2B { para lshift , */
        .long do_self,do_self,do_self,do_self   /* 2C-2F z x c v */
        .long do_self,do_self,do_self,do_self   /* 30-33 b n m , */
        .long do_self,minus,rshift,do_self      /* 34-37 . - rshift * */
        .long alt,do_self,caps,func             /* 38-3B alt sp caps f1 */
        .long func,func,func,func               /* 3C-3F f2 f3 f4 f5 */
        .long func,func,func,func               /* 40-43 f6 f7 f8 f9 */
        .long func,num,scroll,cursor            /* 44-47 f10 num scr home */
        .long cursor,cursor,do_self,cursor      /* 48-4B up pgup - left */
        .long cursor,cursor,do_self,cursor      /* 4C-4F n5 right + end */
        .long cursor,cursor,cursor,cursor       /* 50-53 dn pgdn ins del */
        .long none,none,do_self,func            /* 54-57 sysreq ? < f11 */
        .long func,none,none,none               /* 58-5B f12 ? ? ? */
        .long none,none,none,none               /* 5C-5F ? ? ? ? */
        .long none,none,none,none               /* 60-63 ? ? ? ? */
        .long none,none,none,none               /* 64-67 ? ? ? ? */
        .long none,none,none,none               /* 68-6B ? ? ? ? */
        .long none,none,none,none               /* 6C-6F ? ? ? ? */
        .long none,none,none,none               /* 70-73 ? ? ? ? */
        .long none,none,none,none               /* 74-77 ? ? ? ? */
        .long none,none,none,none               /* 78-7B ? ? ? ? */
        .long none,none,none,none               /* 7C-7F ? ? ? ? */
        .long none,none,none,none               /* 80-83 ? br br br */
        .long none,none,none,none               /* 84-87 br br br br */
        .long none,none,none,none               /* 88-8B br br br br */
        .long none,none,none,none               /* 8C-8F br br br br */
        .long none,none,none,none               /* 90-93 br br br br */
        .long none,none,none,none               /* 94-97 br br br br */
        .long none,none,none,none               /* 98-9B br br br br */
        .long none,unctrl,none,none             /* 9C-9F br unctrl br br */
        .long none,none,none,none               /* A0-A3 br br br br */
        .long none,none,none,none               /* A4-A7 br br br br */
        .long none,none,unlshift,none           /* A8-AB br br unlshift br */
        .long none,none,none,none               /* AC-AF br br br br */
        .long none,none,none,none               /* B0-B3 br br br br */
        .long none,none,unrshift,none           /* B4-B7 br br unrshift br */
        .long unalt,none,uncaps,none            /* B8-BB unalt br uncaps br */
        .long none,none,none,none               /* BC-BF br br br br */
        .long none,none,none,none               /* C0-C3 br br br br */
        .long none,none,none,none               /* C4-C7 br br br br */
        .long none,none,none,none               /* C8-CB br br br br */
        .long none,none,none,none               /* CC-CF br br br br */
        .long none,none,none,none               /* D0-D3 br br br br */
        .long none,none,none,none               /* D4-D7 br br br br */
        .long none,none,none,none               /* D8-DB br ? ? ? */
        .long none,none,none,none               /* DC-DF ? ? ? ? */
        .long none,none,none,none               /* E0-E3 e0 e1 ? ? */
        .long none,none,none,none               /* E4-E7 ? ? ? ? */
        .long none,none,none,none               /* E8-EB ? ? ? ? */
        .long none,none,none,none               /* EC-EF ? ? ? ? */
        .long none,none,none,none               /* F0-F3 ? ? ? ? */
        .long none,none,none,none               /* F4-F7 ? ? ? ? */
        .long none,none,none,none               /* F8-FB ? ? ? ? */
        .long none,none,none,none               /* FC-FF ? ? ? ? */

/*
 * kb_wait waits for the keyboard controller buffer to empty.
 * there is no timeout - if the buffer doesn't empty, we hang.
 */
kb_wait:
        pushl %eax
1:      inb $0x64,%al                 // read status of kbd controller.
        testb $0x02,%al               // test if input buffer is empty (0)
        jne 1b                        // jump to label 1 if not empty.
        popl %eax
        ret
/*
 * This routine reboots the machine by asking the keyboard
 * controller to pulse the reset-line low.
 */
// This subroutine writes the value 0x1234 to physical memory address 0x472. This location is
// the reboot mode flag. During the boot process, the ROM BIOS reads the reboot mode flag and
// directs the next execution based on its value. If the value is 0x1234, the BIOS will skip
// the memory detection process and perform the warm-boot process. If the value is 0, a cold-boot
// process is performed.
reboot:
        call kb_wait                  // wait controller buffer to empty.
        movw $0x1234,0x472            /* don't do memory check */
        movb $0xfc,%al                /* pulse reset and A20 low */
        outb %al,$0x64
die:    jmp die
```

10.2.3 Information

10.2.3.1 PC/AT keyboard interface programming

The keyboard interface on the motherboard of a PC is a dedicated interface that can be seen as a simplified version of the regular synchronous serial port. The interface circuit is called a keyboard controller, which receives the scan code data sent by the keyboard using a serial communication protocol. The keyboard controller used on the motherboard is the Intel 8042 chip or its compatible one. The schematic diagram is shown in Figure 10-7. The independent 8042 chip is not included on the current motherboard, but other integrated circuits on the motherboard will simulate the function of the 8042 chip for compatibility purposes. Therefore, the programming method of the keyboard controller is still applicable to the current motherboard. In addition, the chip output port P2 are used for other purposes. Bit 0 (P20 pin) is used to implement the reset operation of the CPU, and bit 1 (P21 pin) is used to control whether the A20 signal line is turned on or not. When the output port bit 1 is 1, the A20 signal line is turned on (strobe), and 0 is the A20 signal line is disabled.

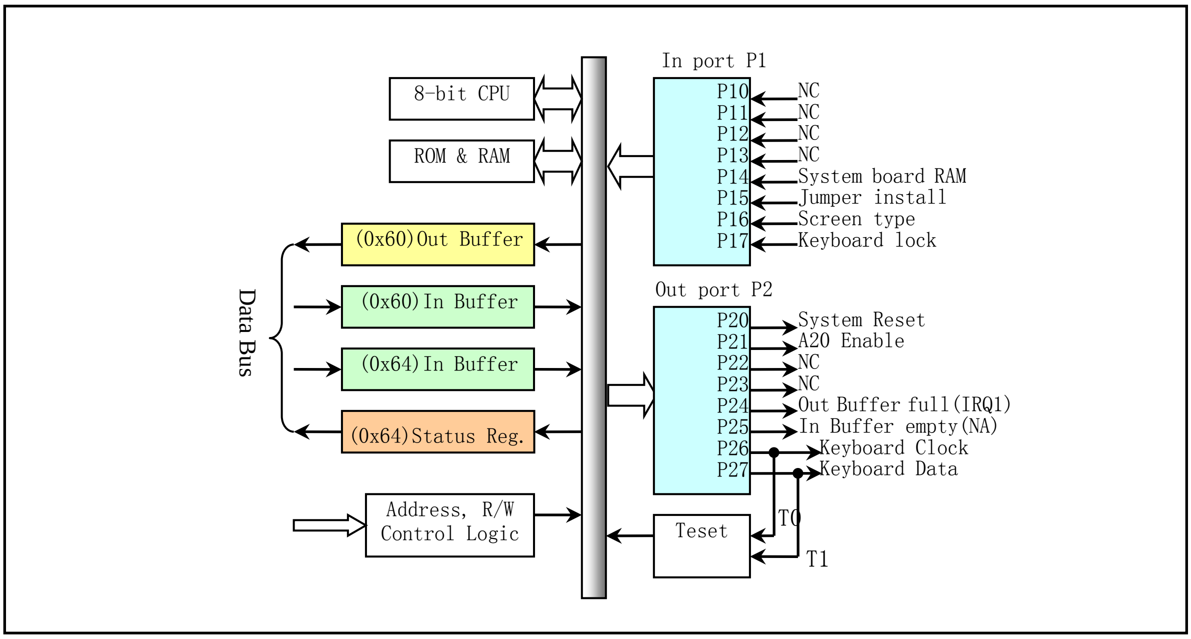


Figure 10-7 Keyboard controller 804X schematic diagram

The range of IO ports assigned to the keyboard controller is 0x60-0x6f, but in fact IBM PC/AT uses only 0x60 and 0x64 port addresses (0x61, 0x62 and 0x63 for XT compatible purposes). See Table 10-1. In addition, the meaning of the read and write operations on the port is different, so there are mainly four different operations. Programming the keyboard controller will involve the status registers, input buffers, and output buffers in the chip.

Table 10-1 Keyboard controller 804X ports

Port	R/W	Name	Purposes
0x60	Read	Data port or Output Buffer	Is an 8-bit read-only register. When the keyboard controller receives a scan code or command response from the keyboard, it sets the status register bit 0 = 1 on the one hand and generates an interrupt IRQ1 on the other hand. Normally it should only be read when the status port bit 0 = 1.
0x60	Write	Input Buffer	Used to send commands and/or subsequent parameters to the keyboard, or to write parameters to the keyboard controller. There are more than 10 keyboard commands, see the instructions after the table. Normally it should only be written when the status port bit 1 = 0.
0x61	Read/ Write		This port is the address of the 8255A output port B (P2) and is hardware reset processing for the PC standard keyboard circuit using the 8255A. This port is used to respond to the received scan code. The method is to first disable the keyboard and then immediately re-enabled the keyboard. The data being manipulated is: Bit 7 =1 Disables the keyboard; =0 allows the keyboard; Bit 6 =0 Forces the keyboard clock to be low, so the keyboard cannot send any

			data.; Bits 5-0 These bits are independent of the keyboard and are used for Programmable Parallel Interface (PPI).
0x64	Read	Status Register	The port is an 8-bit read-only register whose bit field meanings are: Bit 7=1 parity error of the transmitted data from the keyboard (should be 0, odd parity); Bit 6=1 Receive timeout (IRQ1 is not generated by keyboard transfer); Bit 5=1 Transmit timeout (the keyboard is not responding); Bit 4=0 Inhibit Switch. Indicates that the keyboard interface is inhibited; Bit 3=1 The data written to the input buffer is a command (via port 0x64); =0 The data written to the input buffer is a parameter (via port 0x60); Bit 2 System Flag Status: 0 = Power-on or Reset; 1 = Self-test passed; Bit 1=1 Input buffer is full (0x60/64 port has data for 8042); Bit 0 = 1 Output buffer is full (data port 0x60 has data for the system).
0x64	Write	Input Buffer	Write commands to the keyboard controller. It can take one parameter and the parameter is written from port 0x60. There are 12 keyboard controller commands, see the instructions after the table.

10.2.3.2 Keyboard commands

The system writes 1 byte to port 0x60, which is to send a keyboard command. The keyboard should respond within 20 ms of receiving the command, that is, return a command response. Some commands also need to follow a parameter (also written to the port). The command list is shown in Table 10-2. Note that all commands are returned with a 0xfa response code (ACK) if not indicated otherwise.

Table 10-2 Keyboard command list

Command	Has Params	Description
0xed	Yes	Set/reset mode indicator. Set to 1 to open and 0 to close. Parameter byte: Bits 7-3 are all reserved as 0; Bit 2 = caps-lock key; Bit 1 = num-lock key; Bit 0 = scroll-lock key.
0xee	No	Diagnostic response. The keyboard should return 0xee.
0xef		Reserved.
0xf0	Yes	Read/set the scan code set. The parameter byte is equal to 0x00 - select the current scan code set; 0x01 - Select scan code set 1 (for PCs, PS/2 30, etc.); 0x02 - Select scan code set 2 (for AT, PS/2, which is the default); 0x03 - Select scan code set 3.
0xf1		Reserved.
0xf2	No	Read the keyboard identification number (read 2 bytes). The AT keyboard returns the response code 0xfa.
0xf3	Yes	Set the rate and delay time when the scan code is sent continuously. The meaning of the parameter byte is:

		Bit 7 is reserved as 0; Bit 6-5 Delay value: Let C=bit 6-5, then the formula: delay value = (1 + C) * 250ms; Bit 4-0 The rate at which the scan code is continuously transmitted; let B = Bit 4-3; A = Bit 2-0, then a formula:Rate = 1 / ((8 + A) * 2 ^ B * 0.00417). The default value of the parameter is 0x2c.
0xf4	No	Enable keyboard.
0xf5	No	Disable keyboard.
0xf6	No	Set keyboard default parameters.
0xf7-0xfd		Reserved.
0xfe	No	Resend the scan code. This command is issued when the system detects that the keyboard transmits data incorrectly.
0xff	No	Performing a keyboard power-on reset operation is called a Basic Assured Test (BAT). The operation process is: 1. The keyboard responds by sending the command 0xfa immediately after receiving the command; 2. The keyboard controller sets the keyboard clock and data lines high; 3. The keyboard begins to perform BAT operations; 4. If it is completed normally, the keyboard sends 0xaa; otherwise it sends 0xfd and stops scanning.

10.2.3.3 Keyboard Controller Commands

The system writes 1 byte to the input buffer (port 0x64), which sends a keyboard controller command. It can take one parameter, but the parameter is sent by writing to port 0x60, as shown in Table 10-3.

Table 10-3 Keyboard controller command list

Command	Has Params	Description
0x20	No	The last command byte to keyboard controller is placed on port 0x60 for system read.
0x21-0x3f	No	Reads the command in the internal RAM of the controller specified by the lower 5 bits of the command.
0x60-0x7f	Yes	Write the keyboard controller command byte. The parameter byte is: (default is 0x5d) Bit 7 is reserved as 0; Bit 6 IBM PC compatibility mode (parity, conversion to system scan code, single-byte PC BREAK code); Bit 5 PC mode (no parity check for scan code; no conversion to system scan code); Bit 4 disables keyboard operation (making the keyboard clock low); Bit 3 disables override and does not work for keyboard lock conversion; Bit 2 system flag; 1 indicates that the controller is working correctly; Bit 1 is reserved as 0; Bit 0 allows an interrupt to be generated when the output register is full.
0xaa	No	Initialize the keyboard controller self test. Returns 0x55 on success; 0xfc on failure.
0xab	No	Initialize the keyboard interface test. The meaning of the returned byte is: 0x00 is error free;

		0x01 keyboard clock line is low (always low, low stuck); 0x02 The keyboard clock line is high; 0x03 keyboard data line is low; 0x04 The keyboard data line is high.
0xac	No	Diagnostic dump. The 804x 16-byte RAM, output port, and input port status are sequentially output to the system.
0xad	No	Disable keyboard operation (set bit 4 of command byte = 1).
0xae	No	Enable keyboard operation (reset command byte bit 4 = 0).
0xc0	No	Read the input port P1 of 804X and put it at 0x60 for reading;
0xd0	No	Read the output port P2 of 804X and put it at 0x60 for reading;
0xd1	Yes	Write 804X output port P2, the original IBM PC uses the output port bit 2 to control the A20 gate. Note that bit 0 (system reset) should always be set.
0xe0	No	The input of the test terminals T0 and T1 is sent to the output buffer for reading by the system. Bit 1 - Keyboard Data; Bit 0 - Keyboard Clock.
0xed	Yes	Control the status of keyboard LEDs. Set to 1 to open and 0 to close. Parameter byte: Bits 7-3 are all reserved as 0; Bit 2 = Caps-lock key; Bit 1 = Num-lock key; Bit 0 = Scroll-lock key.
0xf0-0xff	No	Send a pulse to the output port. This command sequence controls the output port P20-23 line, see the schematic diagram of the keyboard controller. If you want to output a negative pulse (6 microseconds), set this bit to 0. That is, the lower 4 bits of the command control the output of the negative pulse, respectively. For example, to reset the system, you need to issue the command 0xfe (P20 low).

10.2.3.4 Keyboard Scan Code

The PCs are all non-encoded keyboards. Each key on the keyboard has a position number, from left to right, from top to bottom, and the position code of the PC XT and AT machine keyboard are very different. The microprocessor in the keyboard sends the scan code corresponding to the button to the system. When the key is pressed, the scan code output by the keyboard is called a 'make' scan code, and when the key is released, it is called a 'break' scan code. The scan codes of the keys of the XT keyboard are shown in Table 10-4.

Table 10-4 XT keyboard scan code table

F1	F2	`	1	2	3	4	5	6	7	8	9	0	–	=	\	BS	ESC	NUML	SCRL	SYSR
3B	3C	29	02	03	04	05	06	07	08	09	0A	0B	0C	0D	2B	0E	01	45	46	**
F3	F4	TAB	Q	W	E	R	T	Y	U	I	O	P	[	]			Home	↑	PgUp	PrtSc
3D	3E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B			47	48	49	37
F5	F6	CNTL	A	S	D	F	G	H	J	K	L	;	'	ENTER			←	5	→	–
3F	40	1D	1E	1F	20	21	22	23	24	25	26	27	28	1C			4B	4C	4D	4A
F7	F8	LSHFT	Z	X	C	V	B	N	M	,	.	/	RSHFT				End	↓	PgDn	+
41	42	2A	2C	2D	2E	2F	30	31	32	33	34	35	36				4F	50	51	4E
F9	F10	ALT		Space										CAPLOCK			Ins		Del	
43	44	38		39										3A			52		53	

Each button on the keyboard has a corresponding scan code contained in the lower 7 bits (bits 6-0) of the byte, and the highest bit (bit 7) indicates whether the button is pressed or released. Bit 7 = 0 indicates the scan code just pressed the key, and bit 7 = 1 indicates the scan code after the key is released. For example, if someone just pressed the ESC key, the scan code transmitted to the system would be 1 (1 is the scan code for the ESC key), and when the key is released, a 1+0x80=129 scan code will be generated.

For the standard 83-key keyboard of PC, PC/XT, the scan code is the same as the key number (the position code of the key) and is represented by 1 byte. For example, the "A" key, the key position number is 30, the make code and the scan code are also 30 (0x1e), and the break code is the make code plus 0x80, that is, 0x9e.

The situation is somewhat different for some "extended" keys. When an extended key is pressed, an interrupt will be generated and the keyboard will output an extended scan code prefix of 0xe0, while in the next interrupt an "extended" scan code will be given. For example, for the PC/XT standard keyboard, the scan code of the left control key ctrl is 29 (0x1d), and the "extended" control key ctrl on the right has an extended scan code sequence 0xe0, 0x1d. This rule is also suitable for alt, directional arrow keys.

In addition, there are two keys that are very special, the PrtScn key and the Pause/Break key. Pressing the PrtScn button will send 2 extended characters, 42 (0x2a) and 55 (0x37), to the keyboard interrupt routine, so the actual byte sequence will be 0xe0, 0x2a, 0xe0, 0x37, and when the button is repeatedly generated, the extended character 0xaa is also sent, that is, the sequence 0xe0, 0x2a, 0xe0, 0x37, 0xe0, 0xaa is generated. When the key is released, two extended codes plus 0x80 (0xe0, 0xb7, 0xe0, 0xaa) are resent. When the PrtScn key is pressed, if the shift or ctrl key is also pressed, only 0xe0, 0x37 is sent, and only 0xe0, 0xb7 are sent when released. If the alt key is pressed at the same time, the PrtScn key is like a normal key with scan code 0x54.

For the Pause/Break key, if you press any of the control keys ctrl while pressing the key, the line will be like the extended key 70 (0x46), and in other cases it will send the character sequence 0xe1, 0x1d, 0x45, 0xe1, 0x9d, 0xc5. Pressing the button all the way does not produce a duplicate scan code, and releasing the button does not produce any scan code. Therefore, we can look at and handle this way: scan code 0xe0 means that there is one more character to follow, and scan code 0xe1 means 2 characters followed.

The scan code of the AT keyboard is slightly different from that of PC/XT. When the key is pressed, the scan code of the corresponding key is sent, but when the key is released, two bytes will be sent, the first one is 0xf0, and the second one is the same key scan code. Keyboard designers now use the 8049 as the input processor for the AT keyboard, and for the downward compatibility, the scan code from the AT keyboard is converted to the scan code of the old PC/XT standard keyboard before being sent to the system.

10.3 console.c

10.3.1 Function description

console.c is one of the longest programs in the kernel, but its functionality is relatively simple. All of the subroutines are used to implement the terminal screen write function con_write() and the control operation of the terminal screen display.

The con_write() function is called when a write to a console device is performed. This function manages all control and escape character sequences that provide the entire screen management operation for the application. The implemented escape sequence uses the vt102 terminal specification, which means that when you connect to a non-Linux host using a telnet program, your environment variable should have TERM=vt102. However, the best option for local operations is to set TERM=console because the Linux console provides a superset of vt102 functionality.

The function con_write() is mainly composed of a conversion statement for interpreting the finite-length state automatic escape sequence of one character at a time. In normal mode, the display characters are written directly to the display memory using the current attributes. The function will take a character or sequence of characters from the write buffer queue write_q of the terminal tty_struct data structure, and then display the characters on the terminal screen according to the nature of the characters (normal characters, control characters, escape sequences or control sequences), or perform some screen control operations such as cursor movement and character erasure.

The terminal screen initialization function con_init() sets some basic parameter values about the screen according to the system information obtained when the system starts initialization, and is used for the operation of the con_write() function.

For a description of the terminal device character buffer queue, see the include/linux/tty.h header file, which shows the data structure tty_queue of the character buffer queue, the data structure tty_struct of the terminal, and some control character values. There are also some macro definitions that operate on the buffer queue. See Figure 10-14 for a schematic of the buffer queue and its operation.

10.3.2 Code annotation

Program 10-2 linux/kernel/chr_drv/console.c

```
/*
 *  linux/kernel/console.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      console.c
 *
 * This module implements the console io functions
 *      'void con_init(void)'
 *      'void con_write(struct tty_queue * queue)'
 * Hopefully this will be a rather complete VT102 implementation.
 *
 * Beeping thanks to John T Kohl.
 *
 * Virtual Consoles, Screen Blanking, Screen Dumping, Color, Graphics
 *   Chars, and VT100 enhancements by Peter MacDonald.
 */

/*
 *  NOTE!!! We sometimes disable and enable interrupts for a short while
 * (to put a word in video IO), but this will work even for keyboard
 * interrupts. We know interrupts aren't enabled when getting a keyboard
 * interrupt, as we use trap-gates. Hopefully all is well.
 */

/*
 * Code to check for different video-cards mostly by Galen Hunt,
 * <g-hunt@ee.utah.edu>
 */

   // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
   //     data of the initial task 0, and some embedded assembly function macro statements
   //     about the descriptor parameter settings and acquisition.
   // <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
   //     communication.
   // <linux/config.h> Kernel configuration header file. Define keyboard language and hard
   //     disk type (HD_TYPE) options.
   // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
   //     commonly used functions of the kernel.
   // <asm/io.h> Io header file. Defines the function that operates on the io port in the
   //     form of a macro's embedded assembler.
   // <asm/system.h> System header file. An embedded assembly macro that defines or
   //     modifies descriptors/interrupt gates, etc. is defined.
   // <asm/segment.h> Segment operation header file. An embedded assembly function is defined
   //     for segment register operations.
   // <string.h> String header file. Defines some embedded functions about string operations.
   // <errno.h> Error number header file. Contains various error numbers in the system.
#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/config.h>
#include <linux/kernel.h>

#include <asm/io.h>
#include <asm/system.h>
#include <asm/segment.h>

#include <string.h>
#include <errno.h>

// This symbolic constant defines the default data for the terminal IO structure. For the
// symbolic constants, please refer to the include/termios.h file.
#define DEF_TERMIOS \
(struct termios) { \
        ICRNL, \
        OPOST | ONLCR, \
        0, \
        IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, \
        0, \
        INIT_C_CC \
}


/*
 * These are set up by the setup-routine at boot-time:
 */
// Please refer to the comments on boot/setup.s and the system parameter table read and saved
// by the setup program.

#define ORIG_X             (*(unsigned char *)0x90000)        // original cursor colum no.
#define ORIG_Y             (*(unsigned char *)0x90001)        // original cursor row no.
#define ORIG_VIDEO_PAGE    (*(unsigned short *)0x90004)       // current video page.
#define ORIG_VIDEO_MODE    ((*(unsigned short *)0x90006) & 0xff)           // display mode.
#define ORIG_VIDEO_COLS    (((*(unsigned short *)0x90006) & 0xff00) >> 8)  // screen colums.
#define ORIG_VIDEO_LINES   ((*(unsigned short *)0x9000e) & 0xff)           // screen rows.
#define ORIG_VIDEO_EGA_AX  (*(unsigned short *)0x90008)       // current function.
```

```
#define ORIG_VIDEO_EGA_BX (*(unsigned short *)0x9000a)       // disp mem size, color mode.
#define ORIG_VIDEO_EGA_CX (*(unsigned short *)0x9000c)       // video card properties.

// Defines the monochrome/color display mode type symbol constant.
#define VIDEO_TYPE_MDA     0x10    /* Monochrome Text Display     */
#define VIDEO_TYPE_CGA     0x11    /* CGA Display                 */
#define VIDEO_TYPE_EGAM    0x20    /* EGA/VGA in Monochrome Mode  */
#define VIDEO_TYPE_EGAC    0x21    /* EGA/VGA in Color Mode       */

#define NPAR 16                    // The max nr of arguments in the escape sequence.

int NR_CONSOLES = 0;               // The nr of virtual consoles that the system supports.

extern void keyboard_interrupt(void);      // Keyboard interrupt handler (in keyboard.S).

// The following static variables are some of the global variables used in this file.
static unsigned char    video_type;             /* Type of display being used   */
static unsigned long    video_num_columns;      /* Number of text columns       */
static unsigned long    video_mem_base;         /* Base of video memory         */
static unsigned long    video_mem_term;         /* End of video memory          */
static unsigned long    video_size_row;         /* Bytes per row                */
static unsigned long    video_num_lines;        /* Number of test lines         */
static unsigned char    video_page;             /* Initial video page           */
static unsigned short   video_port_reg;         /* Video register select port   */
static unsigned short   video_port_val;         /* Video register value port    */
static int can_do_colour = 0;                   // flag, can use color function.

// The virtual console structure is defined below. It contains all the current information for
// a virtual console. vc_origin and vc_scr_end are the display memory locations corresponding
// to the start line and the last line used by the virtual console currently being processed
// to perform a fast scroll operation. vc_video_mem_start and vc_video_mem_end are the display
// memory areas used by the current virtual console.
static struct {
        unsigned short  vc_video_erase_char;    // erase char attributes & char (0x0720)
        unsigned char   vc_attr;                // char attribute.
        unsigned char   vc_def_attr;            // default attribute.
        int             vc_bold_attr;           // bold char attribute.
        unsigned long   vc_ques;                // question mark char.
        unsigned long   vc_state;               // state of the escape or control sequence.
        unsigned long   vc_restate;             // next state of escape or control sequence.
        unsigned long   vc_checkin;
        unsigned long   vc_origin;              /* Used for EGA/VGA fast scroll */
        unsigned long   vc_scr_end;             /* Used for EGA/VGA fast scroll */
        unsigned long   vc_pos;                 // The mem location of the current cursor.
        unsigned long   vc_x,vc_y;              // current cursor column, row value.
        unsigned long   vc_top,vc_bottom;       // The top & bottom line nr when scrolling.
        unsigned long   vc_npar,vc_par[NPAR];   // escape sequence param nr and array.
        unsigned long   vc_video_mem_start;     /* Start of video RAM           */
        unsigned long   vc_video_mem_end;       /* End of video RAM (sort of)   */
        unsigned int    vc_saved_x;             // The saved cursor column number.
        unsigned int    vc_saved_y;             // The saved cursor row number.
        unsigned int    vc_iscolor;             // The color display flag.
        char *          vc_translate;           // The character set used.
```

```
} vc_cons [MAX_CONSOLES];

// For ease of reference, the following defines the symbols of the console currently being
// processed, with the same meaning as above. Where currcons is the current virtual terminal
// number in the function argument using the vc_cons[] structure.
#define origin          (vc_cons[currcons].vc_origin)
#define scr_end         (vc_cons[currcons].vc_scr_end)
#define pos             (vc_cons[currcons].vc_pos)
#define top             (vc_cons[currcons].vc_top)
#define bottom          (vc_cons[currcons].vc_bottom)
#define x               (vc_cons[currcons].vc_x)
#define y               (vc_cons[currcons].vc_y)
#define state           (vc_cons[currcons].vc_state)
#define restate         (vc_cons[currcons].vc_restate)
#define checkin         (vc_cons[currcons].vc_checkin)
#define npar            (vc_cons[currcons].vc_npar)
#define par             (vc_cons[currcons].vc_par)
#define ques            (vc_cons[currcons].vc_ques)
#define attr            (vc_cons[currcons].vc_attr)
#define saved_x         (vc_cons[currcons].vc_saved_x)
#define saved_y         (vc_cons[currcons].vc_saved_y)
#define translate       (vc_cons[currcons].vc_translate)
#define video_mem_start (vc_cons[currcons].vc_video_mem_start)
#define video_mem_end   (vc_cons[currcons].vc_video_mem_end)
#define def_attr        (vc_cons[currcons].vc_def_attr)
#define video_erase_char  (vc_cons[currcons].vc_video_erase_char)
#define iscolor         (vc_cons[currcons].vc_iscolor)

int blankinterval = 0;                  // Black screen interval.
int blankcount = 0;                     // Black screen time count.

static void sysbeep(void);              // System beep function.

/*
 * this is what the terminal answers to a ESC-Z or csi0c
 * query (= vt100 response).
 */                                     // csi - Control Sequence Introducer.
// The host requests the terminal to answer a device attribute control sequence by sending a
// device attribute (DA) control sequence with no parameters or parameter 0 ('ESC [c' or
// 'ESC [0c') (ESC-Z acts the same). The terminal sends the following sequence to respond to
// the host.  This sequence (ie 'ESC [?1; 2c') indicates that the terminal is a VT100 compatible
// terminal with advanced video capabilities.
#define RESPONSE "\033[?1;2c"

// Define the character set to use. The upper part is the normal 7-bit ASCII code, which is
// the US character set. The lower part corresponds to the line character in the VT100 terminal
// device, that is, the character set showing the chart line.
static char * translations[] = {
/* normal 7-bit ascii */
        " !\"#$%&'()*+,-./0123456789:;<=>?"
        "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_"
        "`abcdefghijklmnopqrstuvwxyz{|}~ ",
/* vt100 graphics */
```

```
        " !\"#$%&'()*+,-./0123456789:;<=>?"
        "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^ "
        "\004\261\007\007\007\007\370\361\007\007\275\267\326\323\327\304"
        "\304\304\304\304\307\266\320\322\272\363\362\343\\007\234\007 "
};

#define NORM_TRANS (translations[0])
#define GRAF_TRANS (translations[1])

//// Tracks the current position of the cursor.
// Parameters: currcons - the current virtual terminal; new_x - the cursor column number;
// new_y - the cursor line number.
// This function is used to update the current cursor position variable x, y and correct the
// corresponding position pos of the cursor in the display memory. This function first checks
// the validity of the parameters. Exit if the given cursor column number exceeds the maximum
// number of columns in the display, or if the cursor line number is not lower than the maximum
// number of rows displayed. Otherwise, the current cursor variable and the new cursor position
// are updated to correspond to the position pos in the display memory. Note that all variables
// in the function are actually the corresponding fields in the vc_cons[currcons] structure,
// and the following functions are the same.
/* NOTE! gotoxy thinks x==video_num_columns is ok */
static inline void gotoxy(int currcons, int new_x,unsigned int new_y)
{
        if (new_x > video_num_columns || new_y >= video_num_lines)
                return;
        x = new_x;
        y = new_y;
        pos = origin + y*video_size_row + (x<<1);     // One col needs 2 bytes, so x<<1.
}

//// Set the scroll start display memory address.
// First check whether the display card type is a color card, and judge whether the console
// specified by the parameter is the front console, and exit if the two conditions are not met.
// Otherwise, the scroll start address is output to the register on the display card.
static inline void set_origin(int currcons)
{
// First determine the type of display card. For EGA/VGA cards, we can specify the on-screen
// range (area) for scrolling, while the MDA monochrome display card can only perform full-screen
// scrolling. Therefore, only the EGA/VGA card needs to set the display memory address of the
// start line of the scrolling (the starting line is the line corresponding to origin), otherwise
// it will exit directly. In addition, we only operate on the foreground console, so only when
// the current console currcons is the foreground console, we need to set the memory start
// position corresponding to the scroll start line.
        if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
                return;
        if (currcons != fg_console)
                return;
// Then we output 12 to the "display register select port" video_port_reg, which selects the
// display control data register r12 and then writes the scroll start address high byte to it.
// Shifting 9 bits to the right actually means shifting 8 bits to the right and dividing by
// 2 (1 character on the screen is represented by 2 bytes). Then select the display control
// data register r13 and write the low byte of the scroll start address. Shifting 1 bit to the
// right means dividing by 2, which also means that 1 character on the screen is represented
// by 2 bytes of memory. The output value operates relative to the default display memory start
// position video_mem_base, for example, for EGA/VGA color mode, viedo_mem_base = physical
// memory address 0xb8000.
        cli();
        outb_p(12, video_port_reg);       // Select data register r12 to output high byte.
        outb_p(0xff&((origin-video_mem_base)>>9), video_port_val);
        outb_p(13, video_port_reg);       // Select r13 to output the low byte.
        outb_p(0xff&((origin-video_mem_base)>>1), video_port_val);
        sti();
}

//// The content scrolls up one line.
// Move the screen scroll window down one line and add a space character to the new line that
// appears at the bottom of the screen scroll area. The scrolling area must be greater than
// one line. See section 10.3.3.2 for the principle of scrolling operation.
static void scrup(int currcons)
{
// The scrolling area must have at least 2 lines. If the top line number of the scrolling area
// is greater than or equal to the bottom line number, the condition for the rolling operation
// is not satisfied. In addition, for the EGA/VGA card, we can specify the on-screen range (area)
// for scrolling, while the MDA monochrome display card can only perform full-screen scrolling.
// This function handles the EGA and MDA display types separately. If the display type is EGA,
// it is also divided into full-screen window movement and intra-area window movement. Here
// the code first deals with the case where the display card is of the EGA/VGA type.
        if (bottom<=top)
                return;
        if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
        {
// If the move start line top=0, the bottom line = video_num_lines = 25, it means that the entire
// screen window moves down one line. Therefore, the starting memory position origin
// corresponding to the upper left corner of the entire screen window is adjusted to the memory
// position shifted downward by one line, and the memory position corresponding to the current
// cursor and the position of the character pointer scr_end at the end of the screen end are
// also tracked. Finally, the new screen window memory start position origin is written into
// the display controller.
                if (!top && bottom == video_num_lines) {
                        origin += video_size_row;
                        pos += video_size_row;
                        scr_end += video_size_row;
// If the display memory pointer scr_end corresponding to the end of the screen window exceeds
// the end of the actual display memory, the memory data corresponding to all the lines except
// the first line of the screen content is moved to the start position video_mem_start of the
// display memory, and fill in the space character on a new line that appears after the entire
// screen window has been moved down. Then, according to the situation that the screen memory
// data is moved, the start pointer corresponding to the current screen, the cursor position
// pointer, and the corresponding memory pointer scr_end at the end of the screen are readjusted.
//
// This embedded assembly code first moves the memory data corresponding to the (screen character
// line number - 1) line to the display memory start position video_mem_start, and then adds
// a line of space (erase) character data at the subsequent memory location.
// %0 - eax (erase character + attribute); %1 - ecx ((the number of lines of the screen -1)
// corresponds to the number of characters / 2, moving in long words); %2 - edi (displays memory
// start position video_mem_start); %3 - esi (screen window memory start position origin).
// Direction of movement: [edi] -> [esi], move ecx long words.
                        if (scr_end > video_mem_end) {
                                __asm__("cld\n\t"       // clear direction.
                                        "rep\n\t"       // repeat move data.
                                        "movsl\n\t"
                                        "movl _video_num_columns,%1\n\t"
                                        "rep\n\t"       // fill a line of spaces.
                                        "stosw"
                                        ::"a" (video_erase_char),
                                        "c" ((video_num_lines-1)*video_num_columns>>1),
                                        "D" (video_mem_start),
                                        "S" (origin)
                                        :"cx","di","si");
                                scr_end -= origin-video_mem_start;
                                pos -= origin-video_mem_start;
                                origin = video_mem_start;
// If the memory pointer scr_end at the end of the adjusted screen does not exceed the end of
// the display memory video_mem_end, simply fill in the erase line (space character) on the
// new line.
// %0 - eax (erase character + attribute); %1 - ecx (number of screen lines); %2 - edi
// (corresponding memory location at the beginning of the last line).
                        } else {
                                __asm__("cld\n\t"
                                        "rep\n\t"       // fill a line of spaces.
                                        "stosw"
                                        ::"a" (video_erase_char),
                                        "c" (video_num_columns),
                                        "D" (scr_end-video_size_row)
                                        :"cx","di");
                        }
// Then write the new screen window memory start position origin into the display controller.
                        set_origin(currcons);

// Otherwise it means that it is not a full screen move. That is to say, all the lines from
// the specified line top to bottom area are moved up by one line, and the specified line top
// is deleted. At this time, the display memory data corresponding to all the lines of the screen
// from the specified line top to the bottom is directly moved up by one line, and the erasing
// character is filled in the newly appearing line.
// %0 - eax (erase character + attribute); %1 - ecx (the number of long words from the top+1
// line to the bottom line); %2 - edi (the memory location where the top row is located); %3
// - esi (the memory location where the top+1 row is located).
                } else {
                        __asm__("cld\n\t"
                                "rep\n\t"        // repeat from top+1 to bottom.
                                "movsl\n\t"
                                "movl _video_num_columns,%%ecx\n\t"
                                "rep\n\t"        // fill blank char in new line.
                                "stosw"
                                ::"a" (video_erase_char),
                                "c" ((bottom-top-1)*video_num_columns>>1),
                                "D" (origin+video_size_row*top),
                                "S" (origin+video_size_row*(top+1))
                                :"cx","di","si");
                }
        }
// If the display type is not EGA (but MDA), perform the following move operation. Because the
// MDA display card can only scroll through the entire screen, and it will automatically adjust
// beyond the display memory range (that is, the pointer will be automatically scrolled), so
// the memory corresponding to the screen content is not processed separately than the display
// memory range. The processing method is exactly the same as the EGA non-full screen movement.
        else            /* Not EGA/VGA */
        {
                __asm__("cld\n\t"
                        "rep\n\t"
                        "movsl\n\t"
                        "movl _video_num_columns,%%ecx\n\t"
                        "rep\n\t"
                        "stosw"
                        ::"a" (video_erase_char),
                        "c" ((bottom-top-1)*video_num_columns>>1),
                        "D" (origin+video_size_row*top),
                        "S" (origin+video_size_row*(top+1))
                        :"cx","di","si");
        }
}

//// The display content scrolls down one line.
// Move the screen scroll window up one line, and the corresponding screen scroll area content
// moves down one line. A new line will appear above the start line of the move. The processing
// method is similar to scrup() except that the data overlay does not occur when moving the
// memory data, and the copy operation is reversed. That is, copy from the last character on
// the second line of the countdown to the last line, then copy the characters on the third
// line of the countdown to the second line of the last, and so on.
static void scrdown(int currcons)
{
// Similarly, the scrolling area must have at least 2 lines. If the top line number of the
// scrolling area is greater than or equal to the bottom line number of the area, the condition
// for the rolling operation is not satisfied. In addition, for the EGA/VGA card, we can specify
// the on-screen range (area) for scrolling, while the MDA monochrome display card can only
// perform full-screen scrolling. Since the window moves up to the start position of the display
// area memory at most, the display memory pointer scr_end corresponding to the end of the screen
// window does not exceed the actual display memory end, so only the normal memory data movement
// needs to be handled here.
        if (bottom <= top)
                return;
        if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
        {
// %0 - eax (erase character + attribute); %1 - ecx (the number of long words corresponding
// to the number of lines from top to bottom-1); %2 - edi (the last long word position in the
// lower right corner of the window); %3 - Esi (the last long word position in the second line
// of the window countdown).
// Direction of movement: [esi] -> [edi], move ecx long words.
                __asm__("std\n\t"             // set direction flag!!
                        "rep\n\t"
                        "movsl\n\t"
                        "addl $2,%%edi\n\t"   /* %edi has been decremented by 4 */
                        "movl _video_num_columns,%%ecx\n\t"
                        "rep\n\t"            // fill in blanks at above line.
                        "stosw"
                        ::"a" (video_erase_char),
                        "c" ((bottom-top-1)*video_num_columns>>1),
                        "D" (origin+video_size_row*bottom-4),
                        "S" (origin+video_size_row*(bottom-1)-4)
                        :"ax","cx","di","si");
        }
// If it is not the EGA display type, do the following (same as above).
        else                /* Not EGA/VGA */
        {
                __asm__("std\n\t"
                        "rep\n\t"
                        "movsl\n\t"
                        "addl $2,%%edi\n\t"     /* %edi has been decremented by 4 */
                        "movl _video_num_columns,%%ecx\n\t"
                        "rep\n\t"
                        "stosw"
                        ::"a" (video_erase_char),
                        "c" ((bottom-top-1)*video_num_columns>>1),
                        "D" (origin+video_size_row*bottom-4),
                        "S" (origin+video_size_row*(bottom-1)-4)
                        :"ax","cx","di","si");
        }
}

//// The cursor moves down one line at the same column position.
// If the cursor is not on the last line, directly modify the cursor current line variable y++,
// and adjust the cursor corresponding display memory position pos (plus the memory length
// corresponding to one line of characters). Otherwise you need to move the contents of the
// screen window up one line. The function name lf (line feed) refers to the processing control
// character LF.
static void lf(int currcons)
{
        if (y+1<bottom) {
                y++;
                pos += video_size_row;        // Plus the nr of bytes occupied by one line.
                return;
        }
        scrup(currcons);                      // Move the contents up one line.
}

//// The cursor moves up one row in the same column.
// If the cursor is not on the first line of the screen, directly modify the cursor current
// line variable y--, and adjust the cursor corresponding display memory position pos, minus
// the number of bytes of memory length corresponding to a line of characters on the screen.
// Otherwise you need to move the contents of the screen window down one line. The function
// name ri (reverse index) refers to the control character RI or the escape sequence "ESC M".
static void ri(int currcons)
{
        if (y>top) {
                y--;
                pos -= video_size_row;       // Minus the nr of bytes occupied by one line.
                return;
        }
        scrdown(currcons);                   // Move the contents down one line.
}

// The cursor returns to column 0.
// Adjust the cursor corresponding to the memory location pos. The column number of the cursor
// *2 is the length of the memory byte occupied by the 0 column on the line to the column where
// the cursor is located. The function name cr (carriage return) indicates the control character
// being processed.
static void cr(int currcons)
{
        pos -= x<<1;                         // bytes occupied by col 0 to col of cursor.
        x=0;
}

// Erase the previous character of the cursor and move the cursor back one column.
// If the cursor is not in the 0 column, the cursor is backed by 2 bytes corresponding to the
// memory position pos (corresponding to one character on the screen), then the current cursor
// column is decremented by 1, and the character at the position of the cursor is erased.
static void del(int currcons)
{
        if (x) {
                pos -= 2;
                x--;
                *(unsigned short *)pos = video_erase_char;
        }
}

//// Delete part of the content on the screen that is related to the cursor position.
// This function is used to handle ANSI control sequences. The delete character operation
// associated with the cursor position is performed according to the specified control sequence,
// some or all of the displayed characters are erased, and the cursor position is unchanged
// when the character or line is erased.
// The ANSI control sequence processed by this function is: 'ESC [ Ps J'. Among them, Ps = 0
// - means to delete the cursor to the bottom of the screen; 1 - delete the screen to the cursor;
// 2 - delete the entire screen.
// The function name csi_J (CSI - Control Sequence Introducer) indicates that the control
// sequence "CSI Ps J" is processed. The argument 'vpar' corresponds to the value of 'Ps' in
// the above control sequence. For an introduction to terminal control commands, see section
// 10.3.3.3 after the program listing. Commonly used escape sequences and control sequences
// can be found in Appendix 3 of the book.
static void csi_J(int currcons, int vpar)
{
        long count __asm__("cx");                    // set to use register variable.
        long start __asm__("di");

// First, set the number of characters to be deleted and the display memory position at which
// deletion is started according to the above three cases.
        switch (vpar) {
                case 0: /* erase from cursor to end of display */
                        count = (scr_end-pos)>>1;
                        start = pos;
                        break;
                case 1: /* erase from start to cursor */
                        count = (pos-origin)>>1;
                        start = origin;
                        break;
                case 2: /* erase whole display */
                        count = video_num_columns * video_num_lines;
                        start = origin;
                        break;
                default:
                        return;
        }
// Then use the erase character to fill in the place where the character is deleted.
// %0 -ecx (nr of chars deleted); %1 -edi (delete start address); %2 -eax (filled erase char).
        __asm__("cld\n\t"
                "rep\n\t"
                "stosw\n\t"
                ::"c" (count),
                "D" (start),"a" (video_erase_char)
                :"cx","di");
}

//// Delete part of the content related to the cursor position on one line.
// This function erases some or all of the characters in the line of the cursor according to
// the parameters. The erase operation removes characters from the screen without affecting
// other characters. The erased characters are discarded. The cursor position does not change
// when erasing characters or lines.
// ANSI escape sequence: 'ESC [ Ps K' (Ps = 0 to the end of the line; 1 to delete from the beginning;
// 2 to delete the entire line). The function parameter vpar corresponds to the Ps.
static void csi_K(int currcons, int vpar)
{
        long count __asm__("cx");
        long start __asm__("di");

// First, the number of characters to be deleted and the display memory position at which the
// deletion starts are set according to the three conditions in the control sequence.
        switch (vpar) {
                case 0: /* erase from cursor to end of line */
                        if (x>=video_num_columns)
                                return;
                        count = video_num_columns-x;
                        start = pos;
                        break;
                case 1: /* erase from start of line to cursor */
                        start = pos - (x<<1);
                        count = (x<video_num_columns)?x:video_num_columns;
                        break;
                case 2: /* erase whole line */
                        start = pos - (x<<1);
                        count = video_num_columns;
                        break;
                default:
                        return;
        }
// Then use the erase character to fill in the place where the character is deleted.
// %0 - ecx (delete count); %1 -edi (delete address start); %2 -eax (fill erase character).
        __asm__("cld\n\t"
                "rep\n\t"
                "stosw\n\t"
                ::"c" (count),
                "D" (start), "a" (video_erase_char)
                :"cx","di");
}

//// Set the attributes of the display character.
// The control sequence sets the character display attribute according to the parameter. All
// characters sent to the terminal in the future will use the attributes specified here until
// the control sequence is used again to reset the attributes of the character display.
// ANSI escape sequence: 'ESC [ Ps; Ps m'. Where Ps = 0 - default attribute; 1 - bold and bright;
// 4 - underline; 5 - flash; 7 - reverse; 22 - non-bold; 24 - no underline; 25 - no flicker;
// 27 - normal ;30--38 - Set foreground color; 39 - Default foreground color (White);
// 40--48 - Set background color; 49 - Default background color (Black).
void csi_m(int currcons )
{
        int i;

// A control sequence can have multiple different parameters. The parameters are stored in the
// array par[]. The following code cyclically processes each parameter Ps according to the number
// of parameters npar received.
// If Ps = 0, the character attribute that is displayed next to the current virtual console
// is set to the default attribute def_attr. At initialization, def_attr has been set to 0x07
// (white on black).
// If Ps = 1, the character attributes that will be displayed later are set to bold or highlighted.
// If it is a color display, the character attribute or upper 0x08 is used to highlight the
// character; if it is a monochrome display, the character is underlined.
// If Ps = 4, the color and monochrome displays are treated differently. If the color display
// mode is not available at this time, the characters are underlined. If it is a color display,
// if the original vc_bold_attr is not equal to -1, the background color is reset; otherwise,
// the foreground color is reversed. If the foreground color is the same as the background color,
// the foreground color is increased by one and the other color is taken.
        for (i=0;i<=npar;i++)
                switch (par[i]) {
                        case 0: attr=def_attr;break;  /* default */
                        case 1: attr=(iscolor?attr|0x08:attr|0x0f);break;  /* bold */
                        /*case 4: attr=attr|0x01;break;*/  /* underline */
                        case 4: /* bold */
                          if (!iscolor)
                            attr |= 0x01;            // Mono is underlined.
                          else
                          { /* check if forground == background */
                            if (vc_cons[currcons].vc_bold_attr != -1)
                              attr = (vc_cons[currcons].vc_bold_attr&0x0f)|(0xf0&(attr));
                            else
                            { short newattr = (attr&0xf0)|(0xf&(~attr));
                              attr = ((newattr&0xf)==((attr>>4)&0xf)?
                                (attr&0xf0)|(((attr&0xf)+1)%0xf):
                                newattr);
                            }
                          }
                          break;
// If Ps = 5, the character that is subsequently displayed in the current virtual console is
// set to blink, that is, the attribute byte bit 7 is set to 1.
// If Ps = 7, the characters displayed subsequently are set to reverse, that is, the foreground
// and background colors are swapped.
// If Ps = 22, the highlighting of subsequent characters is canceled (cancel bold display).
// If Ps = 24, the underline of the subsequent characters is canceled for monochrome display
// and green is cancelled for color display.
// If Ps = 25, the flashing of the subsequent characters is canceled.
// If Ps = 27, the reverse of the subsequent characters is canceled.
// If Ps = 39, the foreground color of the subsequent characters is reset to default (white).
// If Ps = 49, the background color of the subsequent characters is reset to default (black).
                        case 5: attr=attr|0x80;break;  /* blinking */
                        case 7: attr=(attr<<4)|(attr>>4);break;  /* negative */
                        case 22: attr=attr&0xf7;break; /* not bold */
                        case 24: attr=attr&0xfe;break;  /* not underline */
                        case 25: attr=attr&0x7f;break;  /* not blinking */
                        case 27: attr=def_attr;break; /* positive image */
                        case 39: attr=(attr & 0xf0)|(def_attr & 0x0f); break;
                        case 49: attr=(attr & 0x0f)|(def_attr & 0xf0); break;
// When Ps(par[i]) is another value, the specified foreground or background color is set.
// If Ps = 30..37, the foreground color is set; if Ps=40..47, the background color is set. See
// the instructions following the program list for color values.
                        default:
                          if (!can_do_colour)
                            break;
                          iscolor = 1;
                          if ((par[i]>=30) && (par[i]<=38))        // foreground color.
                            attr = (attr & 0xf0) | (par[i]-30);
                          else  /* Background color */
                            if ((par[i]>=40) && (par[i]<=48))      // background color.
                              attr = (attr & 0x0f) | ((par[i]-40)<<4);
                            else
                                break;
                }
}

//// Set the display cursor.
// The display position of the cursor in the controller is set according to the memory position
// pos of the cursor.
static inline void set_cursor(int currcons)
{
// Since we need to set the display cursor, it means there is a keyboard operation, so we need
// to restore the delay count value of the black screen operation. In addition, the console
// displaying the cursor must be the foreground console.
        blankcount = blankinterval;             // Resets the count of the black screen.
        if (currcons != fg_console)
                return;
// Then use the index register port to select the display control data register r14 (the current
    // display position of the cursor high byte), and then write the current position of the cursor
    // high byte (moving 9 bits to the right means the high byte is moved to the low byte and divided
    // by 2).  This is relative to the default display memory operation. Then use the index register
    // to select r15 and write the low byte of the cursor's current position.
435         cli();
436         outb_p(14, video_port_reg);
437         outb_p(0xff&((pos-video_mem_base)>>9), video_port_val);
438         outb_p(15, video_port_reg);
439         outb_p(0xff&((pos-video_mem_base)>>1), video_port_val);
440         sti();
441 }
442
    // Hide the cursor.
    // Set the cursor to the end of the current virtual console window to hide the cursor.
443 static inline void hide_cursor(int currcons)
444 {
    // First select the display control data register r14 (the current display position of the cursor
    // high byte), and then write the high byte of the cursor position (moving 9 bits to the right
    // means moving the high byte to the low byte and dividing by 2). Then select r15 and write
    // the low byte of the cursor's current position.
445         outb_p(14, video_port_reg);
446         outb_p(0xff&((scr_end-video_mem_base)>>9), video_port_val);
447         outb_p(15, video_port_reg);
448         outb_p(0xff&((scr_end-video_mem_base)>>1), video_port_val);
449 }
450
    //// Send a response sequence to VT100.
    // That is, in response to the host requesting the terminal, the device attribute (DA) is sent
    // to the host. The host requests the terminal to send back a device attribute (DA) control
    // sequence by sending a DA control sequence ('ESC[0c' or 'ESC Z') with no parameters or parameter
    // 0. The terminal then sends back the response sequence defined on line 147 (ie 'ESC [?1; 2c')
    // in response to the host's sequence. This sequence tells the host that the terminal is a VT100
    // compatible terminal with advanced video capabilities. The process is to put the response
    // sequence into the read buffer queue and use the copy_to_cooked() function to process it and
    // put it into the auxiliary (secondary) queue.
451 static void respond(int currcons, struct tty_struct * tty)
452 {
453         char * p = RESPONSE;            // defined on line 147 ('ESC [?1; 2c').
454
455         cli();
456         while (*p) {                    // put response sequence into the read queue.
457                 PUTCH(*p,tty->read_q);  // put one by one. include/linux/tty.h, line 46.
458                 p++;
459         }
460         sti();
461         copy_to_cooked(tty);            // put into the auxiliary queue.tty_io.c，120.
462 }
463
    //// Insert a space character at the cursor.
    // Move all the characters at the beginning of the cursor one space to the right and insert
    // the erase character at the cursor.
464 static void insert_char(int currcons)
465 {
        int i=x;
        unsigned short tmp, old = video_erase_char;      // erase char (with attribute).
        unsigned short * p = (unsigned short *) pos;     // memory position of the cursor.

        while (i++<video_num_columns) {
                tmp=*p;
                *p=old;
                old=tmp;
                p++;
        }
}

//// Insert a line at the cursor.
// Scrolls the screen window down from the line where the cursor is located to the bottom of
// the window. The cursor will be on a new, empty line.
static void insert_line(int currcons)
{
        int oldtop,oldbottom;

// First save the screen window scrolling start line top and last line bottom value, then scroll
// the screen content down one line from the line where the cursor is. Finally, restore the
// original value of the scroll start line 'top' and the last line 'bottom'.
        oldtop=top;
        oldbottom=bottom;
        top=y;                           // set start and end lines of the scroll screen.
        bottom = video_num_lines;
        scrdown(currcons);               // the screen content scrolls down one line
        top=oldtop;
        bottom=oldbottom;
}

//// Delete a character.
// Delete one character at the cursor, and all characters to the right of the cursor are shifted
// to the left by one space.
static void delete_char(int currcons)
{
        int i;
        unsigned short * p = (unsigned short *) pos;

// Returns if the cursor's current column position x exceeds the rightmost column of the screen.
// Otherwise, all characters from the right character of the cursor to the end of the line are
// shifted to the left by one space. Then fill in the erase character at the last character.
        if (x>=video_num_columns)
                return;
        i = x;
        while (++i < video_num_columns) {        // shifted to the left by one space.
                *p = *(p+1);
                p++;
        }
        *p = video_erase_char;                   // finally fill in the erase character.
}

//// Delete one line where the cursor is located.
```

```
// Delete one line where the cursor is located, and then scrolls screen content up one line.
static void delete_line(int currcons)
{
        int oldtop,oldbottom;

// First save the screen scrolling start line 'top' and last line 'bottom', then scroll the
// screen content up one line from the line where the cursor is. Finally restore the original
// value of the screen scrolling start line 'top' and last line 'bottom'.
        oldtop=top;
        oldbottom=bottom;
        top=y;                            // set start and end lines of the scroll screen.
        bottom = video_num_lines;
        scrup(currcons);                  // the screen content scrolls up one line.
        top=oldtop;
        bottom=oldbottom;
}

//// Insert nr characters at the cursor.
// Handling ANSI escape character sequences: 'ESC [ Pn @'. Insert one or more space characters
// at the current cursor. Pn is the number of characters inserted, and the default is 1. The
// cursor will still be at the first inserted space character. The character at the cursor and
// right border will shift to the right and characters beyond the right border will be lost.
// Parameter nr = Pn in the escape sequence.
static void csi_at(int currcons, unsigned int nr)
{
// If the number of inserted characters is greater than one line of characters, it is truncated
// to one line of characters; if the number of inserted characters nr is 0, one character is
// inserted. Then cyclically insert nr space characters.
        if (nr > video_num_columns)
                nr = video_num_columns;
        else if (!nr)
                nr = 1;
        while (nr--)
                insert_char(currcons);
}

//// Insert nr lines at the cursor position.
// Handling ANSI escape sequence: 'ESC [ Pn L'. The control sequence inserts one or more blank
// lines at the cursor. The cursor position does not change after the operation is completed.
// When a blank line is inserted, the line in the scroll area below the cursor moves down. The
// line scrolling out of the display page is lost. Parameter nr = Pn in the escape sequence.
static void csi_L(int currcons, unsigned int nr)
{
// If the number of inserted lines is greater than the maximum number of screen lines, the number
// of lines is cut off to the screen lines; if the number of inserted lines nr is 0, one line
// is inserted.
Then cyclically inserts nr blank lines.
        if (nr > video_num_lines)
                nr = video_num_lines;
        else if (!nr)
                nr = 1;
        while (nr--)
                insert_line(currcons);
```

```
}

//// Delete nr characters at the cursor.
// Handling ANSI escape sequences: 'ESC [Pn P'. This control sequence deletes Pn characters
// from the cursor. When a character is deleted, all characters to the right of the cursor are
// shifted to the left, which produces a space character at the right border. Its properties
// should be the same as the last left-shift character, but here it is simplified, using only
// the default property of the character (black background white foreground space 0x0720) to
// set the space character. Parameter nr = Pn in the escape sequence.
static void csi_P(int currcons, unsigned int nr)
{
// If the number of deleted characters is greater than one line of characters, it is truncated
// to one line of characters; if the number of deleted characters nr is 0, one character is
// deleted. Then iteratively deletes the specified number of characters nr at the cursor.
        if (nr > video_num_columns)
                nr = video_num_columns;
        else if (!nr)
                nr = 1;
        while (nr--)
                delete_char(currcons);
}

//// Delete the nr line at the beginning of the cursor.
// Process ANSI escape sequences: 'ESC [ Pn M'. The control sequence deletes one or more lines
// from the line where the cursor is located in the scroll area. When a line is deleted, the
// line below the deleted line in the scroll area moves up, and 1 blank line is added to the
// bottom line. If Pn is greater than the number of lines remaining on the display page, then
// this sequence only deletes these remaining lines and does not work outside the scroll area.
// Parameter nr = Pn in the escape sequence.
static void csi_M(int currcons, unsigned int nr)
{
// If the number of deleted lines is greater than the maximum number of lines on the screen,
// the number of lines displayed is cut off. If the number of lines to be deleted nr is 0, 1
// line is deleted. Then iteratively deletes the specified nr lines.
        if (nr > video_num_lines)
                nr = video_num_lines;
        else if (!nr)
                nr=1;
        while (nr--)
                delete_line(currcons);
}

//// Save the current cursor position.
static void save_cur(int currcons)
{
        saved_x=x;
        saved_y=y;
}

//// Restore the saved cursor position.
static void restore_cur(int currcons)
{
        gotoxy(currcons,saved_x, saved_y);
```

```
}


// The following enumeration definitions are used in the following con_write() function to
// resolve escape sequences or control sequences. ESnormal is the initial entry state and is
// also the state when the escape or control sequence is processed.
// ESnormal - Indicates that it is in the initial normal state. At this time, if the normal
//    display character is received, the character is directly displayed on the screen; if a
//    control character (such as a carriage return) is received, the cursor position is set.
//    When an escape or control sequence has just been processed, the program will return to
//    this state.
// ESesc - indicates that the escape sequence leading character ESC (0x1b = 033 = 27) was
//    received. If a ’[’ character is received in this state, it is an escape sequence guide
//    code, so it jumps to ESsquare for processing, otherwise the received character is treated
//    as an escape sequence. For selecting the character set escape sequences ’ESC (’ and ’ESC )’,
//    we use a separate state ESsetgraph to handle. For the device control string sequence
//    ’ESC P’, we use a separate state ESsetterm to handle.
// ESsquare - Indicates that a control sequence preamble (’ESC [’) has been received, indicating
//    that a control sequence has been received, so this state performs a zero initialization
//    on the parameter array par[]. If you receive a ’[’ character at this time, it means that
//    you received the ’ESC [[’ sequence, which is the sequence sent by the keyboard function
//    key, so you can jump to Esfunckey to process it. Otherwise we need to prepare to receive
//    the parameters of the control sequence, so set the state Esgetpars and directly enter
//    the state to receive and save the sequence parameter characters.
// ESgetpars - This state indicates that we are going to receive the parameter values of the
//    control sequence at this time. The arguments are represented in decimal numbers, and we
//    convert the received numeric characters to numeric values and save them to the par[] array.
//    If a semicolon ’;’ is received, it remains in this state and the received parameter value
//    is saved in the next item of data par[]. If it is not a numeric character or a semicolon,
//    indicating that all parameters have been obtained, then move to the state ESgotpars to
//    process.
// ESgotpars - Indicates that we have received a complete control sequence. At this point we
//    can process the corresponding control sequence according to the ending character received
//    in this state. However, before processing, if we received ’?’ in the ESsquare state, this
//    sequence is a private sequence of terminal devices. This kernel does not support the
//    processing of this sequence, so we directly restore to the ESNormal state. Otherwise,
//    the corresponding control sequence is executed, and the state is restored to Esnormal
//    after the sequence is processed.
// ESfunckey - Indicates that we have received a sequence from the function keys on the keyboard
//    that we don’t need to display. So we return to the normal state ESnormal.
// ESsetterm - Indicates that it is in the device control string sequence (DCS) state. At this
//    time, if the character ’S’ is received, the initial display character attribute is
//    restored. If the received character is ’L’ or ’l’, the folding line display mode is turned
//    on or off.
// ESsetgraph - Represents the received character set escape sequence ’ESC (’ or ’ESC )’, which
//    are used to specify the character set used by G0 and G1, respectively. At this time, if
//    the character ’0’ is received, the graphic character set is selected as G0 and G1, and
//    if the received character is ’B’, the ordinary ASCII character set is selected as the
//    character set of G0 and G1.
enum { ESnormal, ESesc, ESsquare, ESgetpars, ESgotpars, ESfunckey,
        ESsetterm, ESsetgraph };

//// Console writes function.
```

```
// Characters are taken from the terminal's tty write buffer queue and analyzed for each
// character. If it is a control character or an escape or control sequence, control processing
// such as cursor positioning and character deletion is performed, and the normal character
// is directly displayed at the cursor.
// The parameter tty is the tty structure pointer used by the current console.
void con_write(struct tty_struct * tty)
{
        int nr;
        char c;
        int currcons;

// The function first obtains the console number currcons according to the position of the tty
// in the tty table used by the current console, and then calculates (CHARS()) the number of
// characters nr contained in the current tty write queue, and loops out each character one
// by one for processing. However, if the current console is stopped due to a pause command
// issued by the keyboard or program (such as the button Ctrl-S), then the function stops
// processing the characters in the write queue and exits the function. In addition, if the
// control character CAN (Cancel, ASCII code 24, generated by Ctrl-X) or SUB (Substitute, 26,
// Ctrl-Z) is taken, then if the character is received during the escaping or control sequence,
// then the sequence will not execute and will terminate immediately, with the subsequent
// characters displayed. Note that the con_write() function only processes the characters
// currently in the write queue when fetching the number of characters in the queue. It is
// possible to read the number of characters during a sequence being placed in the write queue,
// so the 'state' may be in the other state of processing the escape or control sequence when
// the function exits last time.
        currcons = tty - tty_table;
        if ((currcons>=MAX_CONSOLES) || (currcons<0))
                panic("con_write: illegal tty");

        nr = CHARS(tty->write_q);               // get nr of chars in the write queue.
        while (nr--) {
                if (tty->stopped)
                        break;
                GETCH(tty->write_q,c);          // get one character.
                if (c == 24 || c == 26)         // Control chars: Cancel or Substitute.
                        state = ESnormal;
                switch(state) {
// If the character extracted is a normal display character, the corresponding display character
// is directly taken out from the current mapped character set, and placed at the display memory
// position where the current cursor is located, that is, the character is directly displayed.
// Then move the cursor position to the right by one character position. Specifically, if the
// character is not a control character or an extended character, ie (31<c<127), then if the
// current cursor is at the end of the line, or at a position other than the end, the cursor
// is moved to the first column of the next line. And adjust the memory pointer pos corresponding
// to the cursor position. The character  is then written to the display memory pos, and the
// cursor is shifted to the right by 1 column, and pos is also moved 2 bytes correspondingly.
                        case ESnormal:
                                if (c>31 && c<127) {            // normal display char.
                                        if (x>=video_num_columns) {     // change line?
                                                x -= video_num_columns;
                                                pos -= video_size_row;
                                                lf(currcons);
                                        }
599                         __asm__("movb %2,%%ah\n\t"          // write char.
600                                 "movw %%ax,%1\n\t"
601                                 ::"a" (translate[c-32]),
602                                 "m" (*(short *)pos),
603                                 "m" (attr)
604                                 :"ax");
605                         pos += 2;
606                         x++;
    // If 'c' is the escape character ESC, the state is converted to ESesc (line 637).
    // If c is LF(10), or VT(11), or FF(12), the cursor moves to the next line.
    // If c is a carriage return CR (13), move the cursor to the head column (column 0).
    // If c is DEL(127), the character to the left of the cursor is erased and the cursor is moved
    // to the erased character position.
    // If c is BS (backspace, 8), move the cursor to the left by 1 column and adjust the cursor
    // corresponding to the memory pointer pos.
607                 } else if (c==27)               // ESC - escape char.
608                         state=ESesc;
609                 else if (c==10 || c==11 || c==12)
610                         lf(currcons);
611                 else if (c==13)                 // CR - carriage return.
612                         cr(currcons);
613                 else if (c==ERASE_CHAR(tty))
614                         del(currcons);
615                 else if (c==8) {                // BS - backspace.
616                         if (x) {
617                                 x--;
618                                 pos -= 2;
619                         }
    // If c is horizontal tab HT(9), the cursor is moved to a multiple of 8. If the number of cursor
    // columns exceeds the maximum columns on the screen, then move the cursor to the next line.
    // If c is the bell BEL (7), the system beep function is called, and the speaker sounds.
    // If c is the control character SO(14) or SI(15), the character set G1 or G0 is selected as
    // the display character set accordingly.
620                 } else if (c==9) {                 // HT - Horizontal Tab.
621                         c=8-(x&7);
622                         x += c;
623                         pos += c<<1;
624                         if (x>video_num_columns) {
625                                 x -= video_num_columns;
626                                 pos -= video_size_row;
627                                 lf(currcons);
628                         }
629                         c=9;
630                 } else if (c==7)                   // BEL - Bell.
631                         sysbeep();
632                 else if (c == 14)                  // SO - Shift out, use G1.
633                         translate = GRAF_TRANS;
634                 else if (c == 15)                  // SI - Shift in, use G0.
635                         translate = NORM_TRANS;
636                 break;
    // If the escape character ESC (0x1b = 27) is received in the ESnormal state, it goes to this
    // state processing. This state processes the control characters or escape characters in c.
    // After processing, the default state will be ESnormal.
```

```
                        case ESesc:
                                state = ESnormal;
                                switch (c)
                                {
                                  case '[':            // ESC [ - CSI sequence.
                                        state=ESsquare;
                                        break;
                                  case 'E':            // ESC E - cursor next line & col 0.
                                        gotoxy(currcons,0,y+1);
                                        break;
                                  case 'M':            // ESC M - moves up one line.
                                        ri(currcons);
                                        break;
                                  case 'D':            // ESC D - moves to next line.
                                        lf(currcons);
                                        break;
                                  case 'Z':            // ESC Z - device property query.
                                        respond(currcons,tty);
                                        break;
                                  case '7':            // ESC 7 - save cursor position.
                                        save_cur(currcons);
                                        break;
                                  case '8':            // ESC 8 - restore cursor position.
                                        restore_cur(currcons);
                                        break;
                                  case '(':  case ')': // ESC (、ESC ) - select char set.
                                        state = ESsetgraph;
                                        break;
                                  case 'P':            // ESC P - set terminal parameters.
                                        state = ESsetterm;
                                        break;
                                  case '#':            // ESC # - modify line attributs.
                                        state = -1;
                                        break;
                                  case 'c':         // ESC c - reset to default settings.
                                        tty->termios = DEF_TERMIOS;
                                        state = restate = ESnormal;
                                        checkin = 0;
                                        top = 0;
                                        bottom = video_num_lines;
                                        break;
                                 /* case '>':   Numeric keypad */
                                 /* case '=':   Appl. keypad */
                                }
                                break;
// If the character '[' is received in the state ESesc, it indicates that it is a CSI control
// sequence, so it goes to the state ESsequare to handle. First, the array par[] used to save
// the parameters of the ESC sequence is cleared, the index variable npar points to the first
// item, and we set the state to start the parameter ESgetpars. However, if the character received
// at this time is '[', it indicates that the sequence sent by the keyboard function key has
// been received, so the next state is set to ESfunckey. If the received character is not '?',
// go directly to the state ESgetpars to handle. If the received character is '?', it indicates
// that the sequence is a private sequence of terminal devices, followed by a function character.
// So go to the next character and go to the state ESgetpars to handle the code.
                        case ESsquare:
                                for(npar=0;npar<NPAR;npar++)      // Initialize para array.
                                        par[npar]=0;
                                npar=0;
                                state=ESgetpars;
                                if (c =='[')       /* Function key */    // 'ESC [['
                                { state=ESfunckey;
                                  break;
                                }
                                if (ques=(c=='?'))
                                        break;
// This state indicates that we are going to receive the parameter values of the control sequence
// at this time. The parameters are represented in decimal numbers, and we convert the received
// numeric characters to values and save them to the par[] array. If you receive a semicolon
// ';', it remains in this state and saves the received parameter value in the next item in
// the array par[]. If it is not a numeric character or a semicolon, indicating that all parameters
// have been obtained, then move to the state ESgotpars to process.
                        case ESgetpars:
                                if (c==';' && npar<NPAR-1) {
                                        npar++;
                                        break;
                                } else if (c>='0' && c<='9') {
                                        par[npar]=10*par[npar]+c-'0';
                                        break;
                                } else state=ESgotpars;
// The ESgotpars state indicates that we have received a complete control sequence. At this
// point we can process the control sequence based on the ending character received in this
// state. However, before processing, if we received '?' in the ESsquare state, this sequence
// is a private sequence of terminal devices. This kernel does not support the processing of
// this sequence, so we directly restore to the ESnormal state. Otherwise go to the corresponding
// control sequence. After the sequence is processed, the state is restored to ESnormal.
                        case ESgotpars:
                                state = ESnormal;
                                if (ques)                 // received '?'
                                { ques =0;
                                  break;
                                }
                                switch(c) {
// If c is 'G' or '`', the first value in par[] represents the column number, and if it is not
// zero, the cursor is shifted to the left by one column.
// If c is 'A', the first value represents the number of lines moved up by the cursor. If the
// parameter is 0, it moves up one line.
// If c is 'B' or 'e', the first parameter represents the number of lines moved down by the
// cursor. If the parameter is 0, it will move down one line.
// If c is 'C' or 'a', the first parameter represents the number of columns to the right of
// the cursor. If the parameter is 0, it is shifted to the right by 1 column.
// If c is 'D', the first parameter represents the number of columns to the left of the cursor.
// If the parameter is 0, it will be shifted to the left by 1 column.
                                        case 'G': case '`': // CSI Pn G -move horizontally
                                                if (par[0]) par[0]--;
                                                gotoxy(currcons,par[0],y);
                                                break;
```

```
                    case 'A':           // CSI Pn A - move up.
                        if (!par[0]) par[0]++;
                        gotoxy(currcons,x,y-par[0]);
                        break;
                    case 'B': case 'e': // CSI Pn B - move down.
                        if (!par[0]) par[0]++;
                        gotoxy(currcons,x,y+par[0]);
                        break;
                    case 'C': case 'a': // CSI Pn C - move right.
                        if (!par[0]) par[0]++;
                        gotoxy(currcons,x+par[0],y);
                        break;
                    case 'D':           // CSI Pn D - move left.
                        if (!par[0]) par[0]++;
                        gotoxy(currcons,x-par[0],y);
                        break;
// If c is 'E', the first value represents the number of lines the cursor moves down and returns
// to column 0. If the value is 0, it will move down one line.
// If c is 'F', the first value represents the number of lines the cursor is moving up and returns
// to column 0. If the parameter is 0, it moves up one line.
// If c is 'd', the first value represents the line number (counted from 0) that the cursor
// is required to have.
// If c is 'H' or 'f', the first value represents the line number to which the cursor is moved,
// and the second parameter represents the column number to which the cursor is moved.
                    case 'E':       // CSI Pn E - move down, col 0.
                        if (!par[0]) par[0]++;
                        gotoxy(currcons,0,y+par[0]);
                        break;
                    case 'F':       // CSI Pn F - move up, col 0.
                        if (!par[0]) par[0]++;
                        gotoxy(currcons,0,y-par[0]);
                        break;
                    case 'd':       // CSI Pn d -  set cursor line no.
                        if (par[0]) par[0]--;
                        gotoxy(currcons,x,par[0]);
                        break;
                    case 'H': case 'f':  // CSI Pn H -set cusor postion.
                        if (par[0]) par[0]--;
                        if (par[1]) par[1]--;
                        gotoxy(currcons,par[1],par[0]);
                        break;
// If the character c is 'J' (sequence 'ESC [ Ps J'), the first parameter represents the way
// the screen is cleared related to the cursor position.
// If c is 'K' ('ESC [ Ps K'), the first parameter represents the way in which the characters
// in the line are deleted related to the cursor position.
// If c is 'L' ('ESC [ Pn L'), it means that n lines are inserted at the cursor position.
// If c is 'M' ('ESC [ Pn M'), it means that n lines are deleted at the cursor position.
// If c is 'P' ('ESC [ Pn P'), it means that n chars are deleted at the cursor position.
// If c is '@' ('ESC [ Pn @'), it means that n chars are inserted at the cursor position.
                    case 'J':       // CSI Pn J - erase chars on screen.
                        csi_J(currcons,par[0]);
                        break;
                    case 'K':       // CSI Pn K - erase chars in a line.
```

```
                                                csi_K(currcons,par[0]);
                                                break;
                                        case 'L':       // CSI Pn L - insert lines.
                                                csi_L(currcons,par[0]);
                                                break;
                                        case 'M':       // CSI Pn M - delete lines.
                                                csi_M(currcons,par[0]);
                                                break;
                                        case 'P':       // CSI Pn P - delete chars.
                                                csi_P(currcons,par[0]);
                                                break;
                                        case '@':       // CSI Pn @ - insert chars.
                                                csi_at(currcons,par[0]);
                                                break;
// If c is 'm' ('ESC [Pn m'), it means changing the display attributes of the characters at
// the cursor, such as bolding.
// If c is 'r' ('ESC [Pn;Pn r'), it means that the starting line number and the ending line
// number of the scrolling are set with two parameters.
// If c is 's' ('ESC [Pn s'), it means that the current cursor position is saved.
// If c is 'u' ('ESC [Pn u'), it means that the cursor is restored to the saved postion.
                                        case 'm':       // CSI Ps m - set char attributes.
                                                csi_m(currcons);
                                                break;
                                        case 'r':       // CSI Pn;Pn r - set scroll range.
                                                if (par[0]) par[0]--;
                                                if (!par[1]) par[1] = video_num_lines;
                                                if (par[0] < par[1] &&
                                                    par[1] <= video_num_lines) {
                                                        top=par[0];
                                                        bottom=par[1];
                                                }
                                                break;
                                        case 's':       // CSI s -  saved cursor postion.
                                                save_cur(currcons);
                                                break;
                                        case 'u':       // CSI u - restore cursor postion.
                                                restore_cur(currcons);
                                                break;
// If the character c is 'l' or 'b', it means setting the screen black screen interval time
// and setting the bold character display, respectively. At this time, par[1] and par[2] are
// feature values, which must be par[1]=par[0]+13;par[2]= par[0]+17 respectively. Under this
// condition, if c is 'l', then par[0] is the number of minutes to delay when starting a black
// screen; if c is 'b', the bold character attribute is set in par[0] value.
                                        case 'l': /* blank interval */
                                        case 'b': /* bold attribute */
                                                  if (!((npar >= 2) &&
                                                  ((par[1]-13) == par[0]) &&
                                                  ((par[2]-17) == par[0])))
                                                    break;
                                                if ((c=='l')&&(par[0]>=0)&&(par[0]<=60))
                                                {
                                                  blankinterval = HZ*60*par[0];
                                                  blankcount = blankinterval;
                                                }
                                                if (c=='b')
                                                  vc_cons[currcons].vc_bold_attr
                                                    = par[0];
                                }
                                break;
// The status ESfunckey indicates that we have received a sequence from the function keys on
// the keyboard. We don't need to display it, so we return to the normal state of ESnormal.
                        case ESfunckey:             // keyboard function key.
                                state = ESnormal;
                                break;
// The state ESsetterm indicates that it is in the device control string (DCS) sequence state.
// At this time, if the character 'S' is received, the initial display character attribute is
// restored. If the character is 'L' or 'l', the folding display mode is turned on or off.
                        case ESsetterm:  /* Setterm functions. */
                                state = ESnormal;
                                if (c == 'S') {
                                        def_attr = attr;
                                        video_erase_char = (video_erase_char&0x0ff) |
                                                           (def_attr<<8);
                                } else if (c == 'L')
                                        ; /*linewrap on*/
                                else if (c == 'l')
                                        ; /*linewrap off*/
                                break;
// The state ESsetgraph indicates that the set character set escape sequence 'ESC (' or 'ESC )'
// has been received. They are used to specify the character set used by G0 and G1, respectively.
// At this time, if the character '0' is received, the graphic character set is selected as
// G0 and G1, and if the received character is 'B', the ordinary ASCII character set is selected
// as the character set of G0 and G1.
                        case ESsetgraph:        // 'CSI (0' or 'CSI (B' - select char set.
                                state = ESnormal;
                                if (c == '0')
                                        translate = GRAF_TRANS;
                                else if (c == 'B')
                                        translate = NORM_TRANS;
                                break;
                        default:
                                state = ESnormal;
                }
        }
//  Finally, set the cursor position in the display controller with the value set above.
        set_cursor(currcons);
}

/*
 *  void con_init(void);
 *
 * This routine initalizes console interrupts, and does nothing
 * else. If you want the screen to clear, call tty_write with
 * the appropriate escape-sequece.
 *
 * Reads the information preserved by setup.s to determine the current display
```

```
832  * type and sets everything accordingly.
833  */
834 void con_init(void)
835 {
836         register unsigned char a;
837         char *display_desc = "????";
838         char *display_ptr;
839         int currcons = 0;                       // current console no.
840         long base, term;
841         long video_memory;
842
    // First, according to the system hardware parameters obtained by the setup.s program, several
    // static global variables specific to this function are initialized (see lines 60-68).
843         video_num_columns = ORIG_VIDEO_COLS;
844         video_size_row = video_num_columns * 2;    // bytes used by a display row.
845         video_num_lines = ORIG_VIDEO_LINES;        // lines of the display.
846         video_page = ORIG_VIDEO_PAGE;              // display page.
847         video_erase_char = 0x0720;                 // char: 0x20, attr: 0x07.
848         blankcount = blankinterval;                // unit: ticks.
849
    // Then, according to whether the display mode is monochrome or color, the display memory start
    // position and the display register index port number and the display register data port number
    // are respectively set. If the obtained BIOS display mode is equal to 7, it means that it is
    // a monochrome display card.
850         if (ORIG_VIDEO_MODE == 7)                 /* Is this a monochrome display? */
851         {
852                 video_mem_base = 0xb0000;         // start addr of monochrome display.
853                 video_port_reg = 0x3b4;           // index register port.
854                 video_port_val = 0x3b5;           // data register port.

    // Then we determine whether the display card is a monochrome card or a color card according
    // to the display mode information obtained by the BIOS interrupt int 0x10 function 0x12. If
    // the return value of the BX register obtained by the interrupt is not equal to 0x10, it means
    // that it is an EGA card, so the initial display type is EGA monochrome. Although there is
    // more display memory on the EGA card, it can only use up to 32KB of display memory with an
    // address range between 0xb0000--0xb8000 in monochrome mode. The code then sets the display
    // description string to 'EGAm' and it will be displayed in the upper right corner of the screen
    // during system initialization.
    // Note that the BX register is used to determine the type of card if it is changed before and
    // after the interrupt int 0x10 is called. If the value of the BL is changed after the interrupt
    // is called, it means that the display card supports the Ah=12h function call, which is a type
    // of VGA or later VGA. If the return value of the interrupt call has not changed, indicating
    // that the display card does not support this function, it indicates that it is a general
    // monochrome display card.
855                 if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
856                 {
857                         video_type = VIDEO_TYPE_EGAM;  // video type (EGA mono).
858                         video_mem_term = 0xb8000;      // video memory end address.
859                         display_desc = "EGAm";
860                 }
    // If the value of the BX register is equal to 0x10, it means a monochrome display card MDA
    // and only 8KB of display memory.
861                 else
```

```
                {
                        video_type = VIDEO_TYPE_MDA;    // MDA mono.
                        video_mem_term = 0xb2000;      // memory end address.
                        display_desc = "*MDA";
                }
        }
// If the display mode is not 7, it indicates a color display card. At this time, the display
// memory starting address used in the text mode is 0xb8000.
        else                            /* If not, it is color. */
        {
                can_do_colour = 1;                      // can use color flag.
                video_mem_base = 0xb8000;               // memory start address.
                video_port_reg  = 0x3d4;                // index register port.
                video_port_val  = 0x3d5;                // data register port.
// Then judge the display card category. If BX is not equal to 0x10, it means that it is an
// EGA card. At this time, a total of 32KB of display memory is available (0xb8000-0xc0000).
// Otherwise, it is a CGA card and can only use 8KB display memory (0xb8000-0xba000).
                if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
                {
                        video_type = VIDEO_TYPE_EGAC;   // EGA type.
                        video_mem_term = 0xc0000;      // mem end address.
                        display_desc = "EGAc";
                }
                else
                {
                        video_type = VIDEO_TYPE_CGA;    // CGA type.
                        video_mem_term = 0xba000;      // meme end address.
                        display_desc = "*CGA";
                }
        }

// Now let's calculate the number of virtual consoles that can be opened on the current display
// card. The number of virtual consoles allowed by the hardware is equal to the total amount
// of memory video_memory divided by the number of bytes occupied by each virtual console. The
// number of display memory occupied by each virtual console is equal to the number of lines
// multiplied by the number of bytes per line of characters: video_num_lines * video_size_row.
// If the number of virtual consoles allowed is greater than the maximum number of system-defined
// MAX_CONSOLES, set the number of virtual consoles to MAX_CONSOLES. If the number of virtual
// consoles thus calculated is 0, it is set to 1.Finally, the total number of video memory divided
// by the number of virtual consoles is the number of bytes of memory occupied by each virtual
// console (vc).
        video_memory = video_mem_term - video_mem_base;
        NR_CONSOLES = video_memory / (video_num_lines * video_size_row);
        if (NR_CONSOLES > MAX_CONSOLES)               // MAX_CONSOLES = 8
                NR_CONSOLES = MAX_CONSOLES;
        if (!NR_CONSOLES)
                NR_CONSOLES = 1;
        video_memory /= NR_CONSOLES;                  // memory of each vc.

        /* Let the user known what kind of display driver we are using */

// Then we display the description string in the upper right corner of the screen. The method
// used is to write the string directly to the corresponding location in the display memory.
```

```
    // First, the display pointer display_ptr points to the last 4 characters of the right end of
    // the first line (each character requires 2 bytes, so minus 8), and then cyclically copies
    // the character of the string, and reserve one attribute byte for each character.
897         display_ptr = ((char *)video_mem_base) + video_size_row - 8;
898         while (*display_desc)
899         {
900                 *display_ptr++ = *display_desc++;
901                 display_ptr++;
902         }
903
904         /* Initialize the variables used for scrolling (mostly EGA/VGA) */
905
    // Note that the current virtual console number currcons has been initialized to 0 at this time.
    // So the following is actually the initialization of all the fields in the structure vc_cons[0].
    // For example, the symbol 'origin' here has been defined as 'vc_cons[0].vc_origin' on line
    // 115 above. Below we first set the default scrolling start memory location video_mem_start
    // and the default scrolling last line memory location (actually they are part of the display
    // memory area occupied by console 0), and then set other properties of virtual console 0.
906         base = origin = video_mem_start = video_mem_base;  // default scroll start address
907         term = video_mem_end = base + video_memory;        // end mem of #0 vc.
908         scr_end = video_mem_start + video_num_lines * video_size_row; // end mem of scroll.
909         top     = 0;                          // top line number of scrolling.
910         bottom  = video_num_lines;            // bottom line number of scrolling.
911         attr = 0x07;                          // default attribute (white on black).
912         def_attr = 0x07;                      // set default char attribute.
913         restate = state = ESnormal;           // current and next state of escape sequence.
914         checkin = 0;
915         ques = 0;                             // got question mark flag.
916         iscolor = 0;                          // color flag.
917         translate = NORM_TRANS;               // char set used (ordinary ASCII table).
918         vc_cons[0].vc_bold_attr = -1;         // bold char attribute flag (-1 not used).
919
    // After setting the current cursor position of console 0 and the memory location pos
    // corresponding to the cursor, we loop through the structure of the remaining virtual consoles.
    // Except for the start and end positions of the display memory occupied by each, their remaining
    // initial values are basically the same as console 0.
920         gotoxy(currcons,ORIG_X,ORIG_Y);
921         for (currcons = 1; currcons<NR_CONSOLES; currcons++) {
922                 vc_cons[currcons] = vc_cons[0];       // copy structure data of #0
923                 origin = video_mem_start = (base += video_memory);
924                 scr_end = origin + video_num_lines * video_size_row;
925                 video_mem_end = (term += video_memory);
926                 gotoxy(currcons,0,0);                 // cursor is at top left.
927         }
    // Finally, set the screen origin (upper left corner) position of the current foreground console
    // and the cursor position in the display controller, and set the keyboard interrupt 0x21 trap
    // gate descriptor. The masking of the keyboard interrupt is then disabled, allowing the IRQ1
    // request signal from the keyboard to be responded to. Finally reset the keyboard controller
    // to allow the keyboard to start working properly.
928         update_screen();                          // update foreground origin & cursor pos.
929         set_trap_gate(0x21,&keyboard_interrupt);  // refer to system.h, line 36.
930         outb_p(inb_p(0x21)&0xfd,0x21);            // cancel masking of keyboard.
931         a=inb_p(0x61);                            // read keyboard port 0x61 (8255A PB).
```

```
        outb_p(a|0x80,0x61);                     // disable keyboard (set bit 7).
        outb_p(a,0x61);                          // enabe it again to reset keyboard.
}

// Update the current foreground console.
// Switch the foreground console to the virtual console specified by fg_console. fg_console
// is the foreground virtual console number set.
void update_screen(void)
{
        set_origin(fg_console);                 // set the scroll start memory address.
        set_cursor(fg_console);                 // set cursor position in the controller.
}

/* from bsd-net-2: */

//// Stop beeping.
// Reset bit 1 and bit 0 of the 8255A PB port. See the timer programming instructions after
// the kernel/sched.c program.
void sysbeepstop(void)
{
        /* disable counter 2 */
        outb(inb_p(0x61)&0xFC, 0x61);
}

int beepcount = 0;                  // beeping counts (ticks).

// Turn on the beep.
// Bit 1 of the 8255A chip PB port is used as the door open signal of the speaker; bit 0 is
// used as the door signal of the 8253 timer 2, and the output pulse of the timer is sent to
// the speaker as the frequency at which the speaker emits sound. Therefore, to make the speaker
// beep, two steps are required: first turn on the PB port (0x61) bit 1 and bit 0 (set), then
// set the timer 2 channel to send a certain timing frequency. See the 8259A interrupt controller
// chip programming method after the boot/setup.s program, and refer to the programming
// instructions for the timer after the kernel/sched.c program.
static void sysbeep(void)
{
        /* enable counter 2 */
        outb_p(inb_p(0x61)|3, 0x61);
        /* set command for counter 2, 2 byte write */
        outb_p(0xB6, 0x43);               // timer control word register port.
        /* send 0x637 for 750 HZ */
        outb_p(0x37, 0x42);               // send high & low count bytes to channel 2.
        outb(0x06, 0x42);
        /* 1/8 second */
        beepcount = HZ/8;
}

//// Copy the screen.
// Copy the screen contents to the user buffer arg specified by the parameter.
// The parameter arg has two purposes, one is to pass the console number, and the other is to
// act as a user buffer pointer.
int do_screendump(int arg)
{
        char *sptr, *buf = (char *)arg;
        int currcons, l;

// The function first verifies the buffer capacity provided by the user, and if it is not enough,
// it expands appropriately. Then take the console number currcons from its beginning. After
// determining that the console number is valid, all the memory contents of the console screen
// are copied to the user buffer.
        verify_area(buf,video_num_columns*video_num_lines);
        currcons = get_fs_byte(buf);
        if ((currcons<1) || (currcons>NR_CONSOLES))
                return -EIO;
        currcons--;
        sptr = (char *) origin;
        for (l=video_num_lines*video_num_columns; l>0 ; l--)
                put_fs_byte(*sptr++,buf++);
        return(0);
}

// Black screen processing.
// When the user does not press any button during the blankInterval interval, the screen is
// blacked out to protect the screen.
void blank_screen()
{
        if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
                return;
/* blank here. I can't find out how to do it, though */
}

// Restore the black screen.
// When the user presses any button, the screen in the black screen state is restored.
void unblank_screen()
{
        if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
                return;
/* unblank here */
}

//// The console print function.
// This function is only used by the kernel display function printk() (kernel/printk.c) to print
// kernel information on the current foreground console. The processing method is to loop out
// the characters in the buffer and control the cursor movement or directly display on the screen
// according to the characteristics of the characters. The argument b is a null-terminated string
// buffer pointer.
void console_print(const char * b)
{
        int currcons = fg_console;
        char c;

// Cycle through the characters in the buffer. If the current character is a newline character,
// a carriage return line feed operation is performed on the cursor; then the next character
// is processed. If it is a carriage return, it will directly perform a carriage return. Then
// go to the next character.
        while (c = *(b++)) {
                if (c == 10) {
                        cr(currcons);
                        lf(currcons);
                        continue;
                }
                if (c == 13) {
                        cr(currcons);
                        continue;
                }
// After reading a character that is not a carriage return or a newline, if you find that the
// current cursor column position has reached the right end of the screen, let the cursor fold
// back to the beginning of the next line. Then place the character at the memory location where
// the cursor is located, which is displayed on the screen. Move the cursor right one column
// to prepare for the next character.
                if (x>=video_num_columns) {
                        x -= video_num_columns;
                        pos -= video_size_row;
                        lf(currcons);
                }
// The register AL is the character to be displayed. Here the attribute byte is placed in AH,
// and then the AX content is stored in the cursor memory location pos, that is, the character
// is displayed at the cursor.
                __asm__("movb %2,%%ah\n\t"          // attribute byte -> ah.
                        "movw %%ax,%1\n\t"          // put value of ax to pos.
                        ::"a" (c),
                        "m" (*(short *)pos),
                        "m" (attr)
                        :"ax");
                pos += 2;
                x++;
        }
//  Finally, set the cursor position in the display controller with the value set above.
        set_cursor(currcons);
}
```

10.3.3 Infomation

10.3.3.1 VGA adapter programming

Here we only give instructions on IBM VGA and its compatible graphics card ports, mainly describing the general programming ports of MDA, CGA, EGA and VGA display control cards. These ports are compatible with the MC6845 chip used by the CGA. The names and uses are shown in Table 10-5. The CGA/EGA/VGA port (0x3d0-0x3df) is taken as an example for description. The port range of the MDA is 0x3b0 - 0x3bf.

The basic steps for programming the display card are: first write the 0--17 value into the index register of the display card (port 0x3d4), select one of the display control internal registers (r0--r17), and the data register port (0x3d5)) corresponds to the internal register. The parameters are then written to the data register port. That is, the data register port of the display card can only operate on one internal register in the display card at a time. The main internal registers of the display card are shown in Table 10-6.

Table 10-5 CGA port register name and functions

Port	R/W	Name and usage
0x3b4/0x3d4	W	6845 index register. Used to select which register (r0-r17) is to be accessed through port 0x3d5.
0x3b5/0x3d5	W	6845 data register. registersr 14-r17 may be read. The function description of each data register is shown in Table 10-6.

0x3b8/0x3d8	R/W	6845 mode control register
		bit 7-6 unused;
		bit 5=1 blinking on;
		bit 4=1 640*200 B/W graphics mode;
		bit 3=1 enable video signal;
		bit 2=1 B/W, 0= color;
		bit 1=1 320*200 graphics, =0 text;
		bit 0=1 80*25 text; =0 40*25 text.

0x3b9/0x3d9	R/W	CGA palette register. Choose the color you are using.
		bit 7-6 unused;
		bit 5=1 Activate the color set: cyan, magenta, white;
		=0 Activate color set: red, green, blue;
		bit 4=1 Enhance graphics, text display background color;
		bit 3=1 Enhanced display of 40*25 borders, 320*200 background, 640*200 foreground color
		bit 2=1 Display red: 40*25 border, 320*200 background, 640*200 foreground;
		bit 1=1 Display green: 40*25 border, 320*200 background, 640*200 foreground;
		bit 0=1 Display blue: 40*25 border, 320*200 background, 640*200 foreground;

0x3ba/0x3da	R	CGA status register.
		bit 7-4 unused;
		bit 3=1 virtical retrace, RAM access OK for next 1.25ms;
		bit 2=1 light pen off, =0 light pen on;
		bit 1=1 light pen trigger set;
		bit 0=1 access memory without disturbing the display; = 0 Do not use memory at this time.

0x3bb/0x3db	W	Clear the light pen latch (reset the light pen register).
0x3dc	R/W	Pre-set the light pen latch (forced light pen strobe is valid).

Table 10-6 MC6845 internal data register and initial values

Reg.	Register name	Unit	R/W	40*25mode	80*25 mode	Graphics mode
r0	Horizontal Total	Char	W	0x38	0x71	0x38
r1	Horizontal Displayed	Char	W	0x28	0x50	0x28
r2	Horizontal Sync Position	Char	W	0x2d	0x5a	0x2d
r3	Horizontal Sync Paulse Width	Char	W	0x0a	0x0a	0x0a
r4	Vertical Total	Char row	W	0x1f	0x1f	0x7f
r5	Vertical Total Adjust	Scan line	W	0x06	0x06	0x06
r6	Vertical Displayed	Char row	W	0x19	0x19	0x64
r7	Vertical Sync Position	Char row	W	0x1c	0x1c	0x70
r8	Interlace Mode		W	0x02	0x02	0x02
r9	Maximum Scan Line Address	Scan line	W	0x07	0x07	0x01

r10	Cursor Start	Scan line	W	0x06	0x06	0x06
r11	Cursor End	Scan line	W	0x07	0x07	0x07
r12	Start Mem Address (High)		W	0x00	0x00	0x00
r13	Start Mem Address (Low)		W	0x00	0x00	0x00
r14	Cursor position (High)		R/W	Vary		
r15	Cursor position (Low)		R/W			
r16	Light Pen (High)		R	Vary		
r17	Light Pen (Low)		R			

10.3.3.2 Principles of scrolling operation

Scrolling refers to moving a piece of text on the specified start and end lines on the screen up (scroll up) or down (scroll down). If you think of the screen as a window that displays the corresponding screen content on the memory, moving the screen contents up is to move the window down the display memory; moving the screen contents down is to move the window up. In the program, the starting position 'origin' of the video memory in the controller is re-set, and the corresponding variables in the adjustment program are adjusted. There are two cases for each of these two operations.

For the scroll up, when the corresponding display memory window of the screen is still within the display memory range after moving downward, that is, the memory block position corresponding to the current screen is always between the memory start position (video_mem_start) and the end position video_mem_end, then we only need to adjust the starting display memory location in the display controller. However, when the position of the memory block corresponding to the screen moves beyond the end of the actual memory (video_mem_end), it is necessary to move the data in the corresponding memory to ensure that all current screen data falls within the display memory range. In this second case, the program moves the memory data corresponding to the screen to the beginning of the actual display memory (video_mem_start).

The actual processing in the program is carried out in three steps. First adjust the screen display starting position origin; then determine whether the corresponding screen memory data exceeds the display memory lower bound (video_mem_end), if it is exceeded, move the screen corresponding memory data to the beginning of the actual display memory (video_mem_start); The new line that appears on the screen is filled with space characters. The operation diagram is shown in Figure 10-8. Figure (a) corresponds to the first simple case, and Figure (b) corresponds to the case when the memory data needs to be moved.

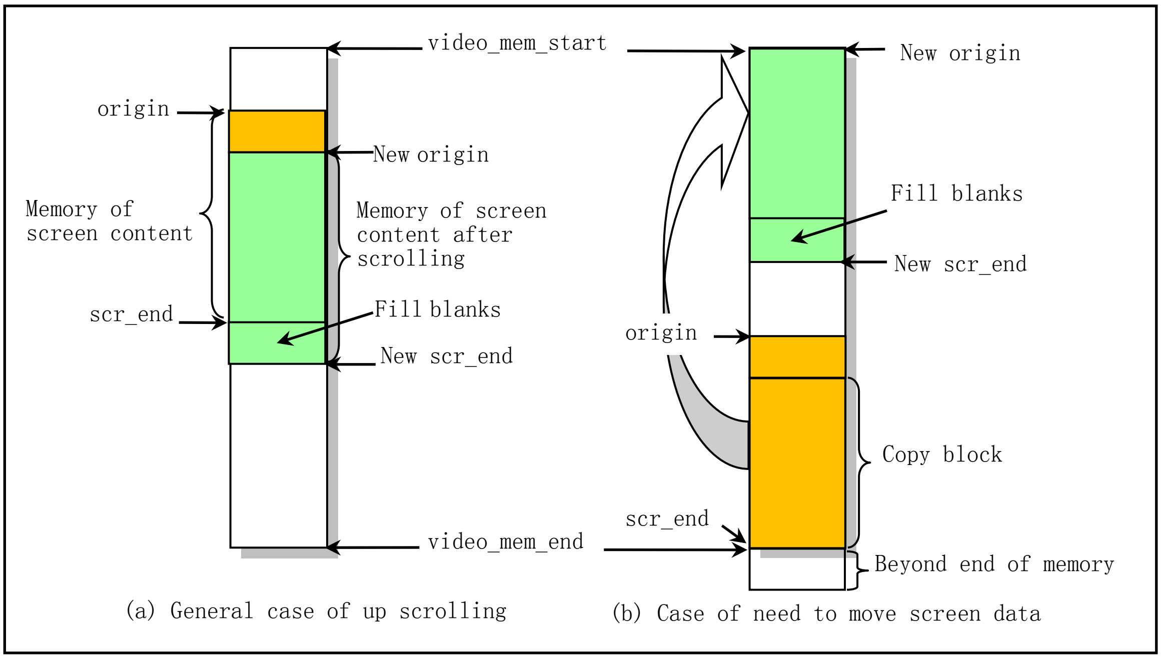


Figure 10-8 Schematic diagram of scroll up operation

Scrolling down the screen is similar to scrolling up and encounters two similar situations. Just because the screen window moves up, a blank line appears at the top of the screen, and when the memory corresponding to the screen content exceeds the display memory range, the screen data memory block needs to be moved down to the end position of the display memory.

10.3.3.3 Terminal Control Commands

Terminal devices usually have two functions, which serve as information input devices (keyboards) and output devices (displays) for the computer. The terminal can have a number of control commands that cause the terminal to perform certain operations rather than just displaying a character on the screen. In this way, the computer can order the terminal to perform operations such as moving the cursor, switching the display mode, and ringing. Terminal control commands can be further divided into two categories: control character commands and ANSI escape control sequences. As we discussed briefly, the console.c (including the keyboard.s above) program in the Linux kernel can actually be seen as a simulated terminal emulator. Therefore, in order to understand the processing of the console.c program, we outline how the program in the ROM in a terminal device handles the code data received from the host. We first briefly describe the structure of the ASCII code table, and then explain how the terminal device handles the received control characters and control sequence string code.

1. Character encoding method

The conventional character terminal uses an 8-bit encoding scheme of ANSI (American National Standards Institute) and ISO (International Standards Organization) standards and a 7-bit code extension technique. ANSI and ISO specify character encoding standards in the computer and communications fields. The ANSI X3.4-1977 and ISO 646-1977 standards define the American Standard Code for Information Interchange, the ASCII code set. The ANSI X3.41-1974 and ISO 2022.2 standards describe code extension techniques for 7-bit and 8-bit code sets. ANSI X3.32, ANSI X3.64-1979 has developed a method for representing terminal control characters using text characters in ASCII code. Although the Linux 0.1x kernel only implements compatibility with VT100 and VT102 terminal devices of Digital Equipment Corporation DEC (now incorporated into Compaq and HP), and these two de facto standard terminal devices only support 7-bit encoding schemes. However, for the sake of completeness and convenience of description, here we also introduce the 8-bit coding scheme.

2. ASCII code table

ASCII code has 7-bit and 8-bit code representation. The 7-bit code table has a total of 128 character codes, as shown in the left half of Table 10-7. Each of the rows represents a value of 4 bits lower in 7 bits, and each column is a high 3 bit value. For example, the binary value of the code 'A' of column 4 and row 1 is 0b0100, 0001 (0x41), and the decimal value is 65.

The characters in the table are divided into two types. One is a control character composed of the first and second columns, and the rest are graphic characters or display characters, text characters. The terminal will process the two types of characters separately. Graphic characters are characters that can be displayed on the screen, while control characters are usually not displayed on the screen. Control characters are used for special control during data communication and text processing. In addition, the DEL character (0x7F) is also a control character, and the space character (0x20) can be either a normal text character or a control character. Control characters and their functions have been standardized by ANSI, and the names are ANSI standard mnemonics. For example: CR (Carriage Return), FF (Form Feed) and CAN (Cancel). Usually the 7-bit encoding method is also applicable to 8-bit encoding. Tables 10-7 are 8-bit code tables (where the left half of the table is identical to the 7-bit code table), with the extension code for the right half not listed.

Table 10-7 8-bit ASCII code table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	`	p			无					
1	SOH	DC1	!	1	A	Q	a	q								
2	STX	DC2	"	2	B	R	b	r								
3	ETX	DC3	#	3	C	S	c	s								
4	EOT	DC4	$	4	D	T	d	t	IND							
5	ENQ	NAK	%	5	E	U	e	u	NEL							
6	ACK	SYN	&	6	F	V	f	v	SSA							
7	BEL	ETB	'	7	G	W	g	w	ESA							
8	BS	CAN	(	8	H	X	h	x	HTS							
9	HT	EM	)	9	I	Y	i	y	HTJ							
A	LF	SUB	*	:	J	Z	j	z	VTS							
B	VT	ESC	+	;	K	[	k	{	PLD	CSI						
C	FF	FS	,	<	L	\	l	|	PLU	ST						
D	CR	GS	-	=	M	]	m	}	RI	OSC						
E	SO	RS	.	>	N	^	n	~	SS2	PM						
F	SI	US	/	?	O	_	o	DEL	SS3	APC						NIL
	C0 codes		GL codes						C1 codes		GR codes					
	7-bit code table								The right half of 8-bit code table							

It has 8 more columns of code than the 7-bit code table, and contains a total of 256 codes. Similar to the 7-bit code table, each row represents the lower 4 bit value of the 8-bit code, and each column represents the high 4-bit value. The left half of the table (column 0 - column 7) is exactly the same as the 7-bit code table, and the 8th bit of their code is 0, so this bit can be ignored. The 8th bit of each code in the right half of the table (column 8 - column 15) is all 1, so these characters can only be used in an 8-bit environment. The 8-bit code table has two control code sets: C0 and C1. There are also two graphic character sets: the left graphic character set GL (Graphic Left) and the right graphic character set GR (Graphic Right).

The functions of the control characters in C0 and C1 cannot be changed, but we can map different display characters to the GL and/or GR areas. Various text character sets that can be used (mapped) are usually stored in the terminal device. We must first do the mapping before using them. For DEC terminal devices that have become the de facto standard, the DEC multi-national character set (ASCII character set and DEC auxiliary character set), the DEC special character set, and the National Replacement Character set (NCR) are usually stored. When the terminal device is turned on, the DEC multi-national character set is used by default.

3. Control functions

In order to direct the terminal device how to process the received data, we need to use the control function of the terminal device. By transmitting a control code or a sequence of control codes, the host can control the display processing of the characters by the terminal device, which are only used to control the display, processing and transmission of text characters, and are not themselves displayed on the screen. Control functions have many uses, such as moving the cursor position on the display, deleting a line of text, changing characters, changing the character set, and setting the terminal operating mode. We can use all the control functions in text mode and use one byte or more bytes to represent the control functions.

It can be considered that all control characters or control character sequences that are not used for display on the screen are control functions. Not all control functions perform their control operations in each ANSI-compliant terminal device, but the device should be able to recognize all control functions and ignore control functions that do not work. So usually a terminal device only implements a subset of the ANSI control functions. Because different devices use different subsets of control functions, compatibility with ANSI standards does not mean that these devices are compatible with each other. Compatibility is only reflected in the fact that various devices use the same control functions.

The single-byte control functions are the control characters in C0 and C1 shown in Table 10-7. Limited control functions are available using the control characters in C0. The control characters in C1 can provide some additional control functions, but they can only be used in an 8-bit environment. Therefore, the VT100 type terminal emulated in the Linux kernel here can only use the control characters in C0. Multi-byte control code can provide a lot of control functions. These multibyte control codes are commonly referred to as Escape Sequences, Control Sequences, and Device Control Strings. Some of these control sequences are common sequences of ANSI standards in the industry, and others are proprietary control sequences designed by manufacturers for their own products. Like the ANSI standard sequence, proprietary control sequence characters also conform to the combined standard of ANSI character codes.

4. Escape Sequences

The host can send an escape sequence to control the display position and attributes of the text characters on the terminal screen. The escape sequence begins with the control character ESC (0x1b) in C0 and is followed by one or more ASCII display characters. The ANSI standard format for escape sequences is as follows:

```
    ESC          I......I            F
    0x1b         0x20--0x2f         0x30--0x7e

    Introducer   Intermediate chars  Final char
                 (0 or multi chars)  (one char)
```

The ESC is an escape sequence introducer defined in the ANSI standard. After receiving the introducer code ESC, the terminal needs to save (rather than display) all subsequent control characters in a certain order.

Intermediate characters are those received after ESC in the range 0x20 -- 0x2f (column 2 in the above ASCII table). Terminals need to save them as part of the control function.

The final character is a character that is received after ESC in the range 0x30 -- 0x7e (column 3 -- 7 in the ASCII table), and the final character indicates the end of a escape sequence. The intermediate and end characters together define the function of a sequence. At this point, the terminal can perform the functions specified by the escape sequence and continue to display the characters that are subsequently received. The final character of the ANSI standard escape sequence ranges from 0x40 to 0x7e (columns 4 - 7 in the ASCII table). The final characters of the proprietary escape sequences defined by each terminal device manufacturer range from 0x30 to 0x3f (column 3 in the ASCII table). For example, here is an escape sequence that specifies G0 as the ASCII character set:

```
    ESC    (     B
    0x1b  0x28  0x42
```

Since the escape sequence uses only 7-bit characters, we can use them in both 7-bit and 8-bit environments. Note that when using escape or control sequences, remember that they define a code sequence rather than a textual representation of the characters. One of the important uses of the escape sequence is to extend the functionality of the 7-bit control character. The ANSI standard allows us to use a 2-byte escape sequence as a 7-bit code extension to represent any control character in C1. This is a very useful feature in applications that require 7-bit compatibility. For example, the control characters CSI and IND in C1 can be represented using a 7-bit code extension form as follows:

```
    C1 char   Escape sequences
    CSI          ESC   [
    0x9b         0x1b  0x5b

    IND          ESC   D
    0x84         0x1b  0x44
```

In general, we can use the above code extension technology in two ways. We can use a 2-character escape sequence to represent any control character in the 8-bit code table C1. The value of the second character is the value of the corresponding character in C1 minus 0x40 (64). Alternatively, we can convert any escape sequence with second character value between 0x40 and 0x5f to produce an 8-bit control character by deleting the control character ESC and adding 0x40 to the second character.

5. Control sequences

Control Sequences start with the control character CSI (0x9b) followed by one or more ASCII graphic characters. The ANSI standard format for the control sequence is as follows:

```
    CSI          P......P           I......I            F
    0x9b         0x30--0x3f         0x20--0x2f          0x40--0x7e

    Introducer   Parameter chars    Intermediate chars  Final char
                 (0 or more chars)  (0 or multi chars)  (one char)
```

The control sequence introducer is the CSI (0x9b) in the 8-bit control character C1. However, since the CSI can also be extended with the 7-bit code extension 'ESC [', all control sequences can be represented using the escaped sequence in which the second character is the left bracket '[' . After receiving the introducer CSI, the terminal needs to save (rather than display) all subsequent control characters in a certain order.

Parameter characters are characters received after CSI in the range 0x30 -- 0x3f (column 3 in the ASCII table), which are used to modify the role or meaning of the control sequence. When the parameter character begins with any '<=>?' (0x3c - 0x3f) character, the terminal will use this control sequence as a proprietary (private) control sequence. The terminal can use two types of parameter characters: numeric characters and select characters. The numeric character parameter represents a decimal number and is represented by Pn. The range is 0 -- 9. The select character parameter comes from a specified parameter list, denoted by Ps. If a control sequence contains more than one parameter, it is separated by a semicolon ';' (0x3b).

Intermediate characters are characters that are received after CSI in the range 0x20 -- 0x2f (column 2 in the ASCII table). Terminals need to save them as part of the control function. Note that the terminal device does not use intermediate characters.

The final character is a character received after CSI in the range 0x40 -- 0x7e (column 4 -- 7 in the ASCII table). The final character indicates the end of the control sequence. The intermediate and finae characters together define the function of a sequence. At this point, the terminal can perform the specified function and continue to display the characters that are subsequently received. The final character of the ANSI standard escape sequence ranges from 0x40 to 0x6f (columns 4 -- 6 in the ASCII table). The final character of the proprietary escape sequences defined by each terminal device manufacturer range from 0x70 to 0x7e (column 7 in the ASCII table). For example, the following sequence defines a control sequence that moves the screen cursor to the specified position (row 5, column 9):

```
    CSI   5    ;    9    H
    0x9b 0x35 0x3b 0x41 0x48
or:
    ESC   [    5    ;    9    H
    0x1b 0x5b 0x35 0x3b 0x39 0x48
```

Figure 10-9 shows an example of a control sequence: cancel the attributes of all characters, then turn on the attributes of underline and reverse: ESC [0;4;7m

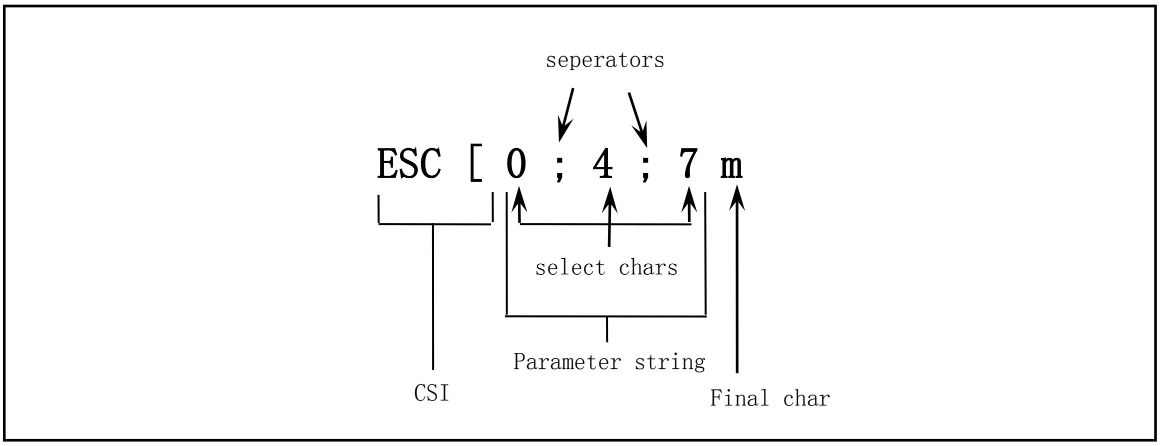


Figure 10-9 Example of control sequence

6. Terminal handling of received characters

Here is a brief description of how the terminal handles the received characters, that is, the response of the terminal to the code sent from the application or host system. The characters received by the terminal can be divided into two categories: graphic (display or text) characters and control characters. Graphic characters are the characters that are received and displayed on the screen. The characters actually displayed on the screen depend on the selected character set. The character set can be selected through control functions.

All data received by the terminal consists of one or more character codes. These data include graphical characters, control characters, escape sequences, control sequences, and device control strings. Most of the data is made up of graphic characters that are only displayed on the screen and has no other effect. Control characters, escape sequences, control sequences, and device control strings are all "control functions," which we can use in our own programs or operating systems to indicate how the terminal processes, transmits, and displays characters. Each control function has a unique name and has a shorthand mnemonic. These names and mnemonics are standard. By default, the terminal's interpretation of a control or display character depends on the ASCII character set used. Note that for unsupported control code, the usual action taken by the terminal is to ignore it. Subsequent characters sent to the terminal that are not described here may have unpredictable consequences.

The appendix of this book gives a description of the commonly used control characters in the C0 and C1 tables, and outlines the actions that will be taken when the terminal receives it. For a particular terminal, it usually does not recognize all control characters in C0 and C1. In addition, the appendix also lists the escape sequences and control sequences used by the console.c program in the Linux 0.1x kernel. All sequences represent the sequence of control functions sent by the host, unless otherwise stated.

10.4 serial.c

10.4.1 Function

The program serial.c implements the system serial port initialization and is ready to use the serial terminal device. The default serial communication parameters are set in the rs_init() initialization function, and the interrupt trap gate (interrupt vector) of the serial port is also set. The rs_write() function is used to send the characters in the serial terminal device write buffer queue to the remote terminal device through the serial line.

The function rs_write() will be called when the character device file is used in the file system. When a program writes to a serial device /dev/tty64 file, the system-call sys_write() is executed (in fs/read_write.c). When the system call determines that the file being read is a character device file, the rw_char() function (in fs/char_dev.c) is called. This function will call rw_tty() using the character device read/write function table (device switch table) according to the information such as the sub-device number of the device being read. This will eventually call the serial terminal write function rs_write() here.

The rs_write() function actually turns on the serial transmit hold register empty interrupt flag, allowing interrupt signals to be sent after the UART sends the data out. The specific send operation is done in the rs_io.s program.

10.4.2 Code annotation

Program 10-3 linux/kernel/chr_drv/serial.c

```
/*
 *  linux/kernel/serial.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      serial.c
 *
 * This module implements the rs232 io functions
 *      void rs_write(struct tty_struct * queue);
 *      void rs_init(void);
 * and all interrupts pertaining to serial IO.
 */

// <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
//     communication.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the
//     data of the initial task 0, and some embedded assembly function macro statements
//     about the descriptor parameter settings and acquisition.
// <asm/system.h> System header file. An embedded assembly macro that defines or
//     modifies descriptors/interrupt gates, etc. is defined.
// <asm/io.h> Io header file. Defines the function that operates on the io port in the
//     form of a macro's embedded assembler.
#include <linux/tty.h>
#include <linux/sched.h>
#include <asm/system.h>
#include <asm/io.h>

// Begin sending when the write queue contains WAKEUP_CHARS characters.
#define WAKEUP_CHARS (TTY_BUF_SIZE/4)

extern void rs1_interrupt(void);       // serial 1 interrupt handler (rs_io.s, 34)
extern void rs2_interrupt(void);       // serial 2 interrupt handler (rs_io.s, 38)

//// Initialize the serial port
// Sets the transmit baud rate (2400 bps) for the specified serial port and allows all interrupt
// sources except the write hold register empty. In addition, when outputting a 2-byte baud
// rate factor, the DLAB bit (bit 7) of the line control register must first be set.
// Parameters: port is the serial port base address, port 1 - 0x3F8; port 2 - 0x2F8.
static void init(int port)
{
        outb_p(0x80,port+3);    /* set DLAB of line control reg */
        outb_p(0x30,port);      /* LS of divisor (48 -> 2400 bps */
        outb_p(0x00,port+1);    /* MS of divisor */
        outb_p(0x03,port+3);    /* reset DLAB */
        outb_p(0x0b,port+4);    /* set DTR,RTS, OUT_2 */
        outb_p(0x0d,port+1);    /* enable all intrs but writes */
        (void)inb(port);        /* read data port to reset things (?) */
}

//// Initialize the serial interrupt routine and the serial interface.
// The gate descriptor setting macro set_intr_gate() in the interrupt descriptor table IDT is
// implemented in include/asm/system.h.
void rs_init(void)
{
```

```
// The following two sentences are used to set the interrupt gate descriptors of the two serial
// ports. rs1_interrupt is the interrupt handler pointer for serial port 1. The interrupt used
// by serial port 1 is int 0x24, and the port 2 is int 0x23. See Table 5-2 and system.h file.
        set_intr_gate(0x24,rs1_interrupt);    // set int gate descriptor of port 1 (IRQ4).
        set_intr_gate(0x23,rs2_interrupt);    // set int gate descriptor of port 2 (IRQ3).
        init(tty_table[64].read_q->data);     // Initialize port 1 (.data is base address).
        init(tty_table[65].read_q->data);     // Initialize port 2
        outb(inb_p(0x21)&0xE7,0x21);          // enable 8259A to respond to IRQ3, IRQ4.
}

/*
 * This routine gets called when tty_write has put something into
 * the write_queue. It must check wheter the queue is empty, and
 * set the interrupt register accordingly
 *
 *      void _rs_write(struct tty_struct * tty);
 */
//// Serial data send write function.
// This function actually only turns on the transmit hold register empty interrupt flag.
// Thereafter, when the transmit holding register is empty, the UART generates an interrupt
// request. In the serial interrupt handler, the program fetches the character at the end of
// the write queue and outputs it to the transmit holding register. Once the UART sends the
// character out, the transmit holding register will be empty and this causes an interrupt request
// again. So as long as there are characters in the write queue, the system repeats the process
// and sends the characters one by one. When all the characters in the write queue are sent
// out, the write queue becomes empty, and the interrupt handler resets the transmit hold register
// interrupt enable flag in the interrupt enable register, thereby again disabling the transmit
// hold register empty to cause interrupt request. At this point, the "loop" send operation
// ends.
void rs_write(struct tty_struct * tty)
{
// If the write queue is not empty, first read the interrupt enable register contents from 0x3f9
// (or 0x2f9), add the transmit hold register interrupt enable flag (bit 1), and then write
// back to the register. Thus, the UART can initiate an interrupt when it is desired to obtain
// the character to be transmitted when the transmit holding register is empty. Just note that
// the write_q.data here is the serial port base address.
        cli();
        if (!EMPTY(tty->write_q))
                outb(inb_p(tty->write_q->data+1)|0x02,tty->write_q->data+1);
        sti();
}

```

10.4.3 Information

10.4.3.1 Universal Asynchronous Serial Communication

The universal asynchronous serial communication transmission method is the most common traditional serial communication method in the industry, and is still widely used in various embedded systems. It uses a dedicated Universal Asynchronous Receive and Transmit (UART) controller chip to implement data transfer. The format of the communication frame used is shown in Figure 10-10. Transferring a character consists of a start bit, data bits, a parity bit, and 1 or 2 stop bits. The start bit plays a synchronous role and the value is always zero. The data bits are the actual data transmitted, that is, the code of one character. Its length can be 5-8 bits. The parity bit is optional and can be set by the program. The stop bit is always 1, and can be set to 1, 1.5 or 2 bits by the program. Before the communication starts to send information, both parties must be set to the same format and use the same transmission rate. For example, it is necessary to have the same number of data bits and stop bits. 在 In the asynchronous communication specification, the transmission 1 is called a MARK and the transmission 0 is called a SPACE. Therefore we also use these two terms in the following description.

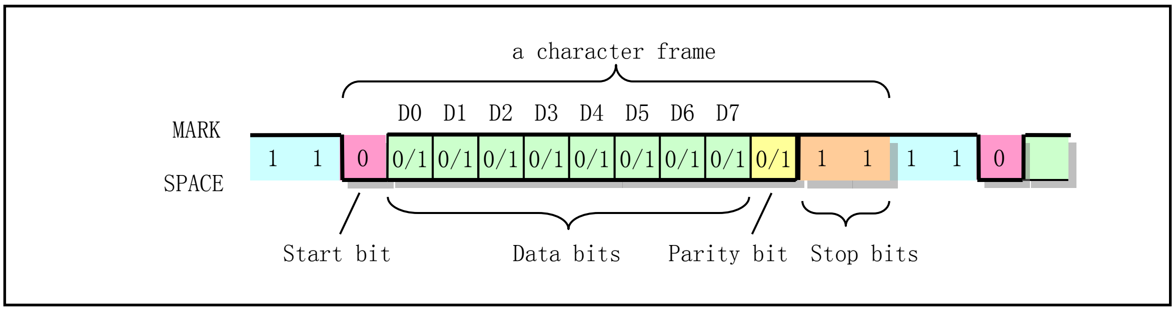


Figure 10-10 Asynchronous serial transmission frame format

When there is no data transmission, the sender is in the MARK state and continues to transmit 1. If data needs to be sent, the sender needs to first send a start bit SPACE of the bit interval. After receiving the SPACE bit, the receiver starts to synchronize with the sender and then receives the subsequent data. If the parity bit is set in the program, the parity bit needs to be received after the data is transmitted. Finally, the stop bit. After the character frame is sent, the next character frame can be sent immediately, or the MARK can be sent temporarily, and then the character frame is sent again.

When receiving a character frame, the receiver may detect one of three types of errors: (1) Parity error. At this point the program should ask the other party to resend the character; (2) Overspeed error. This error occurs because the program fetches characters slower than the receiving speed. At this point, you should modify the program to speed up the character frequency; (3) The frame format is incorrect. This error can occur when the format information requested to be received is incorrect. For example, a SPACE bit was received when a stop bit should be received. In general, in addition to line interference, it is likely that the frame format setting of the communication parties is different.

1. Serial communication interface and UART structure

In order to realize serial communication, the PC usually has two serial interfaces conforming to the RS-232C standard, and uses a universal asynchronous receiver/transmitter (UART) control chip to process the serial data transmission and reception. The serial interface on the PC usually uses a 25-pin DB-25 or a 9-pin DB-9 connector, which is mainly used to connect the MODEM device for operation. Therefore, the RS-232C standard specifies many MODEM-specific interface pins. Please refer to other materials for a detailed description of the RS-232C standard and the working principle of the MODEM device. Here we mainly explain the structure of the UART control chip and prepare it for programming.

Previous PCs used National Semiconductor's NS8250 or NS16450 UART chips. Today's PCs use the 16650A or its compatible chips, but they are all compatible with the NS8250/16450 chip. The main difference between these chips is that the 16650A chip additionally supports FIFO transmission. In this way, the UART can trigger an interrupt after receiving or transmitting up to 16 characters, thus reducing the burden on the system and CPU. However, since the Linux 0.12 we discussed only uses the attributes of NS8250/16450, the FIFO mode is not further explained here.

The UART asynchronous serial port hardware logic used in the PC is shown in Figure 10-11. It can be divided into 3 parts. The first part mainly includes data bus buffer D7 -- D0, internal register select pin A0 -- A2, CPU read / write data select pin DISTR and DOSTR, chip reset pin MR, interrupt request output pin INTRPT and user Pin OUT2 defined to disable/allow interrupts. When OUT2 is 1, the UART can be disabled from issuing an interrupt request signal.

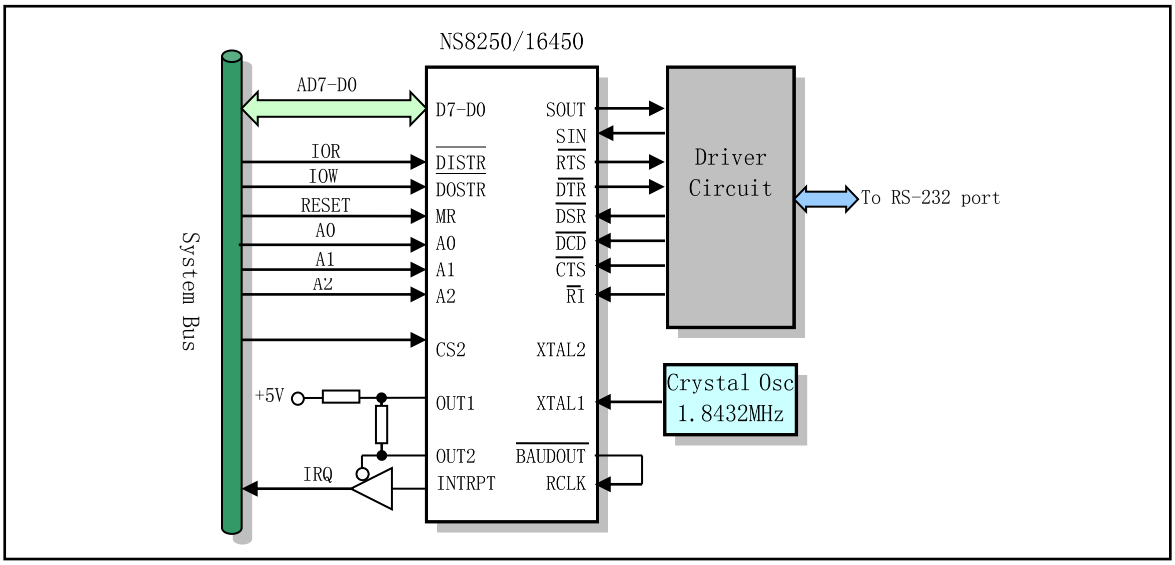


Figure 10-11 NS8250/16450 basic hardware structure diagram

The second part mainly includes the pin part between the UART and the RS-232 interface. These pins are primarily used to receive/transmit serial data and to generate or receive MODEM control signals. The serial output data (SOUT) pin sends a bit stream to the line; the input data (SIN) pin receives the bit stream from the line; the Data Device Ready (DSR) pin is used by the communication device (MODEM) to inform the UART that it has Ready to start receiving data; The request to send (RTS) pin is used to notify the MODEM, the computer requires switching to the sending mode; the clear sending (CTS) is the MODEM telling the computer that it has switched to the ready to receive mode; The Device Carrier Detect (DCD) pin is used to receive MODEM information, which tells the UART that the carrier signal has been received; the Ring Indicator (RI) pin is also used by the MODEM to tell the computer that the communication line is already connected.

The third part is the UART chip clock input circuit part. The UART's operating clock can be generated by connecting a crystal oscillator between pins XTAL1 and XTAL2, or directly from the outside via XTAL1. The PC uses the latter method to directly input the 1.8432MHz clock signal on the XTAL1 pin. The 16 times signal of the UART transmit baud rate is output by the pin BAUDOUT, and the pin RCLK is the baud rate of the received data. Since the two are connected together, the baud rate for sending and receiving data on the PC is the same.

Like the interrupt controller chip 8259A, the UART is also a programmable control chip. By setting its internal registers, we can set the operating parameters of the serial communication and how the UART works. The internal block diagram of the UART is shown in Figure 10-12.

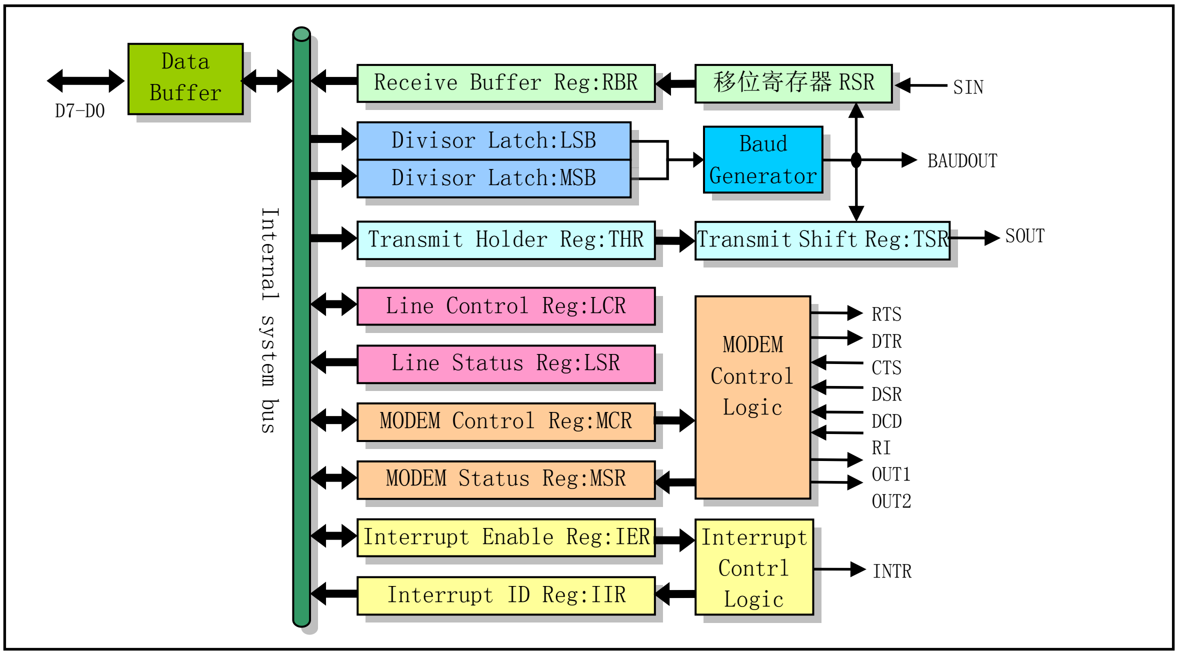


Figure 10-12 Basic block diagram of NS8250 UART internal components

For the NS8250 chip, there are 10 registers that the CPU can access, but the address line A2--A0 used to select these registers can only select up to 8 registers. Therefore, the NS8250 takes a bit (bit 7) in the line control register (LCR) to select the two divisor latch registers LSB and MSB. Bit 7 is referred to as the Divisor Latch Access Bit (DLAB). The purpose of these registers and the access port address are shown in Table 10-8.

Table 10-8 UART internal register ports and purposes

Port address	R/W	condition	Usage
0x3f8 (0x2f8)	W	DLAB=0	Writes transmit holding register, THR, contains char that will be sent.
	R	DLAB=0	Read the receive buffer register RBR. Contains character received.
	R/W	DLAB=1	Read/write baud rate factor low byte (LSB).
0x3f9 (0x2f9)	R/W	DLAB=1	Read/write baud rate factor high byte (MSB).
	R/W	DLAB=0	Read/write interrupt enable register IER. Bits 7-4 all 0, reserved; Bit 3=1 modem status interrupt allowed; Bit 2=1 Receiver line status interrupt enabled; Bit 1=1 Transmit Holding Register Empty Interrupt allowed; Bit 0=1 has received data interrupt enable.
0x3fa (0x2fa)	R		Read interrupt identification register IIR. The interrupt handler is used to determine which of the four types is interrupted. Bits 7-3 all 0 (not used); Bit 2-1 determines the priority of the interrupt; = 11 The receiving status has error and the priority is highest. Read the line status to reset it; = 10 Data has been received, priority 2. Read data can reset it; = 01 Transmit holding register empty, priority 3. Write THR to reset it; = 00 MODEM status changes, priority 4. Read MODEM state to reset it.

			Bit 0 = 0 pending interrupt; =1 no interrupt.
0x3fb (0x2fb)	W		Write the line control register LCR. Bit 7 = 1 Divisor Latch Access Bit (DLAB). = 0 Receiver, transmit hold or interrupt enable register access; Bit 6=1 allows for a break; Bit 5=1 holds the parity bit; Bit 4=1 even parity; =0 odd parity; Bit 3=1 allows parity; =0 no parity; Bit 2=1 now depends on the data bit length. If the data bit length is 5 bits, the stop bit is 1.5 bits; if the data bit length is 6, 7 or 8 bits, the stop bit is 2 bits; = 0 stop bit is 1 bit; Bit 1-0 Data Bit Length: = 00 5 data bits; = 01 6-bit data bits; = 10 7-bit data bits; = 11 8-bit data bits.
0x3fc (0x2fc)	W		Write MODEM control register MCR. Bits 7-5 all 0, reserved; Bit 4=1 chip is in cyclic feedback diagnostic mode of operation; Bit 3=1, auxiliary user specifies output 2, enable INTRIPT to system; Bit 2=1, auxiliary user specifies output 1, and the PC is not used; Bit 1=1 makes the request to send RTS valid; Bit 0 = 1 makes the data terminal ready DTR active.
0x3fd (0x2fd)	R		Read the line status register LSR. Bit 7=0 reserved; Bit 6=1 The transmit shift register is empty; Bit 5=1 Transmit holding register is empty and can get character to send; Bit 4=1 receives a sequence of bits that satisfy the break condition; Bit 3=1 frame format error; Bit 2=1 parity error; Bit 1=1 exceeds the overlay error; Bit 0=1 Receiver data is ready and the system is readable.

0x3fe (0x2fe)	R		Read the MODEM status register MSR. δ indicates a change in the signal or condition.
			Bit 7=1 carrier detect (CD) is valid;
			Bit 6=1 ring indication (RI) is valid;
			Bit 5=1 Data Device Ready (DSR) is valid;
			Bit 4=1 Clear Transmit (CTS) is valid;
			Bit 3=1 detects the δ carrier;
			Bit 2=1 detects the edge of the ring signal;
			Bit 1 = 1 δ Data Device Ready (DSR);
			Bit 0 = 1 δ Clear Transmit (CTS).

2. UART initialization programming method

When the PC is powered on, the system RESET signal resets the UART internal registers and control logic through the MR pin of the NS8250 chip. After that, if you want to use the UART, you need to initialize it to set the working baud rate, number of data bits and working mode of the UART. Below we take the serial port 1 on the PC as an example to illustrate the steps to initialize it. The port base address of the serial port is port = 0x3f8, and the UART chip interrupt pin INTRPT is connected to the interrupt control chip pin IRQ4. Of course, the interrupt descriptor entry of the serial interrupt handler should be set first in the IDT table before initialization.

a) Set the transmission baud rate.

Setting the communication transmission baud rate is to set the values of the two divisor latch registers LSB and MSB, that is, the 16-bit baud rate factor. As can be seen from Table 10-8, to access the two divisor latch registers, we must first set bit 7 DLAB=1 of the line control register LCR, that is, write 0x80 to port+3 (0x3fb). Then we perform output operations on port (0x3f8) and port+1 (0x3f9), and we can write the baud rate factor into LSB and MSB respectively. For a specified baud rate (eg 2400 bps), the baud rate factor is calculated as:

$$\text{Baudrate factor} = \frac{\text{UART Freq.}}{\text{Baudrate} \times 16} = \frac{1.8432\,MHz}{2400 \times 16} = \frac{1843200}{2400 \times 16} = 48$$

So to set the baud rate to 2400pbs, we need to write 48 (0x30) in the LSB and 0 in the MSB. After the baud rate is set, we also need to reset the DLAB bit of the line control register.

b) Set the transfer format.

The serial communication transmission format is defined by the bits in the line control register LCR. The meaning of each of them is shown in Table 10-8. If we need to set the transmission format to no parity, 8 data bits and 1 stop bit, then we need to output the value 0x03 to the LCR. The lowest 2 bits of the LCR indicate the data bit length, and when it is 0b11, the data length is 8 bits.

c) Set the MODEM control register.

Writing to this register sets the mode of operation of the UART and controls the MODEM. There are two modes of UART operation: interrupt mode and query mode. There is also a loop feedback method, but this method is only used to diagnose and test the quality of the UART chip, and cannot be used as an actual communication method. The query method is used in the PC ROM BIOS, but the Linux system discussed in this book uses an efficient interrupt method. Therefore, we will only introduce the operation programming method of the UART in the interrupt mode.

Setting bit 4 of the MCR allows the UART to be in the loop feedback diagnostic mode of operation. In this mode, the UART chip automatically "snap" the input (SIN) and output (SOUT) pins. Therefore, if the data sequence sent at this time is equal to the received sequence, then the UART chip is working properly.

The interrupt mode means that the UART is allowed to send an interrupt request signal to the CPU through the INTRPT pin when the MODEM status changes, or when an error occurs, or when the transmit holding register is empty, or when a character is received. As for allowing interrupt requests to be issued under those conditions, it is determined by the interrupt enable register IER. However, if the UART interrupt request signal can be sent to the 8259A interrupt controller, bit 3 (OUT2) of the MODEM control register MCR needs to be set. Because in the PC, this bit controls the INTRPT pin to the 8259A circuit, see Figure 10-11.

The query mode means that the program receives/transmits the serial data by cyclically polling the contents of the UART register under the condition that the MODEM control register MCR bit 3 (OUT2) is reset. When the MCR bit 3 = 0, the UART can still generate an interrupt request signal on the INTRPT pin under the condition that the MODEM state changes, and the interrupt flag register IIR can be set according to the condition that the interrupt is generated, but the interrupt request signal cannot be send to 8259A. Therefore, the program can only judge the current working state of the UART by querying the contents of the line status register LSR and the interrupt identification register IIR, and perform data receiving and transmitting operations.

Bits 1 and 0 of MCR are used to control the MODEM. When these two bits are set, the data terminal ready DTR pin of the UART and the request to send RTS pin output are valid. To set the UART to interrupt mode and make DTR and RTS valid, then we need to write 0x0b, the binary number 01011, to the MODEM control register.

d) Initialize the interrupt enable register IER

The interrupt enable register IER is used to set the condition that can generate an interrupt, that is, the interrupt source type. There are four types of interrupt sources to choose from, as shown in Table 10-8. If the corresponding bit is 1, it means that the condition is allowed to generate an interrupt, otherwise it is disabled. When an interrupt source type generates an interrupt, the interrupt generated by which interrupt source is specified by bit 2 - bit 1 in the interrupt flag register IIR, and the contents of the specific register can be read and written to reset the UART interrupt. Bit 0 of the IER is used to determine if there is an interrupt currently, and bit 0 = 0 indicates that there is an interrupt to be processed.

In the serial port initialization function of Linux 0.12, the setting allows three types of interrupt sources to generate interrupts (write 0x0d), that is, when the MODEM status changes, when the receiving error occurs, the interrupt is allowed to be generated when the receiver receives the character. However, it is not allowed the transmit hold register empty to generate an interrupt because we have no data to send at this time. When the write queue of the corresponding serial terminal has data to be sent out, the tty_write() function calls the rs_write() function to set the transmit hold register empty enable interrupt flag, thereby during the serial interrupt processing initiated by the interrupt source, the kernel program can start to fetch the character in the write queue and send the output to the transmit holding register for the UART to send out. Once the UART sends the character out, the transmit holding register will empty and cause an interrupt request again. So as long as there are characters in the write queue, the system repeats the process and sends the characters one by one. When all the characters in the write queue are sent out, the write queue becomes empty, and the interrupt handler resets the transmit hold register interrupt enable flag in the interrupt enable register, thereby again disabling the transmit hold register empty to cause an interrupt request. This "loop" send operation also ends.

3. UART interrupt handler programming method

In the Linux kernel, serial terminals use read/write queues to receive and transmit terminal data. The data received from the serial port is placed in the read queue header for the tty_io.c program to read; and the data that needs to be sent to the serial terminal is placed at the write queue pointer. Therefore, the main task of the serial interrupt handler is to put the character in the receive buffer register RBR received by the UART to the pointer at the end of the read queue; the character taken from the pointer at the end of the write queue is placed in the transmit hold register THR of the UART and sent out. At the same time, the serial interrupt handler also needs to handle some other error conditions.

As can be seen from the above description, the UART can generate interrupts with four different interrupt source types. Therefore, when the serial interrupt handler first starts executing, it is only known that an interrupt has occurred, but it is not known which case caused the interrupt. So the first task of the serial interrupt handler is to determine the specific conditions under which the interrupt will be generated. This requires the use of the interrupt flag register IIR to determine the source type from which the current interrupt was generated. Therefore, the serial interrupt handler can be processed separately according to the source type that generates the interrupt using the subroutine address jump table jmp_table[]. The block diagram is shown in Figure 10-13. The structure of the rs_io.s program is basically the same as this block diagram.

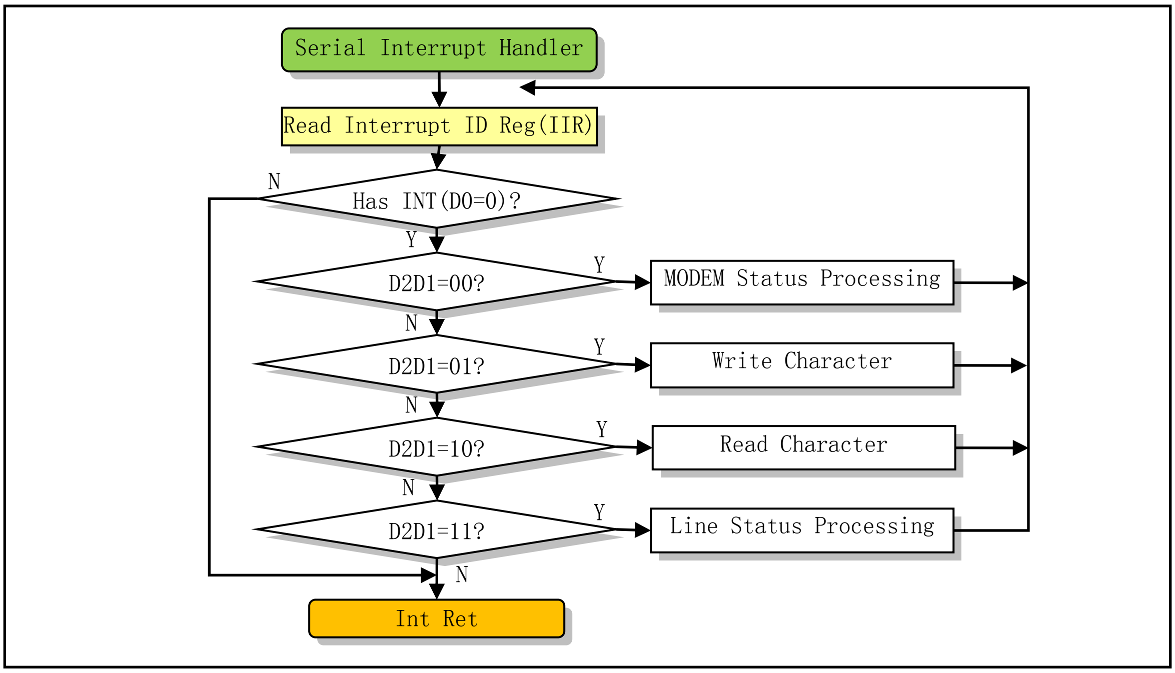


Figure 10-13 Serial communication interrupt processing block diagram

After the content of the IIR is taken out, it is necessary to first judge whether there is an interrupt to be processed according to bit 0. If bit 0 = 0, there is an interrupt that needs to be processed. Then, according to bit 2 and bit 1, the corresponding interrupt source type processing subroutine is called by using the pointer jump table. In each subroutine, the corresponding interrupt source of the UART is reset after processing. After the subroutine returns, this code loops to determine if there are other interrupt sources (bit 0 = 0?). If there are other interrupt sources for this interrupt, bit 0 of IIR is still 0, so the interrupt handler will call the corresponding interrupt source subroutine to continue processing. All interrupt sources that caused this interrupt are processed and reset. At this point, the UART will automatically set the IIR bit 0 = 1, indicating that there is no pending interrupt, so the interrupt handler can exit.

10.5 rs_io.s

10.5.1 Function

The rs_io.s assembly file implements the rs232 serial communication interrupt processing. During the transmission and storage of characters, the interrupt process mainly operates on the read and write buffer queues of the terminal. It stores the characters received from the serial line into the read buffer queue read_q of the serial terminal, or sends out the characters in the write buffer queue write_q to the remote serial terminal device through the serial line.

There are four types of situations that cause the system to generate a serial interrupt: a. because the

MODEM status has changed; b. because the line status has changed; c. because the character was received; d. because the transmit and hold register empty interrupt enable flag is set, there are characters to send. The first two cases of causing an interrupt are reset by reading the corresponding status register value. For the case of receiving a character, the program first puts the character into the read buffer queue read_q, and then calls the copy_to_cooked() function to convert it into the canonical mode character in the character line unit into the auxiliary queue secondary. For the case where a character needs to be sent, the program first takes out a character from the write buffer queue write_q and sends it out, and then judges whether the write buffer queue is empty, and if there are still characters, it performs a transmission operation cyclically.

Therefore, before reading this program, it is best to look at the include/linux/tty.h header file. It gives the data structure tty_queue of the character buffer queue, the data structure tty_struct of the terminal and the values of some control characters. There are also macro definitions that operate on the buffer queue. The buffer queue and its operation diagram are shown in Figure 10-14.

10.5.2 Code annotation

Program 10-4 linux/kernel/chr_drv/rs_io.s

```
/*
 *  linux/kernel/rs_io.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      rs_io.s
 *
 * This module implements the rs232 io interrupts.
 */

.text
.globl _rs1_interrupt,_rs2_interrupt

// size is the length in bytes of the read and write queue buff
size    = 1024                          /* must be power of two !
                                           and must match the value
                                           in tty_io.c!!! */

/* these are the offsets into the read/write buffer structures */
// Corresponds to the offset of each field in the tty_queue structure in the include/linux/tty.h
// file, where rs_addr corresponds to the data field of the tty_queue structure. But for serial
// terminal buffer queues, this field holds the serial port base address (0x3f8 or 0x2f8).
rs_addr = 0                     // field offset of the serial ports (0x3f8 or 0x2f8).
head = 4                        // the head pointer field offset in the buffer.
tail = 8                        // the tail pointer field offset in the buffer.
proc_list = 12                  // wait for the buffered process field offset.
buf = 16                        // buffer field offset.

// When a write buffer queue is full, the kernel puts the process (which fills the queue) into
// a wait state. When there are no more than 256 characters remaining in the write buffer queue,
// the interrupt handler can wake up the waiting process and continue to put characters into
// the write queue.
```

```
startup = 256                   /* chars left in write queue when we restart it */

/*
 * These are the actual interrupt routines. They look where
 * the interrupt is coming from, and take appropriate action.
 */
//// Serial port 1 interrupt handler entry point.
// At initialization, the rs1_interrupt address is placed in the interrupt descriptor 0x24,
// which corresponds to the 8259A interrupt request IRQ4 pin. First, the address of the serial
// terminal 1 (serial port 1) read/write buffer queue pointer in the tty table is first pushed
// onto the stack (tty_io.c, 81), and then jumped to rs_int to continue processing. This will
// allow the processing codes of serial port 1 and serial port 2 to be shared. The character
// buffer queue structure tty_queue can be found in include/linux/tty.h, line 22.
.align 2
_rs1_interrupt:
        pushl $_table_list+8    // serial port 1 r/w queues pointer is pushed onto stack.
        jmp rs_int
.align 2
//// Serial port 2 interrupt handler entry point.
_rs2_interrupt:
        pushl $_table_list+16   // serial port 2 r/w queues pointer is pushed onto stack.

// This code first makes the segment registers ds, es point to the kernel data segment, and
// then takes the serial port base address from the data field of the corresponding read/write
// buffer queue. This address plus 2 is the port address of the interrupt identification register
// IIR. If its bit 0 = 0, it indicates that there is an interrupt to be processed. Then, according
// to bit 2 and bit 1, the corresponding interrupt source type processing subroutine is called
// by using the pointer jump table. Each subroutine will resets the corresponding interrupt
// source of the UART after processing.
// After the subroutine returns, this code loops to determine if there are other interrupt sources
// (bit 0 = 0?). If there are other interrupt sources for this interrupt, bit 0 of the IIR is
// still 0. The interrupt handler then calls the corresponding interrupt source subroutine again
// to continue processing. Until all interrupt sources causing this interrupt are processed
// and reset, the UART will automatically set the IIR bit 0 =1, indicating that there is no
// pending interrupt, so the interrupt handler can exit.
rs_int:
        pushl %edx
        pushl %ecx
        pushl %ebx
        pushl %eax
        push %es
        push %ds                /* as this is an interrupt, we cannot */
        pushl $0x10             /* know that bs is ok. Load it */
        pop %ds                 // Let ds, es point to the kernel data segment.
        pushl $0x10
        pop %es
        movl 24(%esp),%edx      // get serial buffer queue address.
        movl (%edx),%edx        // get read queue struct address -> edx.
        movl rs_addr(%edx),%edx // get serial port 1 (or port 2) base address -> edx.
        addl $2,%edx            /* interrupt ident. reg */
                                // The IIR port address is 0x3fa (0x2fa).
// Get the content of IIR (interrupt identification byte) to determine the source of the
// interrupt. There are 4 types of interrupt conditions determined by bit 2 and bit1: MODEM
```

```
   // status changes; characters to be written (sent); characters to be read (received); line status
   // changes. The code first determine if there is any interrupt to be processed, then processes
   // the corresponding interrupt accordingly.
55 rep_int:
56         xorl %eax,%eax
57         inb %dx,%al              // get interrupt identification byte.
58         testb $1,%al             // bit0 = 0 ? (any interrupts ?)
59         jne end                  // no, jump to end.
60         cmpb $6,%al              /* this shouldn't happen, but ... */
61         ja end                   // if al > 6, jump to end.
62         movl 24(%esp),%ecx       // get serial buffer queue address -> ecx.
63         pushl %edx               // save the IIR port address temperoly.
64         subl $2,%edx             // edx restore to port base address 0x3f8 (0x2f8).
65         call jmp_table(,%eax,2)  /* NOTE! not *4, bit0 is 0 already */
   // The above statement means that when there is an interrupt to be processed, bit 0=0 in the
   // AL, bit 2, bit 1 is the interrupt type, so it is equivalent to multiplying the interrupt
   // type by 2. And because the address has 4 bytes, here multiply by 2, you can get the jump
   // table (line 79) corresponding to each interrupt type address, and then jump there to do the
   // corresponding processing.
   // Allowing the transmission of character interrupts is accomplished by setting the transmit
   // holding register flag. In the serial.c program, when there is data in the write queue, the
   // rs_write() function modifies the contents of the interrupt enable register and adds the
   // transmit hold register interrupt enable flag, causing a serial interrupt to occur when the
   // system needs to send a character.
66         popl %edx                // restore IIR port address 0x3fa (or 0x2fa).
67         jmp rep_int              // jump to loop processing any pending interrupts.

68 end:    movb $0x20,%al           // send EOI instruction to interrupt controller.
69         outb %al,$0x20           /* EOI */
70         pop %ds
71         pop %es
72         popl %eax
73         popl %ebx
74         popl %ecx
75         popl %edx
76         addl $4,%esp             # jump over _table_list entry   # discard the queue address.
77         iret
78
   // The address jump table of each interrupt type processing subroutine. There are 4 types of
   // interrupt conditions: MODEM status changes; characters to be written (sent); characters to
   // be read (received); line status changes.
79 jmp_table:
80         .long modem_status,write_char,read_char,line_status
81
   // This interrupt is caused by a change in the MODEM state. A reset operation is performed on
   // the MODEM status register MSR by reading it.
82 .align 2
83 modem_status:
84         addl $6,%edx             /* clear intr by reading modem status reg */
85         inb %dx,%al              // MODEM status register port: 0x3fe
86         ret
87
   // This serial interrupt is caused by a change in the line state. A reset operation is performed
   // on the line status register LSR by reading it.
   .align 2
   line_status:
           addl $5,%edx             /* clear intr by reading line status reg. */
           inb %dx,%al              // LSR port: 0x3fd
           ret

   // A processing subroutine for interrupts caused by the UART receiving a character. A read from
   // the receive buffer register can reset this interrupt source. This subroutine puts the received
   // character into the head of the read buffer queue read_q and moves the pointer forward by
   // one character position. If the head pointer has reached the end of the buffer, let it fold
   // back to the beginning of the buffer. Finally, the C function do_tty_interrupt() (that is,
   // copy_to_cooked()) is called, and the read characters are processed into a canonical mode
   // buffer queue (secondary buffer).
   // To obtatin the current serial port no, let the current serial port queue address minus the
   // queue talbe address table_list, and then divided by 8. the result 1 is for serial 1, 2 for
   // serial 2.
   .align 2
   read_char:
           inb %dx,%al              // read char in receive buffer register RBR into AL.
           movl %ecx,%edx           // serial buffer queue address -> edx
           subl $_table_list,%edx   // current serial no = (queue address - table_list) / 8
           shrl $3,%edx
           movl (%ecx),%ecx          # read-queue
           movl head(%ecx),%ebx     // get the head pointer in the read queue -> ebx.
           movb %al,buf(%ecx,%ebx)  // put the char at the head position.
           incl %ebx                // and move the head pointer forward by one.
           andl $size-1,%ebx        // modulate the head pointer by the buffer length.
           cmpl tail(%ecx),%ebx     // compare the head with the tail pointer.
           je 1f                    // queue is full if equal, then jump forward.
           movl %ebx,head(%ecx)     // otherwise save the new head pointer.
   1:      addl $63,%edx            // convert serial no to tty no (63 or 64) & push into stack.
           pushl %edx
           call _do_tty_interrupt   // Call tty int handler C function (tty_io.c, line 397).
           addl $4,%esp             // discard paras and ret.
           ret

   // The subroutine of sending characters. This interrupt is caused by setting the transmit holding
   // register enable interrupt flag. Indicates that there are characters in the write queue
   // corresponding to the serial terminal that need to be sent. Then the number of characters
   // currently contained in the write queue is calculated, and if it is less than 256, the process
   // waiting for the write operation is woken up. Then take a character from the end of the write
   // buffer queue and adjust and save the tail pointer. If the write buffer queue is empty, jump
   // to label write_buffer_empty to handle the case.
   .align 2
   write_char:
           movl 4(%ecx),%ecx             # write-queue  // address -> ecx
           movl head(%ecx),%ebx         // get head pointer of write queue to ebx
           subl tail(%ecx),%ebx         // number of chars = head - tail.
           andl $size-1,%ebx             # nr chars in queue
           je write_buffer_empty        // if head == tail, the queue is empty, then jump.
           cmpl $startup,%ebx           // check the number of chars in write queue.
           ja 1f                        // jump if more than 256 chars.
           movl proc_list(%ecx),%ebx     # wake up sleeping process  // get wait proc list.
           testl %ebx,%ebx               # is there any?    # any process waiting to write ?
           je 1f                        // none, then jump forward to label 1.
           movl $0,(%ebx)               // otherwise wake process (change state to runnable).
   1:      movl tail(%ecx),%ebx         // get a char from the tail position to AL.
           movb buf(%ecx,%ebx),%al
           outb %al,%dx                 // output to THR register (port 0x3f8 or 0x2f8).
           incl %ebx                    // move the tail forward by one.
           andl $size-1,%ebx            // modulate and save the tail pointer.
           movl %ebx,tail(%ecx)
           cmpl head(%ecx),%ebx         // tail == head ?
           je write_buffer_empty        // if true, it means queue is empty, then jump.
           ret
   // The following code handles the case where the write buffer queue write_q is empty. If there
   // is a process waiting to write to the serial terminal, it wakes up, and then masks the transmit
   // holding register empty interrupt, and disable the UART to initiate transmit holding register
   // empty interrupt. If the write buffer queue write_q is empty at this time, it means that no
   // characters need to be sent at present. So we should do the following two things. First look
   // at whether there is a process waiting for the write queue to be vacant, and if so, wakes
   // it up. In addition, because the system has no characters to send now, we need to temporarily
   // disable the interrupts generated by transmission hold register THR empty. When a character
   // is placed in the write queue again, the rs_write() function in serial.c will again allow
   // the interrupt to be generated when the transmit holding register is empty, so the UART will
   // "automatically" fetch the characters in the write queue, and send it out.
   .align 2
   write_buffer_empty:
           movl proc_list(%ecx),%ebx     # wake up sleeping process
           testl %ebx,%ebx               # is there any?
           je 1f                        // no process waiting, jump forward to label 1.
           movl $0,(%ebx)               // otherwise wake up the process.
   1:      incl %edx                    // read int enable register IER (0x3f9 or 0x2f9).
           inb %dx,%al
           jmp 1f                       // delay for a while.
   1:      jmp 1f                       // masks transmit hold register empty int (bit 1).
   1:      andb $0xd,%al                /* disable transmit interrupt */
           outb %al,%dx
           ret
```

10.6 tty_io.c

10.6.1 Function

Each tty device has three buffer queues, read queue (read_q), write queue (write_q), and secondary queue (secondary), which are defined in the tty_struct structure (include/linux/tty.h). For each buffer queue, the read operation takes the character from the left end of the buffer queue and moves the buffer tail pointer to the right, while the write operation adds characters to the right end of the buffer queue and also moves the head pointer to the right. If either of these two pointers moves beyond the end of the buffer queue, it will revert to the left and start again, as shown in Figure 10-14.

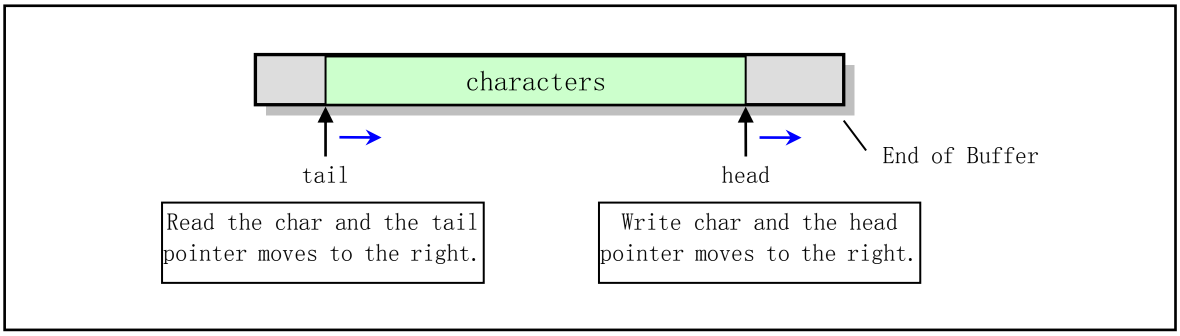

characters
tail
head
End of Buffer
Read the char and the tail pointer moves to the right.
Write char and the head pointer moves to the right.

Figure 10-14 tty character buffer queue operation

The program tty_io.c includes the upper interface functions of the character device. It mainly contains the terminal read/write functions tty_read() and tty_write(), and the canonical mode function copy_to_cooked() of the read operation is also implemented here.

tty_read() and tty_write() will be called when the character device file is used in the file system. For example, when a program reads the /dev/tty file, it will execute the system-call sys_read() (in fs/read_write.c), and the system-call will call rw_char() function (in fs/char_dev.c) when it determines that the file being read is a character device file, the function will call rw_tty() from the character device read/write function table (device switch table) according to the sub-device number of the read device, etc., and finally call the terminal read operation function tty_read().

The copy_to_cooked() function is invoked by the keyboard interrupt handler (via do_tty_interrupt()) to process the characters in the read_q queue according to the character input/output flags (such as INLCR, OUCLC) set in the terminal termios structure, and convert the characters into a sequence of canonical mode lines of characters and stored in the auxiliary queue (canonical mode queue) for reading by tty_read() above. During the conversion process, if the terminal's echo flag L_ECHO is set, the character is also placed in the write queue write_q, and the terminal write function is called to display the character on the screen. If it is a serial terminal, then the write function will be rs_write() (in serial.c, line 53). rs_write() will send the characters in the serial terminal write queue to the serial terminal over the serial line and display it on the screen of the remote serial terminal. The copy_to_cooked() function will also wake up the processes waiting for the auxiliary queue. The steps to implement the function are as follows:

1. If the read queue is empty or the auxiliary queue is full, skip to the last step (step 10), otherwise perform the following operations;
2. Get a character from the tail pointer of the read queue read_q, and move the tail pointer forward by one character position;
3. If it is a carriage return (CR) or line feed (NL) character, the character is converted according to the state of the input flag (ICRNL, INLCR, INOCR) in the terminal termios structure. For example, if you are reading a carriage return character and the ICRNL flag is set, replace it with a newline character;
4. If the uppercase to lowercase flag IUCLC is set, replace the character with the corresponding lowercase character;
5. If the canonical mode flag ICANON is set, the character is processed in canonical mode:
   a. If it is a delete character (^U), delete a line of characters in the auxiliary queue secondary (the queue head pointer is backed up until a carriage return or line feed is encountered or the queue is empty);
   b. If it is an erase character (^H), delete a character at the head pointer in the secondary, and the head pointer moves back by one character position;

   c. If it is a stop character (^S), set the stop flag of the terminal stopped=1;
   d. If it is the start character (^Q), reset the stop flag of the terminal.

6. If the receive keyboard signal flag ISIG is set, generate a signal corresponding to the typed control character for the process;
7. If it is a line ending character (such as NL or ^D), the line count statistics data of the secondary queue is increased by 1;
8. If the local echo flag is set, the character is also placed in the write queue write_q, and the terminal write function is called to display the character on the screen;
9. Put the character in the auxiliary queue secondary, return to the above step 1 to continue to loop through the other characters in the read queue;
10. Finally wake up the processes that sleep on the secondary queue.

You can check the include/linux/tty.h header file when reading the following program. The header file defines the data structure of the tty character buffer queue and some macro operation definitions. It also defines the ASCII code value of the control character.

10.6.2 Code annotation

Program 10-5 linux/kernel/chr_drv/tty_io.c

```
/*
 *  linux/kernel/tty_io.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 'tty_io.c' gives an orthogonal feeling to tty's, be they consoles
 * or rs-channels. It also implements echoing, cooked mode etc.
 *
 * Kill-line thanks to John T Kohl, who also corrected VMIN = VTIME = 0.
 */

// <ctype.h> The character type file. Defines some macros for character type conversion.
// <errno.h> Error number header file. Contains various error numbers in the system.
// <signal.h> Signal header file. Define signal symbol constants, signal structures, and
//     signal manipulation function prototypes.
// <unistd.h> Linux standard header file. Various symbol constants and types are defined
//     and various functions are declared. If __LIBRARY__ is defined, it also includes the
//     system call number and the inline assembly _syscall0().
#include <ctype.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>

 // Give the corresponding bit mask of the alarm signal in the signal bitmap.
#define ALRMMASK (1<<(SIGALRM-1))

// <linux/sched.h> The scheduler header file defines the task structure task_struct, the
//     data of the initial task 0, and some embedded assembly function macro statements
```

```
 //     about the descriptor parameter settings and acquisition.
 // <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
 //     communication.
 // <asm/system.h> System header file. An embedded assembly macro that defines or
 //     modifies descriptors/interrupt gates, etc. is defined.
 // <asm/segment.h> Segment operation header file. An embedded assembly function is defined
 //     for segment register operatio
#include <linux/sched.h>
#include <linux/tty.h>
#include <asm/segment.h>
#include <asm/system.h>

// Kill the process group function (send a signal to the process group). The parameter pgrp
// specifies the process group number; sig specifies the signal; priv is the priority.
// The main purpose of this function is to send the specified signal sig to each process in
// the specified process group pgrp. As long as it is successfully sent to a process, it will
// return 0. Otherwise, if no process is found for the specified process group number pgrp,
// the error number -ESRCH is returned. If the process with the process group number is pgrp
// is found, but the send signal operation fails, the error code for fail sending is returned.
int kill_pg(int pgrp, int sig, int priv);               // kernel/exit.c, line 171.
// Determine if a process group is an orphan process. Returns 0 if not; else returns 1.
int is_orphaned_pgrp(int pgrp);                         // kernel/exit.c, line 232.

// Get one of the three mode flag sets in the termios structure, or use to determine if a flag
// set contains a set flag.
#define _L_FLAG(tty,f)  ((tty)->termios.c_lflag & f)    // local mode flags.
#define _I_FLAG(tty,f)  ((tty)->termios.c_iflag & f)    // input mode flags.
#define _O_FLAG(tty,f)  ((tty)->termios.c_oflag & f)    // output mode flags.

// Take a flag in the special (local) mode flag set of the termios structure.
#define L_CANON(tty)    _L_FLAG((tty),ICANON)    // canonical mode flag.
#define L_ISIG(tty)     _L_FLAG((tty),ISIG)      // signal (INTR, QUIT etc.) flag.
#define L_ECHO(tty)     _L_FLAG((tty),ECHO)      // echo char flag.
#define L_ECHOE(tty)    _L_FLAG((tty),ECHOE)     // echo erase flag in canon mode.
#define L_ECHOK(tty)    _L_FLAG((tty),ECHOK)     // KILL erase line flag in canon mode.
#define L_ECHOCTL(tty)  _L_FLAG((tty),ECHOCTL)   // echo control char flag.
#define L_ECHOKE(tty)   _L_FLAG((tty),ECHOKE)    // KILL erace line & echo flags in canon mode
#define L_TOSTOP(tty)   _L_FLAG((tty),TOSTOP)    // send SIGTTOU signal for backgnd output.

// Get a flag from the termios structure input mode flag set.
#define I_UCLC(tty)     _I_FLAG((tty),IUCLC)     // get uppercase to lowercase flag.
#define I_NLCR(tty)     _I_FLAG((tty),INLCR)     // Map Line feed NL to CR flag on input.
#define I_CRNL(tty)     _I_FLAG((tty),ICRNL)     // Carriage return CR to NL flag.
#define I_NOCR(tty)     _I_FLAG((tty),IGNCR)     // Ignore CR flag.
#define I_IXON(tty)     _I_FLAG((tty),IXON)      // Input control flow flag XON.

// Get a flag from the termios structure output mode flag set.
#define O_POST(tty)     _O_FLAG((tty),OPOST)     // Post-process output.
#define O_NLCR(tty)     _O_FLAG((tty),ONLCR)     // Map NL to CR-NL flag.
#define O_CRNL(tty)     _O_FLAG((tty),OCRNL)     // Map CR to NL flag.
#define O_NLRET(tty)    _O_FLAG((tty),ONLRET)    // NL performs CR function flag.
#define O_LCUC(tty)     _O_FLAG((tty),OLCUC)     // Lowercase to uppercase flag.

```

```
// Get the baud rate in the termios structure control flag. CBAUD is the baud rate mask (0000017).
#define C_SPEED(tty)     ((tty)->termios.c_cflag & CBAUD)
// Determine whether the tty terminal has been hanged up, that is, whether its transmission
// baud rate is B0 (0).
#define C_HUP(tty)       (C_SPEED((tty)) == B0)

#ifndef MIN
#define MIN(a,b) ((a) < (b) ? (a) : (b))
#endif

// The following defines the buffer queue structure array tty_queues used by the tty terminal
// and the tty terminal table structure array tty_table. QUEUES is the maximum number of buffer
// queues used by tty terminals. The pseudo terminal is divided into two types (master and slave).
// Each tty terminal uses three tty buffer queues, which are the read queue for the buffered
// keyboard or serial input read_queue, the write queue for the buffered screen or serial output,
// write_queue, and the auxiliary queue secondary for saving the canonical mode characters.
#define QUEUES  (3*(MAX_CONSOLES+NR_SERIALS+2*NR_PTYS))     // total 54 queues.
static struct tty_queue tty_queues[QUEUES];
struct tty_struct tty_table[256];

// The following sets the starting position of the buffer queue structure used by various types
// of tty terminals in the tty_queues[] array. 8 virtual console terminals occupy the first
// 24 items of the tty_queues[] array (3 X MAX_CONSOLES) (0 -- 23);
// The two serial terminals occupy the next six items (3 X NR_SERIALS) (24 -- 29).
// The four main pseudo terminals occupy the next 12 items (3 X NR_PTYS) (30 -- 41).
// The four slave pseudo terminals occupy the next 12 items (3 X NR_PTYS) (42 -- 53).
#define con_queues tty_queues
#define rs_queues ((3*MAX_CONSOLES) + tty_queues)
#define mpty_queues ((3*(MAX_CONSOLES+NR_SERIALS)) + tty_queues)
#define spty_queues ((3*(MAX_CONSOLES+NR_SERIALS+NR_PTYS)) + tty_queues)

// The following sets the starting position of the tty structure used by various types of tty
// terminals in the tty_table[] array. 8 virtual console terminals can use 64 items (0 -- 63)
// at the beginning of the tty_table[] array;
// The two serial terminals use the next two items (64 -- 65).
// The four main pseudo terminals use items starting from 128, up to 64 items (128 -- 191).
// The four slave pseudo-terminals use items starting from 192, up to 64 items (192 -- 255).
#define con_table tty_table               // Define the console terminal tty table symbol.
#define rs_table (64+tty_table)           // Serial terminal tty table.
#define mpty_table (128+tty_table)        // The main pseudo terminal tty table.
#define spty_table (192+tty_table)        // The slave pseudo terminal tty table.

int fg_console = 0;             // Current foreground console number (range 0--7).

/*
 * these are the tables used by the machine code handlers.
 * you can implement virtual consoles.
 */
// tty read and write buffer queue structure address table. Used by the rs_io.s program to get
// the address of the read-write buffer queue structure.
struct tty_queue * table_list[]={
        con_queues + 0, con_queues + 1,     // foreground console read/write queue address.
        rs_queues + 0, rs_queues + 1,       // serial terminal 1 read/write queue address.
        rs_queues + 3, rs_queues + 4       // serial terminal 2 read/write queue address.
        };

//// Change the foreground console function.
// Set the foreground console to the specified virtual console.
// Parameters: new_console - the new console number specified.
void change_console(unsigned int new_console)
{
// Exit if the console specified by the parameter is already in the foreground or the parameter
// is invalid. Otherwise, the current foreground console number is set, and the foreground
// console read and write queue structure addresses in table_list[] are updated. Finally update
// the current front console screen.
        if (new_console == fg_console || new_console >= NR_CONSOLES)
                return;
        fg_console = new_console;
        table_list[0] = con_queues + 0 + fg_console*3;
        table_list[1] = con_queues + 1 + fg_console*3;
        update_screen();                   // kernel/chr_drv/console.c, line 936.
}

//// If the queue buffer is empty, the process enters an interruptible sleep state.
// Parameters: queue - a pointer to the specified queue. The process needs to call this function
// to verify before fetching the characters in the queue buffer. If the current process has
// no signal to process and the specified queue buffer is empty, let the process enter an
// interruptible sleep state and let the queue's process wait pointer points to the process.
static void sleep_if_empty(struct tty_queue * queue)
{
        cli();
        while (!(current->signal & ~current->blocked) && EMPTY(queue))
                interruptible_sleep_on(&queue->proc_list);
        sti();
}

//// If the queue buffer is full, the process enters an interruptible sleep state.
// Parameters: queue - a pointer to the specified queue.This function needs to be called to
// determine the queue condition before the process writes characters to the queue buffer.
// Returns if the queue buffer is not full. Otherwise, if the process has no signal to process,
// and the free area remaining in the queue buffer is < 128, the process is put into an
// interruptible sleep state, and the queue's process waits pointer points to the process.
static void sleep_if_full(struct tty_queue * queue)
{
        if (!FULL(queue))
                return;
        cli();
        while (!(current->signal & ~current->blocked) && LEFT(queue)<128)
                interruptible_sleep_on(&queue->proc_list);
        sti();
}

//// Wait for a key to be pressed.
// If foreground console read queue is empty, the process enters interruptible sleep state.
void wait_for_keypress(void)
{
        sleep_if_empty(tty_table[fg_console].secondary);
}

//// Copy and convert to character sequences of canonical mode.
// The characters in the specified terminal read queue are copied and converted into the canonical
// mode (cooked mode) characters and stored in the auxiliary queue according to various flags
// set in the terminal termios structure.
void copy_to_cooked(struct tty_struct * tty)
{
        signed char c;

// First check whether the buffer queue pointers in the current terminal tty structure are valid.
// If all three queue pointers are NULL, there is a problem with tty initialization function.
        if (!(tty->read_q || tty->write_q || tty->secondary)) {
                printk("copy_to_cooked: missing queues\n\r");
                return;
        }
// Otherwise, we will properly process each character fetched from the tty read queue buffer
// according to the input and local flags in the terminal termios structure, and then put it
// into the auxiliary queue secondary.
// In the loop body below, if the read queue is already empty or the auxiliary queue is full
// of characters, the loop body is exited. Otherwise, the program takes a character from the
// pointer at the end of the read queue and moves the tail pointer forward by one position.
// It is then processed according to the character code. In addition, if _POSIX_VDISABLE(\0)
// is defined, if the character code value is equal to the value of _POSIX_VDISABLE, the function
// of the corresponding special control character is prohibited.
        while (1) {
                if (EMPTY(tty->read_q))
                        break;
                if (FULL(tty->secondary))
                        break;
                GETCH(tty->read_q,c);           // get a char and the tail moves foreward.

// If the character is a carriage return CR (13), then if the carriage return to newline (line
// feed) flag CRNL is set, the character is converted to a newline NL (10). Otherwise, if the
// Ignore carriage return flag NOCR is set, the character is ignored and continues to process
// other characters. If the character is a newline character NL(10) and the line feed to carriage
// return flag NLCR is set, it is converted to a CR (13). If the uppercase to lowercase input
// flag UCLC is set, the character is converted to lowercase.
                if (c==13) {
                        if (I_CRNL(tty))
                                c=10;
                        else if (I_NOCR(tty))
                                continue;
                } else if (c==10 && I_NLCR(tty))
                        c=13;
                if (I_UCLC(tty))
                        c=tolower(c);
// If the canonical mode flag CANON in the local mode flag set is set, the characters read are
// processed in the following manner.
// First, if the character is the keyboard termination control character KILL (^U), the deletion
// processing is performed on the current line that has been input. The process of deleting
// a line of characters is: If the tty auxiliary queue is not empty, and the last character
// fetched in the auxiliary queue is not a newline character NL(10), and the character is not
// the end-of-file character (^D) or the end-of-file character is not equal to _POSIX_VDISABLE,
// then we loop through the other characters on the line perform deletion processing. During
// the deletion process, if the local echo flag ECHO is set, then the erase control character
// ERASE (^H) needs to be placed in the write queue. If the character is a control character
// (value < 32) and is represented by a two-byte representation (eg ^V), an additional erase
// character ERASE must be placed. The tty write function is then called to output all the
// characters in the write queue to the terminal device. Finally, the tty auxiliary queue head
// pointer is backed by 1 byte.
                if (L_CANON(tty)) {
                        if ((KILL_CHAR(tty) != _POSIX_VDISABLE) &&
                            (c==KILL_CHAR(tty))) {
                                /* deal with killing the input line */
                                while(!(EMPTY(tty->secondary) ||
                                        (c=LAST(tty->secondary))==10 ||
                                        ((EOF_CHAR(tty) != _POSIX_VDISABLE) &&
                                         (c==EOF_CHAR(tty))))) {
                                        if (L_ECHO(tty)) {
                                                if (c<32)          // control char ?
                                                        PUTCH(127,tty->write_q);
                                                PUTCH(127,tty->write_q);
                                                tty->write(tty);
                                        }
                                        DEC(tty->secondary->head);
                                }
                                continue;          // processing other chars in the line.
                        }
// If the character is the delete control character ERASE(^H) and the delete character is not
// equal to _POSIX_VDISABLE, then: if the tty's auxiliary queue is empty, or its last character
// is a newline NL(10), or the end of file character but not equal to _POSIX_VDISABLE, continue
// to process other characters. During the deletion process, if the local echo flag ECHO is
// set, then the erase control character ERASE (^H) needs to be placed in the write queue. If
// the character is a control character (value < 32) and is represented by a two-byte
// representation (eg ^V), an additional erase character ERASE must be placed. The tty write
// function is then called to output all the characters in the write queue to the terminal device.
// Finally, the tty auxiliary queue head pointer is backed by 1 byte, and the code continue
// to process other charachers.

                        if ((ERASE_CHAR(tty) != _POSIX_VDISABLE) &&
                            (c==ERASE_CHAR(tty))) {
                                if (EMPTY(tty->secondary) ||
                                    (c=LAST(tty->secondary))==10 ||
                                    ((EOF_CHAR(tty) != _POSIX_VDISABLE) &&
                                     (c==EOF_CHAR(tty))))
                                        continue;
                                if (L_ECHO(tty)) {
                                        if (c<32)
                                                PUTCH(127,tty->write_q);
                                        PUTCH(127,tty->write_q);
                                        tty->write(tty);
                                }
                                DEC(tty->secondary->head);
                                continue;
                        }
                }
// If the IXON flag is set, the terminal stop/start output control character is activated. If
// this flag is not set, the stop and start characters will be read as normal characters for
// the process. In this code, if the character read is the stop character STOP(^S), set the
// tty stop flag, let the tty pause the output, discard the special control character (not placed
// in the auxiliary queue), and continue to process the other character. If the character is
// the start character START(^Q), the tty stop flag is reset, the tty output is restored, the
// control character is discarded, and other characters continue to be processed.
// For the console, tty->write() is the con_write() function in console.c. So the console will
// immediately pause displaying new characters on the screen (chr_drv/console.c, line 586) due
// to the discovery of stopped=1. For the pseudo terminal, the write operation is suspended
// (chr_drv/pty.c, line 24) because the terminal stopped flag is set. For the serial terminal,
// the transmission should be suspended according to the terminal stopped flag during the sending
// terminal, but this version is not implemented.
                if (I_IXON(tty)) {
                        if ((STOP_CHAR(tty) != _POSIX_VDISABLE) &&
                            (c==STOP_CHAR(tty))) {
                                tty->stopped=1;
                                tty->write(tty);
                                continue;
                        }
                        if ((START_CHAR(tty) != _POSIX_VDISABLE) &&
                            (c==START_CHAR(tty))) {
                                tty->stopped=0;
                                tty->write(tty);
                                continue;
                        }
                }
// If the ISIG flag is set in the input mode flag set, indicating that the terminal keyboard
// can generate a signal, and when the control characters INTR, QUIT, SUSP or DSUSP are received,
// a corresponding signal needs to be generated for the process. If the character is a keyboard
// interrupt (^C), the keyboard interrupt signal SIGINT is sent to all processes in the process
// group of the current process, and the next character continues to be processed. If the
// character is an quit character (^\), the keyboard quit signal SIGQUIT is sent to all processes
// in the process group of the current process, and the next character continues to be processed.
// If the character is a suspend character (^Z), a stop signal SIGTSTP is sent to the current
// process. Similarly, if _POSIX_VDISABLE(\0) is defined, if the character code value is equal
// to the value of _POSIX_VDISABLE during character processing, the function of the corresponding
// special control character is prohibited.
                if (L_ISIG(tty)) {
                        if ((INTR_CHAR(tty) != _POSIX_VDISABLE) &&
                            (c==INTR_CHAR(tty))) {
                                kill_pg(tty->pgrp, SIGINT, 1);
                                continue;
                        }
                        if ((QUIT_CHAR(tty) != _POSIX_VDISABLE) &&
                            (c==QUIT_CHAR(tty))) {
                                kill_pg(tty->pgrp, SIGQUIT, 1);
                                continue;
                        }
                        if ((SUSPEND_CHAR(tty) != _POSIX_VDISABLE) &&
                            (c==SUSPEND_CHAR(tty))) {
```

```
205                                 if (!is_orphaned_pgrp(tty->pgrp))
206                                         kill_pg(tty->pgrp, SIGTSTP, 1);
207                                 continue;
208                         }
209                 }
    // If the character is a newline character NL(10) or a end of file character EOF(4,^D), indicating
    // that a line of characters has been processed, the line number 'secondary.data' currently
    // contained in the auxiliary queue is incremented by one. If you take a line of characters
    // from the auxiliary queue in the function tty_read(), the line number is decremented by one,
    // see line 315.
210                 if (c==10 || (EOF_CHAR(tty) != _POSIX_VDISABLE &&
211                               c==EOF_CHAR(tty)))
212                         tty->secondary->data++;
    // If the echo flag ECHO in the local mode flag set is set, then if the character is a newline
    // NL(10), the newline NL(10) and the CR(13) must also be placed in the tty write queue, and
    // if the character is a control character (value <32) and the echo control character flag ECHOCTL
    // is set, the character '^' and the character c+64 are placed in the tty write queue (^C, ^H,
    // etc. will be displayed); otherwise, the character will be placed directly into the tty write
    // queue. Finally, the tty write operation function is called.
213                 if (L_ECHO(tty)) {
214                         if (c==10) {
215                                 PUTCH(10,tty->write_q);
216                                 PUTCH(13,tty->write_q);
217                         } else if (c<32) {
218                                 if (L_ECHOCTL(tty)) {
219                                         PUTCH('^',tty->write_q);
220                                         PUTCH(c+64,tty->write_q);
221                                 }
222                         } else
223                                 PUTCH(c,tty->write_q);
224                         tty->write(tty);
225                 }
    // The processed characters are placed in the auxiliary queue at the end of each loop.
    // Finally, wake up any processes waiting for the auxiliary queue after exiting the loop body.
226                 PUTCH(c,tty->secondary);
227         }
228         wake_up(&tty->secondary->proc_list);
229 }
230 
231 /*
232  * Called when we need to send a SIGTTIN or SIGTTOU to our process
233  * group
234  *
235  * We only request that a system call be restarted if there was if the
236  * default signal handler is being used.  The reason for this is that if
237  * a job is catching SIGTTIN or SIGTTOU, the signal handler may not want
238  * the system call to be restarted blindly.  If there is no way to reset the
239  * terminal pgrp back to the current pgrp (perhaps because the controlling
240  * tty has been released on logout), we don't want to be in an infinite loop
241  * while restarting the system call, and have it always generate a SIGTTIN
242  * or SIGTTOU.  The default signal handler will cause the process to stop
243  * thus avoiding the infinite loop problem.  Presumably the job-control
244  * cognizant parent will fix things up before continuging its child process.
```

```
 */
//// Signals to all processes in the process group that use the terminal.
// This function is used to send a SIGTTIN or SIGTTOU signal to all processes in the background
// process group when a process in the group accesses the control terminal. Regardless of whether
// the process in the background process group has blocked or ignored these two signals, the
// current process will immediately exit the read and write operations and return.
int tty_signal(int sig, struct tty_struct *tty)
{
// We do not want to stop processes in an orphan process group (see the description on line
// 232 in the file kernel/exit.c). So if the current process group is an orphan process group,
// an error is returned. Otherwise, the specified signal sig is sent to all processes of the
// current process group.
        if (is_orphaned_pgrp(current->pgrp))
                return -EIO;            /* don't stop an orphaned pgrp */
        (void) kill_pg(current->pgrp,sig,1);    // send signal sig.
// If this signal is blocked (masked) by the current process, or ignored, an error is returned;
// Otherwise, if the current process has set a new handler for the signal sig, then we can return
// the information that we can be interrupted; Otherwise, it returns information that can be
// executed after the system-call is restarted.
        if ((current->blocked & (1<<(sig-1))) ||
            ((int) current->sigaction[sig-1].sa_handler == 1))
                return -EIO;            /* Our signal will be ignored */
        else if (current->sigaction[sig-1].sa_handler)
                return -EINTR;          /* We _will_ be interrupted :-) */
        else
                return -ERESTARTSYS;    /* We _will_ be interrupted :-) */
                                        /* (but restart after we continue) */
}

//// tty read function.
// Reads the specified number of characters from the terminal auxiliary queue and places them
// in the user-specified buffer. Parameters: channel - the sub-device number; buf - the user
// buffer pointer; nr - the number of bytes to read. Returns the number of bytes read.
int tty_read(unsigned channel, char * buf, int nr)
{
        struct tty_struct * tty;
        struct tty_struct * other_tty = NULL;
        char c, * b=buf;
        int minimum,time;

// First determine the validity of the function parameters and take the tty structure pointer
// of the terminal. If the three queue pointers of the tty terminal are both NULL, an EIO error
// message is returned. If the tty terminal is a pseudo terminal, another tty structure other_tty
// corresponding to the pseudo terminal needs to be obtained.
        if (channel > 255)
                return -EIO;
        tty = TTY_TABLE(channel);
        if (!(tty->write_q || tty->read_q || tty->secondary))
                return -EIO;
// If the current process uses the tty terminal being processed here, but the process group
// number of the terminal is different from the current process group number, it indicates that
// the current process is a process in the background process group, that is, the process is
// not in the foreground. So we need to stop all processes in the current process group. Therefore,
    // it is necessary to send the SIGTTIN signal to the current process group, and return, waiting
    // to become the foreground process group and then perform the read operation.
    // In addition, if the current terminal is a pseudo terminal, the corresponding other pseudo
    // terminal is other_tty. If the tty here is the primary pseudo terminal, then other_tty is
    // the corresponding slave pseudo terminal, and vice versa.
273         if ((current->tty == channel) && (tty->pgrp != current->pgrp))
274                 return(tty_signal(SIGTTIN, tty));
275         if (channel & 0x80)
276                 other_tty = tty_table + (channel ^ 0x40);

    // The code then sets the read character operation timeout timing value and the minimum number
    // of characters to be read based on the control character array values corresponding to VTIME
    // and VMIN. In non-canonical mode, these two are timeout timing values. VMIN represents the
    // minimum number of characters that need to be read in order to satisfy the read operation.
    // VTIME is a 1/10 second count timing value.
277         time = 10L*tty->termios.c_cc[VTIME];        // set read timeout timing value.
278         minimum = tty->termios.c_cc[VMIN];          // the minimum nr of chars to read.
    // If the tty terminal is in the canonical mode, set the minimum number of characters to be
    // read to be equal to the number of characters to be read, nr, and set the timeout value of
    // the process to read nr characters to a maximum value (no timeout). Otherwise, the terminal
    // is in non-canonical mode. If the minimum number of read characters is set at this time, the
    // temporary read-timeout timing value is set to infinity temporarily, so that the process reads
    // the existing characters in the auxiliary queue first. If the number of characters read is
    // less than minimum, the following code sets the read timeout value of the process according
    // to the specified timeout value time, and waits for the rest of the characters to be read,
    // see line 328.
    // If the minimum number of read characters minimum is not set at this time, it is set to the
    // number of characters to be read nr, and if the timeout timing value is set, the process read
    // timeout timing value timeout is set to the current system time + the specified timeout time,
    // and then the time is reset. In addition, if the minimum number of read characters set above
    // is greater than the number of characters nr to be read by the process, let minimum=nr. That
    // is, for the read operation in the canonical mode, it is not subject to the constraints and
    // control of the corresponding control character values of VTIME and VMIN, and they only function
    // in the non-canonical mode (raw mode) operation.
279         if (L_CANON(tty)) {
280                 minimum = nr;
281                 current->timeout = 0xffffffff;
282                 time = 0;
283         } else if (minimum)
284                 current->timeout = 0xffffffff;
285         else {
286                 minimum = nr;
287                 if (time)
288                         current->timeout = time + jiffies;
289                 time = 0;
290         }
291         if (minimum>nr)
292                 minimum = nr;              // reads up to the required number of chars.

    // Now we start to loop out the characters from the auxiliary queue and put them in the user
    // buffer buf. When the number of bytes to be read is greater than 0, the following loop operation
    // is performed. During the loop, if the current terminal is a pseudo terminal, then we execute
    // the write operation function of its corresponding other pseudo terminal, allowing another
// pseudo terminal to write the character into the current pseudo terminal auxiliary queue
// buffer. That is, the other terminal copies the characters in the write queue buffer to the
// current pseudo terminal read queue buffer, and converts it into the current pseudo terminal
// auxiliary queue after being converted by the canonical mode function.
        while (nr>0) {
                if (other_tty)
                        other_tty->write(other_tty);

// If the tty auxiliary queue is empty, or the canonical mode flag is set and the tty read queue
// is not full, and the number of character lines in the auxiliary queue is 0, then if the process
// read character timeout value (0) is not set, or the current process receives the signal,
// then we exit the loop body first. Otherwise, if the terminal is a slave pseudo terminal and
// its corresponding master pseudo terminal has been hung up, then we also exit the loop body.
// If it is not one of the above two situations, we will let the current process enter the
// interruptible sleep state and continue processing after returning. Since the kernel provides
// data to the user in character line units in the canonical mode, there must be at least one
// line of characters in the auxiliary queue in this mode, that is, the minimum of secondary.data
// is 1.
                cli();
                if (EMPTY(tty->secondary) || (L_CANON(tty) &&
                    !FULL(tty->read_q) && !tty->secondary->data)) {
                        if (!current->timeout ||
                          (current->signal & ~current->blocked)) {
                                sti();
                                break;
                        }
                        if (IS_A_PTY_SLAVE(channel) && C_HUP(other_tty))
                                break;
                        interruptible_sleep_on(&tty->secondary->proc_list);
                        sti();
                        continue;
                }
                sti();
// The following begins the formal execution of the character fetching operation. The number
// of characters to be read nr is successively decremented until nr=0 or the auxiliary buffer
// queue is empty.
// In this loop, the auxiliary queue character c is first taken, and the queue tail pointer
// tail is shifted to the right by one character position. If the character got is a file
// terminator (^D) or a newline character NL(10), the number of character lines contained in
// the auxiliary queue is decremented by one. If the character is a file end character (^D)
// and the canonical mode flag is set, the loop is interrupted, otherwise the file end character
// has not been encountered or is in the raw (non-canonical) mode. In this mode, the user uses
// the character stream as the read object and does not recognize the control characters (such
// as the file terminator). The code then puts the character directly into the user data buffer
// buf and decrements the number of characters to be read by one. At this time, if the number
// of characters to be read is already 0, the loop is interrupted. In addition, if the terminal
// is in canonical mode and the character read is a newline NL(10), the loop is also exited.
// In addition, as long as the nr characters have not been taken, and the auxiliary queue is
// not empty, the characters in the queue are continuously read.
                do {
                        GETCH(tty->secondary,c);
                        if ((EOF_CHAR(tty) != _POSIX_VDISABLE &&
                             c==EOF_CHAR(tty)) || c==10)
```

```
                                tty->secondary->data--;
                        if ((EOF_CHAR(tty) != _POSIX_VDISABLE &&
                             c==EOF_CHAR(tty)) && L_CANON(tty))
                                break;
                        else {
                                put_fs_byte(c,b++);
                                if (!--nr)
                                        break;
                        }
                        if (c==10 && L_CANON(tty))
                                break;
                } while (nr>0 && !EMPTY(tty->secondary));

// At this point, if the tty terminal is in canonical mode, we may have read a newline or
// encountered a end of file character. If it is in non-canonical mode, then we have read nr
// characters, or the auxiliary queue has been taken out. So we first wake up the process waiting
// for the read queue, and then see if the timeout timing value is set. If the timeout timing
// value is not 0, we will wait for a certain amount of time for other processes to write characters
// to the read queue. So we set the process read timeout timing value to the system current
// time jiffies + read timeout time. Of course, if the terminal is in canonical mode, or if
// nr characters have been read, we can exit this big loop directly.
                wake_up(&tty->read_q->proc_list);
                if (time)
                        current->timeout = time+jiffies;
                if (L_CANON(tty) || b-buf >= minimum)
                        break;
        }
// At this point, the tty character loop operation ends, so we reset the process's read timeout
// timing value timeout. If the current process has received a signal and no characters have
// been read yet, it is returned by restarting the system-call number, otherwise it returns
// the number of characters read (b-buf).
        current->timeout = 0;
        if ((current->signal & ~current->blocked) && !(b-buf))
                return -ERESTARTSYS;
        return (b-buf);
}

//// tty write function.
// Put the characters in the user buffer into the tty write queue buffer.
// Parameters: channel - the sub-device number; buf - the buffer pointer; nr - the number of
// bytes written. Returns the number of bytes written.
int tty_write(unsigned channel, char * buf, int nr)
{
        static cr_flag=0;
        struct tty_struct * tty;
        char c, *b=buf;

// First determine the validity of the parameters and take the tty structure pointer of the
// terminal. If the three queue pointers of the tty terminal are both NULL, an EIO error message
// is returned.
        if (channel > 255)
                return -EIO;
        tty = TTY_TABLE(channel);
        if (!(tty->write_q || tty->read_q || tty->secondary))
                return -EIO;
// If TOSTOP is set in the terminal local mode flag set, it means that the background process
// output needs to send a signal SIGTTOU. At this time, if the current process uses the tty
// terminal being processed here, but the process group number of the terminal is different
// from the current process group number, it means that the current process is a process in
// the background process group, that is, the process is not in the foreground. So we need to
// stop all processes in the current process group. Therefore, it is necessary to send the SIGTTOU
// signal to the current process group, and return, waiting to become the foreground process
// group and then perform the write operation.
        if (L_TOSTOP(tty) &&
            (current->tty == channel) && (tty->pgrp != current->pgrp))
                return(tty_signal(SIGTTOU, tty));

// Now we start looping out the characters from the user buffer buf and putting them into the
// write queue buffer. When the number of bytes to be written is greater than 0, the following
// loop operation is performed. During the loop, if the tty write queue is full at this time,
// the current process enters an interruptible sleep state. If the current process has a signal
// to process, exit the loop body.
        while (nr>0) {
                sleep_if_full(tty->write_q);
                if (current->signal & ~current->blocked)
                        break;
// When the number of characters to be written nr is still greater than 0 and the tty write
// queue buffer is not full, the following operations are performed cyclically. First take 1
// byte from the user buffer. If the execution output processing flag OPOST in the terminal
// output mode flag set is set, the post-processing operation on the character is performed.
                while (nr>0 && !FULL(tty->write_q)) {
                        c=get_fs_byte(b);
                        if (O_POST(tty)) {
// If the character is a carriage return '\r' (CR, 13) and the carriage return conversion line
// character OCRNL is set, the character is replaced with a newline character '\n' (NL, 10);
// Otherwise, if the character is a newline character '\n' and the line feed return function
// flag ONLRET is set, the character is replaced with a carriage return character '\r'.
                                if (c=='\r' && O_CRNL(tty))
                                        c='\n';
                                else if (c=='\n' && O_NLRET(tty))
                                        c='\r';
// If the character is a newline character '\n' and the carriage return flag cr_flag is not
// set, but the line feed to the car-new line flag ONLCR is set, the cr_flag flag is set and
// a CR is placed in the write queue. Then we continue to process the next character.
// If the lowercase to uppercase flag OLCUC is set, the character is converted to uppercase
// character.
                                if (c=='\n' && !cr_flag && O_NLCR(tty)) {
                                        cr_flag = 1;
                                        PUTCH(13,tty->write_q);
                                        continue;
                                }
                                if (O_LCUC(tty))
                                        c=toupper(c);
                        }
// Next, the user data buffer pointer b is forwarded by 1 byte; the number of bytes to be written
// is reduced by one byte; the cr_flag flag is reset, and the byte is placed in the tty write
// queue.
                        b++; nr--;
                        cr_flag = 0;
                        PUTCH(c,tty->write_q);
                }
// If the required characters are all written, or the write queue is full, the program exits
// the loop. At this point, the corresponding tty write function is called to display the
// characters in the write queue on the console screen or send them out through the serial port.
// If the currently processed tty is a console terminal, then tty->write() is con_write(); if
// the tty is a serial terminal, tty->write() is rs_write() function. If there are still bytes
// to write, then we need to wait for the characters in the write queue to be taken. So call
// the scheduler here and go to other tasks first.
                tty->write(tty);
                if (nr>0)
                        schedule();
        }
        return (b-buf);                 // finally returns the number of bytes written.
}

/*
 * Jeh, sometimes I really like the 386.
 * This routine is called from an interrupt,
 * and there should be absolutely no problem
 * with sleeping even in an interrupt (I hope).
 * Of course, if somebody proves me wrong, I'll
 * hate intel for all time :-). We'll have to
 * be careful and see to reinstating the interrupt
 * chips before calling this, though.
 *
 * I don't think we sleep here under normal circumstances
 * anyway, which is good, as the task sleeping might be
 * totally innocent.
 */
//// Function called in interrupt handler - character canonical mode processing.
// Parameters: tty - the specified tty terminal number.
// Copy or convert the characters in the specified tty terminal queue into canonical (cooked)
// mode characters and store them in the auxiliary queue. This function is called in the serial
// port read character interrupt (rs_io.s, 110) and the keyboard interrupt (kerboard.S, 76).
void do_tty_interrupt(int tty)
{
        copy_to_cooked(TTY_TABLE(tty));
}

//// Character device initialization function. Empty, ready for future expansion.
void chr_dev_init(void)
{
}

//// tty terminal initialization function.
// Initialize all terminal buffer queues, initialize serial terminal and console terminal.
void tty_init(void)
{
        int i;
```

```
409
    // First initialize the buffer queue structure of all terminals and set the initial value. For
    // serial terminal read/write buffer queues, set their data field to the serial port base address.
    // Serial port 1 is 0x3f8, and serial port 2 is 0x2f8. Then initially set the tty structure
    // of all terminals. The initial value of the special character array c_cc[] is defined in the
    // include/linux/tty.h file.
410             for (i=0 ; i < QUEUES ; i++)
411                     tty_queues[i] = (struct tty_queue) {0,0,0,0,""};
412             rs_queues[0] = (struct tty_queue) {0x3f8,0,0,0,""};      // read queue.
413             rs_queues[1] = (struct tty_queue) {0x3f8,0,0,0,""};      // write queue.
414             rs_queues[3] = (struct tty_queue) {0x2f8,0,0,0,""};
415             rs_queues[4] = (struct tty_queue) {0x2f8,0,0,0,""};
416             for (i=0 ; i<256 ; i++) {
417                     tty_table[i] =  (struct tty_struct) {
418                             {0, 0, 0, 0, 0, INIT_C_CC},
419                             0, 0, 0, NULL, NULL, NULL, NULL
420                     };
421             }
    // Then initialize the console terminal (console.c, line 834). We put con_init() here because
    // we need to determine the number of virtual consoles in the system, NR_CONSOLES, based on
    // the type of display card and the amount of video memory. This value is used in the subsequent
    // console tty structure initialization loop. For the tty structure of the console, the 425--430
    // lines are the termios structure fields contained in the tty structure. The input mode flag
    // set is initialized to an ICRNL flag; the output mode flag is initialized to include a
    // post-processing flag OPOST and a flag ONLCR that converts NL to CRNL; the local mode flag
    // set is initialized to include IXON, ICANON, ECHO, ECHOCTL, and ECHOKE flags; The control
    // character array c_cc[] is set to contain the initial value INIT_C_CC. The read buffer, write
    // buffer, and auxiliary buffer queue structures in the tty structure of the console terminal
    // are initialized on line 435, which respectively point to the corresponding structure items
    // in the tty queue structure array tty_table[], see the description on lines 61--73.
422             con_init();
423             for (i = 0 ; i<NR_CONSOLES ; i++) {
424                     con_table[i] = (struct tty_struct) {
425                             {ICRNL,         /* change incoming CR to NL */
426                             OPOST|ONLCR,    /* change outgoing NL to CRNL */
427                             0,                              // control mode flag set.
428                             IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE, // local flag set.
429                             0,              /* console termio */    // 0 -- TTY.
430                             INIT_C_CC},                             // control char array c_cc[]
431                             0,              /* initial pgrp */
432                             0,              /* initial session */
433                             0,              /* initial stopped */
434                             con_write,                              // console write function.
435                             con_queues+0+i*3,con_queues+1+i*3,con_queues+2+i*3
436                     };
437             }
    // Then initialize the fields in the tty structure of the serial terminal. Line 450 initializes
    // the read/write and auxiliary buffer queue structures in the serial terminal tty structure,
    // which point to the corresponding structure items in the tty buffer queue structure array
    // tty_table[]. See the description on line 61--73.
438             for (i = 0 ; i<NR_SERIALS ; i++) {
439                     rs_table[i] = (struct tty_struct) {
440                             {0, /* no translation */   // input mode flag set
```

```
                        0,   /* no translation */   // output mode flag set.
                        B2400 | CS8,                // control mode flag set.2400bps,8bits.
                        0,                          // local mode flag set.
                        0,                          // line procedure, 0 -- TTY.
                        INIT_C_CC},                 // control character array.
                        0,                          // initial process group.
                        0,                          // init session.
                        0,                          // initial stopped flag.
                        rs_write,                   // serial port write function.
                        rs_queues+0+i*3,rs_queues+1+i*3,rs_queues+2+i*3  // three queues.
                };
        }
// Then we reinitialize the tty structure used by the pseudo terminal. The pseudo terminal is
// paired, that is, a master pseudo terminal is equipped with a slave pseudo terminal. Therefore,
// they must be initialized. In the following loop, we first initialize the tty structure of
// each master pseudo terminal, and then initialize its corresponding tty structure of each
// slave pseudo terminal.
        for (i = 0 ; i<NR_PTYS ; i++) {
                mpty_table[i] = (struct tty_struct) {
                        {0,  /* no translation */   // input mode flag set.
                        0,   /* no translation */   // output mode flag set.
                        B9600 | CS8,              // control mode flag set. 9600bps, 8bits.
                        0,                          // local mode flag set.
                        0,                          // line procedure, 0--TTY.
                        INIT_C_CC},                 // control character array.
                        0,                          // initial process group.
                        0,                          // initial session.
                        0,                          // initial stopped flag.
                        mpty_write,                 // master pseudo write function.
                        mpty_queues+0+i*3,mpty_queues+1+i*3,mpty_queues+2+i*3
                };
                spty_table[i] = (struct tty_struct) {
                        {0,  /* no translation */
                        0,   /* no translation */
                        B9600 | CS8,
                        IXON | ISIG | ICANON,       // local mode flag set.
                        0,
                        INIT_C_CC},
                        0,
                        0,
                        0,
                        spty_write,                 // slave pseudo write function.
                        spty_queues+0+i*3,spty_queues+1+i*3,spty_queues+2+i*3
                };
        }
// Finally, the serial interrupt handler and serial interfaces 1 and 2 (serial.c, line 37) are
// initialized, and the number of virtual consoles NR_CONSOLES and the number of pseudo terminals
// NR_PTYS contained in the system are displayed.
        rs_init();
        printk("%d virtual consoles\n\r",NR_CONSOLES);
        printk("%d pty's\n\r",NR_PTYS);
}

```

10.6.3 Information

10.6.3.1 Control characters VTIME, VMIN

In non-canonical mode, these two values are the timeout timing value and the minimum number of characters read. VMIN indicates the minimum number of characters that need to be read in order to satisfy the read operation. VTIME is a one-tenth second count timeout value. When both are set, the read operation will wait until at least one character is read. If VMIN characters are received before the timeout, the read operation is satisfied. If the timeout has expired before the VMIN characters are received, the characters that have been received at this time are returned to the user. If only VMIN is set, the read operation will not return until VMIN characters are read. If only VTIME is set, the read will return immediately after reading at least one character or timing out. If neither is set, the read will return immediately, giving only the number of bytes currently read. See the termios.h file for details.

10.7 tty_ioctl.c

10.7.1 Function

This program is used for the control operation of character devices and implements the function tty_ioctl(). By using this function, the user program can modify information such as setting flags in the termios structure of the specified terminal. The tty_ioctl() function will be called by the input and output control system in sys_ioctl() in fs/ioctl.c to implement a file system-based unified device access interface.

Instead of using the sys_ioctl() system-call directly, the general user program uses the associated functions implemented in the library file. For example, for the terminal IO control command TIOCGPGRP that obtains the terminal process group number (ie, the foreground process group number), the library file libc uses the command to call the sys_ioctl() system-call to implement the function tcgetpgrp(). Therefore, ordinary users only need to use tcgetpgrp() to achieve the same purpose. Of course, we can also use the library function ioctl() to achieve the same functionality.

10.7.2 Code annotation

Program 10-6 linux/kernel/chr_drv/tty_ioctl.c

```
/*
 *  linux/kernel/chr_drv/tty_ioctl.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <errno.h> Error number header file. Contains various error numbers in the system.
// <termios.h> Terminal input and output function header file. It mainly defines the terminal
//     interface that controls the asynchronous communication port.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
//     communication.
// <asm/io.h> Io header file. Defines the function that operates on the io port in the form of
//     a macro’s embedded assembler.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined
//     for segment register operations.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//     descriptors/interrupt gates, etc. is defined.
#include <errno.h>
#include <termios.h>

#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/tty.h>

#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>

// The following lines gives two external function prototypes and an array of baud rate factors
// for the serial ports. The first function is used to obtain the session number to which the
// process group belongs according to the process group number pgrp (defined in kernel/exit.c,
// line 161). The second function tty_signal() is used to signal to all processes in the process
// group that use the specified tty terminal, which is defined in chr_drv/tty_io.c, line 246.
// The baud rate factor array (or array of divisors) gives the correspondence between the baud
// rate and the baud rate factor. For example, when the baud rate is 2400 bps, the corresponding
// factor is 48 (0x30); the factor of 9600 bps is 12 (0x1c). See the instructions after the
// program list.
extern int session_of_pgrp(int pgrp);
extern int tty_signal(int sig, struct tty_struct *tty);

static unsigned short quotient[] = {
        0, 2304, 1536, 1047, 857,
        768, 576, 384, 192, 96,
        64, 48, 24, 12, 6, 3
};

//// Modify the transmission baud rate.
// Parameters: tty - the tty data structure corresponding to the terminal. When the divisor
// latch flag DLAB is set, the baud rate factor low byte and the high byte are respectively
// written to the UART through the ports 0x3f8 and 0x3f9 of the serial port 1, and the DLAB
// bit is reset after the writing. For serial port 2, the two ports are 0x2f8 and 0x2f9.
static void change_speed(struct tty_struct * tty)
{
        unsigned short port,quot;

// The function first checks if the terminal specified by the parameter tty is a serial terminal,
// and if not, exits. For the tty structure of the serial terminal, the data field of the read
// queue stores the base address of the serial port (0x3f8 or 0x2f8), and the read_q.data field
// value of the general console terminal is 0. Then we obtain the baud rate index number that
// has been set from the control mode flag set of the terminal termios structure, and obtain
// the corresponding baud rate factor value from the baud rate factor array quotient[]. CBAUD
// is the baud rate bits mask in the control mode flag set.
```

```
        if (!(port = tty->read_q->data))
                return;
        quot = quotient[tty->termios.c_cflag & CBAUD];
// Then, the baud rate factor is written into the baud rate factor latch of the UART chip
// corresponding to the serial port. Before writing, we must first set the divisor latch access
// bit DLAB (bit 7) of the line control register LCR, and then write the 16-bit baud rate factor
// low and high byte to port 0x3f8, 0x3f9 respectively. Finally reset the DLAB flag bit of the
// LCR.
        cli();
        outb_p(0x80,port+3);            /* set DLAB */
        outb_p(quot & 0xff,port);       /* LS of divisor */
        outb_p(quot >> 8,port+1);       /* MS of divisor */
        outb(0x03,port+3);              /* reset DLAB */
        sti();
}

//// Clear the tty buffer queue.
// Parameter: queue is the specified buffer queue pointer.
// Let the buffer head pointer be equal to the tail pointer, so as to clear the buffer.
static void flush(struct tty_queue * queue)
{
        cli();
        queue->head = queue->tail;
        sti();
}

static void wait_until_sent(struct tty_struct * tty)
{
        /* do nothing - not implemented */
}

static void send_break(struct tty_struct * tty)
{
        /* do nothing - not implemented */
}

//// Obtain terminal termios structure information.
// Parameters: tty - specifies the terminal's tty structure pointer; termios - the user buffer
// that used to hold the termios structure.
static int get_termios(struct tty_struct * tty, struct termios * termios)
{
        int i;

// First, verify that the memory area indicated by the user buffer pointer is sufficient. If
// not, allocate memory. Then copy the termios structure information of the specified terminal
// to the user buffer. Finally returns 0.
        verify_area(termios, sizeof (*termios));           // kernel/fork.c, line 24.
        for (i=0 ; i< (sizeof (*termios)) ; i++)
                put_fs_byte( ((char *)&tty->termios)[i] , i+(char *)termios );
        return 0;
}

//// Set the information in the termios structure of the terminal.
```

```
// Parameters: tty - specifies the terminal's tty structure pointer; termios - user data area
// termios structure pointer.
static int set_termios(struct tty_struct * tty, struct termios * termios,
                        int channel)
{
        int i, retsig;

        /* If we try to set the state of terminal and we're not in the
           foreground, send a SIGTTOU.  If the signal is blocked or
           ignored, go ahead and perform the operation.  POSIX 7.2) */
// If the process group number of the tty terminal of the current process is different from
// the process group number of the process, that is, the current process terminal is not in
// the foreground, indicating that the current process attempts to modify the termios structure
// of the uncontrolled terminal. Therefore, according to the requirements of the POSIX standard,
// the SIGTTOU signal needs to be sent here to allow the process using this terminal to temporarily
// stop execution, so that we can modify the termios structure first. However, if the send
// function tty_signal() returns a value of ERESTARTSYS or EINTR, then the operation will be
// performed again later.
        if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
                retsig = tty_signal(SIGTTOU, tty);          // chr_drv/tty_io.c, line 246.
                if (retsig == -ERESTARTSYS || retsig == -EINTR)
                        return retsig;
        }
// The termios structure information in the user data area is then copied to the termios structure
// of the specified terminal tty structure. Because the user may have modified the terminal
// serial port transmission baud rate, the baud rate in the serial UART chip is modified again
// according to the baud rate information in the control mode flag c_cflag in the termios
// structure, and finally returns 0. .
        for (i=0 ; i< (sizeof (*termios)) ; i++)
                ((char *)&tty->termios)[i]=get_fs_byte(i+(char *)termios);
        change_speed(tty);
        return 0;
}

//// Get the information in the termio structure.
// Parameters: tty - specifies the terminal's tty structure pointer; termio - the user buffer
// that holds the termio structure information.
static int get_termio(struct tty_struct * tty, struct termio * termio)
{
        int i;
        struct termio tmp_termio;

// First verify that the user's buffer capacity is sufficient, if not enough, then allocate
// memory. The information for the termios structure is then copied into the temporary termio
// structure. These two structures are basically the same, but the data types of the input,
// output, control, and local flagsets are different. The former's data type is long, while
// the latter is short. Therefore, the purpose of copying to the temporary termio structure
// is for data type conversion. Finally, the information in the temporary termio structure is
// copied byte by byte into the user buffer and returns 0.
        verify_area(termio, sizeof (*termio));
        tmp_termio.c_iflag = tty->termios.c_iflag;
        tmp_termio.c_oflag = tty->termios.c_oflag;
        tmp_termio.c_cflag = tty->termios.c_cflag;
        tmp_termio.c_lflag = tty->termios.c_lflag;
        tmp_termio.c_line = tty->termios.c_line;
        for(i=0 ; i < NCC ; i++)
                tmp_termio.c_cc[i] = tty->termios.c_cc[i];
        for (i=0 ; i< (sizeof (*termio)) ; i++)
                put_fs_byte( ((char *)&tmp_termio)[i] , i+(char *)termio );
        return 0;
}

/*
 * This only works as the 386 is low-byt-first
 */
//// Set the termio structure information of the terminal.
// Parameters: tty - specifies the terminal's tty structure pointer; termio - the termio
// structure in the user data area. This function is used to copy the information of the user
// buffer termio into the terminal's termios structure and return 0.
static int set_termio(struct tty_struct * tty, struct termio * termio,
                        int channel)
{
        int i, retsig;
        struct termio tmp_termio;

// As with set_termios(), if the process group number of the terminal used by the process is
// different from the process group number of the process, that is, the current process terminal
// is not in the foreground, indicating that the current process attempts to modify the termios
// structure of the uncontrolled terminal. Therefore, according to the requirements of the POSIX
// standard, the SIGTTOU signal needs to be sent here to allow the process using this terminal
// to temporarily stop execution, so that we can modify the termios structure first. However,
// if the send function tty_signal() returns a value of ERESTARTSYS or EINTR, then the operation
// will be performed again later.
        if ((current->tty == channel) && (tty->pgrp != current->pgrp)) {
                retsig = tty_signal(SIGTTOU, tty);
                if (retsig == -ERESTARTSYS || retsig == -EINTR)
                        return retsig;
        }
// Then copy the termio structure information in the user data area to the temporary termio
// structure, and then copy the information into the tty's termios structure. The purpose of
// this is to convert the type of the mode flag set, that is, from the short integer type of
// termio to the long integer type of termios. But the c_line and c_cc[] fields of the two
// structures are identical.
        for (i=0 ; i< (sizeof (*termio)) ; i++)
                ((char *)&tmp_termio)[i]=get_fs_byte(i+(char *)termio);
        *(unsigned short *)&tty->termios.c_iflag = tmp_termio.c_iflag;
        *(unsigned short *)&tty->termios.c_oflag = tmp_termio.c_oflag;
        *(unsigned short *)&tty->termios.c_cflag = tmp_termio.c_cflag;
        *(unsigned short *)&tty->termios.c_lflag = tmp_termio.c_lflag;
        tty->termios.c_line = tmp_termio.c_line;
        for(i=0 ; i < NCC ; i++)
                tty->termios.c_cc[i] = tmp_termio.c_cc[i];
// Finally, because the user may have modified the terminal serial port speed, the baud rate
// in the serial UART chip is then modified according to the baud rate information in the control
// mode flag c_cflag in the termios structure, and returns 0.
        change_speed(tty);
```

```
130         return 0;
131 }
132
    //// tty terminal device input and output control function.
    // Parameters: dev - device number; cmd - ioctl command; arg - operation parameter pointer.
    // The function first finds the tty structure of the corresponding terminal according to the
    // device number, and then processes it according to the control command cmd.
133 int tty_ioctl(int dev, int cmd, int arg)
134 {
135         struct tty_struct * tty;
136         int     pgrp;
137
    // First, the tty sub-device number is obtained according to the device number, thereby obtaining
    // the tty structure of the terminal. If the major device number is 5 (control terminal), the
    // tty field of the process is the tty sub-device number. At this time, if the process's tty
    // sub-device number is a negative number, it indicates that the process does not have a control
    // terminal, that is, the ioctl call cannot be issued, and an error message is displayed and
    // the machine is stopped. If the major device number is not 5 but 4, we can get the sub-device
    // number from the device number. The sub-device number can be 0 (console terminal), 1 (serial
    // 1 terminal), or 2 (serial 2 terminal).
138         if (MAJOR(dev) == 5) {
139                 dev=current->tty;
140                 if (dev<0)
141                         panic("tty_ioctl: dev<0");
142         } else
143                 dev=MINOR(dev);
    // Then according to the sub-device number and the tty table, we can get the tty structure of
    // the corresponding terminal. So let tty point to the tty structure corresponding to the
    // sub-device number, and then separately process according to the ioctl command cmd provided
    // by the parameter. The second half of line 144 is used to select the corresponding tty structure
    // in the tty_table[] table based on the sub-device number dev. If dev = 0, it means that the
    // foreground terminal is being used, so we can directly use the terminal number fg_console
    // as the tty_table[] entry index to get the tty structure. If dev is greater than 0, then it
    // should be considered in two cases: (1) dev is the virtual terminal number; (2) dev is the
    // serial terminal number or pseudo terminal number. For virtual terminals, the tty structure
    // in tty_table[] is indexed by dev-1(0 -- 63), and for other types of terminals, their tty
    // structure index entry is dev. For example, if dev = 64, indicating a serial terminal 1, its
    // tty structure is tty_table[dev]. If dev = 1, the tty structure of the corresponding terminal
    // is tty_table[0]. See lines 70-73 of the tty_io.c program.
144         tty = tty_table + (dev ? ((dev < 64)? dev-1:dev) : fg_console);

    // TCGETS: Get the termios structure information of the corresponding terminal. At this point
    // the parameter arg is the user buffer pointer.
    // TCSETSF: Before setting the termios, we need to wait for all data in the output queue to
    // be processed and flush the input queue. Then perform the operation of setting the termios.
    // TCSETSW: Before setting the termios, we need to wait for all the data in output queue to
    // be handled. This form is required for situations where modifying params affects the output.
    // TCSETS: Set the corresponding terminal termios structure information. At this point the
    // parameter arg is the user buffer pointer that holds the termios structure.
    // TCGETA: Get the information in the termio structure of the corresponding terminal. At this
    // point the parameter arg is the user buffer pointer.
    // TCSETAF: Before setting termio, we need to wait for all data in the output queue to be processed
    // and the input queue is flushed. Then perform the setting terminal termio operation.
    // TCSETAW: Before setting the termios, we need to wait for all data in the output queue to
    // be processed. This form is required for situations where modifying params affects the output.
    // TCSETA: Set the termio structure information of the corresponding terminal. At this point
    // the parameter arg is the user buffer pointer that holds the termio structure.
    // TCSBRK: If the value of the parameter arg is 0, wait for the output queue to be processed
    // and then send a break.
            switch (cmd) {
                    case TCGETS:
                            return get_termios(tty,(struct termios *) arg);
                    case TCSETSF:
                            flush(tty->read_q);     /* fallthrough */
                    case TCSETSW:
                            wait_until_sent(tty);   /* fallthrough */
                    case TCSETS:
                            return set_termios(tty,(struct termios *) arg, dev);
                    case TCGETA:
                            return get_termio(tty,(struct termio *) arg);
                    case TCSETAF:
                            flush(tty->read_q);    /* fallthrough */
                    case TCSETAW:
                            wait_until_sent(tty);  /* fallthrough */
                    case TCSETA:
                            return set_termio(tty,(struct termio *) arg, dev);
                    case TCSBRK:
                            if (!arg) {
                                    wait_until_sent(tty);
                                    send_break(tty);
                            }
                            return 0;
    // Start/stop flow control. If the parameter arg is TCOOFF (Terminal Control Output OFF), the
    // output is suspended; if it is TCOON, the pending output is restored. At the same time as
    // suspending or restoring the output, the characters in the write queue need to be output to
    // speed up the user interaction response. If arg is TCIOFF (Terminal Control Input ON), the
    // input is suspended; if it is TCION, the pending input is re-opened.
                    case TCXONC:
                            switch (arg) {
                            case TCOOFF:
                                    tty->stopped = 1;    // stop the terminal output.
                                    tty->write(tty);     // write queue output.
                                    return 0;
                            case TCOON:
                                    tty->stopped = 0;    // restore the terminal output.
                                    tty->write(tty);
                                    return 0;
    // If the parameter arg is TCIOFF, it means that the terminal is required to stop input, so
    // we put the STOP character into the terminal write queue, and the input will be suspended
    // when the terminal receives the character. If the parameter is TCION, it means that a START
    // character is sent to let the terminal resume the transmission.
    // STOP_CHAR(tty) is defined as ((tty)->termios.c_cc[VSTOP]), which is used to obtain the
    // corresponding value of the control character array of the terminal termios structure. If
    // the kernel defines _POSIX_VDISABLE(\0), then when an item value equals the value of
    // _POSIX_VDISABLE, it means that the corresponding special character is prohibited. So here
    // directly check if the value is 0 or not to determine whether to put the stop control character
    // into the terminal write queue. The same is true of the following.
                            case TCIOFF:
                                    if (STOP_CHAR(tty))
                                            PUTCH(STOP_CHAR(tty),tty->write_q);
                                    return 0;
                            case TCION:
                                    if (START_CHAR(tty))
                                            PUTCH(START_CHAR(tty),tty->write_q);
                                    return 0;
                            }
                            return -EINVAL; /* not implemented */
    // Flushes data that has been written but not yet sent, or received but not yet read. If the
    // parameter arg is 0, the input queue is flushed ; if it is 1, the output queue is flushed;
    // if it is 2, the input and output queues are both flushed.
                    case TCFLSH:
                            if (arg==0)
                                    flush(tty->read_q);
                            else if (arg==1)
                                    flush(tty->write_q);
                            else if (arg==2) {
                                    flush(tty->read_q);
                                    flush(tty->write_q);
                            } else
                                    return -EINVAL;
                            return 0;
    // TIOCEXCL: Sets the terminal serial line dedicated mode.
    // TIOCNXCL: Resets the terminal serial line dedicated mode.
    // TIOCSCTTY: Set tty as the control terminal. (TIOCNOTTY - no control terminal).
    // TIOCGPGRP: Read terminal process group number (that is, read the foreground process group
    // number). First verify the user buffer length, then copy the pgrp field of the terminal tty
    // to the user buffer. At this point the parameter arg is the user buffer pointer.
                    case TIOCEXCL:
                            return -EINVAL; /* not implemented */
                    case TIOCNXCL:
                            return -EINVAL; /* not implemented */
                    case TIOCSCTTY:
                            return -EINVAL; /* set controlling term NI */
                    case TIOCGPGRP:                       // implement function tcgetpgrp().
                            verify_area((void *) arg,4);
                            put_fs_long(tty->pgrp,(unsigned long *) arg);
                            return 0;

    // Set the terminal process group number pgrp (that is, set the foreground process group number).
    // The parameter arg is now a pointer to the process group number pgrp in the user buffer. The
    // prerequisite for executing this command is that the process must have a control terminal.
    // If the current process does not have a control terminal, or dev is not its control terminal,
    // or the control terminal is now the terminal dev being processed, but the session number of
    // the process is different from the session number of the terminal dev, then a terminalless
    // error message is returned.
    // Then we get the process group number from the user buffer and verify the validity of the
    // group number. If the group number pgrp is less than 0, an invalid group number error message
    // is returned; if the session number of pgrp is different from the current process, an permission
    // error message is returned. Otherwise we can set the process group number of the terminal
    // to prgp. At this point prgp becomes the foreground process group.
                    case TIOCSPGRP:                        // implement function tcsetpgrp().
                            if ((current->tty < 0) ||
                                (current->tty != dev) ||
                                (tty->session != current->session))
                                    return -ENOTTY;
                            pgrp=get_fs_long((unsigned long *) arg); // get from user buffer.
                            if (pgrp < 0)
                                    return -EINVAL;
                            if (session_of_pgrp(pgrp) != current->session)
                                    return -EPERM;
                            tty->pgrp = pgrp;
                            return 0;
    // TIOCOUTQ: Returns the number of characters that remains in the output queue. First verify
    // the user buffer length, then copy the number of characters in the queue to the user. At this
    // point the parameter arg is the user buffer pointer.
    // TIOCINQ: Returns the number of characters not yet read in the input auxiliary queue. The
    // user buffer length is first verified, and then the number of characters in the auxiliary
    // queue is copied to the user. At this point the parameter arg is the user buffer pointer.
                    case TIOCOUTQ:
                            verify_area((void *) arg,4);
                            put_fs_long(CHARS(tty->write_q),(unsigned long *) arg);
                            return 0;
                    case TIOCINQ:
                            verify_area((void *) arg,4);
                            put_fs_long(CHARS(tty->secondary),
                                    (unsigned long *) arg);
                            return 0;
    // TIOCSTI: Simulate terminal input operations. The command takes a pointer to a character as
    // a parameter and assumes that the character was typed on the terminal. The user must have
    // superuser privileges or read permission on the control terminal.
    // TIOCGWINSZ: Read terminal device window size (see the winsize structure in termios.h).
    // TIOCSWINSZ: Sets the terminal device window size information (see winsize structure).
                    case TIOCSTI:
                            return -EINVAL; /* not implemented */
                    case TIOCGWINSZ:
                            return -EINVAL; /* not implemented */
                    case TIOCSWINSZ:
                            return -EINVAL; /* not implemented */
    // TIOCMGET: Returns the current status bit flag set for the MODEM status control pin.
    // (see lines 185 -- 196 in termios.h).
    // TIOCMBIS: Sets the state of a single MODEM state control pin (true or false).
    // TIOCMBIC: Resets the state of a single MODEM state control pin.
    // TIOCMSET: Sets the state of the MODEM status pin. If a bit is set, the corresponding status
    // pin will be asserted.
                    case TIOCMGET:
                            return -EINVAL; /* not implemented */
                    case TIOCMBIS:
                            return -EINVAL; /* not implemented */
                    case TIOCMBIC:
                            return -EINVAL; /* not implemented */
                    case TIOCMSET:
                            return -EINVAL; /* not implemented */
```

```
// TIOCGSOFTCAR: Read the software carrier detect flag (1 - on; 0 - off).
// TIOCSSOFTCAR: Set the software carrier detect flag (1 - on; 0 - off).
                case TIOCGSOFTCAR:
                        return -EINVAL; /* not implemented */
                case TIOCSSOFTCAR:
                        return -EINVAL; /* not implemented */
                default:
                        return -EINVAL;
        }
}

```

10.7.3 Information

10.7.3.1 Baud Rate and Baud Rate Factor

The baud rate is calculated as: baud rate = 1.8432MHz / (16 X baud rate factor).
The corresponding relationship between the common baud rate and the baud rate factor is shown in Table 10-9.

Table 10-9 Baud rate and baud rate factor table

Baud rate	Baud rate factor		Baud rate	Baud rate factor	
	MSB,LSB	Value		MSB,LSB	Value
50	0x09,0x00	2304	1200	0x00,0x60	96
75	0x06,0x00	1536	1800	0x00,0x40	64
110	0x04,0x17	1047	2400	0x00,0x30	48
134.5	0x03,0x59	857	4800	0x00,0x18	24
150	0x03,0x00	768	9600	0x00,0x1c	12
200	0x02,0x40	576	19200	0x00,0x06	6
300	0x01,0x80	384	38400	0x00,0x03	3
600	0x00,0xc0	192			

10.8 Summary

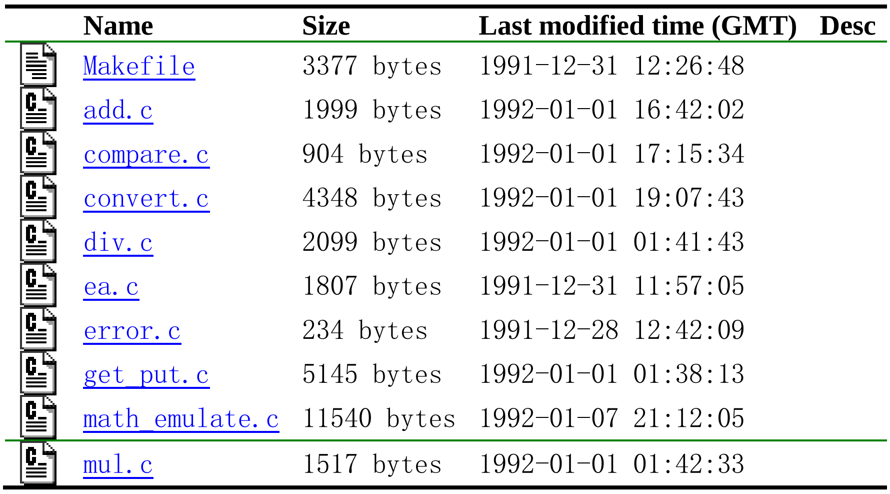
This chapter describes the implementation of the character device in detail, and introduces the working principle and implementation code of the local terminal device and the dumb terminal based on serial communication. We first give the hierarchical relationship of modules accessing character devices in the UNIX operating system, as well as several common types of terminals. Then combined with the termios data structure and three buffer queue structures used by the terminal, the definitions of the canonical mode (cooked mode) and the non-canonical mode (raw mode) and its program implementation are described in detail, and the console terminal and serial terminal are explained. We then introduced and annotated each program in detail in the order of the files.

In the next chapter, we will detail the basic functions of the math coprocessor and the basic data types used, and specify the implementation code for the software emulation of the math coprocessor in the Linux kernel.

11 Math Coprocessor (math)

The kernel directory kernel/math directory contains the math coprocessor simulation programs, which contains a total of nine C programs, as shown in Listing 11-1. The content of this chapter is closely related to the specific hardware structure, so the reader needs to have a deep knowledge of Intel CPU and coprocessor instruction code structure. Fortunately, this content has little to do with the kernel implementation, so skipping this chapter does not prevent the reader from a complete understanding of the kernel implementation. However, if you understand the contents of this chapter, it will be very helpful to implement system-level applications (such as assembly and disassembly) and to develop coprocessor floating-point handlers.

List 11-1 linux/kernel/math



Name	Size	Last modified time (GMT)	Desc
Makefile	3377 bytes	1991-12-31 12:26:48	
add.c	1999 bytes	1992-01-01 16:42:02	
compare.c	904 bytes	1992-01-01 17:15:34	
convert.c	4348 bytes	1992-01-01 19:07:43	
div.c	2099 bytes	1992-01-01 01:41:43	
ea.c	1807 bytes	1991-12-31 11:57:05	
error.c	234 bytes	1991-12-28 12:42:09	
get_put.c	5145 bytes	1992-01-01 01:38:13	
math_emulate.c	11540 bytes	1992-01-07 21:12:05	
mul.c	1517 bytes	1992-01-01 01:42:33	

11.1 Function Description

Performing computationally intensive operations on a computer can usually be done in three ways. One is to perform calculations directly using CPU normal instructions. Since general CPU instructions are a class of general-purpose instructions, the use of these instructions for complex and large-scale computation requires complex computational subroutines, and generally only those skilled in mathematics and computers can program these subroutines. Another method is to configure a dedicated math coprocessor chip for the CPU. Using a coprocessor chip can greatly simplify the difficulty of mathematical programming, and the speed and efficiency of the operation will be doubled, but additional hardware investment is required. Another method is to use an emulator at the kernel level of the system to simulate the computational functions of the coprocessor. This method may be the one with the lowest speed and efficiency, but it is as easy to use as a coprocessor to program the calculation program, and can run the program without any changes to the program on a machine with a coprocessor and therefore has good code compatibility.

In the early days of Linux 0.1x and even Linux 0.9x kernel development, the math coprocessor chip 80387 (or its compatible chip) was expensive and has always been a luxury in ordinary PCs. Therefore, unless there is a large amount of scientific calculations or where it is particularly needed, the 80387 chip will not be installed in a general PC. Although the math coprocessor feature is built into the current Intel processor, the coprocessor emulator code is no longer required in the current operating system. But because the 80387 emulation program is completely based on the analog 80387 chip processing structure and analysis of instruction code structure. So after learning the contents of this chapter, we can not only fully understand the programming method of the 80387 coprocessor, but also help to write system-level programs like assembler and disassembler.

If the 80386 PC does not include the 80387 math coprocessor chip, then when the CPU executes a coprocessor instruction, it will cause a "device nonexistent" exception interrupt. The processing code for this exception procedure starts at line 158 of sys_call.s. If the operating system has already set the EM bit of the CPU control register CR0 at initialization, then the math_emulate() function in the math_emulate.c program is called to "interpret" each instruction of the coprocessor with the software.

The math coprocessor emulator math_emulate.c in the Linux 0.12 kernel fully emulates the way the 80387 chip executes coprocessor instructions. Before processing a coprocessor instruction, the program first creates a "soft" 80387 environment in memory using data structures and other types, including emulating all 80387 internal stack accumulator groups ST[], control word registers CWD, status word register SWD, and feature word TWD (TAG word) registers, Then, the current coprocessor instruction opcode causing the exception is analyzed, and the corresponding mathematical simulation operation is performed according to the specific opcode. Therefore, before describing the processing of the math_emulate.c program, it is necessary to introduce the internal structure and basic working principle of the 80387.

11.1.1 Floating point data type

This section focuses on the types of floating point data used by the coprocessor. First, let's briefly review several representations of integers, and then explain several standard representations of floating-point numbers and temporary real-number representations used in operations in 80387.

1. Integer data type

For Intel 32-bit CPUs, there are three basic unsigned data types: byte, word, and double word, with 8, 16, and 32 bits, respectively. The representation of an unsigned number is simple. Each bit in a byte represents a binary number and has different weights depending on where it is located. For example, an 8-bit unsigned binary number 0b10001011 can be expressed as:

$$U = 0b10001011 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 139$$

It corresponds to the decimal number 139. The one with the smallest weight (2^0) is usually called the Least Significant Bit (LSB), and the bit with the largest weight (2^7) is called the Most Significant Bit (MSB).

Moreover, there are usually three types of integer data representations with negative values in the computer: Two's complement, biased number, and Sign magnitude representation. Table 11-1 gives some of the values represented by these three forms.

Table 11-1 Several representations of integers

Decimal	2's complement	Biased (127)	Sign magnitude
128	Not Available (NA)	0b11111111	NA
127	0b01111111	0b11111110	0b01111111
126	0b01111110	0b11111101	0b01111110

2	0b00000010	0b10000001	0b00000010
1	0b00000001	0b10000000	0b00000001
0	0b00000000	0b01111111	0b00000000
-0	NA	NA	0b10000000
-1	0b11111111	0b01111110	0b10000001
-2	0b11111110	0b01111101	0b10000010
-126	0b10000010	0b00000001	0b11111110
-127	0b10000001	0b00000000	0b11111111
-128	0b10000000	NA	NA

The 2's complement notation is the integer representation used by most computer CPUs today, because the simple addition of unsigned numbers of the CPU is also applicable to data operations in this format. Using this notation, a negative number is the number plus one after each bit is inverted. The MSB bit is the sign bit of the number. MSB = 0 for a positive number; MSB = 1 for a negative number. The 80386 CPU has 8-bit (1 byte), 16-bit (1 word) and 32-bit (double word) 2's complement data types. The data range that can be represented by each is: -128 -- 127, -32768 -- 32767 , -2147483648 -- 2146473647.

The biased representation of numbers is typically used to represent index field values in the floating point format. Adding a number to the specified biased value is the biased number representation of the number. As can be seen from Table 11-1, the numerical values of this representation have an order of magnitude of unsigned numbers. Therefore, this representation is easy to compare numerical values, that is, the large value of the biased representation value is always a large number of unsigned values, while the other two representations are not.

The signed number representation has a bit dedicated to the representation of sign (0 for positive number and 1 for negative number), while other bits are identical to the values represented by unsigned integers. The valid number (mantissa) portion of a floating point number uses the representation method, and the sign bit represents the sign of the entire floating point number.

In addition, a format we call a temporary integer type is used in the 80387 emulator, as shown in Figure 11-1. It is 10 bytes long and can represent a 64-bit integer data type. The lower 8 bytes can represent a maximum of 63 unsigned numbers, while the highest 2 bytes use only the most significant bits to indicate the positive or negative value. For 32-bit integer values, the lower 4 bytes are used, and the 16-bit integer value is represented by the lower 2 bytes.

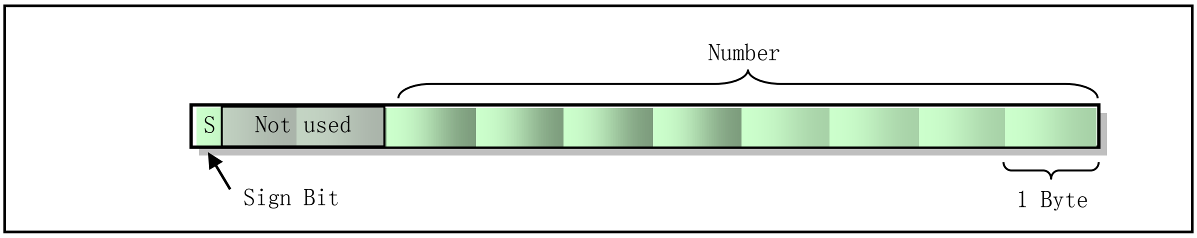


Figure 11-1 Temporary integer format supported by the emulator

2. 2. BCD data type

The BCD (Binary Coded Decimal) code value is a binary coded decimal value. For compressed BCD code, each byte can represent a two-digit decimal number, where each 4 bits represents a digit of 0-9. For example, the compressed BCD code representation of the decimal number 59 is 0x01011001. For uncompressed BCD codes, each byte uses only the lower 4 bits to represent a 1-digit decimal number.

The 80387 coprocessor supports the representation and operation of a 10-byte compressed BCD code, which can represent an 18-bit decimal number, as shown in Figure 11-2. Similar to the temporary integer format, where the highest byte uses only the sign bit (most significant bit) to indicate the positive or negative of the value, and the remaining bits are not used. If the BCD code data is negative, the highest valid bit of at the highest byte is used to indicate a negative value. Otherwise all bits of the highest byte are 0.

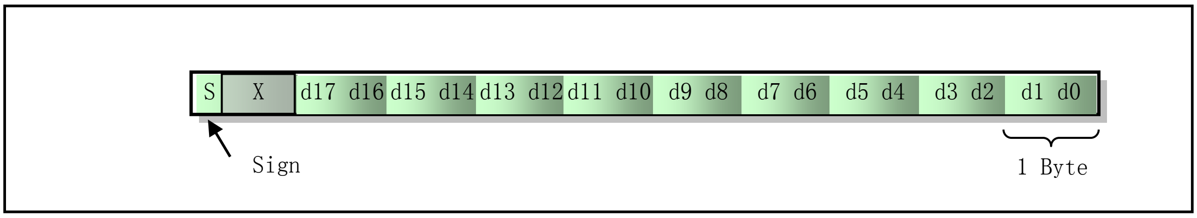


Figure 11-2 BCD code data type supported by 80387

3. Floating point data type

A number with an integer part and a fractional (mantissa) part is called a real number or a floating point number. In fact, an integer is a real number with a fraction part of 0 and is a subset of the real set. Since computers use fixed-length bits to represent a number, computers are not able to accurately represent all real numbers. Since the number of bits allocated to the fractional part is not fixed in order to represent the most accurate real value in a fixed length bit when the computer represents a real number, that is, the decimal point can be "floating", so the real data type represented by the computer also known as floating point numbers. In order to facilitate the porting of the program, the floating point number representation specified by IEEE Standard 754 (or 854) is currently used in the computer to represent the real number.

The general format of this real number representation is shown in Figure 11-3. It consists of a Significand part, an Exponent part, and a Sign bit. Significand is composed of an integer 1 and a fraction part. The 80387 coprocessor supports three real types, and the number of bits used in each part is shown in Figure 11-4. With the exception of the 80-bit temporary real (or called extended-real) format, all of these data types exist in memory only. When they are loaded into 80387 coprocessor data registers, they are converted into temporary-real format and operated on in that format.

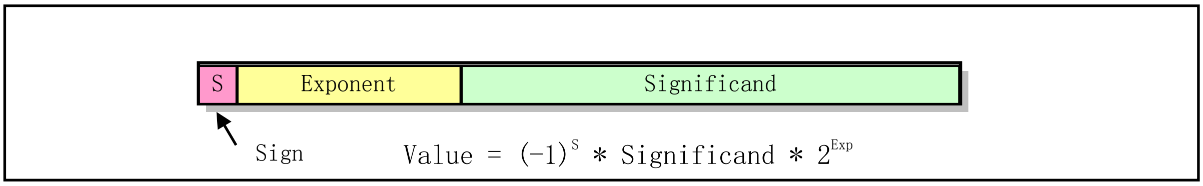


Figure 11-3 Floating-point general format

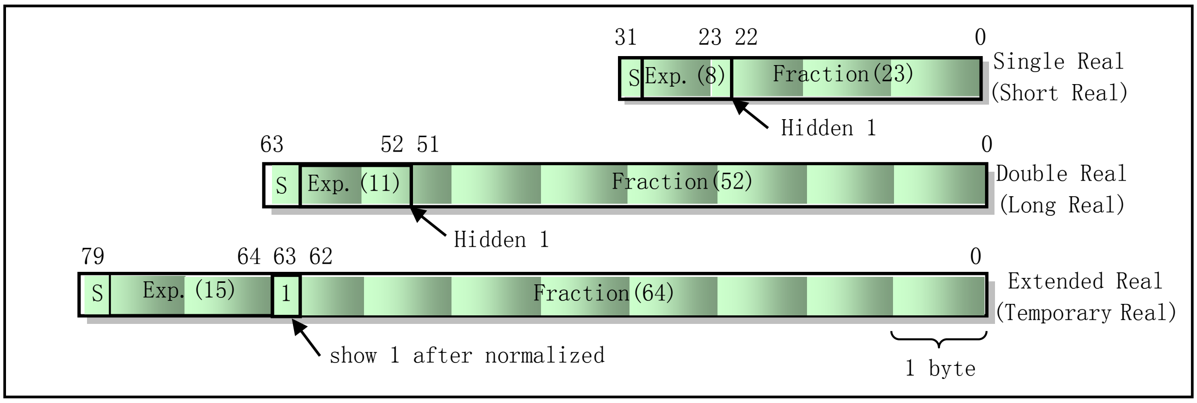


Figure 11-4 Real number format used by the 80387 coprocessor

In Figure 11-4, S is a sign bit of one bit. S=1 indicates a negative real number; S=0 indicates a positive real number. The Significand gives the significant digits or mantissa of the real number. When using the exponential form, a real number can be expressed in a variety of forms. For example, the decimal real number 10.34 can be expressed as 1034.0 x 10^-2, or 10.34 x 10^0, or 1.034 x 10^1 or 0.1034 x 10^2, and the like. In order for the calculation to get the maximum precision value, we always normalize the real number, that is, adjust the exponent value of the real number so that the binary most significant value is always 1, and the decimal point is on the right side. Therefore, the correct normalization result of the above example is 1.034 x 10^1. For binary numbers it is 1.XXXXX * 2^N (where X is 1 or 0). If we always use this form to represent a real number, then the left of the decimal point must be 1. So in the short real (single precision) and long real (double precision) formats of 80387, this '1' does not need to be explicitly expressed. Therefore, in a binary real number of a short real number or a long real number, 0x0111...010 is actually 0x1.0111...010.

The exponent field in the format contains the power of 2 required to represent a number as a normalized form. As mentioned earlier, in order to facilitate comparison of the size of the numbers, the 80387 uses the biased number form to store the exponent value. The biased bases of the short real number, long real number, and the extended real number (temporary real number) are 127, 1023, and 16383, respectively, and the corresponding hexadecimals are 0x7F, 0x3FF, and 0x3FFF, respectively. Therefore, a short real exponent value of 0b10000000 actually represents the 2^1 (0b01111111 + 0b00000001).

In addition, the temporary real number is the format of the number represented by the 80387 internal operation. Its most significant number 1 is explicitly placed at bit 63, and regardless of the data type you are giving (for example, integer, short real or BCD code, etc.), 80387 will convert it to this temporary real number format. The purpose of the 80387 is to maximize accuracy and minimize overflow exceptions during the operation. Explicitly indicating 1 is because 80387 does need this bit during the operation (used to represent very small values). When a short or long real number entered into 80387 is converted to a temporary real number format, a 1 is explicitly placed at bit 63.

During the processing, the program defines a data structure for the temporary real number in the linux/math_emu.h file:

```
typedef struct {
     long a,b;         // a is lower 32 bits, b is upper 32 bits (including 1 fixed bit).
     short exponent;
} temp_real;
```

Among them, the 64-bit mantissa is represented by two long variables. The variable a is the lower 32 bits, and b is the upper 32 bits (including the 1 fixed bit). In addition, in order to solve the problem of byte alignment of the gcc compiler in the data structure at the time, another structure with the same function is defined.

```
typedef struct {
        short m0,m1,m2,m3;
        short exponent;
} temp_real_unaligned;
```

4. Special real numbers

Similar to the case where some values in the above table cannot be represented, some values expressed in real numbers also have special meaning. For temporary real numbers in the 80-bit length format, 80387 does not use all of the range values it can represent. Table 11-2 is all possible values that can be represented by the temporary real number in use of 80387, where the 1 bit of the left side of the dashed line of the significant number column indicates the bit 63 of the temporary real number, that is, the bit of the value 1 is explicitly indicated. Short real numbers and long real numbers do not have this bit, so there are no pseudo denormalized categories in the table. Let's take a look at some of the special values: zero, infinity, denormalized, pseudo-normalized, and the signalling NaN (Not a Number) and quiet NaN.

Table 11-2 The type and range of values that temporary real numbers represent.

Sign	Biased Exponent	Significand		Types
0/1	11...11	1	11...11	NaNs - QNaNs
0/1	11...11	1	...	Quiet NaNs
0/1	11...11	1	10...00	Indefinite
0/1	11...11	1	01...11	Signalling NaNs - SNaNs
0/1	11...11	1	...	
0/1	11...11	1	00...01	
0/1	11...11	1	00...00	Infinite
0/1	11...10	1	11...11	Normals
0/1	...	1	...	
0/1	00...01	1	00...00	
0/1	00...00	1	11...11	Pseudo-Denormals
0/1	00...00	1	...	
0/1	00...00	1	00...00	
0/1	00...00	0	11...11	Denormals
0/1	00...00	0	...	
0/1	00...00	0	00...01	
0/1	00...00	0	00...00	Zero

Zero is the value where the exponent and the significand are both 0, and the remaining exponents with a value of 0 are reserved, that is, the value of the exponent of 0 cannot represent a normal real value. Infinite are values where the exponent value is all 1, the significand value is all zeros, and all remaining values with exponent values of 0x11...11 are also reserved.

The denormals number is a special class value used to represent very small values. It can indicate a progressive underflow or a loss of progressive accuracy. The value is usually required to be represented as a normalized number (left shift until the most significant bit of the significant number is bit 1), whereas the most significant digit of the non-normalized number is not 1. At this time, the biased exponent 0x00...00 is a special representation of the short real number, the long real number, and the temporary real number exponent value of 2^-126, 2^-1022, 2^-16382, respectively. This representation is special because the biased exponent value 0x00...01 also represents the same exponent values 2^-126, 2^-1022, 2^-16382 for the three real types, respectively.

The pseudo-denormals class value is the value of the most significant bit of the significand is 1, and the bit of the denormals class value is 0. Pseudo-denormal numbers are rare, they can be represented by normalized class numbers but not. Since it has been explained above that the special biased exponent 0x00...00 has the same value as the exponent 0x00...01 of the normalized number, the pseudo denormalized class number can be expressed as a normalized class value.

Another special case is NaN (Not a Number). NaN comes in two forms: it produces signals (Signaling NaN) and does not produce signals or is called Quiet NaN. An invalid operation exception is thrown when a signalling NaN (SNaN) is used for operation, while a quiet NaN (QNaN) does not. SNaN can be used to determine that all variables have been initialized before use. The method is that the program can initialize the variables to SNaN values, so that if an uninitialized value is used during the operation, an exception is thrown. Of course, NaN class values can also be used to store other information.

80387 itself does not produce a value of the SNaN class, but it will produce a value of the QNaN class. When a invalid operation exception occurs, 80387 will generate a QNaN class value, and the result of the operation will be an indefinite type value. The indefinite value is a special QNaN class value. Each data type has a number that represents an indefinite value. For integers, the largest negative number is used to represent its indefinite.

There are also some temporary real values that are not supported by 80387, that is, those that are not listed in the above table. If 80387 encounters these unsupported values, an invalid operation exception will be thrown.

11.1.2 Math Coprocessor Function and Structure

Although the 80386 is a general-purpose microprocessor, its instructions are not very suitable for mathematical calculations. Therefore, if you use 80386 to perform mathematical calculations, you need to compile very complex programs, and the execution efficiency is relatively low. As an auxiliary processing chip of the 80386, the 80387 greatly expands the programmer's programming range. What the programmers were not likely to do before, using the coprocessor, can be done easily and quickly and accurately.

The 80387 has a special set of registers that allow the 80387 to operate directly on orders of magnitude larger or smaller than the 80386 can handle. The 80386 uses a binary complement to represent a number. This method is not suitable for representing decimals. The 80387 does not use the 2's complement method to represent the value. It uses the 80-bit (10-byte) format specified by IEEE Standard 754. This format not only has broad compatibility, but also the ability to represent large (or small) values in binary. For example, it can represent values as large as 1.21 X 10^4932 and can handle numbers as small as 3.3 X 10^-4932. The 80387 does not maintain a fixed decimal point position. If the value is small, it uses more decimal places; if the value is large, it uses a few decimal places. Therefore the position of the decimal point can be "floating". This is also the origin of the term "floating point".

To support floating point operations, the 80387 contains three sets of registers, as shown in Figure 11-5. (1) Eight 80-bit data registers (accumulators) that can be used to temporarily store eight floating-point operands, and that these accumulators can perform stack operations; (2) Three 16-bit status and control registers: a status word register SWD, a control word register CWD, and a feature (TAG) register; (3) Four 32-bit error pointer registers (FIP, FCS, FOO, and FOS) are used to determine the instruction and memory operands that caused the 80387 internal exception.

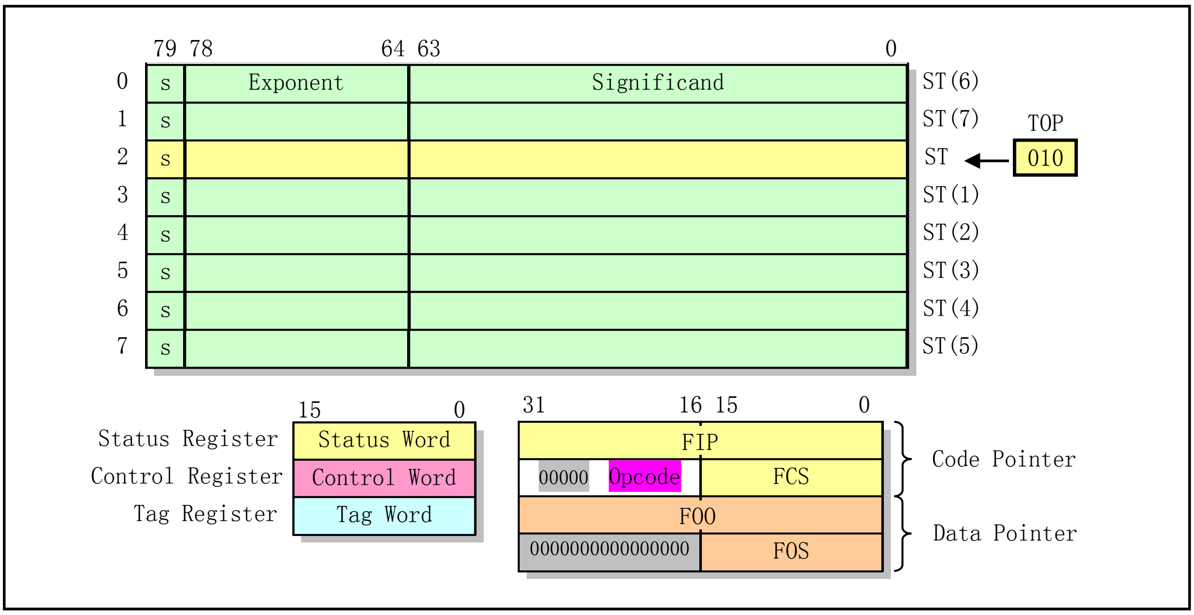


Figure 11-5 80387 registers

1. Stack floating point accumulator

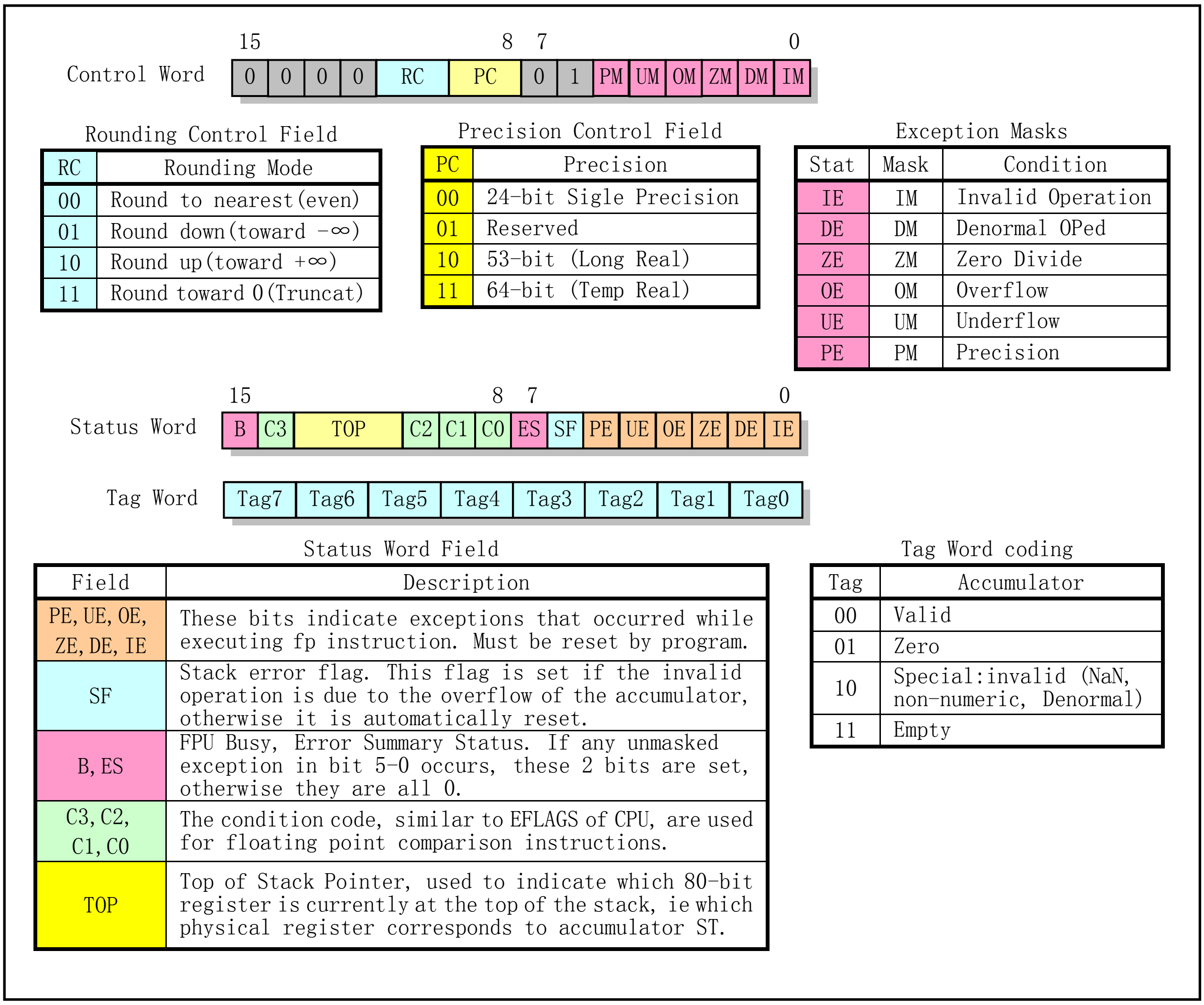
During the execution of floating-point instructions, eight 80-bit physical register sets are used as stack accumulators. Although each 80-bit register has a fixed physical order position (ie 0--7 on the left), the current top of the stack is indicated by ST (ie ST(0)). The remaining accumulators under ST are indicated by the name ST(i) (i = 1 -- 7). As for which 80-bit physical register is the current stack top ST, it is specified by the specific operation process. The 3-bit field named TOP in the status word register (shown in Figure 11-6) contains the absolute position of the 80-bit physical register corresponding to the current top ST. A push or load operation will decrement the TOP field value by one and store the new value in the new ST. After the push operation, the original ST becomes ST(1), and the original ST(7) becomes the current ST. That is, the names of all accumulators have changed from the original ST(i) to ST((i+1)&0x7). A pop or store operation will read the value from the current TOP register ST and store it in memory, and increment the TOP field value by 1. Therefore, after the pop operation, the original ST (ie ST(0)) becomes ST(7), and the original ST(1) becomes the new ST. That is, the names of all accumulators are changed from the original ST(i) to ST((i-1)& 0x7).

ST acts like an accumulator because it is used as an implicit operand for all floating point instructions. If there is another operand, then the second operand can be one of any remaining accumulators ST(i), or a memory operand. Each accumulator in the stack provides an 80-bit space stored in a temporary real format for a real number, with the most significant bit being the sign bit, bits 78--64 being the 15-bit exponent field, and bits 63--0 being 64-bit valid number field.

Floating-point instructions are designed to take full advantage of this accumulator stack mode. A floating-point load instruction (FLD, etc.) reads an operand from memory and pushes it onto the stack, while a floating-point store instruction takes a value from the current top of the stack and writes it to memory. If the value in the stack is no longer needed, the pop operation can also be performed at the same time. Operations such as summation and multiplication will take the contents of the current ST register as one operand and the other from other registers or memory, and save the result in ST after the calculation. There is also a type of "operate and pop-up" operation for computing between ST and ST(1). This form of operation performs a pop-up and then places the result in a new ST.

2. Status and Control Register

Three 16-bit registers (TAG words, control words, and status words) control the operation of floating point instructions and provide status information for them. Their specific format is shown in Figure 11-6, which will be explained one by one below.



Control Word

15					8	7							0
0	0	0	0	RC	PC	0	1	PM	UM	OM	ZM	DM	IM

Rounding Control Field

RC	Rounding Mode
00	Round to nearest(even)
01	Round down(toward -∞)
10	Round up(toward +∞)
11	Round toward 0(Truncat)

Precision Control Field

PC	Precision
00	24-bit Sigle Precision
01	Reserved
10	53-bit (Long Real)
11	64-bit (Temp Real)

Exception Masks

Stat	Mask	Condition
IE	IM	Invalid Operation
DE	DM	Denormal OPed
ZE	ZM	Zero Divide
OE	OM	Overflow
UE	UM	Underflow
PE	PM	Precision

Status Word

15						8	7						0
B	C3	TOP	C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

Tag Word

Tag7	Tag6	Tag5	Tag4	Tag3	Tag2	Tag1	Tag0

Status Word Field

Field	Description
PE, UE, OE, ZE, DE, IE	These bits indicate exceptions that occurred while executing fp instruction. Must be reset by program.
SF	Stack error flag. This flag is set if the invalid operation is due to the overflow of the accumulator, otherwise it is automatically reset.
B, ES	FPU Busy, Error Summary Status. If any unmasked exception in bit 5-0 occurs, these 2 bits are set, otherwise they are all 0.
C3, C2, C1, C0	The condition code, similar to EFLAGS of CPU, are used for floating point comparison instructions.
TOP	Top of Stack Pointer, used to indicate which 80-bit register is currently at the top of the stack, ie which physical register corresponds to accumulator ST.

Tag Word coding

Tag	Accumulator
00	Valid
01	Zero
10	Special:invalid (NaN, non-numeric, Denormal)
11	Empty

Figure 11-6 Control, Status, and Tag Register Format

A. Control Word

The 16-bit Control Word can be used to program various processing options to control the operation of the 80387. It can be divided into three parts: (1) RC (Rounding Control) of bits 11--10 is a rounding control field for rounding the calculation result; (2) The PC (Precision Control) of bits 9-8 is a precision control field for adjusting the accuracy of the calculation result before saving to the specified memory location. All other operations use temporary real format precision, or use the precision specified by the instruction; (3) Bit 5--0 is the exception mask bit used to control coprocessor exception handling. These 6 bits correspond to the six anomalies that may occur in 80387, each of which can be masked separately. If a particular exception occurs and its corresponding mask bit is not set, then 80387 will notify the CPU of the exception and will cause the CPU to generate an exception int 16. However, if the corresponding mask bit is set, then 80387 will handle and correct the abnormal problem itself without notifying the CPU. This register can be read and written at any time. The specific meaning of each bit is shown in Figure 11-6.

B. Status Word

During operation, the 80387 sets the bits in the Status Word for the program to detect specific conditions. When an exception occurs, it lets the CPU determine the cause of the exception. Because all six exceptions of the coprocessor will cause the CPU to generate the same exception int16.

C. Tag Word

The Tag Word register contains eight 2-bit Tag fields for eight physical floating point data registers. These tag fields indicate that the corresponding physical register contains a valid, zero, or special floating point value, respectively, or is empty. Special values are those that are infinite, non-numeric, denormalized, or unsupported. The tag field can be used to detect the overflow of the accumulator stack. If the push operation decrements TOP and points to a non-empty register, an overflow on the stack occurs. If a pop operation attempts to read or pop an empty register, it will cause an underflow of the stack. Both the overflow and underflow of the stack will throw an invalid operation exception.

3. Error-Pointer Register

The error-pointer register is a four 32-bit 80387 register containing 80387 pointers to the last executed instruction and the data used, as shown in Figure 11-6. The first two registers FIP (FPU Instruction Pointer) and FCS (FPU Code Selector) are pointers to the two opcodes in the last executed instruction (ignoring the prefix code). FCS is the segment selector and opcode, and FIP is the intra-segment offset. The last two registers, FOO (FPU Operand Offset) and FOS (FPU Operand Selector), are the last pointers to the instruction memory operands. In FOS is the segment selector, and FOO is the intra-segment offset value. If the last executed coprocessor instruction does not contain a memory operand, the last two register values are useless. The instructions FLDENV, FSTENV, FNSTENV, FRSTOR, FSAVE, and FNSAVE are used to load and store the contents of these four registers. The first three instructions load or store a total of 28 bytes of content: control word, status word and feature word, and four error pointer registers. The control word, status word, and feature word are all operated in 32 bits, and the upper 16 bits are 0. The last three instructions are used to load or store all 108 bytes of register contents of the coprocessor.

4. 4. Floating-point instruction format

The simulation of the coprocessor is to analyze the specific floating-point instruction opcode and operands, and use the 80386 ordinary instructions to perform the corresponding emulation operations according to the structure of each instruction. There are more than 70 instructions in the math coprocessor 80387, which are divided into 5 categories, as shown in Table 11-3. The operation code of each instruction has 2 bytes, and the upper 5 bits of the first byte are binary 11011. The 5-bit value (0x1b or decimal 27) is exactly the ASCII code value of the character ESC (escape), so all math coprocessor instructions are visually referred to as ESC escape instructions. The same ESC bit can be ignored when simulating floating-point instructions, as long as the value of the lower 11 bits is determined.

Table 11-3 Floating-point instruction type

	1st Byte	2nd Byte	Option Fields

1	1 1 0 1 1	OPA		1	MOD		1	OPB		R/M			SIB	DISP
2	1 1 0 1 1	MF		OPA	MOD		OPB			R/M			SIB	DISP
3	1 1 0 1 1	d	P	OPA	1	1	OPB			ST(i)				
4	1 1 0 1 1	0	0	1	1	1	1	OP						
5	1 1 0 1 1	0	1	1	1	1	1	OP						
	15 -- 11	10	9	8	7	6	5	4	3	2	1	0		

The meanings of the fields in the table are as follows (refer to the 80X86 processor manual for the specific meaning and detailed description of these fields):

a) OP (Operation opcode) is the instruction opcode. In some instructions, it is divided into two parts: OPA and OPB.

b) MF (Memory Format) is a memory format: 00-32-bit real number; 01-32-bit integer; 10-64-bit real number; 11-64-bit integer.

c) P(Pop) indicates whether to perform a popup process after the operation: 0 - no need; 1 - pop up the stack after the operation.

d) d (destination) indicates the accumulator that saves the result of the operation: 0 - ST(0); 1 - ST(i).

e) MOD (Mode) and R/M (Register/Memory) are the operation mode field and the operand position field.

f) SIB (Scale Index Base) and DISP (Displacement) are optional follow-up fields with MOD and R/M field instructions.

In addition, all assembly language mnemonics for floating point instructions start with the letter F, for example: FADD, FLD, and so on. There are also some standard representations as follows:

a) FI All instructions for operating integer data start with FI, such as FIADD, FILD, etc.

b) FB All instructions for operating BCD type data start with FB, such as FBLD, FBST, etc.

c) FxxP All instructions that perform a pop operation are terminated with the letter P, such as FSTP, FADDP, etc.

d) FxxPP All instructions that perform two pop operations are terminated with the letter PP, such as FCOMPP, FUCOMPP, etc.;

e) e) FNxx Except for instructions starting with FN, all instructions will detect unmasked operation exceptions before execution. Instructions that start with FN do not detect arithmetic anomalies, such as FNINIT, FNSAVE, and so on.

11.2 math-emulation.c

11.2.1 Function

All functions in the math_emulate.c program can be divided into three categories: the first class is "device does not exist" exception handler interface function math_emulate(), this class has only one such function; The second type is the floating-point instruction emulation processing main function do_emu(), and this class only has this one function; In addition, all functions are simulation operation auxiliary functions, including functions in the other C programs.

In a PC that does not include a 80387 coprocessor chip, if the emulation flag EM = 1 is set in CR0 during kernel initialization, then the CPU will generate an exception int7 when the CPU encounters a floating-point instruction, and the math_emulate(long ___false) function at line 476 in this program is called during the exception processing.

In the math_emulate() function, if it is judged that the current process has not used the co-processing operation of the simulation, the 80387 control word, the status word and the feature word (Tag Word) of the simulation are initialized, and all 6 coprocessor exception mask bits in the control word are set, and the status word and the tag word are reset, and then call the simulation processing main function do_emu(). The parameters used when calling are the return address pointers that call the math_emulate() function during interrupt processing as follows. The info structure is actually a structure of some data in the stack that has been gradually pushed onto the stack since the CPU generated the interrupt int7, so it is basically the same as the distribution of data in the kernel stack when the system-call is executed. See line 11 of the include/linux/math_emu.h file and the beginning of kernel/sys_call.s.

```
11 struct info {
12         long ___math_ret;          // caller (int7) return address.
13         long ___orig_eip;          // a place where the original EIP is temporarily saved.
14         long ___edi;               // registers pushed on stack during int7 processing.
15         long ___esi;
16         long ___ebp;
17         long ___sys_call_ret;      // process system-call's return code.
18         long ___eax;               // below are the same as stacks of the system-call.
19         long ___ebx;
20         long ___ecx;
21         long ___edx;
22         long ___orig_eax;          // its -1 if not a system-call.
23         long ___fs;
24         long ___es;
25         long ___ds;
26         long ___eip;               // pushed by CPU automatically.
27         long ___cs;
28         long ___eflags;
29         long ___esp;
30         long ___ss;
31 };
```

The do_emu() function (line 52) first determines whether there is a coprocessor internal exception for the simulation based on the status word. If there is one, set the busy bit B (bit 15) of the status word, otherwise reset the busy bit B. Then, the two-byte floating-point instruction code that generates the coprocessor exception is obtained from the EIP field in the above info structure, and is used for performing software simulation operation processing after masking the ESC code (binary 11011) bit portion therein. For ease of processing, the function uses five switch statements to process five types of floating point instruction codes. For example, the first switch statement (line 75) is used to handle floating-point instructions that do not involve addressing memory operands, while the last two switch statements (lines 419, 432) are dedicated to handling instructions with operands related to memory. For the latter type of instruction, the basic flow of the process is to first obtain the effective address of the memory operand according to the addressing mode byte in the instruction code, and then read the corresponding data from the effective address (integer, Real number or BCD code value). The read value is then converted to a temporary real number format used by the 80387 internal processing. After the calculation is completed, the value of the temporary real number format is converted into the original data type, and finally saved to the user data area.

In addition, when simulating a specific floating point instruction, if the floating point instruction is found to be invalid, the program will immediately call the abandon execution function __math_abort(). This function will send the specified signal to the current process, and modify the stack pointer esp to point to the return address (___math_ret) of the math_emulate() function in the interrupt process, and immediately return to the interrupt handler.

11.2.2 **Code annotation**

Program 11-1 linux/kernel/math/math_emulate.c

```
/*
 * linux/kernel/math/math_emulate.c
 *
 * (C) 1991 Linus Torvalds
 */

/*
 * Limited emulation 27.12.91 - mostly loads/stores, which gcc wants
 * even for soft-float, unless you use bruce evans' patches. The patches
 * are great, but they have to be re-applied for every version, and the
 * library is different for soft-float and 80387. So emulation is more
 * practical, even though it's slower.
 *
 * 28.12.91 - loads/stores work, even BCD. I'll have to start thinking
 * about add/sub/mul/div. Urgel. I should find some good source, but I'll
 * just fake up something.
 *
 * 30.12.91 - add/sub/mul/div/com seem to work mostly. I should really
 * test every possible combination.
 */

/*
 * This file is full of ugly macros etc: one problem was that gcc simply
 * didn't want to make the structures as they should be: it has to try to
 * align them. Sickening code, but at least I've hidden the ugly things
 * in this one file: the other files don't need to know about these things.
 *
 * The other files also don't care about ST(x) etc - they just get addresses
 * to 80-bit temporary reals, and do with them as they please. I wanted to
 * hide most of the 387-specific things here.
 */

// <signal.h> Signal header file. Define signal symbol constants, signal structures, and
//     signal manipulation function prototypes.
// <linux/math_emu.h> Coprocessor emulation header file. A coprocessor data structure and
//     a floating point representation structure are defined.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
//     commonly used functions of the kernel.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined
//     for segment register operations.
#include <signal.h>
```

```

#define __ALIGNED_TEMP_REAL 1
#include <linux/math_emu.h>
#include <linux/kernel.h>
#include <asm/segment.h>

#define bswapw(x) __asm__("xchgb %%al,%%ah":"=a" (x):"" ((short)x))  // exchange 2 bytes.
#define ST(x) (*__st((x)))                                // get simulated ST(x) value.
#define PST(x) ((const temp_real *) __st((x)))            // get pointr to the simulated ST(x).

/*
 * We don't want these inlined - it gets too messy in the machine-code.
 */
// The following are simulation functions for floating-point instructions of the same name.
static void fpop(void);
static void fpush(void);
static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b);
static temp_real_unaligned * __st(int i);

// Perform floating-point instruction simulation.
// The function first checks if there is an unmasked exception flag set in the simulated I387
// structure status word register, and if so, sets the busy flag B in the status word.  Then
// save the instruction pointer, and take out the 2-byte floating-point instruction code code
// at the pointer EIP, and then analyze the code and process it according to its meaning.  For
// different code types, Linus uses several different switch blocks for simulation processing.
// The argument is a pointer to the info structure.
static void do_emu(struct info * info)
{
        unsigned short code;
        temp_real tmp;
        char * address;

// The function first checks if the exception flag is set in the status word register, and if
// so, sets the busy flag B (bit 15) in the status word, otherwise resets the B flag.  Then we
// save the original instruction pointer.  Then, let's see if the code that executes this function
// is user code.  If not (ie the caller's code segment selector is not equal to 0x0f), then there
// is code in the kernel that uses floating point instructions, but this is not allowed.  The
// kernel then shuts down after displaying the CS, EIP, and "Needs Mathematical Simulation in
// the Kernel" information for the floating-point instructions.
        if (I387.cwd & I387.swd & 0x3f)
                I387.swd |= 0x8000;              // sets the busy flag B.
        else
                I387.swd &= 0x7fff;              // clear the flag B.
        ORIG_EIP = EIP;                          // save the original EIP.
/* 0x0007 means user code space */
        if (CS != 0x000F) {                      // if not in user code space, STOP.
                printk("math_emulate: %04x:%08x\n\r",CS,EIP);
                panic("Math emulation needed in kernel");
        }
// Then we get the 2-byte floating-point instruction code at the pointer EIP.  Since the Intel
// CPU stores data in the "Little endien" style, the code fetched at this time is exactly reversed
// from the first and second bytes of the instruction.  So we need to swap the order of the two
// bytes in 'code'.  Then mask out the ESC bit in the first code byte (binary 11011).
```

```
    // Next, the floating point instruction pointer EIP is saved to the fip field in the TSS segment
    // i387 structure, and the CS is saved to the fcs field, and the slightly processed floating
    // point instruction code is placed in the upper 16 bits of the fcs field. These values are
    // saved so that the program can be processed like a real coprocessor in the event of a simulated
    // processor exception. Finally, let the EIP point to the subsequent floating point instruction
    // or operand.
68          code = get_fs_word((unsigned short *) EIP);  // get 2 bytes floating-point code.
69          bswapw(code);                                // exchange bytes.
70          code &= 0x7ff;                               // Mask the ESC part in the code.
71          I387.fip = EIP;                              // save EIP, code selector and code.
72          *(unsigned short *) &I387.fcs = CS;
73          *(1+(unsigned short *) &I387.fcs) = code;
74          EIP += 2;                                    // point to next instruction code.

    // Then we analyze the code and process it according to its meaning. For different code type
    // values, Linus uses several different switch blocks for processing.
    // (1) First, if the instruction opcode has a fixed code value (independent of the register
    // or the like), it is processed below.
    // The macro math_abort() is used to terminate the coprocessor emulation operation, defined
    // in the file linux/math_emu.h, line 52. The actual implementation code is at line 488 of the
    // program. See the description before line 50 of the linux/math_emu.h file.
75          switch (code) {
76                  case 0x1d0: /* fnop */    // like nop.
77                          return;
78                  case 0x1d1: case 0x1d2: case 0x1d3:  // invalid code, send signal and exit.
79                  case 0x1d4: case 0x1d5: case 0x1d6: case 0x1d7:
80                          math_abort(info,1<<(SIGILL-1));
81                  case 0x1e0:               // FCHS - Change the ST sign bit: ST = -ST.
82                          ST(0).exponent ^= 0x8000;
83                          return;
84                  case 0x1e1:               // FABS - get absolute value. ST = |ST|.
85                          ST(0).exponent &= 0x7fff;
86                          return;
87                  case 0x1e2: case 0x1e3:   // invalid code. send signal and exit.
88                          math_abort(info,1<<(SIGILL-1));
89                  case 0x1e4:               // FTST - Test TS and set Cn in status word.
90                          ftst(PST(0));
91                          return;
92                  case 0x1e5:               // FXAM - Check TS and modify Cn in status word.
93                          printk("fxam not implemented\n\r");
94                          math_abort(info,1<<(SIGILL-1));
95                  case 0x1e6: case 0x1e7:   // invalid code. send signal and exit.
96                          math_abort(info,1<<(SIGILL-1));
97                  case 0x1e8:               // FLD1 - Load constant 1.0 to accumulator ST.
98                          fpush();
99                          ST(0) = CONST1;
100                         return;
101                 case 0x1e9:               // FLDL2T - Load constant Log2(10) to ST.
102                         fpush();
103                         ST(0) = CONSTL2T;
104                         return;
105                 case 0x1ea:               // FLDL2E - Load constant Log2(e) to ST.
106                         fpush();
```

```
                        ST(0) = CONSTL2E;
                        return;
                case 0x1eb:               // FLDPI - Load constant Pi to ST.
                        fpush();
                        ST(0) = CONSTPI;
                        return;
                case 0x1ec:               // FLDLG2 - Load constant Log10(2) to ST.
                        fpush();
                        ST(0) = CONSTLG2;
                        return;
                case 0x1ed:               // FLDLN2 - Load constant Loge(2) to ST.
                        fpush();
                        ST(0) = CONSTLN2;
                        return;
                case 0x1ee:               // FLDZ - Load constant 0.0 to ST.
                        fpush();
                        ST(0) = CONSTZ;
                        return;
                case 0x1ef:               // invalid code. send signal and exit.
                        math_abort(info,1<<(SIGILL-1));
                case 0x1f0: case 0x1f1: case 0x1f2: case 0x1f3:
                case 0x1f4: case 0x1f5: case 0x1f6: case 0x1f7:
                case 0x1f8: case 0x1f9: case 0x1fa: case 0x1fb:
                case 0x1fc: case 0x1fd: case 0x1fe: case 0x1ff:
                        printk("%04x fxxx not implemented\n\r",code + 0xc800);
                        math_abort(info,1<<(SIGILL-1));
                case 0x2e9:               // FUCOMPP - no order comparison.
                        fucom(PST(1),PST(0));
                        fpop(); fpop();
                        return;
                case 0x3d0: case 0x3d1:   // FNOP - on 387. !! should be 0x3e0, 0x3e1.
                        return;
                case 0x3e2:               // FCLEX - Clears exception flag in status word.
                        I387.swd &= 0x7f00;
                        return;
                case 0x3e3:               // FINIT - Initializes the coprocessor.
                        I387.cwd = 0x037f;
                        I387.swd = 0x0000;
                        I387.twd = 0x0000;
                        return;
                case 0x3e4:               // FNOP - on 80387.
                        return;
                case 0x6d9:               // FCOMPP - compares ST(1) with ST, pops twice.
                        fcom(PST(1),PST(0));
                        fpop(); fpop();
                        return;
                case 0x7e0:               // FSTSW AX - Saves status word to AX register.
                        *(short *) &EAX = I387.swd;
                        return;
        }
```

```
// (2) Next, we process the instruction that the last 3 bits of the 2nd byte are REG. That is,
// the code in the form of "11011,XXXXXXXX,REG", and the prefix "11011" alread be cleared.
// The macro real_to_real(a, b) is used for assignment between two temporary reals, defined
// in file linux/math_emu.h, line 72.
        switch (code >> 3) {
                case 0x18:                          // FADD ST, ST(i)
                        fadd(PST(0),PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 0x19:                          // FMUL ST, ST(i)
                        fmul(PST(0),PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 0x1a:                          // FCOM ST(i)
                        fcom(PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 0x1b:                          // FCOMP ST(i)
                        fcom(PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(0));
                        fpop();
                        return;
                case 0x1c:                          // FSUB ST, ST(i)
                        real_to_real(&ST(code & 7),&tmp);
                        tmp.exponent ^= 0x8000;
                        fadd(PST(0),&tmp,&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 0x1d:                          // FSUBR ST, ST(i)
                        ST(0).exponent ^= 0x8000;
                        fadd(PST(0),PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 0x1e:                          // FDIV ST, ST(i)
                        fdiv(PST(0),PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 0x1f:                          // FDIVR ST, ST(i)
                        fdiv(PST(code & 7),PST(0),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 0x38:                          // FLD ST(i)
                        fpush();
                        ST(0) = ST((code & 7)+1);
                        return;
                case 0x39:                          // FXCH ST(i)
                        fxchg(&ST(0),&ST(code & 7));
                        return;
                case 0x3b:                          // FSTP ST(i)
                        ST(code & 7) = ST(0);
                        fpop();
                        return;
                case 0x98:                          // FADD ST(i), ST
                        fadd(PST(0),PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        return;
                case 0x99:                // FMUL ST(i), ST
                        fmul(PST(0),PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        return;
                case 0x9a:                // FCOM ST(i)
                        fcom(PST(code & 7),PST(0));
                        return;
                case 0x9b:                // FCOMP ST(i)
                        fcom(PST(code & 7),PST(0));
                        fpop();
                        return;
                case 0x9c:                // FSUBR ST(i), ST
                        ST(code & 7).exponent ^= 0x8000;
                        fadd(PST(0),PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        return;
                case 0x9d:                // FSUB ST(i), ST
                        real_to_real(&ST(0),&tmp);
                        tmp.exponent ^= 0x8000;
                        fadd(PST(code & 7),&tmp,&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        return;
                case 0x9e:                // FDIVR ST(i), ST
                        fdiv(PST(0),PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        return;
                case 0x9f:                // FDIV ST(i), ST
                        fdiv(PST(code & 7),PST(0),&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        return;
                case 0xb8:                // FFREE ST(i)
                        printk("ffree not implemented\n\r");
                        math_abort(info,1<<(SIGILL-1));
                case 0xb9:                // FXCH ST(i)
                        fxchg(&ST(0),&ST(code & 7));
                        return;
                case 0xba:                // FST ST(i)
                        ST(code & 7) = ST(0);
                        return;
                case 0xbb:                // FSTP ST(i)
                        ST(code & 7) = ST(0);
                        fpop();
                        return;
                case 0xbc:                // FUCOM ST(i)
                        fucom(PST(code & 7),PST(0));
                        return;
                case 0xbd:                // FUCOMP ST(i)
                        fucom(PST(code & 7),PST(0));
                        fpop();
                        return;
                case 0xd8:                // FADDP ST(i), ST
                        fadd(PST(code & 7),PST(0),&tmp);
```

```
                        real_to_real(&tmp,&ST(code & 7));
                        fpop();
                        return;
                case 0xd9:                // FMULP ST(i), ST
                        fmul(PST(code & 7),PST(0),&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        fpop();
                        return;
                case 0xda:                // FCOMP ST(i)
                        fcom(PST(code & 7),PST(0));
                        fpop();
                        return;
                case 0xdc:                // FSUBRP ST(i), ST
                        ST(code & 7).exponent ^= 0x8000;
                        fadd(PST(0),PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        fpop();
                        return;
                case 0xdd:                // FSUBP ST(i), ST
                        real_to_real(&ST(0),&tmp);
                        tmp.exponent ^= 0x8000;
                        fadd(PST(code & 7),&tmp,&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        fpop();
                        return;
                case 0xde:                // FDIVRP ST(i), ST
                        fdiv(PST(0),PST(code & 7),&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        fpop();
                        return;
                case 0xdf:                // FDIVP ST(i), ST
                        fdiv(PST(code & 7),PST(0),&tmp);
                        real_to_real(&tmp,&ST(code & 7));
                        fpop();
                        return;
                case 0xf8:                // FFREE ST(i)
                        printk("ffree not implemented\n\r");
                        math_abort(info,1<<(SIGILL-1));
                        fpop();
                        return;
                case 0xf9:                // FXCH ST(i)
                        fxchg(&ST(0),&ST(code & 7));
                        return;
                case 0xfa:                // FSTP ST(i)
                case 0xfb:                // FSTP ST(i)
                        ST(code & 7) = ST(0);
                        fpop();
                        return;
        }
```

// (3) Next, we process the code in the form of the second byte 7--6 is MOD, the bit 2--0 is
// R/M, that is, "11011, XXX, MOD, XXX, R/M". The MOD will be processed in each subroutine,
// so first let the code AND 0xe7 (ie 0b11100111) to mask out the MOD.

```
        switch ((code>>3) & 0xe7) {
                case 0x22:                // FST - Saves single precision real (short real)
                        put_short_real(PST(0),info,code);
                        return;
                case 0x23:                // FSTP - Saves single precision real (short real)
                        put_short_real(PST(0),info,code);
                        fpop();
                        return;
                case 0x24:                // FLDENV - Load status and control registers, etc.
                        address = ea(info,code);   // get efficient address.
                        for (code = 0 ; code < 7 ; code++) {
                                ((long *) & I387)[code] =
                                   get_fs_long((unsigned long *) address);
                                address += 4;
                        }
                        return;
                case 0x25:                // FLDCW - Load control word.
                        address = ea(info,code);
                        *(unsigned short *) &I387.cwd =
                                get_fs_word((unsigned short *) address);
                        return;
                case 0x26:            // FSTENV - Store status and control registers, etc.
                        address = ea(info,code);
                        verify_area(address,28);
                        for (code = 0 ; code < 7 ; code++) {
                                put_fs_long( ((long *) & I387)[code],
                                        (unsigned long *) address);
                                address += 4;
                        }
                        return;
                case 0x27:                // FSTCW - Store control word.
                        address = ea(info,code);
                        verify_area(address,2);
                        put_fs_word(I387.cwd,(short *) address);
                        return;
                case 0x62:                // FIST - Stores short integer.
                        put_long_int(PST(0),info,code);
                        return;
                case 0x63:                // FISTP - Stores short integer.
                        put_long_int(PST(0),info,code);
                        fpop();
                        return;
                case 0x65:                // FLD - Load an extended (temporary) real number.
                        fpush();
                        get_temp_real(&tmp,info,code);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 0x67:                // FSTP - Store the temporary real.
                        put_temp_real(PST(0),info,code);
                        fpop();
                        return;
                case 0xa2:                // FST - Stores double precision (long) real.
                        put_long_real(PST(0),info,code);
                        return;
                case 0xa3:                      // FSTP - Stores double precision (long) real.
                        put_long_real(PST(0),info,code);
                        fpop();
                        return;
                case 0xa4:                  // FRSTOR - Restores all 108 bytes register contents.
                        address = ea(info,code);
                        for (code = 0 ; code < 27 ; code++) {
                                ((long *) & I387)[code] =
                                   get_fs_long((unsigned long *) address);
                                address += 4;
                        }
                        return;
                case 0xa6:                  // FSAVE - Saves all 108 bytes register contents.
                        address = ea(info,code);
                        verify_area(address,108);
                        for (code = 0 ; code < 27 ; code++) {
                                put_fs_long( ((long *) & I387)[code],
                                        (unsigned long *) address);
                                address += 4;
                        }
                        I387.cwd = 0x037f;
                        I387.swd = 0x0000;
                        I387.twd = 0x0000;
                        return;
                case 0xa7:                      // FSTSW - Store status word.
                        address = ea(info,code);
                        verify_area(address,2);
                        put_fs_word(I387.swd,(short *) address);
                        return;
                case 0xe2:                      // FIST - Stores short integer.
                        put_short_int(PST(0),info,code);
                        return;
                case 0xe3:                      // FISTP - Stores short integer.
                        put_short_int(PST(0),info,code);
                        fpop();
                        return;
                case 0xe4:                      // FBLD - Loads the number of BCD type.
                        fpush();
                        get_BCD(&tmp,info,code);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 0xe5:                      // FILD - Loads a long integer.
                        fpush();
                        get_longlong_int(&tmp,info,code);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 0xe6:                      // FBSTP - Stores number of BCD type.
                        put_BCD(PST(0),info,code);
                        fpop();
                        return;
                case 0xe7:                      // BISTP - Stores a long integer.
                        put_longlong_int(PST(0),info,code);
                        fpop();
                        return;
        }
// (4) The second type of floating point instructions are processed below. First, the number
// of the specified type is taken according to the MF of the bit 10--9 of the instruction code,
// and then separately processed according to the combined value of the OPA and the OPB. That
// is, the instruction code in the form of "11011, MF, 000, XXX, R/M" is processed.
        switch (code >> 9) {
                case 0:                   // MF = 00, short real (32-bit real).
                        get_short_real(&tmp,info,code);
                        break;
                case 1:                   // MF = 01, short integer (32-bit integer).
                        get_long_int(&tmp,info,code);
                        break;
                case 2:                   // MF = 10, long real (64-bit real).
                        get_long_real(&tmp,info,code);
                        break;
                case 4:           // MF = 11, a long integer (64-bit). should be 'case 3'!
                        get_short_int(&tmp,info,code);
        }
// (5) Process the OPB code in the second byte of the floating point instruction.
        switch ((code>>3) & 0x27) {
                case 0:                           // FADD
                        fadd(&tmp,PST(0),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 1:                           // FMUL
                        fmul(&tmp,PST(0),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 2:                           // FCOM
                        fcom(&tmp,PST(0));
                        return;
                case 3:                           // FCOMP
                        fcom(&tmp,PST(0));
                        fpop();
                        return;
                case 4:                           // FSUB
                        tmp.exponent ^= 0x8000;
                        fadd(&tmp,PST(0),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 5:                           // FSUBR
                        ST(0).exponent ^= 0x8000;
                        fadd(&tmp,PST(0),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 6:                           // FDIV
                        fdiv(PST(0),&tmp,&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
                case 7:                           // FDIVR
                        fdiv(&tmp,PST(0),&tmp);
                        real_to_real(&tmp,&ST(0));
                        return;
        }
// Process the instruction code of the form "11011, XX, 1, XX, 000, R/M".
        if ((code & 0x138) == 0x100) {            // FLD, FILD
                        fpush();
                        real_to_real(&tmp,&ST(0));
                        return;
        }
// The rest are invalid instructions.
        printk("Unknown math-insns: %04x:%08x %04x\n\r",CS,EIP,code);
        math_abort(info,1<<(SIGFPE-1));
}

// The 80387 emulation interface function called in exception Interrupts int7.
// If the current process has not used the coprocessor, set the use of the coprocessor flag
// used_math, and then initialize the 80387 control word, status word and tag word. Finally,
// use the return address when int 7 invokes this function as a parameter to call the floating
// point instruction emulation function do_emu(). The parameter ___false is _orig_eip.
void math_emulate(long ___false)
{
        if (!current->used_math) {
                current->used_math = 1;
                I387.cwd = 0x037f;
                I387.swd = 0x0000;
                I387.twd = 0x0000;
        }
/* &___false points to info->___orig_eip, so subtract 1 to get info */
        do_emu((struct info *) ((&___false) - 1));
}

// Terminate the simulation.
// When an invalid instruction code or an unimplemented instruction is processed, the function
// first restores the original EIP of the program and then sends a specified signal to the current
// process. Finally, the stack pointer is pointed to the return address when int7 invokes this
// function, and directly returns to the interrupt. This function will be used in the macro
// math_abort(). See the linux/math_emu.h file at line 50 for more instructions.
void __math_abort(struct info * info, unsigned int signal)
{
        EIP = ORIG_EIP;
        current->signal |= signal;
        __asm__("movl %0,%%esp ; ret"::"g" ((long) info));
}

// Accumulator stack pop-up operation.
// Increase the TOP field of the status word by 1 and modulo 7 .
static void fpop(void)
{
        unsigned long tmp;

        tmp = I387.swd & 0xffffc7ff;
        I387.swd += 0x00000800;
        I387.swd &= 0x00003800;
        I387.swd |= tmp;
}

// Accumulator stack Push operation.
// The TOP field of status word is decremented by 1 (ie, 7 is added) and modulo 7.
static void fpush(void)
{
        unsigned long tmp;

        tmp = I387.swd & 0xffffc7ff;
        I387.swd += 0x00003800;
        I387.swd &= 0x00003800;
        I387.swd |= tmp;
}

// Swap the values of the two accumulator registers.
static void fxchg(temp_real_unaligned * a, temp_real_unaligned * b)
{
        temp_real_unaligned c;

        c = *a;
        *a = *b;
        *b = c;
}

// Get the ST(i) memory pointer in the I387 structure.
// Take the TOP field value in the status word, add the specified physical data register number
// and modulo 7, and finally return the pointer corresponding to ST(i).
static temp_real_unaligned * __st(int i)
{
        i += I387.swd >> 11;            // Get the TOP field in the status word.
        i &= 7;
        return (temp_real_unaligned *) (i*10 + (char *)(I387.st_space));
}

```

11.3 error.c

11.3.1 Function

When the coprocessor detects that it has an error, it will notify the CPU via the 80387 chip ERROR pin. The error.c program is used to process the error signal sent by the coprocessor, mainly to execute the math_error() function.

11.3.2 Code annotation

Program 11-2 linux/kernel/math/error.c

```
/*
 * linux/kernel/math/error.c
 *
 * (C) 1991 Linus Torvalds
 */

// <signal.h> Signal header file. Define signal symbol constants, signal structures, and
//     signal manipulation function prototypes.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the
//     data of the initial task 0, and some embedded assembly function macro statements
//     about the descriptor parameter settings and acquisition.
#include <signal.h>

#include <linux/sched.h>

// Coprocessor error handling function called in int16.
// The following code is used to handle the error sent by the coprocessor. It causes 80387 to
// clear all exception flags and busy bits in the status word. If the last task used coprocessor,
// then set the coprocessor error signal flag. After returning, the function will jump to the
// system-call interrupt return place of ret_from_sys_call to continue execution.
void math_error(void)
{
        __asm__("fnclex");          // clear all exception flags and busy bit in status word.
        if (last_task_used_math)    // if used coprocessor, set signal flag.
                last_task_used_math->signal |= 1<<(SIGFPE-1);
}

```

11.4 ea.c

11.4.1 Function

The ea.c program is used to calculate the effective address used by the operand when simulating floating-point instructions. In order to analyze the effective address information in an instruction, we must have an understanding of the instruction encoding method. The general encoding format for Intel processor instructions is shown in Figure 11-7.

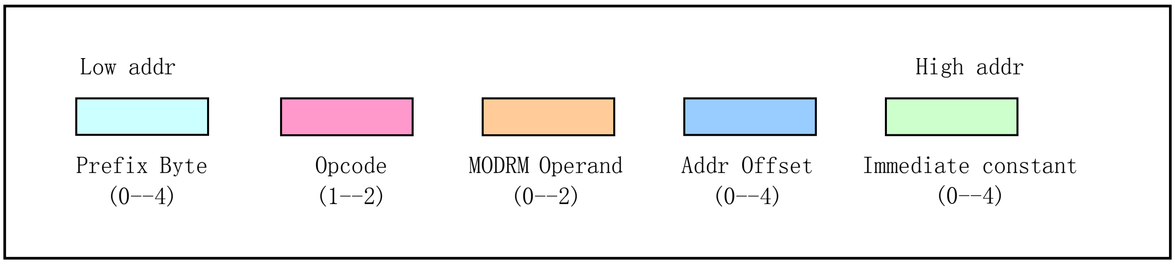
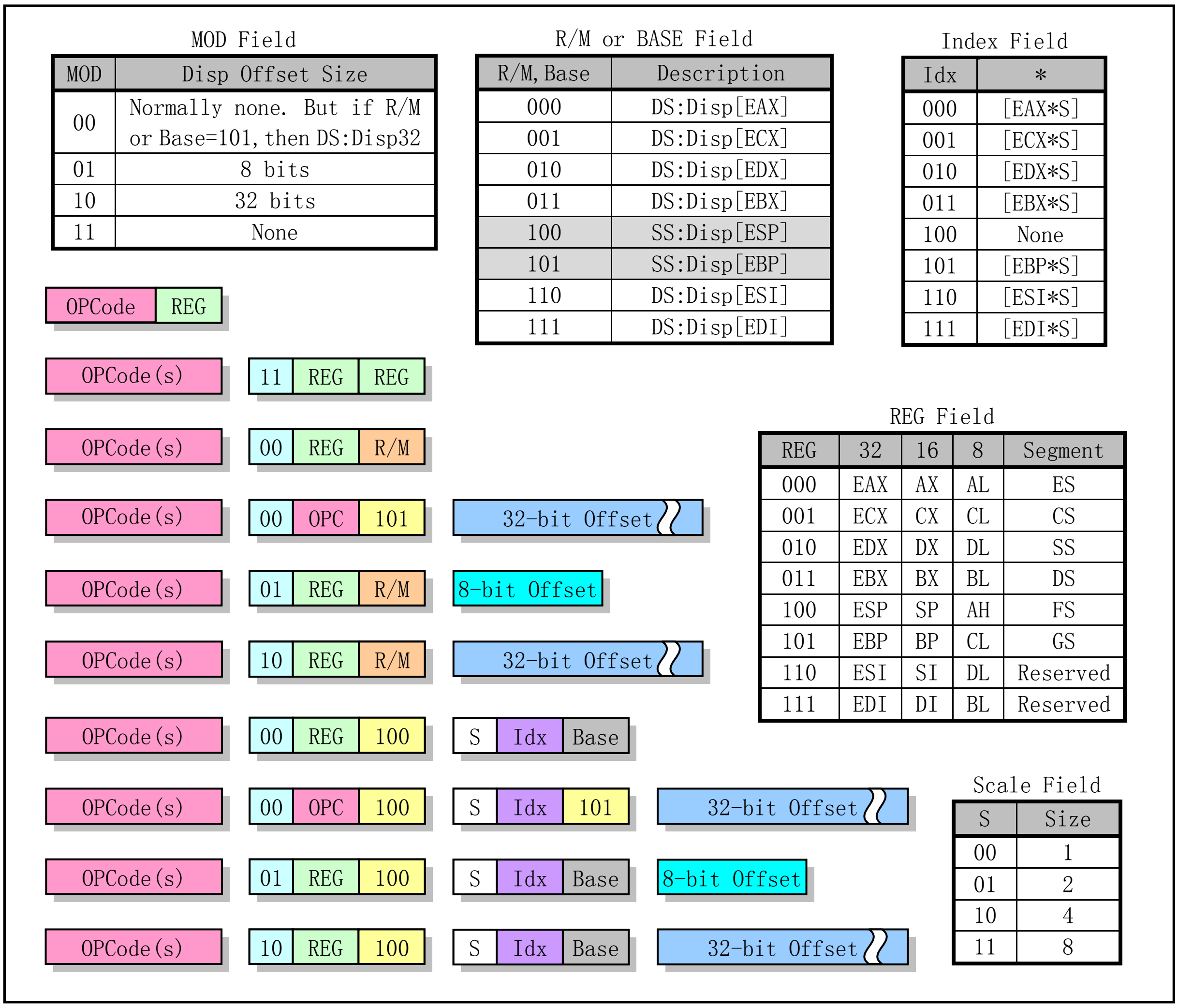


Figure 11-7 General instruction encoding format

As can be seen from the figure, each instruction can have up to 5 fields. The prefix field can consist of 0 to 4 bytes and is used to modify the next instruction. The opcode field is the main field that indicates the operation of the instruction. Each instruction must have at least 1 byte of operation code. If necessary, the instruction opcode field indicates whether or not to follow a MODRM operand indicator, which is used to explicitly indicate the type and number of operands. For memory operands, the address displacement field is used to give the offset of the operand. The MOD subfield of the MODRM field indicates whether the instruction contains an address offset field and its length. The immediate constant field gives the operand required by the instruction opcode, which is the simplest operand given in the instruction. See the Intel manual for a detailed description of the immediate operands, register operands, and memory operand encoding.

The encoding format of all instructions is shown schematically in Figure 11-8. The figure shows the instruction formats for 10 different encoding methods. Wherein, the symbol OPCode or OPC is an operation code; REG is a register field; the R/M field is used to indicate a register as an operand, or is combined with a MOD field to specify an addressing mode. Some MODR/M encodings require a second addressing byte (called SIB byte) to indicate the addressing mode. This byte has three subfield contents: (1) Scale - scale (S) field specifies the scale factor; (2) index - index (The Idx) field specifies the index register; (3) the Base - Base field specifies the base address register.



MOD Field

MOD	Disp Offset Size
00	Normally none. But if R/M or Base=101, then DS:Disp32
01	8 bits
10	32 bits
11	None

R/M or BASE Field

R/M, Base	Description
000	DS:Disp[EAX]
001	DS:Disp[ECX]
010	DS:Disp[EDX]
011	DS:Disp[EBX]
100	SS:Disp[ESP]
101	SS:Disp[EBP]
110	DS:Disp[ESI]
111	DS:Disp[EDI]

Index Field

Idx	*
000	[EAX*S]
001	[ECX*S]
010	[EDX*S]
011	[EBX*S]
100	None
101	[EBP*S]
110	[ESI*S]
111	[EDI*S]

REG Field

REG	32	16	8	Segment
000	EAX	AX	AL	ES
001	ECX	CX	CL	CS
010	EDX	DX	DL	SS
011	EBX	BX	BL	DS
100	ESP	SP	AH	FS
101	EBP	BP	CL	GS
110	ESI	SI	DL	Reserved
111	EDI	DI	BL	Reserved

Scale Field

S	Size
00	1
01	2
10	4
11	8

OPCode REG
OPCode(s) 11 REG REG
OPCode(s) 00 REG R/M
OPCode(s) 00 OPC 101 32-bit Offset
OPCode(s) 01 REG R/M 8-bit Offset
OPCode(s) 10 REG R/M 32-bit Offset
OPCode(s) 00 REG 100 S Idx Base
OPCode(s) 00 OPC 100 S Idx 101 32-bit Offset
OPCode(s) 01 REG 100 S Idx Base 8-bit Offset
OPCode(s) 10 REG 100 S Idx Base 32-bit Offset

Figure 11-8 Instruction encoding and addressing format summary

The ea() function in this program is used to calculate the effective address based on the addressing mode byte in the instruction. It first takes the MOD field and the R/M field value in the instruction code. If

MOD=0b11, it means a single-byte instruction with no offset field. If the R/M field = 0b100 and MOD is not 0b11, it means 2-byte address mode addressing. At this time, the function sib() which processes the second operand instruction byte SIB (Scale, Index, Base) is called to obtain the offset value and return. If the R/M field is 0b101 and MOD is 0, it indicates a single-byte address mode encoding followed by a 32-byte offset value. For the rest of the case, it is processed according to the MOD.

11.4.2 Code annotation

Program 11-3 linux/kernel/math/ea.c

```
/*
 * linux/kernel/math/ea.c
 *
 * (C) 1991 Linus Torvalds
 */

/*
 * Calculate the effective address.
 */

// <stddef.h> The standard definition header file. NULL, offsetof(TYPE, MEMBER) are defined.
// <linux/math_emu.h> Coprocessor emulation header file. A coprocessor data structure and
//     a floating point representation structure are defined.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined
//     for segment register operations.
#include <stddef.h>

#include <linux/math_emu.h>
#include <asm/segment.h>

// The offset position of each register in the info structure. offsetof() is used to find the
// offset position of the specified field in the structure. See the include/stddef.h file.
static int __regoffset[] = {
        offsetof(struct info,___eax),
        offsetof(struct info,___ecx),
        offsetof(struct info,___edx),
        offsetof(struct info,___ebx),
        offsetof(struct info,___esp),
        offsetof(struct info,___ebp),
        offsetof(struct info,___esi),
        offsetof(struct info,___edi)
};

// Get the contents of the register at the specified position in the info structure.
#define REG(x) (*(long *)(__regoffset[(x)]+(char *) info))

// Get the value of the second operand indication byte SIB (Scale, Index, Base).
static char * sib(struct info * info, int mod)
{
        unsigned char ss,index,base;
        long offset = 0;

// The SIB byte is first taken from the user code segment, then the field bit values are taken.
```

```
        base = get_fs_byte((char *) EIP);
        EIP++;
        ss = base >> 6;                         // scale size.
        index = (base >> 3) & 7;
        base &= 7;
// If the index code is 0b100, it means there is no index offset value. Otherwise index offset
// value offset = register content * scale factor.
        if (index == 4)
                offset = 0;
        else
                offset = REG(index);
        offset <<= ss;
// If the MOD in the previous MODRM byte is not zero, or if Base is not equal to 0b101, it means
// that there is an offset value in the register specified by base. Therefore, the offset needs
// to be added to the contents of the base corresponding register. If MOD=1, the offset value
// is 1 byte (8-bit). Otherwise, if MOD=2, or base=0b101, the offset value is 4 bytes.
        if (mod || base != 5)
                offset += REG(base);
        if (mod == 1) {
                offset += (signed char) get_fs_byte((char *) EIP);
                EIP++;
        } else if (mod == 2 || base == 5) {
                offset += (signed) get_fs_long((unsigned long *) EIP);
                EIP += 4;
        }
// Finally save and return the offset value.
        I387.foo = offset;
        I387.fos = 0x17;
        return (char *) offset;
}

// Calcating the effective address base on the addressing mode byte in the instruction code.
char * ea(struct info * info, unsigned short code)
{
        unsigned char mod,rm;
        long * tmp = &EAX;
        int offset = 0;

// First take the MOD field and the R/M field value in the instruction code. If MOD=0b11, it
// means a single-byte instruction with no offset field. If the R/M field = 0b100 and MOD is
// not 0b11, it means 2-byte address mode addressing, so call sib() to find the offset value
// and return.
        mod = (code >> 6) & 3;                  // MOD field.
        rm = code & 7;                          // R/M field.
        if (rm == 4 && mod != 3)
                return sib(info,mod);
// If the R/M field is 0b101 and MOD is 0, it indicates a single-byte address mode encoding
// followed by a 32-byte offset value. Then take the 4-byte offset value in the user code, save
// it and return it.
        if (rm == 5 && !mod) {
                offset = get_fs_long((unsigned long *) EIP);
                EIP += 4;
                I387.foo = offset;
                I387.fos = 0x17;
                return (char *) offset;
        }
// For the rest of the case, it is processed according to the MOD. First, the value of the contents
// of the corresponding register of the R/M code is taken out as the pointer tmp. For MOD=0,
// there is no offset value. For MOD=1, the code is followed by a 1-byte offset value. For MOD=2,
// there is a 4-byte offset after the code. Finally save and return the valid address value.
        tmp = & REG(rm);
        switch (mod) {
                case 0: offset = 0; break;
                case 1:
                        offset = (signed char) get_fs_byte((char *) EIP);
                        EIP++;
                        break;
                case 2:
                        offset = (signed) get_fs_long((unsigned long *) EIP);
                        EIP += 4;
                        break;
                case 3:
                        math_abort(info,1<<(SIGILL-1));
        }
        I387.foo = offset;
        I387.fos = 0x17;
        return offset + (char *) *tmp;
}

```

11.5 convert.c

11.5.1 Function

The convert.c program contains data type conversion functions during the 80387 emulation operation. Before performing the simulation calculation, we need to convert the integer or real type provided by the user into the temporary real number format used in the simulation, and then convert back to the original format after the simulation is completed. For example, Figure 11-9 shows a schematic diagram of a conversion of a short real into a temporary real number format.

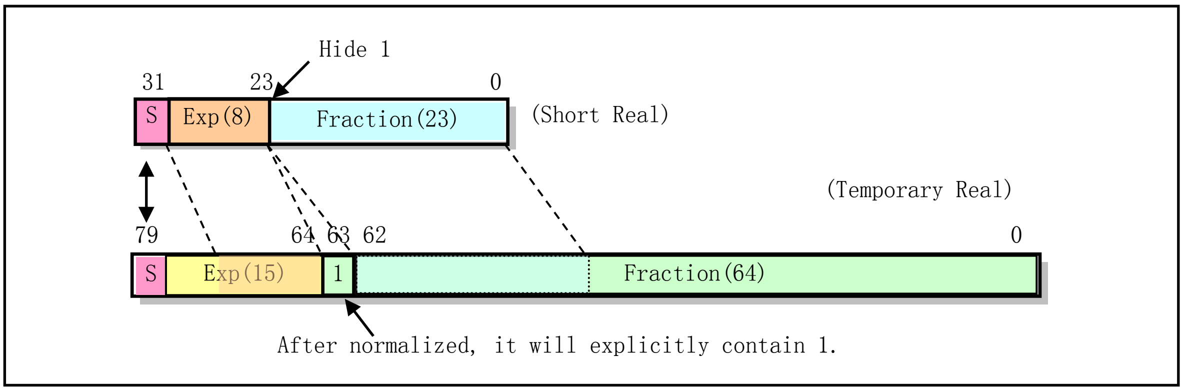


Figure 11-9 Conversion diagram of short real to temporary real format

11.5.2 Code annotation

Program 11-4 linux/kernel/math/convert.c

```
/*
 * linux/kernel/math/convert.c
 *
 * (C) 1991 Linus Torvalds
 */

#include <linux/math_emu.h>

/*
 * NOTE!!! There is some "non-obvious" optimisations in the temp_to_long
 * and temp_to_short conversion routines: don't touch them if you don't
 * know what's going on. They are the adding of one in the rounding: the
 * overflow bit is also used for adding one into the exponent. Thus it
 * looks like the overflow would be incorrectly handled, but due to the
 * way the IEEE numbers work, things are correct.
 *
 * There is no checking for total overflow in the conversions, though (ie
 * if the temp-real number simply won't fit in a short- or long-real.)
 */

// Convert short real to temporary real number.
// The short real number is 32 bits long, its significant number (mantissa) is 23 bits long,
// the exponent is 8 bits, and there is 1 sign bit.
void short_to_temp(const short_real * a, temp_real * b)
{
// First we handle the case where the short real number is 0. If it is 0, the significand
// the temporary real number b is set to 0. The sign bit of the temporary real number is then
// set according to the short real sign bit, ie the most significant bit of exponent.
        if (!(*a & 0x7fffffff)) {
                b->a = b->b = 0;                // set significand of the temp real = 0.
                if (*a)
                        b->exponent = 0x8000;   // set sign bit.
                else
                        b->exponent = 0;
                return;
        }
// For a general short real, first determine the exponent value corresponding to the temporary
// real number. Here we need to use the concept of the integer number biased representation
// method, see section 11.1. The biase number of the short real exponent is 127, while the biase
// of the temporary real exponent is 16383. Therefore, after extracting the exponent value in
// the short real, it is necessary to change the biase value to 16383. This forms the exponent
// value in the temporary real format. Also, if the short real number is negative, you need
// to set the sign bit of the temporary real (bit 79). Next set the mantissa. The method is
// to shift the short real number to the left by 8 bits, and let the 23 most significant digit
// of the mantissa be at the bit 62 of the temporary real number. Bit 63 of the temporary real
// mantissa needs to be set to 1, which requires an OR 0x80000000 operation. Finally, the low
// 32-bit significant number of the temporary real is cleared.
```

```
        b->exponent = ((*a>>23) & 0xff)-127+16383;   // change biased value to 16383.
        if (*a<0)
                b->exponent |= 0x8000;               // add negative sign if need.
        b->b = (*a<<8) | 0x80000000;                 // put mantissa, set 63th bit.
        b->a = 0;
}

// Convert long real to temporary real.
// The method is exactly the same as short_to_temp().However, the long real is 64 bits long,
// its significant number (mantissa) is 52 bits long, the exponent is 11 bits, and there is
// 1 sign bit. In addition, the long real exponent biased value is 1023.
void long_to_temp(const long_real * a, temp_real * b)
{
        if (!a->a && !(a->b & 0x7fffffff)) {
                b->a = b->b = 0;
                if (a->b)
                        b->exponent = 0x8000;        // set sign bit.
                else
                        b->exponent = 0;
                return;
        }
        b->exponent = ((a->b >> 20) & 0x7ff)-1023+16383;  // change biased value to 16383.
        if (a->b<0)
                b->exponent |= 0x8000;
        b->b = 0x80000000 | (a->b<<11) | (((unsigned long)a->a)>>21);    // place 1.
        b->a = a->a<<11;
}

// Convert temporary real to short real.
// The procedure is the opposite of short_to_temp() but we requires handling precision and
// rounding issues.
void temp_to_short(const temp_real * a, short_real * b)
{
// If the exponent part is 0, the short real number is set to -0 or 0 depending on whether or
// not there is a sign bit. refer to Table 11-2.
        if (!(a->exponent & 0x7fff)) {
                *b = (a->exponent)?0x80000000:0;
                return;
        }
// First, the exponent portion is processed, that is, the amount of the exponent bias (16383)
// is replaced by the biased amount 127 of the short real number, and the sign bit is set if
// it is a negative number.
        *b = ((((long) a->exponent)-16383+127) << 23) & 0x7f800000;
        if (a->exponent < 0)                       // set sign if negative.
                *b |= 0x80000000;
// Then get the highest 23 bits of the signifcand from the temporary real number and perform
// the rounding operation according to the rounding setting in the control word.
        *b |= (a->b >> 8) & 0x007fffff;            // get higher 23bits of the temp real.
        switch (ROUNDING) {
                case ROUND_NEAREST:
                        if ((a->b & 0xff) > 0x80)
                                ++*b;
                        break;
                case ROUND_DOWN:
                        if ((a->exponent & 0x8000) && (a->b & 0xff))
                                ++*b;
                        break;
                case ROUND_UP:
                        if (!(a->exponent & 0x8000) && (a->b & 0xff))
                                ++*b;
                        break;
        }
}

// Convert temporary real to long real number.
// The long real is 64 bits long, its significant number (mantissa) is 52 bits long, the exponent
// is 11 bits, and there is 1 sign bit, and the exponent biased value is 1023.
void temp_to_long(const temp_real * a, long_real * b)
{
        if (!(a->exponent & 0x7fff)) {
                b->a = 0;
                b->b = (a->exponent)?0x80000000:0;
                return;
        }
        b->b = (((0x7fff & (long) a->exponent)-16383+1023) << 20) & 0x7ff00000;
        if (a->exponent < 0)
                b->b |= 0x80000000;
        b->b |= (a->b >> 11) & 0x000fffff;
        b->a = a->b << 21;
        b->a |= (a->a >> 11) & 0x001fffff;
        switch (ROUNDING) {
                case ROUND_NEAREST:
                        if ((a->a & 0x7ff) > 0x400)
                                __asm__("addl $1,%0 ; adcl $0,%1"
                                        :"=r" (b->a),"=r" (b->b)
                                        :"" (b->a),"1" (b->b));
                        break;
                case ROUND_DOWN:
                        if ((a->exponent & 0x8000) && (a->b & 0xff))
                                __asm__("addl $1,%0 ; adcl $0,%1"
                                        :"=r" (b->a),"=r" (b->b)
                                        :"" (b->a),"1" (b->b));
                        break;
                case ROUND_UP:
                        if (!(a->exponent & 0x8000) && (a->b & 0xff))
                                __asm__("addl $1,%0 ; adcl $0,%1"
                                        :"=r" (b->a),"=r" (b->b)
                                        :"" (b->a),"1" (b->b));
                        break;
        }
}

// Convert temporary real to a temporary integer format.
// Temporary integers are also represented by 10 bytes. The lower 8 bytes are unsigned integer
// values, and the upper 2 bytes represent exponent value and sign bit. If the highest significant
// byte of the upper 2 bytes is 1, it indicates a negative number; if the bit is 0, it indicates
// a positive number.
void real_to_int(const temp_real * a, temp_int * b)
{
        int shift =  16383 + 63 - (a->exponent & 0x7fff);
        unsigned long underflow;

        b->a = b->b = underflow = 0;
        b->sign = (a->exponent < 0);
        if (shift < 0) {
                set_OE();
                return;
        }
        if (shift < 32) {
                b->b = a->b; b->a = a->a;
        } else if (shift < 64) {
                b->a = a->b; underflow = a->a;
                shift -= 32;
        } else if (shift < 96) {
                underflow = a->b;
                shift -= 64;
        } else
                return;
        __asm__("shrdl %2,%1,%0"                   // 32-bit shift right.
                :"=r" (underflow),"=r" (b->a)
                :"c" ((char) shift),"" (underflow),"1" (b->a));
        __asm__("shrdl %2,%1,%0"
                :"=r" (b->a),"=r" (b->b)
                :"c" ((char) shift),"" (b->a),"1" (b->b));
        __asm__("shrl %1,%0"
                :"=r" (b->b)
                :"c" ((char) shift),"" (b->b));
        switch (ROUNDING) {
                case ROUND_NEAREST:
                        __asm__("addl %4,%5 ; adcl $0,%0 ; adcl $0,%1"
                                :"=r" (b->a),"=r" (b->b)
                                :"" (b->a),"1" (b->b)
                                ,"r" (0x7fffffff + (b->a & 1))
                                ,"m" (*&underflow));
                        break;
                case ROUND_UP:
                        if (!b->sign && underflow)
                                __asm__("addl $1,%0 ; adcl $0,%1"
                                        :"=r" (b->a),"=r" (b->b)
                                        :"" (b->a),"1" (b->b));
                        break;
                case ROUND_DOWN:
                        if (b->sign && underflow)
                                __asm__("addl $1,%0 ; adcl $0,%1"
                                        :"=r" (b->a),"=r" (b->b)
                                        :"" (b->a),"1" (b->b));
                        break;
        }
}
```

```
// Convert a temporary integer to a temporary real format.
void int_to_real(const temp_int * a, temp_real * b)
{
// Since the original value is an integer, when converting to a temporary real, the exponent
// adds 63 in addition to the biased amount of 16383. This means that the significand needs
// to be multiplied by 2^63, which means that the significand are also integer values.
    b->a = a->a;
    b->b = a->b;
    if (b->a || b->b)
        b->exponent = 16383 + 63 + (a->sign? 0x8000:0);
    else {
        b->exponent = 0;
        return;
    }
// The normal real number after the conversion format is normalized, that is, the high significant
// bit of the significand is not zero.
    while (b->b >= 0) {
        b->exponent--;
        __asm__("addl %0,%0 ; adcl %1,%1"
            :"=r" (b->a),"=r" (b->b)
            :"" (b->a),"1" (b->b));
    }
}

```

11.6 add.c

11.6.1 Function

The add.c program is used to handle the addition during the simulation. In order to calculate the mantissa of the floating-point number, we need to first symbolize the mantissa, and then perform non-symbolization after the calculation, and then resume using the temporary real format to represent the floating-point number. A schematic diagram of the symbolization and non-symbolic format conversion of floating-point mantissas is shown in Figure 11-10.

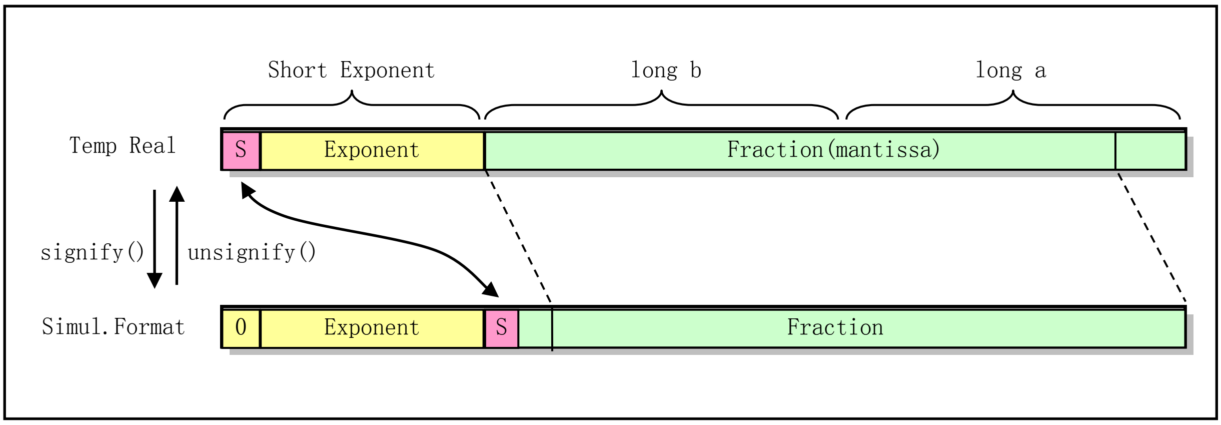


Figure 11-10 Transformation between temporary real and simulation formats

11.6.2 Code annotation

Program 11-5 linux/kernel/math/add.c

```
1  /*
2   * linux/kernel/math/add.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * temporary real addition routine.
9   *
10  * NOTE! These aren't exact: they are only 62 bits wide, and don't do
11  * correct rounding. Fast hack. The reason is that we shift right the
12  * values by two, in order not to have overflow (1 bit), and to be able
13  * to move the sign into the mantissa (1 bit). Much simpler algorithms,
14  * and 62 bits (61 really - no rounding) accuracy is usually enough. The
15  * only time you should notice anything weird is when adding 64-bit
16  * integers together. When using doubles (52 bits accuracy), the
17  * 61-bit accuracy never shows at all.
18  */
19
20 #include <linux/math_emu.h>
21
   // Get a negative number (two's complement) representation of a number.
   // The operation is to invert the mantissa (significand) of the temporary real number and add
   // 1 to it. The parameter 'a' is a pointer to a temporary real structure. The combination of
   // its fields a and b is the significand of the temporary real.
22 #define NEGINT(a) \
23 __asm__("notl %0 ; notl %1 ; addl $1,%0 ; adcl $0,%1" \
24         :"=r" (a->a),"=r" (a->b) \
25         :"" (a->a),"1" (a->b))
26
   // Signified the mantissa.
   // That is, transform the temporary real number into an exponential and integer representation
   // to facilitate the simulation operation. So we call it the simulation format here.
   // The operation is to move the 64-bit binary mantissa right by 2 bits (so the exponent needs
   // to add 2). Since the highest bit of the exponent field is the sign bit, if the exponent value
   // is less than zero, the number is a negative. So we represent the mantissa in complements
   // (take negative) and then take the exponent to a positive value. At this time, the mantissa
   // contains not only the significand shifted by 2 bits but also the sign bit of the value.
27 static void signify(temp_real * a)
28 {

   // On line 30: %0 is a->a; %1 is a->b. The assembly instruction "shrdl $2, %1, %0" performs
   // a double-precision (64-bit) right shift, which shifts the combined mantissa <b, a> to the
   // right by 2 bits. Since this move does not change the value in %1 (a->b), it also needs to
   // be shifted to the right by 2 bits.
29         a->exponent += 2;                        // increases exponent by 2.
30         __asm__("shrdl $2,%1,%0 ; shrl $2,%1"   // mantissa is shifted to right by 2 bits.
31                 :"=r" (a->a),"=r" (a->b)
```

```
                : "" (a->a), "1" (a->b));
        if (a->exponent < 0)
                NEGINT(a);                     // be negative.
        a->exponent &= 0x7fff;                 // remove the sign bit (if any).
}

// Unsignified the mantissa.
// Convert the emulation format to a temporary real format. That is, the real number represented
// by the exponent and the integer is converted into a temporary real format.
static void unsignify(temp_real * a)
{
// For the number with a value of 0, do not process, just return. Otherwise, we first reset
// the sign bit of the temporary rea, and then determine whether the high 32-bit field a->b
// of the mantissa has a sign bit. If so, add a sign bit to the exponent field and also represent
// (complement) the mantissa as an unsigned number. Finally, the mantissa is normalized, and
// the exponent value is decremented accordingly. That is, we perform a left shift operation
// so that the most significant bit of the mantissa is not 0 (the last a->b value appears to
// be like a negative value).
        if (!(a->a || a->b)) {                 // ret if zero.
                a->exponent = 0;
                return;
        }
        a->exponent &= 0x7fff;                 // reset the sign bit.
        if (a->b < 0) {                        // if negative, let mantissa be a positive.
                NEGINT(a);
                a->exponent |= 0x8000;         // add a sign bit.
        }
        while (a->b >= 0) {
                a->exponent--;                 // the mantissa is normalized.
                __asm__("addl %0,%0 ; adcl %1,%1"
                        :"=r" (a->a),"=r" (a->b)
                        :"" (a->a),"1" (a->b));
        }
}

// Simulate floating-point addition instruction.
// Temporary real number parameter: src1 + src2 -> result.
void fadd(const temp_real * src1, const temp_real * src2, temp_real * result)
{
        temp_real a,b;
        int x1,x2,shift;

// First take the exponential values x1, x2 of the two numbers (with the sign bit removed),
// then let the variable a be equal to the maximum value, and let the variable 'shift' equal
// the exponential difference (ie, the multiple of 2 of the difference).
        x1 = src1->exponent & 0x7fff;
        x2 = src2->exponent & 0x7fff;
        if (x1 > x2) {
                a = *src1;
                b = *src2;
                shift = x1-x2;
        } else {
                a = *src2;
                b = *src1;
                shift = x2-x1;
        }
// If the difference between the two is too large, greater than or equal to 2^64, we can ignore
// the small number (ie b value). So you can return value a directly. Otherwise, if the difference
// is greater than or equal to 2^32, then we can ignore the lower 32-bit value of the small
// value b. So we shift b's high long field value b.b to the right by 32 bits, that is, put
// it in b.a. Then increase the exponent of b by 32 times. That is, the exponent difference
// is subtracted by 32. After this adjustment, the mantissas of the two added numbers fall
// substantially in the same range.
        if (shift >= 64) {
                *result = a;
                return;
        }
        if (shift >= 32) {
                b.a = b.b;
                b.b = 0;
                shift -= 32;
        }
// Then we make a fine adjustment to adjust the exponent of the two to the same value. The
// adjustment method is to shift the mantissa of the small value b to the right by 'shift' bits.
// Thus the exponents of the two are the same, in the same order of magnitude. So we can add
// the two mantissas. Before adding, we need to convert them into a simulation format and convert
// it back to the temporary real format after the addition operation.
        __asm__("shrdl %4,%1,%0 ; shrl %4,%1"     // Double precision (64 bit) right shift.
                :"=r" (b.a),"=r" (b.b)
                :"" (b.a),"1" (b.b),"c" ((char) shift));
        signify(&a);                              // change format.
        signify(&b);
        __asm__("addl %4,%0 ; adcl %5,%1"         // adding by using normal instructions.
                :"=r" (a.a),"=r" (a.b)
                :"" (a.a),"1" (a.b),"g" (b.a),"g" (b.b));
        unsignify(&a);                            // change back to temp real format.
        *result = a;
}

```

11.7 compare.c

11.7.1 Function

The compare.c program is used to compare the size of two temporary real numbers in the accumulator during the simulation.

11.7.2 Code annotation

Program 11-6 linux/kernel/math/compare.c

```
/*
 * linux/kernel/math/compare.c
 *
 * (C) 1991 Linus Torvalds
 */

/*
 * temporary real comparison routines
 */

#include <linux/math_emu.h>

// Reset the C3, C2, C1, and C0 condition bits in the status word.
#define clear_Cx() (I387.swd &= ~0x4500)

// The temporary real is normalized, that is, expressed as an exponent and a significand.
// For example: 102.345 is expressed as 1.02345 X 10^2. 0.0001234 is expressed as 1.234 X 10^-4.
// Of course, these numbers are represented in binary in the function.
static void normalize(temp_real * a)
{
        int i = a->exponent & 0x7fff;           // get the exponent (ignore sign bit).
        int sign = a->exponent & 0x8000;        // get sign.

// If the 64-bit significant number (mantissa) of the temporary real a is 0, then a is equal
// to 0. So clear the exponent of a and return.
        if (!(a->a || a->b)) {
                a->exponent = 0;
                return;
        }
// If the mantissa of a has a zero-valued bit at the far left, shift the mantissa to the left
// and adjust the exponent value (decrement). Until the MSB (most significant bit) of the b
// field is 1 (when b appears as a negative value). Finally add the sign bit.
        while (i && a->b >= 0) {
                i--;
                __asm__("addl %0,%0 ; adcl %1,%1"
                        :"=r" (a->a),"=r" (a->b)
                        :"" (a->a),"1" (a->b));
        }
        a->exponent = i | sign;
}

// Simulate floating-point instructions FTST (Floating-point Test).
// That is, the top stack accumulator ST(0) is compared with 0, and the condition bits in the
// status word are set according to the comparison result. If ST > 0.0, C3, C2, and C0 are
// respectively 000; if ST < 0.0, the condition bit is 001; if ST == 0.0, the condition bit
// is 100; if not, the condition bit is 111.
void ftst(const temp_real * a)
{
        temp_real b;

// First, the condition flag in the status word is cleared, and the comparison value b (ST)
// is normalized. If b is not equal to zero and the sign bit is set (is a negative number),
// condition bit C0 is set, otherwise condition bit C3 is set.
        clear_Cx();
        b = *a;
        normalize(&b);
        if (b.a || b.b || b.exponent) {
                if (b.exponent < 0)
                        set_C0();
        } else
                set_C3();
}

// Simulate floating-point instruction FCOM (Floating-point Compare).
// Compare the two parameters src1, src2 and set the condition bits according to the comparison
// result. If src1 > src2, C3, C2, and C0 are respectively 000; if src1 < src2, the condition
// bit is 001; if the two are equal, the condition bit is 100.
void fcom(const temp_real * src1, const temp_real * src2)
{
        temp_real a;

        a = *src1;
        a.exponent ^= 0x8000;                   // invert the sign bit.
        fadd(&a,src2,&a);                       // add the two (ie subtract).
        ftst(&a);                               // test results, set the condition.
}

// Simulate floating-point instruction FUCOM (no order comparison).
// This function is used for comparison operations where one of the operands is a NaN.
void fucom(const temp_real * src1, const temp_real * src2)
{
        fcom(src1,src2);
}

```

11.8 get_put.c

11.8.1 Function

The get_put.c program handles all access to user memory: fetch and store instructions/real values/BCD values. This is the only part that involves other data formats. All other operations in the simulation process use the temporary real number format.

11.8.2 Code annotation

Program 11-7 linux/kernel/math/get_put.c

```
/*
 * linux/kernel/math/get_put.c
 *
 * (C) 1991 Linus Torvalds
 */

/*
 * This file handles all accesses to user memory: getting and putting
 * ints/reals/BCD etc. This is the only part that concerns itself with
 * other than temporary real format. All other cals are strictly temp_real.
 */
// <signal.h> Signal header file. Define signal symbol constants, signal structures, and
//     signal manipulation function prototypes.
// <linux/math_emu.h> Coprocessor emulation header file. A coprocessor data structure and
//     a floating point representation structure are defined.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
//     commonly used functions of the kernel.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined
//     for segment register operations.
#include <signal.h>

#include <linux/math_emu.h>
#include <linux/kernel.h>
#include <asm/segment.h>

// Get a short real (single precision real number) in user memory.
// According to the content in the addressing mode byte in the floating-point instruction code
// and the content in the current register in the info structure, the effective address
// (see file math/ea.c) where the short real number is located is obtained, and then the
// corresponding real value is read from the user data area. Finally, the user's short real
// number is converted into a temporary real number (math/convert.c) and returned.
// Parameters: tmp - a pointer to a temporary real; info - info structure pointer;
// code - instruction code.
void get_short_real(temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        short_real sr;

        addr = ea(info,code);                        // calculate effective address.
        sr = get_fs_long((unsigned long *) addr);    // get from user data area.
        short_to_temp(&sr,tmp);                      // convert to temp real format.
}

// Get a long real (double precision real number) in user memory.
// This function is handled in the same way as get_short_real().
void get_long_real(temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        long_real lr;

        addr = ea(info,code);                        // get effective address.
        lr.a = get_fs_long((unsigned long *) addr);  // get long real.
        lr.b = get_fs_long(1 + (unsigned long *) addr);
        long_to_temp(&lr,tmp);                       // convert to temp real format.
}

// Take the temporary real number in the user's memory.
// First, according to the contents of the addressing mode byte in the floating-point instruction
// code and the contents of the current register in the info structure, obtain the effective
// address (math/ea.c) where the temporary real number is located, and then read the
// corresponding temporary real value from the user data area.
// Parameters: tmp - a pointer to a temporary real number; info - info structure pointer;
// code - instruction code.
void get_temp_real(temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;

        addr = ea(info,code);                           // get effective address.
        tmp->a = get_fs_long((unsigned long *) addr);
        tmp->b = get_fs_long(1 + (unsigned long *) addr);
        tmp->exponent = get_fs_word(4 + (unsigned short *) addr);
}

// Get a short integer in the user's memory.
// The function first obtains the effective address of the short integer according to the content
// in the addressing mode byte in the floating-point instruction code and the contents of the
// current register in the info structure, and then reads the corresponding integer value from
// the user data area, and save as a temporary integer format. Finally, convert the temporary
// integer value to a temporary real number.
// Temporary integers are also represented by 10 bytes. The lower 8 bytes are unsigned integer
// values, and the upper 2 bytes represent exponent value and sign bits. If the highest
// significant byte of the upper 2 bytes is 1, it means a negative number; if the most significant
// bit is 0, it means a positive number.
// Parameters: tmp - a pointer to a temporary real number; info - info structure pointer;
// code - instruction code.
void get_short_int(temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        temp_int ti;

        addr = ea(info,code);                 // get effective in the instruction code.
        ti.a = (signed short) get_fs_word((unsigned short *) addr);
        ti.b = 0;
        if (ti.sign = (ti.a < 0))             // set sign bit if it is a negative.
                ti.a = - ti.a;                // and unsignify the mantissa part.
        int_to_real(&ti,tmp);                 // change to temp real.
}

// Get a long integer in the user's memory and convert it to a temporary real format.
// This function is handled in the same way as get_short_int() above.
void get_long_int(temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        temp_int ti;

        addr = ea(info,code);
        ti.a = get_fs_long((unsigned long *) addr);
        ti.b = 0;
        if (ti.sign = (ti.a < 0))
                ti.a = - ti.a;
        int_to_real(&ti,tmp);
}

// Get a 64-bit long integer (extended long integer) in user memory.
// First, according to the contents of the addressing mode byte in the floating-point instruction
// code and the contents of the current register in the info structure, the effective address
// of the 64-bit long integer is obtained, and then the corresponding integer value is read
// from the user data area and saved as a temporary integer. Finally, convert the temporary
// integer value to a temporary real number.
// Parameters: tmp - temporary real pointer; info - info pointer; code - instruction code.
void get_longlong_int(temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        temp_int ti;

        addr = ea(info,code);                         // get effective address.
        ti.a = get_fs_long((unsigned long *) addr);   // get 64-bit longlong integer.
        ti.b = get_fs_long(1 + (unsigned long *) addr);
        if (ti.sign = (ti.b < 0))                     // set sign bit if it is a negative,
                __asm__("notl %0 ; notl %1\n\t"       // and complement, carry adjustment.
                        "addl $1,%0 ; adcl $0,%1"
                        :"=r" (ti.a),"=r" (ti.b)
                        :"" (ti.a),"1" (ti.b));
        int_to_real(&ti,tmp);                         // change to temporary real.
}

// Multiply a 64-bit integer (such as N) by 10.
// This macro is used in the conversion of the following BCD code values into a temporary real
// number format. The method is: N<<1 + N<<3.
#define MUL10(low,high) \
__asm__("addl %0,%0 ; adcl %1,%1\n\t" \
"movl %0,%%ecx ; movl %1,%%ebx\n\t" \
"addl %0,%0 ; adcl %1,%1\n\t" \
"addl %0,%0 ; adcl %1,%1\n\t" \
"addl %%ecx,%0 ; adcl %%ebx,%1" \
:"=a" (low),"=d" (high) \
:"" (low),"1" (high):"cx","bx")

// 64-bit addition. Add the 32-bit unsigned number val to the 64-bit <high,low>.
#define ADD64(val,low,high) \
__asm__("addl %4,%0 ; adcl $0,%1":"=r" (low),"=r" (high) \
:"" (low),"1" (high),"r" ((unsigned long) (val)))

// Take the BCD code value in the user's memory and convert it to temporary real format.
// The function first obtains the effective address of the BCD code according to the content
// in the addressing mode byte in the floating-point instruction code and the content in the
// registers in the info structure, and then reads the corresponding BCD code of 10 bytes from
// the user data area (where 1 byte is used for sign) and is converted to a temporary integer
// form. Finally, the temporary integer value is converted to a temporary real number.
// Parameters: tmp - a pointer to a temporary real number; info - info structure pointer;
// code - instruction code.
void get_BCD(temp_real * tmp, struct info * info, unsigned short code)
```

```
{
        int k;
        char * addr;
        temp_int i;
        unsigned char c;

// Get the memory effective address of the BCD code value, and then start processing from the
// last BCD code byte (most significant bit). First obtain the sign bit of the BCD code value
// and set the sign bit of the temporary integer. The 9-byte BCD code value is then converted
// to a temporary integer format, and finally the temporary integer value is converted to a
// temporary real number.
        addr = ea(info,code);                      // get effective address.
        addr += 9;                                 // point to the last (10th) byte.
        i.sign = 0x80 & get_fs_byte(addr--);       // get sign bit.
        i.a = i.b = 0;
        for (k = 0; k < 9; k++) {                  // change to temporary integer.
                c = get_fs_byte(addr--);
                MUL10(i.a, i.b);
                ADD64((c>>4), i.a, i.b);
                MUL10(i.a, i.b);
                ADD64((c&0xf), i.a, i.b);
        }
        int_to_real(&i,tmp);                       // change to temporary real.
}

// Save the result in the short (single precision) real format to the user data area.
// The function first obtains the effective address addr in the instruction code, and then
// converts the result of the temporary real into a short real format and stores it at the location
// of addr.
// Parameters: tmp - temporary real format result value; info - info structure pointer;
// code - instruction code.
void put_short_real(const temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        short_real sr;

        addr = ea(info,code);                       // get effective address.
        verify_area(addr,4);                        // verify the data area.
        temp_to_short(tmp,&sr);                     // convert to short real.
        put_fs_long(sr,(unsigned long *) addr);     // store at location addr.
}

// Save the result in the long (double precision) real format to the user data area.
// This function is handled in the same way as put_real_real() above.
void put_long_real(const temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        long_real lr;

        addr = ea(info,code);
        verify_area(addr,8);
        temp_to_long(tmp,&lr);
        put_fs_long(lr.a, (unsigned long *) addr);
        put_fs_long(lr.b, 1 + (unsigned long *) addr);
}

// Save the result in the temporary real format to the user data area.
// The function first obtains the address addr for saving the result, and after verifying that
// there is enough (10 bytes) of user memory at the address, then stores the temporary real
// number to the addr.
// Parameters: tmp - temporary real format result value; info - info structure pointer;
// code - instruction code.
void put_temp_real(const temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;

        addr = ea(info,code);                          // get effective address.
        verify_area(addr,10);                          // verify there is enough uesr memory.
        put_fs_long(tmp->a, (unsigned long *) addr); // store temp real to user area.
        put_fs_long(tmp->b, 1 + (unsigned long *) addr);
        put_fs_word(tmp->exponent, 4 + (short *) addr);
}

// Save the result in the short integer format to the user data area.
// The function first obtains the address addr for saving the result, and then converts the
// result of the temporary real format into a temporary integer format. If it is a negative
// number, set the integer sign bit. Finally, the integer is saved to the user memory.
// Parameters: tmp - temporary real format result value; info - info structure pointer;
// code - instruction code.
void put_short_int(const temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        temp_int ti;

        addr = ea(info,code);                          // get effective address.
        real_to_int(tmp,&ti);                          // change to temp integer.
        verify_area(addr,2);                           // verify user area (need 2 bytes)
        if (ti.sign)
                ti.a = -ti.a;                          // negative the result.
        put_fs_word(ti.a,(short *) addr);              // store to user data area.
}

// Save the result in a long integer format to the user data area.
// This function is handled in the same way as put_short_int() above.
void put_long_int(const temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        temp_int ti;

        addr = ea(info,code);                          // get effective address.
        real_to_int(tmp,&ti);                          // change to temp integer.
        verify_area(addr,4);                        // verify user area (4 bytes).
        if (ti.sign)                                // if signed, negative the result.
                ti.a = -ti.a;
        put_fs_long(ti.a,(unsigned long *) addr);    // store to the user data area.
}

// Save the result in the 64-bit integer format to the user data area.
// The function first obtains the address addr for saving the result, and then converts the
// result of the temporary real format into a temporary integer format. If it is a negative
// number, set the integer sign bit. Finally, the integer is saved to the user memory.
// Parameters: tmp - temporary real format result value; info - info structure pointer;
// code - instruction code.
void put_longlong_int(const temp_real * tmp,
        struct info * info, unsigned short code)
{
        char * addr;
        temp_int ti;

        addr = ea(info,code);                       // get effective address.
        real_to_int(tmp,&ti);                       // change to temporary integer.
        verify_area(addr,8);                        // verify that 8 bytes are available.
        if (ti.sign)                               // if signed, change to negative value.
                __asm__("notl %0 ; notl %1\n\t"     // invert and add one.
                        "addl $1,%0 ; adcl $0,%1"
                        :"=r" (ti.a),"=r" (ti.b)
                        :"" (ti.a),"1" (ti.b));
        put_fs_long(ti.a,(unsigned long *) addr);    // store to the user data area.
        put_fs_long(ti.b,1 + (unsigned long *) addr);
}

// The unsigned number <high, low> is divided by 10 and the remainder is placed in rem.
#define DIV10(low,high,rem) \
__asm__("divl %6 ; xchgl %1,%2 ; divl %6" \
        :"=d" (rem),"=a" (low),"=b" (high) \
        :"" (0),"1" (high),"2" (low),"c" (10))

// Save the result in the BCD code format to the user data area.
// This function first obtains the address addr for saving the result and verifies the user
// space for storing the 10-byte BCD code. The result of the temporary real number format is
// then converted to data in BCD code format and saved to the user memory. If it is negative,
// the most significant bit of the highest memory byte is set.
// Parameters: tmp - temporary real format result value; info - info structure pointer;
// code - instruction code.
void put_BCD(const temp_real * tmp,struct info * info, unsigned short code)
{
        int k,rem;
        char * addr;
        temp_int i;
        unsigned char c;

        addr = ea(info,code);                       // get the address for storing result.
        verify_area(addr,10);                       // verify memory space.
        real_to_int(tmp,&i);                        // change to temporary integer.
        if (i.sign)                               // if signed, set MSB bit of high byte.
                put_fs_byte(0x80, addr+9);
        else                                      // otherwise reset the MSB bit.
                put_fs_byte(0, addr+9);
        for (k = 0; k < 9; k++) {                 // change to BCD code and stored.
                DIV10(i.a,i.b,rem);
                c = rem;
                DIV10(i.a,i.b,rem);
                c += rem<<4;
                put_fs_byte(c,addr++);
        }
}

```

11.9 mul.c

11.9.1 Function

The functions in the mul.c program use the CPU's normal arithmetic instructions to simulate the 80387 multiplication.

11.9.2 Code annotation

Program 11-8 linux/kernel/math/mul.c

```
/*
 * linux/kernel/math/mul.c
 *
 * (C) 1991 Linus Torvalds
 */

/*
 * temporary real multiplication routine.
 */

#include <linux/math_emu.h>

// Shift the 16-byte value at the parameter c pointer to the left by 1 bit (multipl by 2).
static void shift(int * c)
{
        __asm__("movl (%0),%%eax ; addl %%eax,(%0)\n\t"
                "movl 4(%0),%%eax ; adcl %%eax,4(%0)\n\t"
                "movl 8(%0),%%eax ; adcl %%eax,8(%0)\n\t"
                "movl 12(%0),%%eax ; adcl %%eax,12(%0)"
                ::"r" ((long) c):"ax");
}

// Two temporary reals are multiplied, the result is placed at the c pointer (16 bytes).
static void mul64(const temp_real * a, const temp_real * b, int * c)
{
        __asm__("movl (%0),%%eax\n\t"
                "mull (%1)\n\t"
                "movl %%eax,(%2)\n\t"
                "movl %%edx,4(%2)\n\t"
                "movl 4(%0),%%eax\n\t"
                "mull 4(%1)\n\t"
                "movl %%eax,8(%2)\n\t"
                "movl %%edx,12(%2)\n\t"
                "movl (%0),%%eax\n\t"
                "mull 4(%1)\n\t"
                "addl %%eax,4(%2)\n\t"
                "adcl %%edx,8(%2)\n\t"
                "adcl $0,12(%2)\n\t"
                "movl 4(%0),%%eax\n\t"
                "mull (%1)\n\t"
                "addl %%eax,4(%2)\n\t"
                "adcl %%edx,8(%2)\n\t"
                "adcl $0,12(%2)"
                ::"b" ((long) a),"c" ((long) b),"D" ((long) c)
                :"ax","dx");
}

// Simulation floating-point calculation instruction FMUL (Floating-point Multiply).
// Temporary reals src1 * scr2 -> result.
void fmul(const temp_real * src1, const temp_real * src2, temp_real * result)
{
        int i,sign;
        int tmp[4] = {0,0,0,0};

// First determine the sign of the multiplication of the two numbers. The sign is equal to the
// sign bit XOR of both. Then calculate the multiplied exponent value. The exponent needs to
// be added when multiplying. However, since the exponent is stored in a biased number format,
// the biased amount is added twice when the exponents of the two numbers are added, so it is
// necessary to subtract a biased number ( The biased number of the temporary real is 16383).
        sign = (src1->exponent ^ src2->exponent) & 0x8000;
        i = (src1->exponent & 0x7fff) + (src2->exponent & 0x7fff) - 16383 + 1;

// If the resulting exponent becomes negative, it means that the two numbers are multiplied
// to produce an underflow. So directly return the signed zero value.
// If the result exponent is greater than 0x7fff, it indicates that an overflow occurred, so
// the status word overflow exception flag is set and returned.
        if (i<0) {
                result->exponent = sign;
                result->a = result->b = 0;
                return;
        }
        if (i>0x7fff) {
                set_OE();
                return;
        }
// If the mantissas of two numbers are multiplied and the result is not 0, the resulting mantissa
// is normalized. That is, the resulting mantissa value is shifted to the left so that the most
// significant bit is 1, and the exponent is adjusted accordingly. If the mantissa of the 16-byte
// result obtained by multiplying the mantissa of the two numbers is 0, the exponent is also
// set to 0. Finally, the multiplication result is saved in the temporary real 'result'.
        mul64(src1,src2,tmp);
        if (tmp[0] || tmp[1] || tmp[2] || tmp[3])
                while (i && tmp[3] >= 0) {
                        i--;
                        shift(tmp);
                }
        else
                i = 0;
        result->exponent = i | sign;
        result->a = tmp[2];
        result->b = tmp[3];
}

```

11.10 div.c

11.10.1 Function

The div.c program uses the CPU normal calculation instructions to simulate the division of the 80387 coprocessor.

11.10.2 Code annotation

Program 11-9 linux/kernel/math/div.c

```
/*
 * linux/kernel/math/div.c
 *
 * (C) 1991 Linus Torvalds
 */

/*
 * temporary real division routine.
 */

#include <linux/math_emu.h>

// Shift the contents of the 4 bytes to the left by 1 bit (multiply by 2).
static void shift_left(int * c)
{
        __asm__ __volatile__("movl (%0),%%eax ; addl %%eax,(%0)\n\t"
                "movl 4(%0),%%eax ; adcl %%eax,4(%0)\n\t"
                "movl 8(%0),%%eax ; adcl %%eax,8(%0)\n\t"
                "movl 12(%0),%%eax ; adcl %%eax,12(%0)"
                ::"r" ((long) c):"ax");
}

// Shift the contents of the 4 bytes pointed to by pointer c to the right by 1 bit.
```

```
static void shift_right(int * c)
{
        __asm__("shrl $1,12(%0) ; rcrl $1,8(%0) ; rcrl $1,4(%0) ; rcrl $1,(%0)"
                ::"r" ((long) c));
}

// 16-byte subtraction function.
// 16-byte subtraction, (a - b) -> a. Finally, ok is set according to whether there is a borrow
// flag (CF=1). If there is no borrow (CF=0) then ok = 1, otherwise ok = 0.
static int try_sub(int * a, int * b)
{
        char ok;

        __asm__ __volatile__("movl (%1),%%eax ; subl %%eax,(%2)\n\t"
                "movl 4(%1),%%eax ; sbbl %%eax,4(%2)\n\t"
                "movl 8(%1),%%eax ; sbbl %%eax,8(%2)\n\t"
                "movl 12(%1),%%eax ; sbbl %%eax,12(%2)\n\t"
                "setae %%al":"=a" (ok):"c" ((long) a), "d" ((long) b));
        return ok;
}

// 16-byte division function.
// Parameter a / b -> c. The method is to simulate multi-byte division using subtraction.
static void div64(int * a, int * b, int * c)
{
        int tmp[4];
        int i;
        unsigned int mask = 0;

        c += 4;
        for (i = 0 ; i<64 ; i++) {
                if (!(mask >>= 1)) {
                        c--;
                        mask = 0x80000000;
                }
                tmp[0] = a[0]; tmp[1] = a[1];
                tmp[2] = a[2]; tmp[3] = a[3];
                if (try_sub(b,tmp)) {
                        *c |= mask;
                        a[0] = tmp[0]; a[1] = tmp[1];
                        a[2] = tmp[2]; a[3] = tmp[3];
                }
                shift_right(b);
        }
}

// Simulate floating point instruction FDIV.
// Temporary real division: src1 / src 2 -> result.
void fdiv(const temp_real * src1, const temp_real * src2, temp_real * result)
{
        int i,sign;
        int a[4],b[4],tmp[4] = {0,0,0,0};

        sign = (src1->exponent ^ src2->exponent) & 0x8000;
        if (!(src2->a || src2->b)) {
                set_ZE();
                return;
        }
        i = (src1->exponent & 0x7fff) - (src2->exponent & 0x7fff) + 16383;
        if (i<0) {
                set_UE();
                result->exponent = sign;
                result->a = result->b = 0;
                return;
        }
        a[0] = a[1] = 0;
        a[2] = src1->a;
        a[3] = src1->b;
        b[0] = b[1] = 0;
        b[2] = src2->a;
        b[3] = src2->b;
        while (b[3] >= 0) {
                i++;
                shift_left(b);
        }
        div64(a,b,tmp);
        if (tmp[0] || tmp[1] || tmp[2] || tmp[3]) {
                while (i && tmp[3] >= 0) {
                        i--;
                        shift_left(tmp);
                }
                if (tmp[3] >= 0)
                        set_DE();
        } else
                i = 0;
        if (i>0x7fff) {
                set_OE();
                return;
        }
        if (tmp[0] || tmp[1])
                set_PE();
        result->exponent = i | sign;
        result->a = tmp[2];
        result->b = tmp[3];
}
```

11.11 Summary

This chapter describes the method and code implementation of the Linux kernel for emulating the 80387 math coprocessor chip. First, we introduced several commonly used integer and floating point types, described the 80387 runtime and the temporary real types used in software simulation, and gave their specific representation format. Then we introduce the composition and working mathod of the math coprocessor, and explain the working implementation of each emulation code program with the math_emulate.c program as the main line.

In the next chapter we provide a comprehensive introduction to the MINIX file system used by kernel 0.12. After a brief description of the MINIX file system structure and various file types, we will detail how the cache used to access the file system works. It also briefly describes the role of each underlying file and function and the main functions for accessing data in various files, including file and directory management functions provided by system calls. Later, before starting to comment on the code in detail, an example of a simple file system on the block device was specifically presented and described.

12 File System (fs)

This chapter covers the implementation code for the file system in the Linux kernel and the cache manager for block devices. When developing the Linux 0.12 kernel file system, Mr. Linus mainly referred to the MINIX operating system at the time, and used the 1.0 version of the MINIX file system. Therefore, when reading the contents of this chapter, you can first read about the MINIX file system. For the introduction of the working principle and implementation method of the cache, you can first browse the book "Design of UNIX Operating System" by Mr. M.J.Bach.

There are 18 source files associated with the file system implementation, as shown in Listing 12-1. The cross-references between these files and the functions in them can be found in Section 5.10.

List 12-1 linux/fs

Filename	Size	Last modified time (GMT)	Desc.
Makefile	7176 bytes	1992-01-12 19:49:06	
bitmap.c	4007 bytes	1992-01-11 19:57:29	
block_dev.c	1763 bytes	1991-12-09 21:11:23	
buffer.c	9072 bytes	1991-12-06 20:21:00	
char_dev.c	2103 bytes	1991-11-19 09:10:22	
exec.c	9908 bytes	1992-01-13 23:36:33	
fcntl.c	1455 bytes	1991-10-02 14:16:29	
file_dev.c	1852 bytes	1991-12-01 19:02:43	
file_table.c	122 bytes	1991-10-02 14:16:29	
inode.c	7166 bytes	1992-01-10 22:27:26	
ioctl.c	1136 bytes	1991-12-21 01:58:35	
namei.c	18958 bytes	1992-01-12 04:09:58	
open.c	4862 bytes	1992-01-08 20:01:36	
pipe.c	2834 bytes	1992-01-10 22:18:11	
read_write.c	2802 bytes	1991-11-25 15:47:20	
select.c	6381 bytes	1992-01-13 22:25:23	
stat.c	1875 bytes	1992-01-11 20:39:19	
super.c	5603 bytes	1991-12-09 21:11:34	
truncate.c	1692 bytes	1992-01-11 19:47:28	

12.1 Main Functions

The file system is an important part of the operating system and is the place where the operating system stores a large amount of programs and data for a long time. When the system loads the executable program, it needs to be quickly read from the file system to the memory to run. Some temporary files generated during the system operation also need to be dynamically saved in the file system. Therefore, the file system needs to use high-speed devices to store programs and data, so the operating system usually uses a block device that can store a large amount of information as a device of the file system. In addition, UNIX-like operating systems usually access devices through device files, so the composition and implementation of the file system is very complicated.

The procedures noted in this chapter are large, but through the analysis of the Linux source code directory structure in Section 5.10 (see Figure 5-29), we can divide these files into four parts for discussion. The first part is about the high-speed buffer (cache) management program, which mainly implements the function of high-speed data access to block devices such as hard disks. This part of the content is concentrated in the buffer.c program; The second part of the code describes the low-level generic functions of the file system. The management of file index nodes, the allocation and release of disk data blocks, and the conversion algorithm of file names and i nodes are described; The third part of the program is about reading and writing data in the file, including access to data in character devices, pipes, and block read and write files; The fourth part of the program mainly involves the implementation of the system-call interface of the file, mainly related to the system-calls of file opening, closing, creation and related file directory operations.

Let's first introduce the basic structure of the MINIX file system, and then explain the four parts separately.

12.1.1 MINIX file system

At present, the version of the MINIX operating system is 2.0, and the file system used is version 2.0. It is different from the version before the 1.5 version of the system, and its capacity has been expanded. However, since the Linux kernel annotated in this book uses the MINIX file system version 1.0, only the version 1.0 file system is briefly introduced here.

The MINIX file system is basically the same as the standard UNIX file system. It consists of six parts: (1) Boot block, (2) Super block, (3) i node bitmap, (4) Logic block bitmap, (5) i nodes , (6) Data Blocks. For a common disk block device, the distribution of its parts is shown in Figure 12-1.

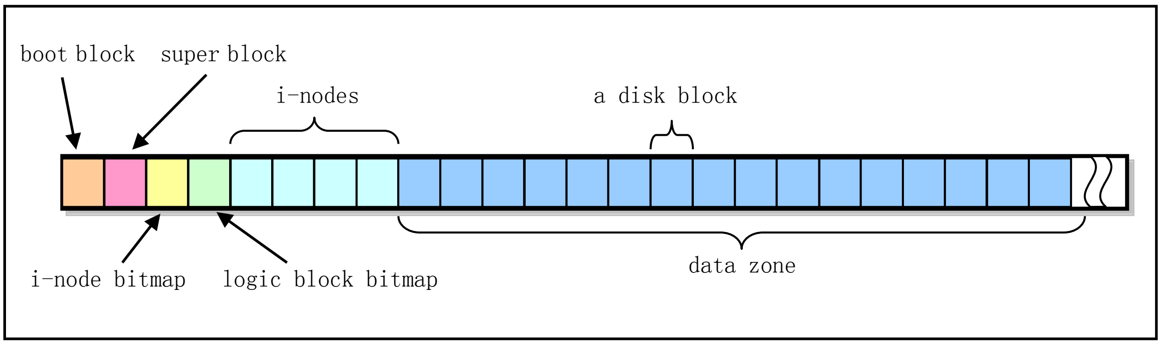


Figure 12-1 Schematic diagram of the layout of each part of the MINIX file system

In the figure, the entire block device is divided into disk blocks in units of 1 KB, so for a 360 KB floppy disk device, there are 360 disk blocks in the above figure, and each square represents one disk block. As we will see in the following description, in the MINIX 1.0 file system, the disk block size is exactly the same as the logical block size, which is also 1 KB. Therefore 360KB discs also contain 360 logical blocks. We will sometimes mix these two names in the discussion that follows.

The boot block is an execution code and data disk block that can be automatically read by the ROM BIOS when the computer is powered on. However, not all disk devices in a system are used as boot devices, so for disks that are not used for booting, this disk block may contain no code. However, any disk block device must have boot block space to maintain the uniformity of the MINIX file system format. That is, the file system simply leaves a space for storing the boot block on the block device. If you put the kernel image file in the file system, you can store the actual bootloader in the first block of the device where the filesystem resides (ie the boot blockspace), and let it get and load the kernel image file in the filesystem.

For a large-capacity hard disk block device, several partitions can usually be divided on it, and a different complete file system can be stored in each partition, as shown in Figure 12-2. The figure shows that there are 4 partitions, which store the FAT32 file system, NTFS file system, MINIX file system and EXT2 file system. The first sector of the hard disk is the master boot sector, which stores the hard disk boot code and partition table information. The information in the partition table indicates the type of each partition on the hard disk, the starting position parameter and the ending position parameter, and the total number of sectors occupied in the hard disk. See the hard disk partition table structure after the kernel/blk_drv/hd.c file.

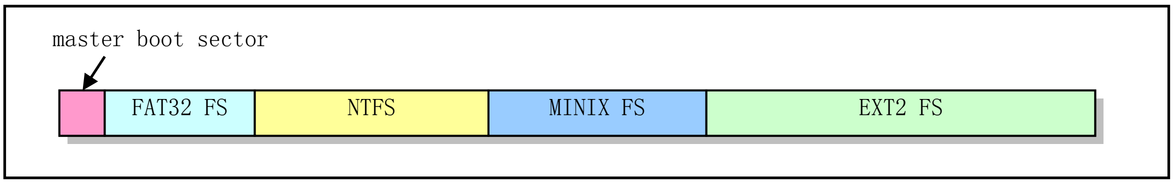


Figure 12-2 Partitions and file systems on hard disk devices

The super block is used to store the structure information of the file system on the disk device, and describes the size of each part. The structure is shown in Figure 12-3. Where s_ninodes represents the total number of i-nodes on the device; s_nzones represents the total number of logical blocks in the logical block on the device; s_imap_blocks and s_zmap_blocks represent the number of disk blocks occupied by the i-node bitmap and the logical block bitmap, respectively; s_firstdatazone represents the first logical block number occupied by the beginning of the data area on the device; S_log_zone_size is the number of disk blocks contained in each logical block represented by a base 2 logarithm. For the MINIX 1.0 file system, this value is 0, so the size of its logical block is equal to the disk block size, which is 1 KB. s_max_size is the maximum file length in bytes, which does not exceed 4GB. Of course, this length value will be limited by the disk capacity. S_magic is the file system magic number used to indicate the type of file system. For the MINIX 1.0 file system, its magic number is 0x137f.

In the Linux 0.12 system, the loaded file system superblock is stored in the superblock table (array) super_block[]. There are 8 entries in this table, so the Linux 0.12 system can load up to 8 file systems at the same time. The superblock table will be initialized in the mount_root() function of the super.c program. In the read_super() function, a superblock will be set in the table for the newly loaded filesystem, and the superblock will be released in the put_super() function.

	Field name	Data type	Description
available in disk and memory	s_ninodes	short	Number of i-nodes
	s_nzones	short	Number of zones.
	s_imap_blocks	short	Number of blocks for i-node bitmap
	s_zmap_blocks	short	Number of blocks for logic bitmap
	s_firstdatazone	short	Fisrt logic block no. in data zone.
	s_log_zone_size	short	Log$_2$(DiskBlocks/LogicBlocks)
	s_max_size	long	Maximum file size
	s_magic	short	FS magic number (0x137f)
available only in memory.	s_imap[8]	buffer_head *	i-node bitmap block array in cache.
	s_zmap[8]	buffer_head *	Logic block bitmap block array in cache.
	s_dev	short	Device no of the super block.
	s_isup	m_inode *	i-node of the installed fs.
	s_imount	m_inode *	i-node the fs installed to.
	s_time	long	Modified time.
	s_wait	task_struct *	Tasks waiting for this superblock.
	s_lock	char	Locked flag.
	s_rd_only	char	Readonly flag.
	s_dirt	char	Modified flag (dirty flag)

Figure 12-3 Super block structure of MINIX file system

The logic block bitmap is used to describe the usage of each data block on the disk. In addition to the first bit (bit 0), each bit in the logical block bitmap represents, in turn, a logical block in the data area on the disk. Therefore, bit 1 of the logic block bitmap represents the first data disk block in the data area on the disk, not the first disk block (boot block) on the disk. When a data disk block is occupied, the corresponding bit in the logic block bitmap is set. Since the function to find the free disk block returns a value of 0 when all disk data blocks are occupied, the lowest bit (bit 0) of the logic block bitmap is idle and is set to 1 when the file system is created.

From the structure of the super block, we can also see that the logic block bitmap uses up to 8 buffer blocks (s_zmap[8]), and each block size is 1024 bytes, and each bit represents the occupation status of one disk block. Therefore a buffer block can represent 8192 disk blocks, and 8 buffer blocks can represent a total of 65,536 disk blocks, so the maximum block device capacity (length) that MINIX file system 1.0 can support is 64 MB.

The i-node is used to store index information for each file and directory name on the disk device. The i-node bitmap is used to indicate whether the i-node is used, and each bit represents an i-node. For a size of 1K block, a disk block can represent the usage of 8192 i-nodes. Similar to the case of a logical block bitmap, since the function that looks for an idle i-node returns a value of 0 when all i-nodes are used, the lowest bit (bit 0) of the first byte of the i-node bitmap and the corresponding i-node 0 is left idle, and the bit position in the corresponding bit map of the i-node 0 is set to 1 in advance when the file system is created. Therefore, only the state of 8191 i-nodes can be represented in the first i-node bitmap block.

The i-node part of the disk holds the index node of the file or directory name in the file system, and each file or directory name has an i-node. Each i-node structure stores information about the corresponding file or directory, such as the file host's id (uid), the file's group id (gid), the file length, the access modification time, and the location of the file data block on the disk. A total of 32 bytes are used for the entire i-node structure, as shown in Figure 12-4.

	Field name	Data type	Description
fields in disk and memory total 32 bytes	i_mode	short	File types & attributes (rwx bits)
	i_uid	short	File owner's user id.
	i_size	long	File size (bytes)
	i_mtime	long	Modified time (from 1970.1.1:0,sec)
	i_gid	char	File group id
	i_nlinks	char	Number of links (how many directory entries point to this i-node).
	i_zone[9]	short	Logic block no. occupied by file: zone[0]-zone[6] direct block no. zone[7] indirect block no. zone[8] 2^{nd} indirect block no. Note: for device file, zone[0] is device no. of the specified device file.
fields available only in memory	i_wait	task_struct *	tasks waiting for the i-node.
	i_atime	long	Last access time.
	i_ctime	long	i-node modified time.
	i_dev	short	Device no the i-node belongs.
	i_num	short	i-node no.
	i_count	short	Reference count of the i-node. 0 - idle.
	i_lock	char	i-node is locked.
	i_dirt	char	i-node has been modified (dirty flag).
	i_pipe	char	i-node is used for pipe.
	i_mount	char	i-node was installed with other fs.
	i_seek	char	Search flag (for lseek).
	i_update	char	i-node updated flag.

Figure 12-4 i-node structure of MINIX file system version 1.0

The i_mode field is used to save the file type and access rights properties. Its bits 15-12 are used to save the file type, bits 11-9 hold the information set when the file is executed, and bits 8-0 indicate the access rights of the file, as shown in Figure 12-5. See the file include/sys/stat.h on lines 20 -- 50 and file include/fcntl.h for details.

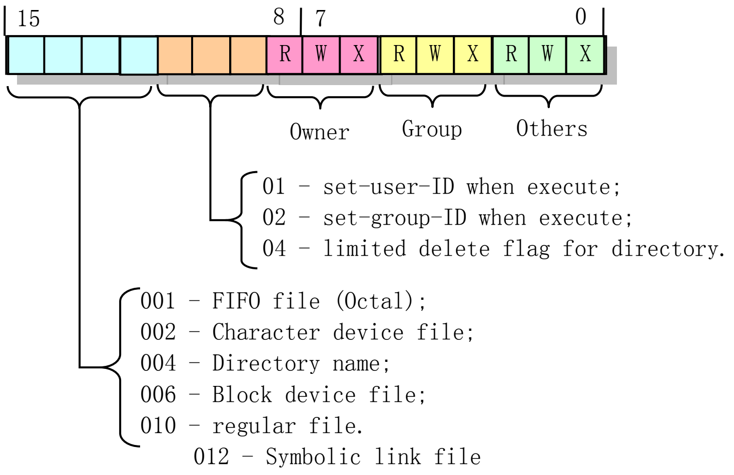


Figure 12-5 i-node mode field content

The data in the file is stored in the data area of the disk block, and a file name is associated with the data disk block through the corresponding i-node. The number of these disk blocks is stored in the logical block array i_zone[] of the i-node. The i_zone[] array is used to store the disk block number of the file corresponding to the i-node. i_zone[0] to i_zone[6] are used to store the 7 disk block numbers at the beginning of the file, called direct blocks. If the file length is less than or equal to 7K bytes, the disk block it uses can be quickly found according to its i-node. If the file is larger, you need to use an indirect block (i_zone[7]), which holds the additional block number. For the MINIX file system it can store 512 disk block numbers, so 512 disk blocks can be addressed. If the file is larger, you need to use a secondary indirect disk block (i_zone[8]). The primary disk block of the secondary indirect block functions similarly to the one-time indirect disk block, so 512*512 disk blocks can be addressed using the secondary indirect disk block, as shown in Figure 12-6. So for MINIX file system version 1.0, the maximum length of a file is (7 + 512 + 512 * 512) = 262,663 KB.

In addition, for device files in the /dev/ directory, they do not occupy the data disk blocks in the disk data area, that is, the length of their files is zero. The i-node of the device file name is only used to save the attributes and device numbers of the device it defines. The device number is stored in zone[0] of the device file i node.

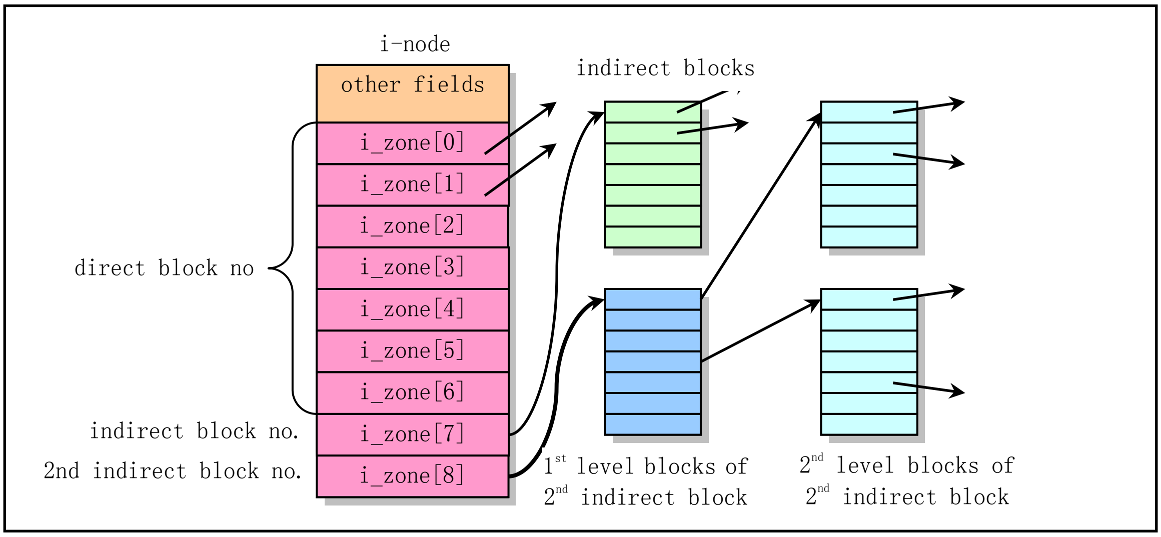


Figure 12-6 The function of the logical block array of the i-node

Here again, when all i-nodes are used, the function that looks for the idle i-node will return a value of 0. Therefore, the lowest bit of the i-node bitmap and the i-node 0 are not used. The structure of i-node 0 is initialized to all zeros, and the bit position of 0 in the i-node bitmap is set when the file system is created.

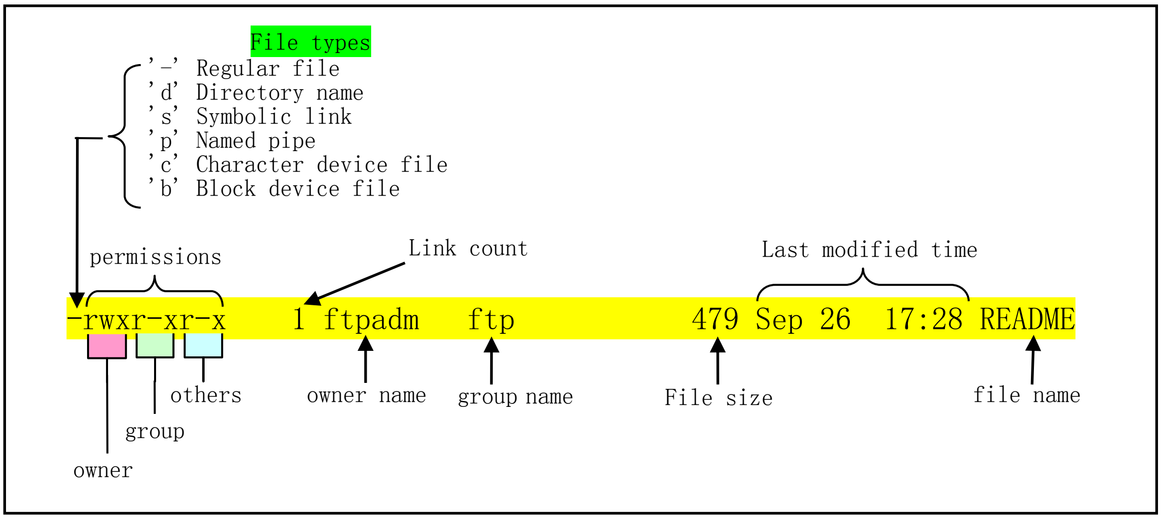
For a PC, the length of one sector (512 bytes) is generally used as the data block length of the block device. The MINIX file system treats two consecutive sectors of data (1024 bytes) as one block of data, called a disk block, and its length is the same as the buffer block length in the cache. The number is counted from the first disk block on the disk, that is, the boot block is the 0th disk block.

The above logical block or block is a power multiple of 2 of the disk block. A logical block length can be equal to 1, 2, 4 or 8 disk block lengths. For the Linux kernel discussed in this book, the length of the logic block is equal to the length of the disk block, so the two terms have the same meaning in the code comments. However, the term data logic block (or data disk block) refers to the disk block numbered from the first data disk block in the data portion of the disk device.

12.1.2File Types, Attributes, and Directory Items

12.1.2.1 File types and attributes

Files in UNIX-like operating systems can be generally divided into six categories: (1) regular file; (2) directory name; (3) symbolic link file; (4) named pipe file; (5) character device file; (6) block device files. If you execute the "ls -l" command at the shell command line prompt, we can know the type of the file from the listed file status information, as shown in Figure 12-7.



```
-rwxr-xr-x    1 ftpadm    ftp         479 Sep 26   17:28 README
```

Figure 12-7 File information displayed by the command 'ls –l'

In the figure, the first character of the file information displayed indicates the type of the listed file. For example, the '-' in the figure indicates that the file is a regular (normal) file.

A regular file ('-') is a file that the file system does not interpret and can contain byte streams of any length. For example, source files, binary executables, documents, and script files.

The directory ('d') also appears as a file in the UNIX-like operating system, but file system management interprets its contents so that people can see which files are contained in a directory and how they are organized together to form a hierarchical file system.

A symbolic link ('s') is used to reference another file with a different file name. A symbolic link can connect to a file in another file system. Deleting a symbolic link does not affect the file being connected. There is also a connection method called "hard link". It has the same status as the linked file in the symbolic link described here, and is treated as a general file, but cannot be linked across file systems (or devices) and increments the link count value of the file. See the description of the link count below.

A named pipe ('p') file is a file created when the system creates a named pipe and can be used for communication between unrelated processes.

Character device ('c') files are used to access character devices, such as tty terminals, memory devices, and network devices, by operating files.

A block device ('b') file is used to access devices such as hard disks, floppy disks, and the like. In UNIX-like operating systems, block device files and character device files are generally stored in the /dev directory of the system.

In the Linux kernel, the file type information is stored in the i_mode field of the corresponding i-node and is represented by the upper 4 bits. When working with the file system, the system uses some macros that determine file types, such as S_ISBLK, S_ISDIR, etc. These macros are defined in the include/sys/stat.h file.

In the figure, the file type character is followed by three sets of file permission attributes composed of one set of three characters, which are used to indicate the access rights of the file host, the same group of users, and other users to the file. 'rwx' indicates permission to read, write, and execute the file, respectively. For directory name files, executable means that you can enter the directory. When operating on the permissions of a file, they are generally represented in octal. For example, '755' (0b111, 101, 101) indicates that the file host can read/write/execute the file, and the same group of users and others can read and execute the file. In the Linux 0.12 source code, the file permission information is also stored in the i_mode field of the i-node, and the lower 9 bits of the field are used to represent the three sets of permissions, and are often represented by the variable mode. Macros for file permissions are defined in the file include/fcntl.h.

The 'Link Count' field in the figure indicates the number of times the file was referenced by a hard link. When the count is reduced to zero, the file is deleted. 'Username' indicates the name of the file host, and 'Group name' is the name of the group to which the user belongs.

12.1.2.2 File System Directory Item Structure

The Linux 0.12 system uses the MINIX file system version 1.0. Its directory structure and directory entry structure is the same as that of the traditional UNIX file system, and is defined in the include/linux/fs.h file. In a directory of the file system, directory entries for all file names in the directory are stored in data blocks of the directory file. For example, the directory entry for all file names under the directory name root/ is stored in the data block of the root/ directory name file. All file name information in the root directory of the file system is stored in the data block of the specified i-node (ie, the i-node of number 1). The file name directory entry structure is as follows:

```
// Defined in the include/linux/fs.h file.
#define NAME_LEN 14                              // maximum name length.
#define ROOT_INO 1                               // root i-node.

// File directory entry structure.
struct dir_entry {
        unsigned short inode;                    // i-node number.
        char name[NAME_LEN];                     // filename.
};
```

Each file directory entry includes only a file name string of 14 bytes in length and a 2-byte i-node number corresponding to the file name. Therefore, a logical disk block can store 1024/16=64 directory entries. Other information about the file is stored in the i-node structure specified by the i-node number. The structure mainly includes information such as file access attribute, host, length, access save time, and disk block. The i-node of each inode number is located at a fixed location on the disk.

When a file is opened, the file system will find its i-node number according to the given file name, and find the disk block location where the file is located by its corresponding i-node information, as shown in Figure 12-8. For example, to find the i-node number of the file name /usr/bin/vi, the file system first starts from the root directory with the fixed i-node number (1), that is, the name is found from the data block of the i-node number 1. The directory entry of usr, thus getting the i node number of the file /usr. According to the i-node number, the file system can smoothly obtain the directory /usr, and the directory entry of the file name bin can be found therein. In this way, we also know the i-node number of /usr/bin, so we can know the location of the directory /usr/bin directory, and find the directory entry of the vi file in the directory. Finally, we can get the i-node number of the file path name /usr/bin/vi, so that the i-node structure information can be obtained from the disk.

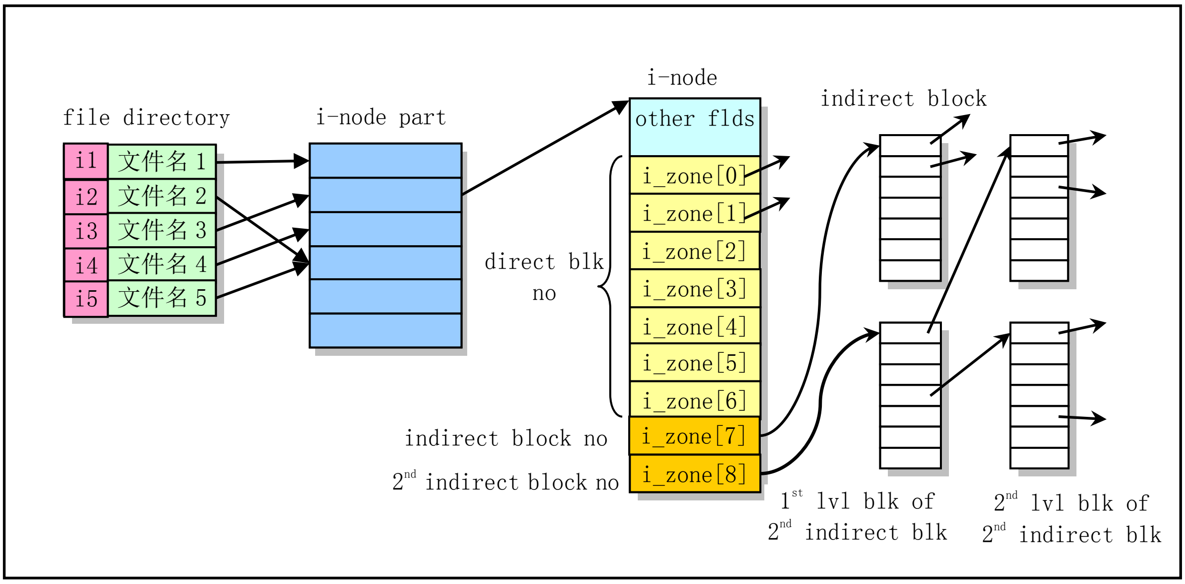


Figure 12-8 Find the file disk block location by file name

If you look at the distribution of a file on disk, the process of searching for a file block information can be represented by Figure 12-9 (where boot blocks, super blocks, i-nodes, and logical block bitmaps are not shown).

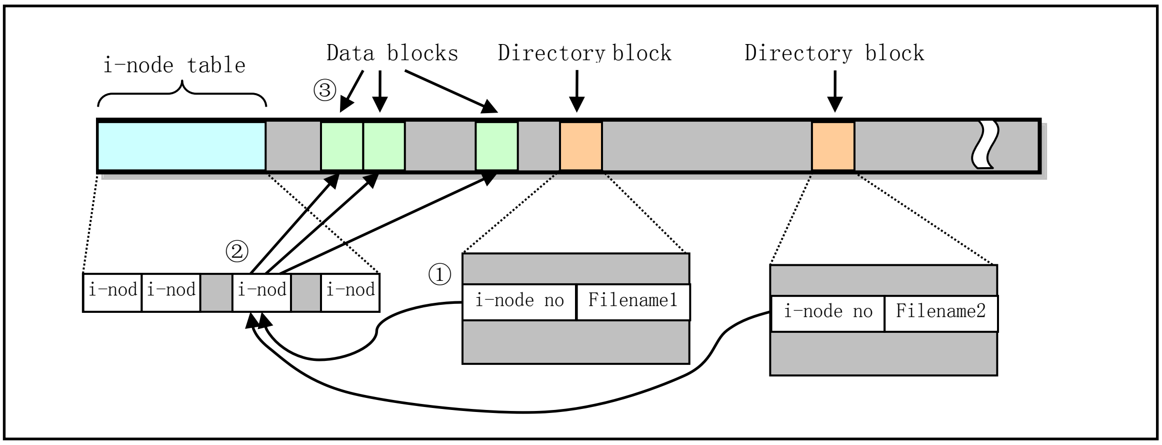


Figure 12-9 Get its data block from the file name

We can find the corresponding directory entry by the file name specified by the user program. According to the i-node number in the directory entry, the corresponding i-node structure in the i-node table can be found. The i-node structure contains the block number information of the file data, so that the data information corresponding to the file name can be finally obtained. In the figure, there are two directory entries pointing to the same i-node, so the same data on the disk can be obtained based on the two file names. There is a link count field i_nlinks in each i-node structure that records the number of directory entries pointing to the i-node, that is, the hard link count value of the file. In this case, the value of this field is 2. When performing a delete file operation, the kernel will actually delete the data of the file on the disk only when the i-node link count value is equal to zero. In addition, since the i-node number in the directory entry can only be used in the current file system, the directory entry of one file system cannot be used to point to the i-node in another file system, that is, the hard link cannot cross the file system.

Unlike hard links, file name directory entries for symbolic link types do not directly point to the corresponding i-node. The symbolic link directory entry stores the pathname string of a file in the data block of the corresponding file. When accessing a symbolic link directory entry, the kernel reads the contents of the file and then accesses the specified file based on the pathname string. Therefore, symbolic links may not limited to a file system. We can create a symbolic link in a file system that points to a file name in another file system.

There are also two special file directory entries in each directory, whose names are fixed to '.' and '..' respectively. The '.' directory entry gives the i-node number of the current directory, and the '..' directory entry gives the i-node number of the parent directory. Therefore, when a relative path name is given, the file system can use these two special directory entries for lookup operations. For example, to find '../kernel/Makefile', you can first get the i-node number of the parent directory according to the '..' directory entry of the current directory, and then perform the lookup operation according to the process described above.

For a directory entry for each directory file, the link count field in its i-node also indicates the number of directory entries connected to that directory. Therefore, the link count value for each directory file is at least 2. One of them is a link to a directory entry that contains a directory file, and the other is a link to a '..' directory entry in the directory. For example, if we create a subdirectory named 'mydir' in the current directory, the link between the current directory and the subdirectory is shown in Figure 12-10.

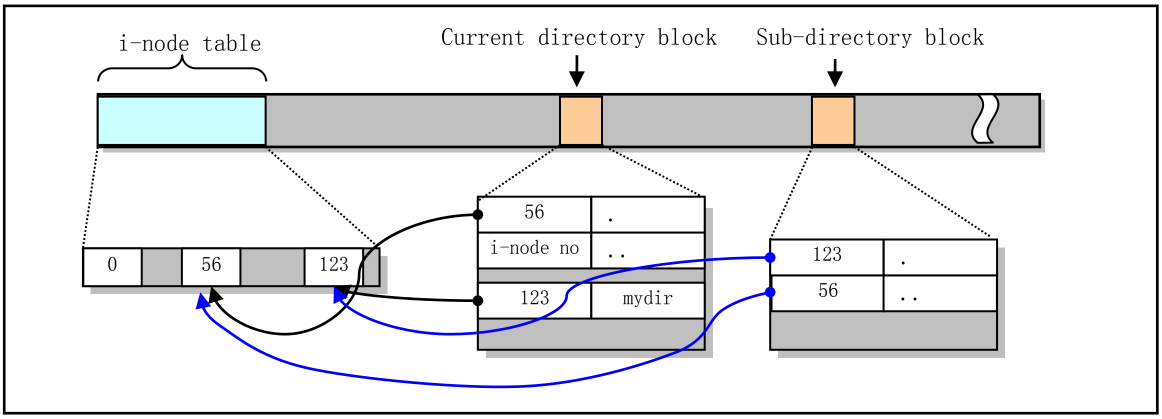


Figure 12-10 Directory file entries and subdirectory links

The figure shows that we have created a 'mydir' subdirectory in the directory with i node number 56, and the i node number of this subdirectory is 123. The '.' directory entry in the 'mydir' subdirectory points to its own i-node 123, while its '..' directory entry points to the i-node 56 of its parent directory. It can be seen that since the directory entry of a directory always has two links, if there are subdirectories, the number of i-node links of the parent directory is equal to the number of 2+ subdirectories.

12.1.2.3 Directory Structure Example

Take the Linux 0.12 system as an example, let's look at its root directory structure. After running the Linux 0.12 system in the bochs simulation system, we first list its file system root directory entries, including the implied '.' and '..' directory entries. Then we use the hexdump command to view the contents of the data block of the '.' or '..' file, you can see the contents of each directory item contained in the root directory.

```
[/usr/root]# cd /
[/]# ls -la
total 10
```

```
drwxr-xr-x  10 root     root          176 Mar 21  2004 .
drwxr-xr-x  10 root     4096          176 Mar 21  2004 ..
drwxr-xr-x   2 root     4096          912 Mar 21  2004 bin
drwxr-xr-x   2 root     root          336 Mar 21  2004 dev
drwxr-xr-x   2 root     root          224 Mar 21  2004 etc
drwxr-xr-x   8 root     root          128 Mar 21  2004 image
drwxr-xr-x   2 root     root           32 Mar 21  2004 mnt
drwxr-xr-x   2 root     root           64 Mar 21  2004 tmp
drwxr-xr-x  10 root     root          192 Mar 29  2004 usr
drwxr-xr-x   2 root     root           32 Mar 21  2004 var

[/]# hexdump .
0000000 0001 002e 0000 0000 0000 0000 0000 0000       // .
0000010 0001 2e2e 0000 0000 0000 0000 0000 0000       // ..
0000020 0002 6962 006e 0000 0000 0000 0000 0000       // bin
0000030 0003 6564 0076 0000 0000 0000 0000 0000       // dev
0000040 0004 7465 0063 0000 0000 0000 0000 0000       // etc
0000050 0005 7375 0072 0000 0000 0000 0000 0000       // usr
0000060 0115 6e6d 0074 0000 0000 0000 0000 0000       // mnt
0000070 0036 6d74 0070 0000 0000 0000 0000 0000       // tmp
0000080 0000 6962 2e6e 656e 0077 0000 0000 0000       // idle, not used.
0000090 0052 6d69 6761 0065 0000 0000 0000 0000       // image
00000a0 007b 6176 0072 0000 0000 0000 0000 0000       // var
00000b0
[/]#
```

After executing the 'hexdump.' command, all directory entries contained in the i-node data block are listed. Each row corresponds to a directory entry. The first two bytes of each line are the i-node number, and the next 14 bytes are the file name or directory name string. If the i-node number in a directory entry is 0, it means that the directory entry is not used, or the corresponding file has been deleted or removed. The i-node numbers of the first two directory entries ('.' and '..') are all number 1. This is a special feature of the file system root directory structure, which is different from the rest of the subdirectory structure.

Now let's look at the etc/ subdirectory entry. Also using the hexdump command on the etc/ directory, we can display the directory entries contained in the etc/ subdirectory, as shown below.

```
[/]# ls etc -la
total 32
drwxr-xr-x   2 root     root          224 Mar 21  2004 .
drwxr-xr-x  10 root     root          176 Mar 21  2004 ..
-rw-r--r--   1 root     root          137 Mar  4  2004 group
-rw-r--r--   1 root     root        11801 Mar  4  2004 magic
-rw-r--r--   1 root     root           11 Jan 22 18:12 mtab
-rw-r--r--   1 root     root          142 Mar  5  2004 mtools
-rw-r--r--   1 root     root          266 Mar  4  2004 passwd
-rw-r--r--   1 root     root          147 Mar  4  2004 profile
-rw-r--r--   1 root     root           57 Mar  4  2004 rc
-rw-r--r--   1 root     root         1034 Mar  4  2004 termcap
-rwx--x--x   1 root     root        10137 Jan 15  1992 update

[/]# hexdump etc
0000000 0004 002e 0000 0000 0000 0000 0000 0000        // .
```

```
0000010 0001 2e2e 0000 0000 0000 0000 0000 0000        // ..
0000020 0007 6372 0000 0000 0000 0000 0000 0000        // rc
0000030 000b 7075 6164 6574 0000 0000 0000 0000        // update
0000040 0113 6574 6d72 6163 0070 0000 0000 0000        // termcap
0000050 00ee 746d 6261 0000 0000 0000 0000 0000        // mtab
0000060 0000 746d 6261 007e 0000 0000 0000 0000        // not used.
0000070 007c 616d 6967 0063 0000 0000 0000 0000        // magic
0000080 0016 7270 666f 6c69 0065 0000 0000 0000        // profile
0000090 007e 6170 7373 6477 0000 0000 0000 0000        // passwd
00000a0 0081 7267 756f 0070 0000 0000 0000 0000        // group
00000b0 01ee 746d 6f6f 736c 0000 0000 0000 0000        // mtools
00000c0
[/]#
```

At this point, we can see that the data block corresponding to the etc/ directory name i-node contains the directory entry information of all the files in the subdirectory. The i-node of the directory entry '.' is the i-node number 4 of the etc/ directory entry, and the i-node of '..' is the i-node number 1 of the etc/parent directory.

12.1.3 High speed buffer (Buffer cache)

A high-speed buffer is a must for the file system to access data in a block device. In order to access data in the file system on the block device, the kernel can access the block device each time for a read or write operation. But the time of each I/O operation is very slow compared to the processing speed of the memory and CPU. In order to improve the performance of the system, the kernel opens up a high-speed data buffer (buffer cache) in memory and divides it into buffer blocks equal in size to the disk data block for use and management, in order to reduce the number of times of accessing block devices. In the Linux kernel, the buffer cache is located between the kernel code area and the main memory area, as shown in Figure 5-5. The data block in each block device that has been used recently is stored in the buffer cache. When you need to read data from a block device, the buffer manager first looks in the buffer cache. If the corresponding data is already in the buffer, there is no need to read it from the block device again. If the data is not in the buffer cache, a command to read the block device is issued to read the data into the cache. When data needs to be written to the block device, the system will request a free buffer block in the cache to temporarily store the data. As for when to actually write the data to the device, it is achieved through the device data synchronization function.

The program that implements the buffer cache in the Linux kernel is buffer.c. Other programs in the file system call its block read and write functions by specifying the device number and data logic block number that need to be accessed. These interface functions are: block read function bread (), block advance read-ahead function breada () and page block read function bread_page (). The page block read function reads the number of buffer blocks (4 blocks) that can be accommodated in one page of memory at a time.

12.1.4 File System Lower Level Functions

The underlying processing functions of the file system are contained in the following five files:

- The bitmap.c program includes a release and occupancy handler for the i-node bitmap and the logic block bitmap. The functions that operate on the i-node bitmap are free_inode() and new_inode(), and the functions that operate on the logic block bitmap are free_block() and new_block().
- The truncate.c program includes a function truncate() that truncates the length of the data file to zero. It cuts the file length on the device specified by the i-node to 0 and releases the device logic block occupied by the file data.

- The inode.c program includes a function iput() that allocates the i-node function and a function iput() that puts back the memory i-node, and a function bmap() that takes the logical block number of the file data block on the device.
- The namei.c program mainly includes the function namei(). This function maps the given file pathname to its i-node using iget(), iput(), and bmap().
- The super.c program is designed to handle file system superblocks, including the functions get_super(), put_super(), and free_super(). Also included are several file system load/unload handlers and system-calls such as sys_mount().

The hierarchical relationship between functions in these files is shown in Figure 12-11.

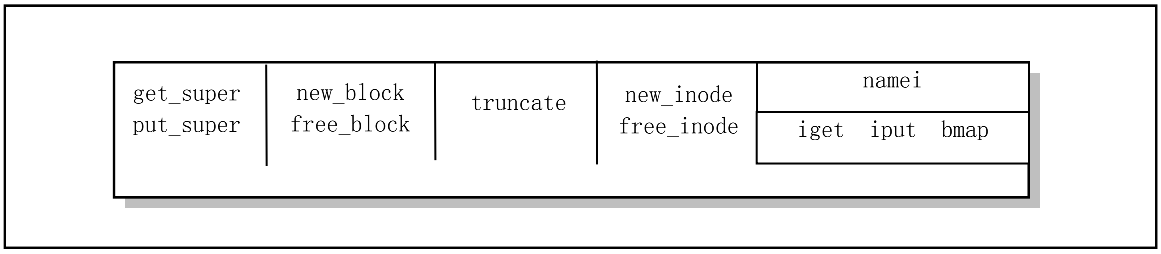


Figure 12-11 File system low-level function hierarchy

12.1.5 File Data Access Operations

The access operation code for the data in the file mainly involves five files: block_dev.c, file_dev.c, char_dev.c, pipe.c, and read_write.c. The read and write functions involved are shown in Figure 12-12. The first four files can be thought of as block devices, character devices, pipeline devices, and interface programs for ordinary files and file read/write system-calls. They collectively implement the read() and write() system-calls in read_write.c. By judging the attributes of the file being manipulated, the two system-calls will call the relevant processing functions in these files to operate.

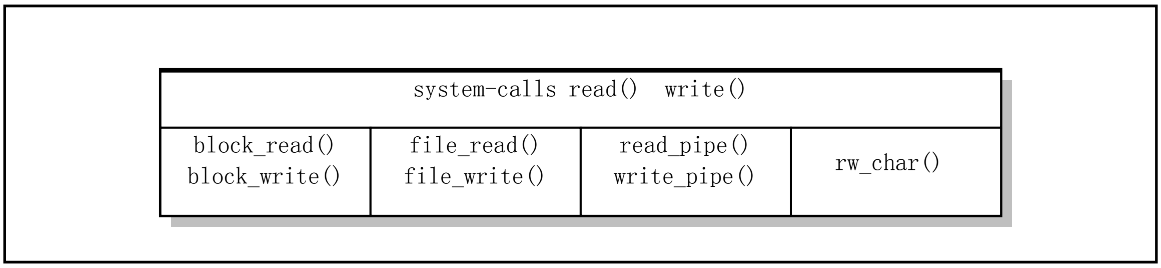


Figure 12-12 File data access functions

The functions block_read() and block_write() in the block_dev.c program are used to read and write data from a block device special file. The parameters used specify the device number to be accessed, the starting position and length of the read and write.

The file_read() and file_write() functions in the file_dev.c program are used to access general regular files. By giving the i-node and file structure of the file, we can know the device number where the file is located and the current read-write pointer of the file.

The pipe read and write functions read_pipe() and write_pipe() are implemented in the pipe.c file. It also implements the system-call pipe() that creates an anonymous pipe. Pipes are mainly used to transfer data between processes in a first-in, first-out manner, or they can be used to synchronize processes. There are two types of pipes: named pipes and unnamed pipes. Named pipes are created using the open call of the file system, while unnamed pipes are created using the system-call pipe(). When using pipes, the regular file read(), write(), and close() functions are used. Only the descendants of the pipe() call can share access to the unnamed pipe, and all processes can access the named pipe as long as the permission is granted.

For pipe read and write, it can be seen that one process writes data to one end of the pipe, while another process reads data from the other end of the pipe. The way the kernel accesses the data in the pipeline is exactly the same as the way it accesses the data in a normal formal file. The difference between allocating storage for pipes and allocating space for regular files is that pipes use only direct blocks of i-nodes. The kernel manages the direct block of the i-node as a circular queue of the pipeline, and ensures the order of first-in first-out by modifying the read-write pointer.

For character device files, the system-calls read() and write() call the rw_char() function in char_dev.c. Character devices include console terminals (tty), serial terminals (ttyx), and memory character devices.

In addition, the kernel uses the file structure, the file table file_table[], and the in-memory i-node table inode_table[] to manage operational access to the file. The definition of these data structures and tables can be found in the header file include/linux/fs.h. The file structure is defined as follows.

```
struct file {
         unsigned short f_mode;              // file operating mode (RW bits)
         unsigned short f_flags;             // file open and control flags.
         unsigned short f_count;             // file handler count.
         struct m_inode * f_inode;           // point to i-node in memory (the v-node).
         off_t f_pos;                        // current read/write position.
};
struct file file_table[NR_FILE]              // file table array, total 64 items.
```

It is used to establish a relationship between a file handle and an i-node entry in a memory i-node table. The file type and access attribute field f_mode have the same meaning as the i_mode field in the file i-node structure, as described above; The f_flags field is a combination of some open operation control flags given by the parameter flag in the open file function open() and control function fcntl(), which are defined in file include/fcntl.h. There are some of the following flags:

```
  // file access mode used by open() and fcntl(). Only one of three can be used at the same time.
#define O_RDONLY         00             // open file in read only mode.
#define O_WRONLY         01             // open file in write only mode.
#define O_RDWR           02             // open file with read/write mode.
   // File creation flags for open(). Can be used with the above access mode in a 'bit or' mode.
#define O_CREAT        00100            // create if not exist (not used by fcntl).
#define O_EXCL         00200            // exclude using file flag.
#define O_NOCTTY       00400            // no control tty.
#define O_TRUNC        01000            // truncate to 0 if file exist and in write mode.
#define O_APPEND       02000            // open in append mode, point to eof.
#define O_NONBLOCK     04000            // open in nonblack mode.
#define O_NDELAY       O_NONBLOCK
```

The file reference count field f_count in the file structure indicates the number of times the file is referenced by the file handle; The memory i-node structure field f_inode points to the memory i-node structure item in the corresponding i-node table of this file. The file table is an array of file structure items in the kernel. In the Linux 0.12 kernel, the file table can have up to 64 items, so the entire system can open up to 64 files at the same time. In the process data structure (that is, the process control block or process descriptor), the file structure pointer array filp[NR_OPEN] field of the open file is specifically defined. Where NR_OPEN = 20, so each process can open up to 20 files at the same time. The sequence number of the pointer array item corresponds to the file descriptor (file handle), and the pointer of the item points to the file item opened in the file table. For example, filp[0] is the file structure pointer corresponding to the currently open file descriptor 0 (handle 0) of the process.

The i-node table inode_table[NR_INODE] in the kernel is an array of memory i-node structures, where NR_INODE = 32, so at the moment the kernel can only store 32 memory i-node information at the same time. The relationship between the file opened by a process and the kernel file table and the corresponding memory i-node can be represented by Figure 12-13. One file in the figure is opened as standard input for the process (file handle 0) and the other is opened as standard output (file handle 1).

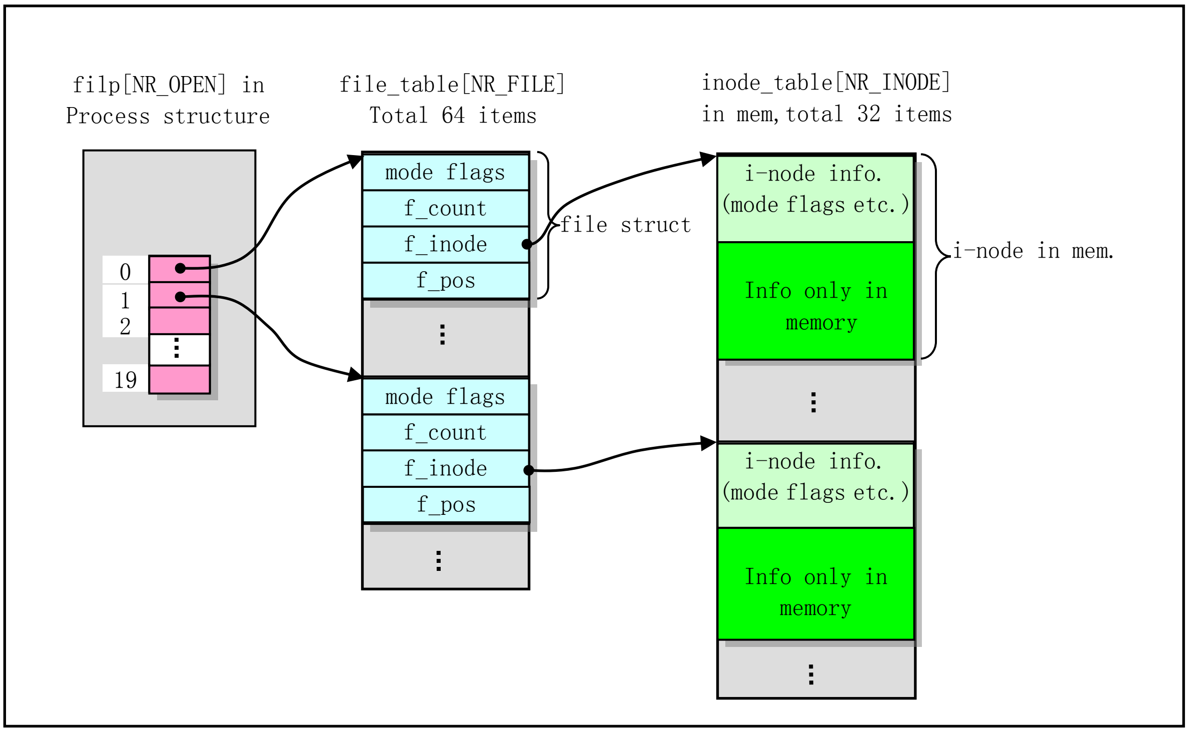


Figure 12-13 The structures used by the process to open the file

12.1.6 File and Directory Management System-Calls

User access to the file system are achieved through the system-calls provided by the kernel. The upper layer implementation of the system-calls for the file basically consists of the five files listed in Figure 12-14.

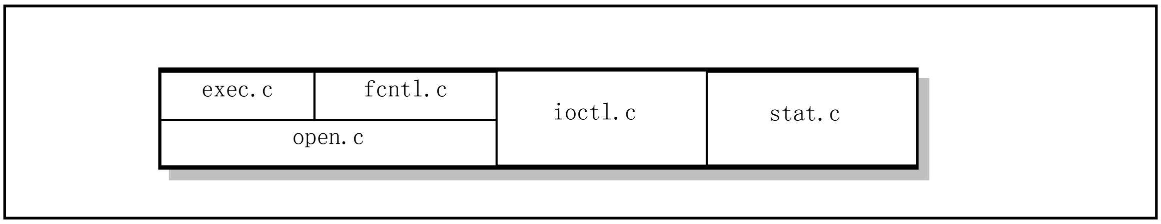


Figure 12-14 File system upper layer operating programs

The open.c file is used to implement system-calls related to file operations. There are mainly file creation, opening and closing, file host and attribute modification, file access permission modification, file time modification, and system file root change.

The exec.c program implements the loading and execution of binary executables and shell script files. The main function is do_execve(), which is the C handler for the system-call interrupt (int 0x80) function number __NR_execve(), and is the main implementation function of the exec() function cluster.

Fcntl.c implements the file control system-call fcntl(), and two file handle duplication system-calls dup() and dup2(). Dup2() needs to specify the value of the new handle, while dup() returns the unused handle with the smallest current value. The handle duplication operation is mainly used for standard input/output redirection and pipeline operations of files.

The ioctl.c file implements the input/output control system-call ioctl(), which mainly calls the tty_ioctl() function to control the I/O of the terminal.

The stat.c file is used to implement the file status information system-calls stat() and fstat(). Stat() uses the file name to get the information, and fstat() uses the file handle (descriptor) to get the information.

12.1.7 Analysis of a File System in 360KB Floppy Disk

In order to deepen the understanding of the file system structure shown in Figure 12-1, we used the Linux 0.12 system to create a MINIX 1.0 file system in the 360KB floppy image, which only stores a file named hello.c.

We first run the following command in the Linux 0.12 system in the bochs environment to create a file system.

```
[/usr/root]# mkfs /dev/fd1 360                  // Create a 360KB file system in the 2nd fd.
120 inodes                                      // 120 i-nodes and 360 disk (logic) blocks.
360 blocks
Firstdatazone=8 (8)                             // Starting block number of the data zone is 8.
Zonesize=1024                                   // Block size is 1024 bytes.
Maxsize=268966912                               // Maximum file szie(obviously incorrect).

[/usr/root]# mount /dev/fd1 /mnt                // Mount to the /mnt directory and copy a file.
[/usr/root]# cp hello.c /mnt
[/usr/root]# ll -a /mnt                         // There are 3 directory entries.
total 3
drwxr-xr-x   2 root     root          48 Feb 23 17:48 .
drwxr-xr-x  10 root     root         176 Mar 21  2004 ..
-rw-------   1 root     root          74 Feb 23 17:48 hello.c

[/usr/root]# umount /dev/fd1                    // Unmount the file system.
```

```
[/usr/root]#
```

In the above series of operations, after executing the mkfs command on the floppy disk (image file) in the second floppy drive, a MINIX file system is created on the disk. It can be seen from the content displayed after the execution of the command that the file system contains a total of 120 i-nodes and 360 disk blocks. The first block number in the data zone of the disk is 8, and the size of the logical block is 1024 bytes, which is the same with disk block, and the file maximum size that can be stored is 268,966,912 bytes (obviously wrong).

Then we use the mount command to mount the device to the directory /mnt, and then copy the file hello.c to it, and then unmount the file system. Now we have created a MINIX file system with only one file, which is stored in the disk image file (diskb.img) corresponding to the second floppy drive of bochs.

Now let's look at the specific content of this file system. For convenience, here we use the hexdump command in the Linux 0.12 system to observe the contents. You can also exit the bochs system and view it using an editing program such as notepad++ that can modify the binary. After executing the hexdump command on the device /dev/fd1, the following is displayed (slightly sorted out).

```
[/usr/root]# hexdump /dev/fd1 | more
0000000 44eb 4d90 6f74 6c6f 2073 0020 0102 0001  // 0x0000 - 0x03ff (1KB) is boot block.
0000010 e002 4000 f00b 0009 0012 0002 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
*                                                // the data in this range is 0.
0000400 0078 0168 0001 0001 0008 0000 1c00 1008  // 0x0400 - 0x07ff (1KB) is super block.
0000410 137f 0000 0000 0000 0000 0000 0000 0000
0000420 0000 0000 0000 0000 0000 0000 0000 0000
*
0000800 0007 0000 0000 0000 0000 0000 0000 ff00  // 0x0800 - 0x0bff (1KB) is i-node bitmap.
0000810 ffff ffff ffff ffff ffff ffff ffff ffff
*                                                // the data in this range is 1.
0000c00 0007 0000 0000 0000 0000 0000 0000 0000  // 0x0c00 - 0x0fff (1KB),logical blk bitmap.
0000c10 0000 0000 0000 0000 0000 0000 0000 0000
0000c20 0000 0000 0000 0000 0000 0000 fffe ffff
0000c30 ffff ffff ffff ffff ffff ffff ffff ffff
*
0001000 41ed 0000 0030 0000 c200 421c 0200 0008  // 0x1000 - 0x1fff (4KB) is 120 i-nodes.
0001010 0000 0000 0000 0000 0000 0000 0000 0000
0001020 8180 0000 004a 0000 c200 421c 0100 0009
0001030 0000 0000 0000 0000 0000 0000 0000 0000
*                                                //// The following is data block contents.
0002000 0001 002e 0000 0000 0000 0000 0000 0000  // 0x2000 - 0x23ff (1KB), root i-node data
0002010 0001 2e2e 0000 0000 0000 0000 0000 0000
0002020 0002 6568 6c6c 2e6f 0063 0000 0000 0000
0002030 0000 0000 0000 0000 0000 0000 0000 0000
*
0002400 6923 636e 756c 6564 3c20 7473 6964 2e6f  // 0x2400 - 0x27ff (1KB), the hello.c file.
0002410 3e68 0a0a 6e69 2074 616d 6e69 2928 7b0a
0002420 090a 7270 6e69 6674 2228 6548 6c6c 2c6f
0002430 7720 726f 646c 5c21 226e 3b29 090a 6572
0002440 7574 6e72 3020 0a3b 0a7d 0000 0000 0000
0002450 0000 0000 0000 0000 0000 0000 0000 0000
--More--
```

Now let's analyze the contents given above one by one. According to Figure 12-1, the first disk block of the MINIX 1.0 file system is a boot block, so disk block 0 (0x0000 - 0x03ff, 1KB) is the boot block content. Regardless of whether your disk is used to boot the system, each newly created file system will retain a boot disk block. For newly created disk image files, the boot disk block contents should all be zero. The content in the boot disk block in the above display data is the data left in the original image file, that is, the mkfs command does not modify the contents of the boot disk block when the file system is created.

Disk block 1 (0x0400 - 0x07ff, 1 KB) is the content of the super block. According to the super block data structure of the MINIX file system (see Figure 12-3), we can know the file system super block information listed in Table 12-1, which has a total of 18 bytes. Since the logarithm of the number of disk blocks per logical block to 2 is 0, for the MINIX file system, the disk block size is equal to the logical block (block) size.

Table 12-1 MINIX file system super block in 360KB disk

Field Name	Description	Content or Value
s_ninodes	Number of i-nodes	0x0078 (=120 in decimal)
s_nzones	Zone block (logical block) number	0x0168 (=360 in decimal)
s_imap_blocks	Number of blocks occupied by i-node bitmap	0x0001
s_zmap_blocks	Number of blocks in zone bitmap	0x0001
s_firstdatazone	First data block number	0x0008
s_log_zone_size	Log2(blocks/zone)	0x0000
s_max_size	Maximum file size	0x10081c00 = 268966912 Bytes
s_magic	FS magic number	0x137f

Disk block 2 (0x0800 - 0x0bff, 1 KB) contains i-node bitmap information. Since there are a total of 120 i-nodes in the file system, and each bit represents an i-node structure, the file system actually occupies 120/8 = 15 bytes in the 1 KB-sized disk block. The bit value of 0 indicates that the corresponding i-node structure is not occupied, and 1 indicates that it is occupied or reserved. The remaining unused byte bits in the disk block are initialized to 1 by the mkfs command.

From the data of the disk block 2, we can see that the first byte value is 0x07 (0b0000111), that is, the first three bits of the i-node bitmap have been occupied. As can be seen from the foregoing description, the first bit (bit 0) is reserved. The 2nd and 3rd bits respectively indicate that the file system's No. 1 and No. 2 i-nodes have been used, that is, the following i-node areas already contain two i-node structures. In fact, node 1 is used as the root i-node of the file system, and node 2 is used for the only file hello.c on the file system, the contents of which are described later.

Disk block 3 (0x0c00 - 0x0fff, 1 KB) is the logical block bitmap content. Since the disk capacity is only 360KB, the file system actually uses 360 bits, which is 360/8 = 45 bytes. The number of blocks for special purposes is: 1 boot block + 1 super block + 1 i-node bitmap block + 1 logical block bitmap block + 4 i-node data blocks = 8. Considering that the logic block bitmap only indicates that the disk block is occupied in the data area on the disk, after removing the number of function blocks that have been used, the actual number of bits required for the logic block bitmap is 360 - 8 = 352 bits (occupying 44 bytes), plus the remaining unused bit 0, the bitmap requires a total of 353 bits. This is why the last (45th) byte (0xfe) has only 1 bit being 0.

Therefore, when we know the bit offset value nr of a logic block in the logic block bitmap, then the corresponding actual disk disk block number block is equal to nr + 8 -1, ie block = nr + s_firstdatazone - 1. When we want to find the bit offset value (ie, the block number in the data area) nr in the logical block bitmap for a disk block number block, it is nr = block - s_firstdatazone +1.

Similar to the i-node bitmap, the first 3 bits of the first byte of the logic block bitmap are already occupied. The first (bit 0) bit is reserved, and the 2nd and 3rd bits indicate that 2 disk blocks (logical blocks) have been used in the disk data zone. In fact, the first disk block in the disk data zone represented by bit 1 is used for the root i-node to store data information (directory entries), and the second disk block represented by bit 2 is used for the number 2 i-node to save relevant data information. Please note that the data information mentioned here refers to the data content managed by the i-node, and is not the information of the i-node structure. The structure information of the i-node itself will be stored in the disk block in the i-node area dedicated to the i-node structure, that is, the disk disk block 4--7.

Disk block 4--7 (0x1000 - 0x1fff, 4KB) 4 disk blocks are used to store i-node structure information. Since the file system is provided with 120 i-nodes, and each i-node occupies 32 bytes (see Figure 12-4), a total of 120 x 32 = 3840 bytes is required, that is, 4 disk blocks are required. From the data shown above, we can see that the first 32 bytes hold the contents of root i-node, and the subsequent 32 bytes store the contents of node 2, as shown in Table 12-2 and Table 12-3.

Table 12-2 No. 1 root i-node structure content

Field Name	Description	Value
i_mode	File’s mode	0x41ed (drwxr-xr-x)
i_uid	File’s owner id	0x0000
i_size	File size	0x00000030 (48 bytes)
i_mtime	Modified time	0x421cc200 (Feb 23 17:48)
i_gid	File group id	0x00
i_nlinks	Number of links	0x02
i_zone[9]	Array of logical block numbers	zone[0] = 0x0008, remaining items are all 0.

Table 12-3 No. 2 i-node structure content

Field Name	Description	Value
i_mode	File’s mode	0x8180 (-rw-------)
i_uid	File’s owner id	0x0000
i_size	File size	0x0000004a (74 bytes)
i_mtime	Modified time	0x421cc200 (Feb 23 17:48)
i_gid	File group id	0x00
i_nlinks	Number of links	0x01
i_zone[9]	Araay of logical block numbers	zone[0] = 0x0009, remaining items are all 0.

It can be seen that the data block of root i node has only one block, and its logical block number is 8, located on the first block in the disk data zone, and the length is 30 bytes. From the previous section, a directory entry length is 16 (0x10) bytes, so there are 3 directory entries (0x30 bytes) coexisting in this logical block. Because it is a directory, its number of links is 2.

The data block of the No. 2 i-node is also only one block, and is located in the second block of the disk data zone, and the disk block number is 9. The length of the data stored therein is 74 bytes, which is the byte length of the hello.c file.

The disk block 8 (0x2000 - 0x23ff, 1 KB) is the data of the root i-node, and there are three directory item structure information (48 bytes), as shown in Table 12-4.

Table 12-4 Data content of root i-node

No.	I-node No.	File Name
1	0x0001	0x2e （.）
2	0x0001	0x2e,0x2e （..）
3	0x0002	0x68,0x65,0x6c,0x6c,0x6f,0x2e,0x63 （hello.c）

Disk block 9 (0x2400 - 0x27ff, 1 KB) is the content of the hello.c file. It contains 74 bytes of text information.

12.2 buffer.c

From this section, we will explain and annotate the programs in the fs/ directory one by one. According to the description in Section 1 of this chapter, the programs in this chapter can be divided into four parts: (1) cache management; (2) file underlying operations; (3) file data access; and (4) file high-level access control. Here we first describe the high-speed buffer (or buffer cache) management procedure of part 1. This section contains only one program buffer.c.

12.2.1 Function

Below we first explain the specific location of the cache in physical memory, and then describe the process of its initialization, Then we give the structure of the buffer cache, the linked list structure and the hash table structure used. Then, the important buffer block acquisition function gltblk() and the block read function bread() are described in detail. Finally, the buffer cache access process and the synchronous operation method are explained from a system perspective.

1. Buffer cache physical location

The buffer.c program is used to operate and manage the buffer cache (pool). The buffer cache is located between the kernel code block and the main memory area, as shown in Figure 12-15. The buffer cache acts as a bridge between the block device and other programs in the kernel. In addition to the block device driver, if the kernel program needs to access the data in the block device, it needs to go through the cache to operate indirectly.

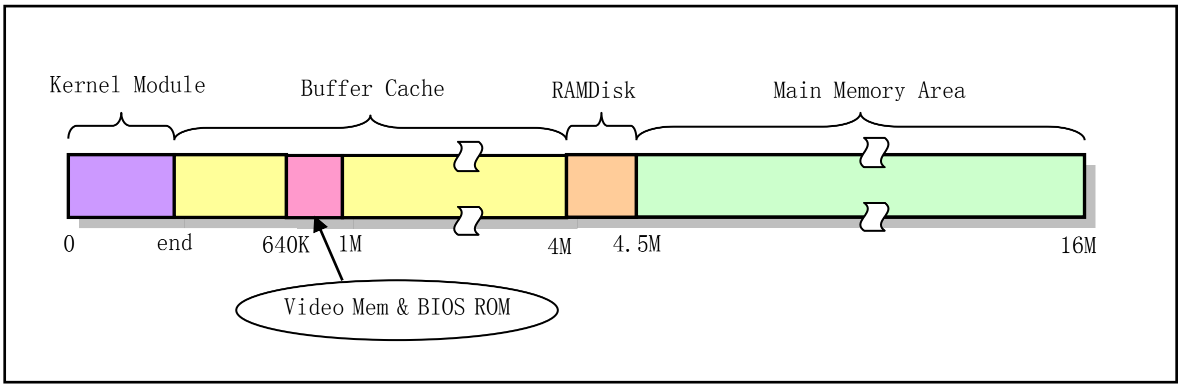


Figure 12-15 The location of the buffer cache in physical memory

The starting position of the cache in the figure begins with the end of the kernel module 'end', and 'end' is an external variable set by the linker ld during the kernel module link. This symbol is not defined in the kernel code. When the linker generates the system module, the ld program sets the address of 'end', which is equal to 'data_start + datasize + bss_size', which is the first effective address after the end of the bss segment, which is the end of the kernel module. In addition, the linker also sets two external variables, 'etext' and 'edata', which represent the first address after the code segment and the first address after the data segment.

2. Buffer cache initialization

The entire cache is divided into 1024-byte buffer blocks, which is exactly the same size as the disk logical block on the block device. The buffer cache is managed by a hash table and a linked list containing all the buffer blocks. During the buffer initialization process, the initialization program starts from both ends of the entire buffer, and sets the buffer block header structure and the corresponding buffer block respectively, as shown in Figure 12-16. The high end of the buffer is divided into buffer blocks of 1024 bytes, and the buffer header structure buffer_head (include/linux/fs.h, line 68) corresponding to each buffer block is respectively established at the low end. This header structure is used to describe the attributes of the corresponding buffer block and is used to join all buffer headers into a linked list. This setting operation continues until there is not enough memory in the buffer to divide the buffer block.

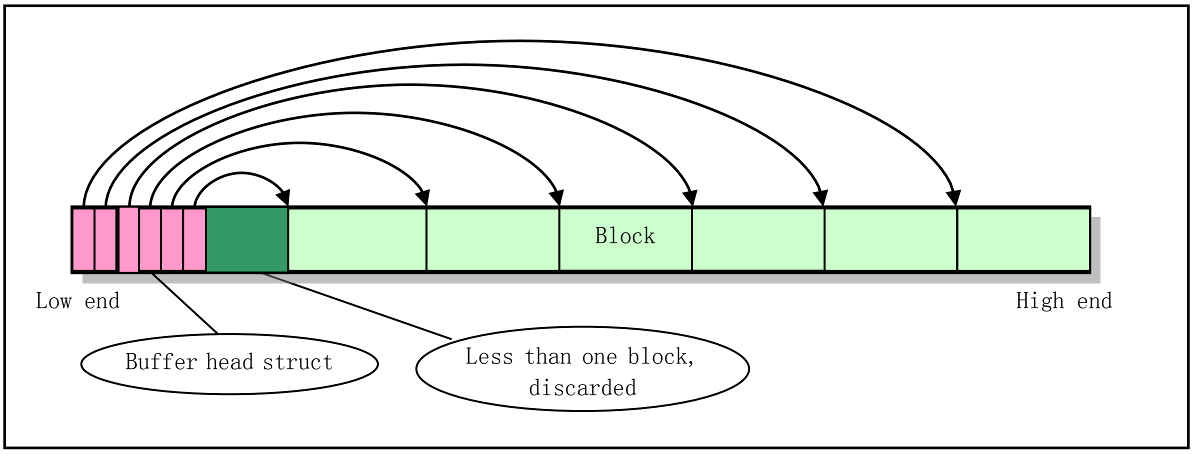


Figure 12-16 Initialization of the buffer cache

3. Buffer cache structure and linked list

The buffer_head of all buffer blocks is linked into a doubly linked list structure, as shown in Figure 12-17. The free_list pointer in the figure is the head pointer of the linked list, pointing to the first "most idle" buffer block in the free block list, which is the least recently used buffer block. The back pointer b_prev_free of the buffer block points to the last buffer block in the buffer block list, that is, the buffer block that has just been used recently. The buffer block header data structure is:

```
struct buffer_head {
        char * b_data;                       // Points to buffer block data area (1024 bytes)
        unsigned long b_blocknr;             // Block No.
        unsigned short b_dev;                // Device no.of the data source (0 = free).
        unsigned char b_uptodate;            // Indicates if the data has been updated.
        unsigned char b_dirt;                // Modified flag: 0 -clean, 1 -modified (dirty).
        unsigned char b_count;               // The nr of users who use the block.
        unsigned char b_lock;                // Whether the block is locked. 0-ok, 1-lock
        struct task_struct * b_wait;         // Point to the task waiting for the buffer.
        struct buffer_head * b_prev;         // The previous block on the hash queue.
        struct buffer_head * b_next;         // The next block on the hash queue.
        struct buffer_head * b_prev_free;    // The previous block on the free list.
        struct buffer_head * b_next_free;    // The next block on the free list.
};
```

The field b_lock is a lock flag indicating that the driver is modifying the contents of the buffer block, so the buffer block is busy and is being locked. This flag is independent of other flags of the buffer block and is mainly used in the blk_drv/ll_rw_block.c program to lock the buffer block when updating the data information in the buffer block. Because when updating the data in the buffer block, the current process will voluntarily go to sleep waiting, so that other processes have the opportunity to access the buffer block. Therefore, in order to prevent other processes from using the data, it is necessary to lock the buffer block before going to sleep.

The field b_count is the count used by the buffer manager, indicating the number of times the corresponding buffer block is being used (referenced) by each process, so this field is used for the program reference count management of the buffer block, and is also independent of other flags of the buffer block. When the reference count is not 0, the buffer manager cannot release the corresponding buffer block. A free block is a block with b_count = 0. When b_count = 0, it means that the corresponding buffer block is not used (free), otherwise it is being used. For the block of the program application, if the buffer manager can obtain the existing specified block from the hash table, the b_count of the block is incremented by 1 (b_count++). If the buffer block is an unused block that is re-applied, the b_count in its header structure is set equal to one. When a program releases its reference to a block, the number of references to that block is decremented accordingly (b_count--). Since the flag b_lock indicates that other programs are using and locking the specified buffer block, the b_count must be greater than 0 for the buffer block where b_lock is set.

The field b_dirt is a dirty flag indicating whether the content in the buffer block has been modified to be different from the corresponding data block on the block device (delayed write). The field b_uptodate is a data update (valid) flag indicating whether the data in the buffer block is valid. Both flags are set to 0 when the block is initialized or released, indicating that the buffer block is invalid at this time. B_dirt = 1, b_uptodate = 0 when data is written to the buffer but has not yet been written to the device. The data becomes valid when the data is written to the block device or just after reading the buffer block from the block device, ie b_uptodate = 1. Please note that there is a special case where both b_dirt and b_uptodate are 1 when a new device buffer block is applied, indicating that the data in the buffer block is different from the block device, but the data is still valid (updated).

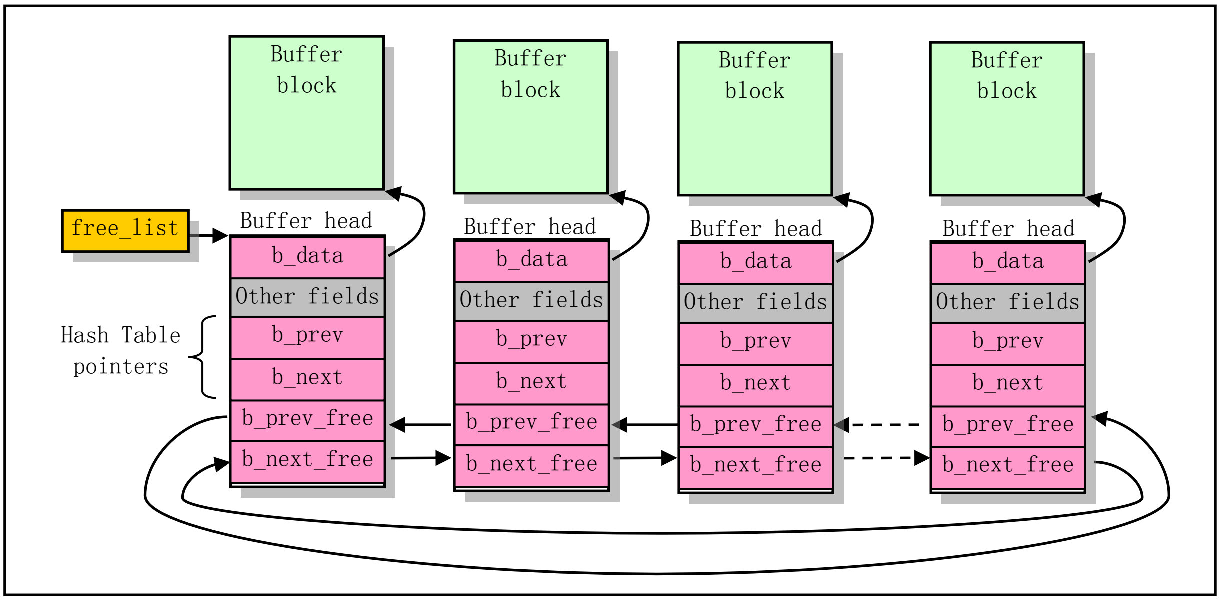


Figure 12-17 Bidirectional circular linked free list of all buffer blocks

The "other fields" in the buffer header structure in the figure include the block device number and the logical block number of the buffered data, and these two fields uniquely determine the block device and its data corresponding to the data in the buffer block. There are also several status flags: the data valid (update) flag, the modified flag, the number of processes the data is used, and whether the buffer block is locked.

When the kernel code uses the buffer block in the buffer cache, it specifies the device number (dev) and the logical block number of the data, and operates by calling the buffer block read function bread(), bread_page(), or breada(). These functions use the buffer search management function getblk() to find matching or free blocks in all buffer blocks. This function will be highlighted below. When the system releases the buffer block, you need to call the brelse() function. The call hierarchy of all these buffer block data access and management functions can be described in Figures 12-18.

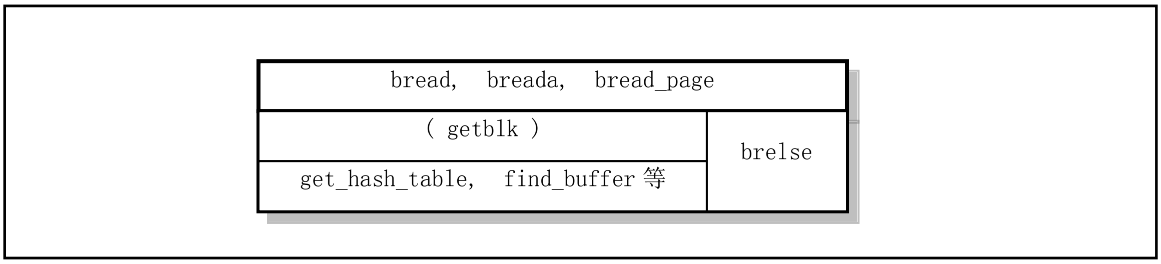


Figure 12-18 Hierarchical relationship between buffer management functions

4. Hash queue of the buffer cache

In order to quickly and efficiently find out in the buffer cache whether the requested data block has been read into the buffer, the buffer.c program uses a hash array table structure with 307 buffer_head pointer entries. The hash function used by the hash table is a combination of the device number and the logical block number. The specific hash function used in the program is: (device number ^ logical block number) Mod 307. In Figure 12-17, the pointers b_prev and b_next are bidirectional links used in hash tables for hashing between multiple buffer blocks on the same item, that is, the buffer blocks with the same hash value calculated by the hash function are linked in the same array item on the linked list. For details on how the buffer block operates on the hash queue, see the description in Chapter 3 of the Unix Operating System Design. For the state of a dynamically changing hash table structure at a certain time, see Figure 12-19.

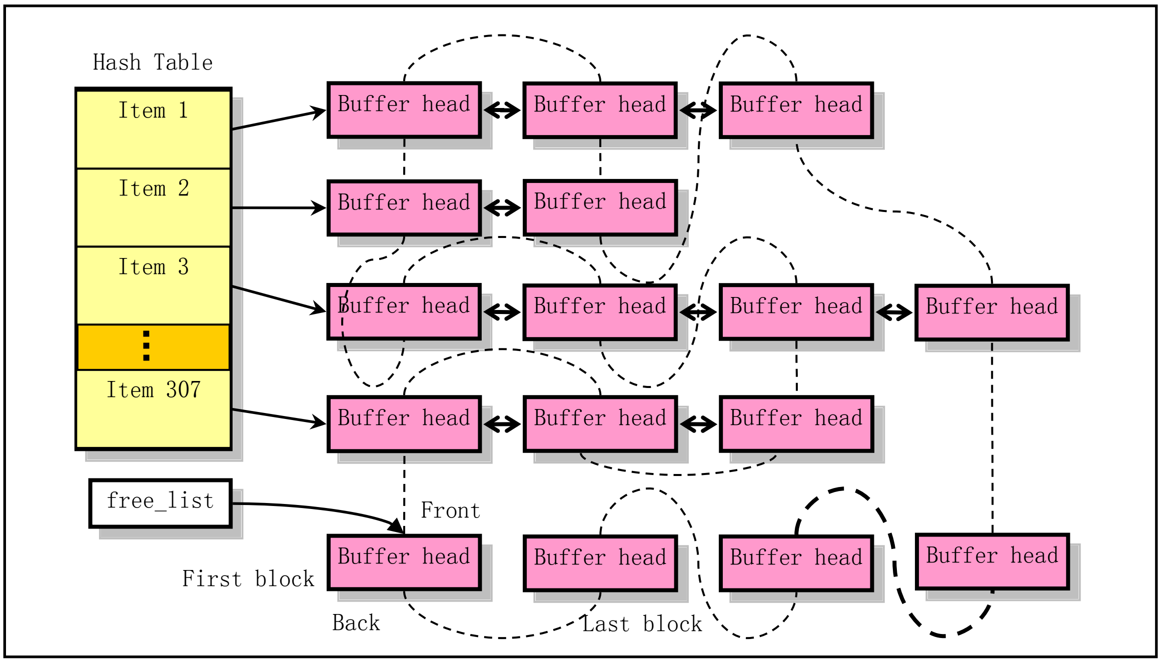


Figure 12-19 Schematic diagram of buffer block hash queue at a certain moment

Wherein, the double arrow horizontal line indicates a bidirectional link pointer that is hashed between the buffer block header structures in the same hash entry. The dotted line represents a bidirectional circular linked list of all the buffer blocks in the buffer cache (the so-called free linked list). The free_list is the head pointer at the most free buffer block of the list. In fact, this doubly linked list is a least recently used (LRU) linked list. Below we will explain the buffer block search function getblk() in detail.

5. Buffer block get function

The three functions mentioned above all call getblk() at execution time to get the appropriate free buffer block. The function first invokes the get_hash_table() function to search the hash table queue for the buffer block of the specified device number and logical block number. If the specified block exists, it immediately returns a pointer to the corresponding block header structure; If it doesn't exist, we start from the free-list header and scan the free-list to find a free buffer. In the search process, we also compare the found free buffer blocks, and according to the weights given by the combination of the modified flag and lock flag, it is determined which free block is most suitable. If the found free block is neither modified nor locked, then we don't have to keep looking. If no free block is found at this time, the current process is put to sleep, and it is searched again when it continues to execute. If the free block is locked, the process also needs to go to sleep, waiting for other (block driver) to unlock. If the buffer block is occupied by other processes during sleep waiting, then just start searching for the buffer block again. Otherwise, it is judged whether the buffer block has been modified, and if so, the block is written to the disk and waiting for the block to be unlocked. At this point, if the buffer block is once again occupied by another process, then once again, it is completely abandoned, so we have to start executing getblk() again.

After experiencing the above toss, there may be another unexpected situation at this time, that is, while we are sleeping, other processes may have added the buffer block we need to the hash queue, so we need to search for the last time the hash queue here. If we really find the buffer block we need in the hash queue, then we have to make the above judgment on the buffer block we found, so again, we need to start executing getblk() again.

Finally, we found a free buffer block that was not used by the process, was not locked, and was clean (the modified flag was not set). So we set the number of references to the block, reset several other flags, and then remove the buffer header structure of the block from the free-list queue. After the device number and corresponding logical number to which the buffer block belongs are set, it is inserted into the header of the corresponding table entry of the hash table and linked to the end of the free-list queue. Since the search for free blocks starts from the free-list header, this operation of first removing from the free-list queue and using the most recently used buffer block and then reinserting into the tail of the free-list achieves the least recently used LRU algorithm. Finally, the pointer to the buffer block header is returned. The entire getblk() process can be seen in Figure 12-20.

From the above analysis, it can be seen that each time the function acquires a new free buffer block, it will move it to the end of the linked list pointed to by the free_list header pointer. That is, the closer the buffer block is to the end of the linked list, the closer time it is used. Therefore, if the corresponding buffer block is not found in the hash table, the search will start from the free_list header when searching for a new free buffer block. It can be seen that the algorithm for the kernel to obtain the buffer block uses the following strategy:

- If the specified buffer block exists in the hash table, it indicates that the available buffer block has been obtained, so it returns directly;
- Otherwise, we need to start the search from the free_list header in the linked list, that is starting from the least recently used buffer block.

Therefore, the most ideal situation is to find a buffer block that is completely free, that is, a buffer block with b_dirt and b_lock flags of 0; But if these two conditions are not met, then we need to calculate a value based on the b_dirt and b_lock flags. Because device operations are often time consuming, we need to increase the weight of b_dirt in your calculations. Then we wait on the buffer block with the smallest result value (if the buffer block is already locked). Finally, when the flag b_lock is 0, it indicates that the original content of the buffer block that has been waiting has been written to the block device. So getblk() gets a free buffer block.

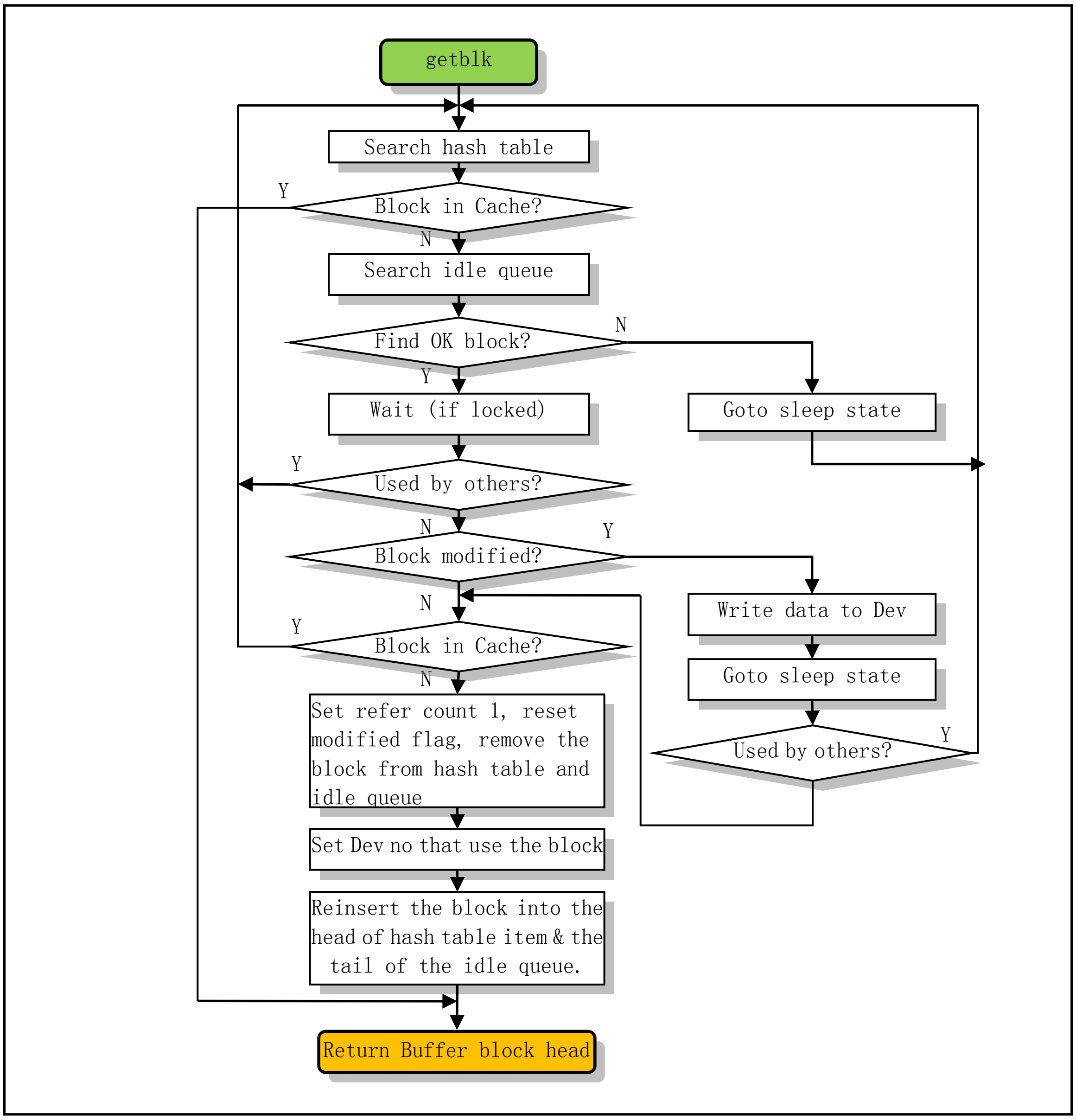


Figure 12-20 Getblk() function execution flow chart

6. Buffer block read function

From the above processing we can see that the buffer block returned by getblk() may be a new free block, or it may just be a buffer block containing the data we need, and it already exists in the buffer cache. Therefore, for the read data block operation (bread()), it is necessary to judge the update flag of the buffer block to see if the included data is valid. If it is valid, the data block can be directly returned to the applied program. Otherwise, you need to call the device's low-level block read-write function (ll_rw_block()), and at the same time let yourself go to sleep, waiting for the data to be read into the buffer block, and after waking up, judge whether the data is valid. If it is valid, this data can be returned to the application program, otherwise the read operation to the device fails and no data is fetched. Thus, the buffer block is released and a NULL value is returned. Figure 12-21 is a block diagram of the bread() function. The breada() and bread_page() functions are similar to the bread() function.

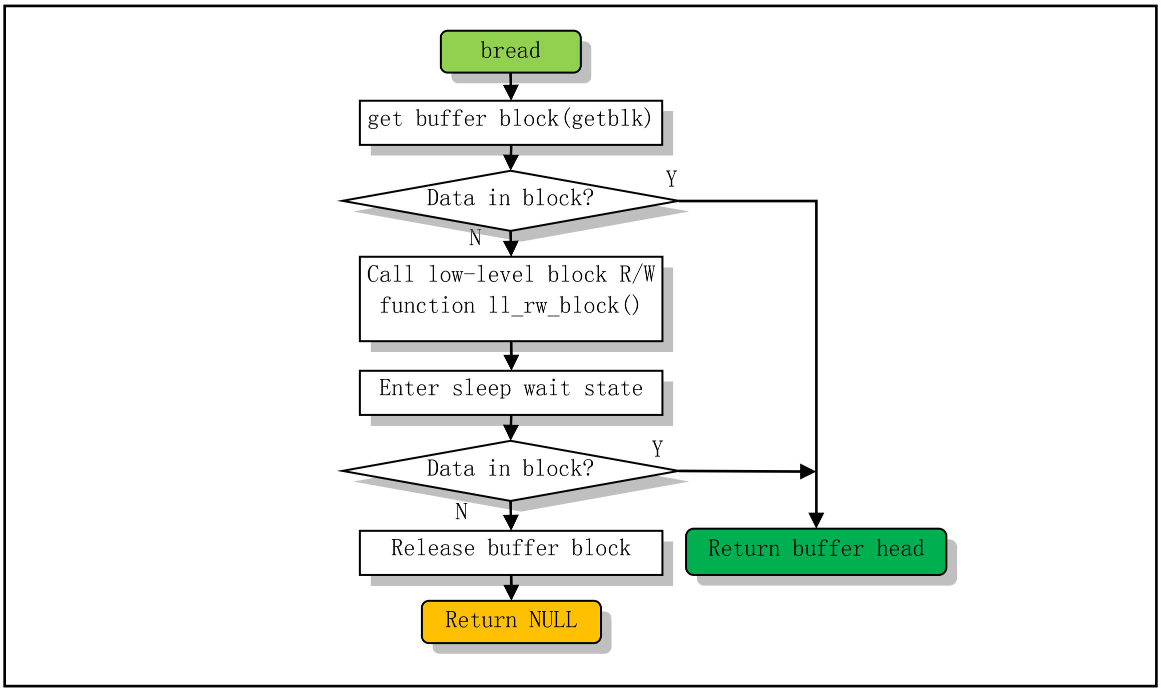


Figure 12-21 Bread() function execution flow diagram

When the program no longer needs to use data from a buffer block, the brelse() function can be invoked to free the buffer block and wake up the process that is asleep due to waiting for the buffer block. Note that the buffer blocks in the free list are not all free. Therefore, it can be used only when the disk is refreshed, unlocked, and no other process references (reference count = 0).

7. buffer access method and synchronization operation

In summary, the high-speed buffer plays an important role in improving access efficiency and increasing data sharing for block devices. In addition to the driver, the read and write operations of other upper-level programs on the block device by the kernel need to be implemented indirectly through the cache manager. The main link between them is through the bread() function in the cache manager and the low-level interface function ll_rw_block() of the block device. If the upper program wants to access the block device data, it applies to the buffer manager through bread(). If the required data is already in the buffer cache, the buffer manager will return the data directly to the program. If the required data is not yet in the buffer, the buffer manager will apply to the block device driver through ll_rw_block(), and let the process corresponding to the program sleep and wait. After the block device driver puts the specified data into the cache, the buffer manager returns the data to the upper program, as shown in Figure 12-22.

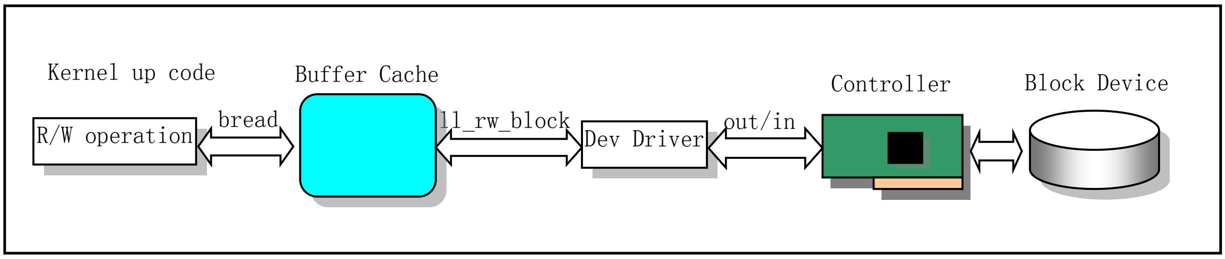


Figure 12-22 Kernel block device access operation

For update and synchronization operations, its main function is to make some buffer block contents in memory consistent with information on block devices such as disks. The main function of sync_inodes() is to match the i-node information in the inode_table with that on the disk, but it needs to go through the intermediate link of the system buffer cache. In fact, any synchronization operation is divided into two phases:

(1) The data structure information is synchronized with the buffer block in the cache buffer and is independently responsible for the driver;

(2) The synchronization problem between the data block and the corresponding block of the disk in the buffer cache is handled by the buffer management program here.

The sync_inodes() function does not deal directly with the disk. It can only advance to the buffer, which is only responsible for synchronizing with the information in the buffer. The rest of the operation requires the buffer manager to be responsible for execution. In order for sync_inodes() to know which i-nodes are different from those on the disk, you must first make the contents of the buffer consistent with the contents on the disk. This way sync_indes() knows which disk inodes need to be modified and updated by comparing with the latest data in the buffer of the current disk. Finally, the second synchronization operation of the buffer cache and the disk device is performed, so that the data in the memory is truly synchronized with the data in the block device.

12.2.2 **Code annotation**

Program 12-1 linux/fs/buffer.c

```
/*
 *  linux/fs/buffer.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  'buffer.c' implements the buffer-cache functions. Race-conditions have
 * been avoided by NEVER letting a interrupt change a buffer (except for the
 * data, of course), but instead letting the caller do it. NOTE! As interrupts
 * can wake up a caller, some cli-sti sequences are needed to check for
 * sleep-on-calls. These should be extremely quick, though (I hope).
 */

/*
 * NOTE! There is one discordant note here: checking floppies for
 * disk change. This is where it fits best, I think, as it should
 * invalidate changed floppy-disk-caches.
 */

// <stdarg.h> Standard parameter header file. Define a list of variable parameters in the
//     form of macros. It mainly describes one type (va_list) and three macros (va_start,
//     va_arg and va_end) for the vsprintf, vprintf, and vfprintf functions.
// <linux/config.h> Kernel configuration header file. Define keyboard language and hard
//     disk type (HD_TYPE) options.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the
//     data of the initial task 0, and some embedded assembly function macro statements
//     about the descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
//     commonly used functions of the kernel.
```

```
   // <asm/system.h> System header file. An embedded assembly macro that defines or
   //     modifies descriptors/interrupt gates, etc. is defined.
   // <asm/io.h> Io header file. Defines the function that operates on the io port in the
   //     form of a macro's embedded assembler.
21 #include <stdarg.h>
22
23 #include <linux/config.h>
24 #include <linux/sched.h>
25 #include <linux/kernel.h>
26 #include <asm/system.h>
27 #include <asm/io.h>
28
   // The value of the variable 'end' below is generated by the linker ld at compile time to indicate
   // the end position of the kernel module, as shown in Figure 12-15. We can find this value from
   // the System.map file generated when the kernel was compiled. It is used here to indicate that
   // the buffer cache starts at the end of the kernel code.
   // The buffer_wait variable on line 33 is the queue head pointer for the task that is asleep
   // while waiting for the free buffer block. It has a different effect than the b_wait pointer
   // in the buffer block header structure. When a task requests a buffer block and happens to
   // encounter a lack of available free buffer blocks, the task is added to the buffer_wait sleep
   // wait queue. The b_wait is a wait queue header that is dedicated to the task waiting for the
   // specified buffer block (ie, the buffer block corresponding to b_wait).
29 extern int end;
30 struct buffer_head * start_buffer = (struct buffer_head *) &end;
31 struct buffer_head * hash_table[NR_HASH];         // NR_HASH = 307
32 static struct buffer_head * free_list;            // free block link list head pointer.
33 static struct task_struct * buffer_wait = NULL;   // queue for waiting a free block.

   // The following defines the number of buffer blocks contained in the system buffer cache. Here,
   // NR_BUFFERS is a constant symbol defined on line 48 in the linux/fs.h file, which is defined
   // as the variable nr_buffers, which is declared as a global variable on line 172 in the fs.h
   // file. Mr. Linus wrote the code in such a way as to implicitly indicate that nr_buffers is
   // a "constant" that does not change after the kernel buffer is initialized. It will be set
   // in the initialization function buffer_init() (line 371).
34 int NR_BUFFERS = 0;                               // The number of buffer blocks.
35
   //// Wait for the specified buffer block to unlock.
   // If the specified buffer block bh has been locked, then we let the process sleep uninterrupted
   // in the buffer queue b_wait. When the buffer block is unlocked, all processes on its waiting
   // queue will be woken up. Although it goes to sleep after the interrupt is disabled (cli),
   // doing so does not affect the response to interrupts in other process contexts. Because each
   // process saves the value of the register EFLAGS in its own TSS segment, the value of the current
   // EFLAGS in the CPU changes as the process switches. A process that uses sleep_on() to go to
   // sleep needs to wake up explicitly with wake_up().
36 static inline void wait_on_buffer(struct buffer_head * bh)
37 {
   // If it has been locked, the process goes to sleep and waits for it to unlock.
38         cli();                            // disable int.
39         while (bh->b_lock)
40                 sleep_on(&bh->b_wait);
41         sti();                            // enable int.
42 }
43
```

```
//// Device data synchronization
// Synchronize data in device and cache. sync_inodes() is defined in file inode.c, line 59.
// The function first calls the i-node synchronization function to write all modified i-nodes
// in the memory i-node table to the buffer cache. Then, the entire cache buffer is scanned,
// and a write disk request is generated for the buffer block that has been modified, and the
// buffered data is written into the disk, so that the data in the cache is synchronized with
// the device.
int sys_sync(void)
{
        int i;
        struct buffer_head * bh;

        sync_inodes();                          /* write out inodes into buffers */
        bh = start_buffer;                      // points to the beginning of cache.
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
                wait_on_buffer(bh);     // Wait for the buffer to be unlocked (if locked).
                if (bh->b_dirt)
                        ll_rw_block(WRITE,bh);  // Generate a write device block request.
        }
        return 0;
}

//// Synchronize the cached data with data on the specified device.
// This function first searches all buffer blocks in the buffer cache and writes to the disk
// for the buffer of the specified device dev if its data has been modified (synchronous
// operation). The in-memory i-node table data is then written to the buffer cache. Then perform
// the same write operation as above for the specified device dev again.
int sync_dev(int dev)
{
        int i;
        struct buffer_head * bh;

// First, the data synchronization operation is performed on the device specified by the
// parameter, so that the data on the device is synchronized with the data in the buffer cache.
// The method is to scan all buffer blocks in the cache, and to check whether the buffer block
// of the specified device dev is locked, and if it is locked, sleep and wait for it to be unlocked.
// Then it is judged whether the buffer block is still the buffer block of the designated device,
// and it has been modified (the b_dirt flag is set), and if so, the write operation is performed
// on it. Because the buffer block may have been released or used for other purposes during
// our sleep, it is necessary to check again if the buffer block is still a buffer block for
// the specified device before proceeding.
        bh = start_buffer;                   // points to the beginning of cache.
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
                if (bh->b_dev != dev)        // ignore blocks that doesn’t belong to dev.
                        continue;
                wait_on_buffer(bh);          // wait for the buffer to be unlocked.
                if (bh->b_dev == dev && bh->b_dirt)
                        ll_rw_block(WRITE,bh);
        }
// Then write the i-node data into the buffer cache, and synchronize the inode in the inode_table
// with the information in the buffer cache. The data in the cache is then synchronized again
// with the data in the device after it has been updated. The two-pass sync operation is here
// to improve the efficiency of kernel execution. The first pass of the buffer synchronization
// can make many "dirty blocks" in the memory clean, so that the synchronization operation of
// the i-node can be performed efficiently. This second buffer synchronization synchronizes
// the buffer blocks that are dirty due to the i-node synchronization operation with the data
// in the device.
        sync_inodes();
        bh = start_buffer;
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
                if (bh->b_dev != dev)
                        continue;
                wait_on_buffer(bh);
                if (bh->b_dev == dev && bh->b_dirt)
                        ll_rw_block(WRITE,bh);
        }
        return 0;
}

//// Invalidate the data of the specified device in the buffer cache.
// Scan all buffer blocks in the cache. For the buffer block of the specified device, reset
// its valid (update) flag and the modified flag.
void inline invalidate_buffers(int dev)
{
        int i;
        struct buffer_head * bh;

        bh = start_buffer;
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
                if (bh->b_dev != dev)          // Ignore blocks that doesn't belong to dev.
                        continue;
                wait_on_buffer(bh);            // wait for the buffer to be unlocked.
// Since the process has performed a sleep wait, it is necessary to determine whether the buffer
// still belongs to the specified device.
                if (bh->b_dev == dev)
                        bh->b_uptodate = bh->b_dirt = 0;
        }
}

/*
 * This routine checks whether a floppy has been changed, and
 * invalidates all buffer-cache-entries in that case. This
 * is a relatively slow routine, so we have to try to minimize using
 * it. Thus it is called only upon a 'mount' or 'open'. This
 * is the best way of combining speed and utility, I think.
 * People changing diskettes in the middle of an operation deserve
 * to loose :-)
 *
 * NOTE! Although currently this is only for floppies, the idea is
 * that any additional removable block-device will use this routine,
 * and that mount/open needn't know that floppies/whatever are
 * special.
 */
//// Check if the disk has changed and invalidate the buffer block if it has been replaced.
void check_disk_change(int dev)
{
        int i;

// First check if it is a floppy device, because the code now only supports floppy removable
// media. If not, quit. Then test if the floppy disk has been replaced, and if not, exit. The
// function floppy_change() is on line 139 of blk_drv/floppy.c.
        if (MAJOR(dev) != 2)
                return;
        if (!floppy_change(dev & 0x03))
                return;
// The floppy disk has been replaced, so the buffer blocks occupied by the i-node bitmap and
// the logical block bitmap of the corresponding device is released; and the i-node of the device
// and the cache block occupied by the data block information are invalidated.
        for (i=0 ; i<NR_SUPER ; i++)
                if (super_block[i].s_dev == dev)
                        put_super(super_block[i].s_dev);
        invalidate_inodes(dev);
        invalidate_buffers(dev);
}

// Below is the hash function definition and the calculation macro for the hash table entry.
// The main role of the hash table is to reduce the time it takes to find elements. By establishing
// a correspondence between the storage location of the element and the keyword (hash function),
// we can immediately find the specified element directly through the function calculation.
// The guiding principle for building a hash function is to try to ensure that the probability
// of hashing to any array item is substantially equal. There are many ways to create a hash
// function. Here Linux 0.12 mainly uses the most commonly used keyword residual remainder
// method. Because the buffer block we are looking for has two conditions, the device number
// 'dev' and the buffer block number 'block', the hash function designed must contain these
// two key values. The XOR operation of the two keywords here is just one way to calculate the
// key value. Performing a modulo operation (%) on the key values ensures that the values
// calculated by the function are all within the range of the function array items.
#define _hashfn(dev,block) (((unsigned)(dev^block))%NR_HASH)
#define hash(dev,block) hash_table[_hashfn(dev,block)]

//// Remove the buffer block from the hash queue and the free list.
// The hash queue is a doubly linked list structure, and the buffer block free list is a
// bidirectional circular linked list structure.
static inline void remove_from_queues(struct buffer_head * bh)
{
/* remove from hash-queue */
        if (bh->b_next)
                bh->b_next->b_prev = bh->b_prev;
        if (bh->b_prev)
                bh->b_prev->b_next = bh->b_next;
// If the buffer block is the first block of the queue, then the corresponding entry of the
// hash table points to the next buffer block in the queue.
        if (hash(bh->b_dev,bh->b_blocknr) == bh)
                hash(bh->b_dev,bh->b_blocknr) = bh->b_next;
/* remove from free list */
        if (!(bh->b_prev_free) || !(bh->b_next_free))
                panic("Free block list corrupted");
        bh->b_prev_free->b_next_free = bh->b_next_free;
        bh->b_next_free->b_prev_free = bh->b_prev_free;
```

```
    // If the free list header points to this buffer block, it is directed to the next buffer block.
145         if (free_list == bh)
146                 free_list = bh->b_next_free;
147 }
148 
    //// Insert the buffer block at the end of the free list and put it into the hash queue.
149 static inline void insert_into_queues(struct buffer_head * bh)
150 {
151 /* put at end of free list */
152         bh->b_next_free = free_list;
153         bh->b_prev_free = free_list->b_prev_free;
154         free_list->b_prev_free->b_next_free = bh;
155         free_list->b_prev_free = bh;
156 /* put the buffer in new hash-queue if it has a device */
    // Note that when an item of the hash table is inserted for the first time, the hash() value
    // is definitely NULL, so bh->b_next obtained on line 161 is definitely NULL. So on line 163,
    // we should only assign b_prev a bh value if bh->b_next is not NULL. That is, before the 163th
    // line, a sentence "if (bh->b_next)" should be added. The error was corrected after the kernel
    // 0.96 version.
157         bh->b_prev = NULL;
158         bh->b_next = NULL;
159         if (!bh->b_dev)
160                 return;
161         bh->b_next = hash(bh->b_dev,bh->b_blocknr);
162         hash(bh->b_dev,bh->b_blocknr) = bh;
163         bh->b_next->b_prev = bh;          // "if (bh->b_next)" should be added before this.
164 }
165 
    //// Use hash table to look up the buffer block for a given device and a specified block number
    // in the buffer cache. Returns the buffer block pointer if found, otherwise returns NULL.
166 static struct buffer_head * find_buffer(int dev, int block)
167 {
168         struct buffer_head * tmp;
169 
    // Search the hash table for buffer block with the specified device number and block number.
170         for (tmp = hash(dev,block) ; tmp != NULL ; tmp = tmp->b_next)
171                 if (tmp->b_dev==dev && tmp->b_blocknr==block)
172                         return tmp;
173         return NULL;
174 }
175 
176 /*
177  * Why like this, I hear you say... The reason is race-conditions.
178  * As we don't lock buffers (unless we are readint them, that is),
179  * something might happen to it while we sleep (ie a read-error
180  * will force it bad). This shouldn't really happen currently, but
181  * the code is ready.
182  */
    //// Use the hash table to find the buffer block.
    // Use the hash table to find the specified buffer block in the buffer cache. If found, the
    // block is locked and the block header pointer is returned.
183 struct buffer_head * get_hash_table(int dev, int block)
184 {
        struct buffer_head * bh;

        for (;;) {
// Look for the buffer block for the given device and the specified block in the buffer cache.
// If it is not found, return NULL and exit.
                if (!(bh=find_buffer(dev,block)))
                        return NULL;
// If the desired block is found, its reference count is incremented by one, and then waiting
// for the block to be unlocked (if it has been locked). Since it has gone through a sleep state,
// it is necessary to verify the correctness of the block and return the buffer head pointer.
// If the device or block number of the buffer block changes during sleep, its reference count
// is revoked and the search is made again.
                bh->b_count++;
                wait_on_buffer(bh);       // Wait for the buffer to be unlocked (if locked).
                if (bh->b_dev == dev && bh->b_blocknr == block)
                        return bh;
                bh->b_count--;
        }
}

/*
 * Ok, this is getblk, and it isn't very clear, again to hinder
 * race-conditions. Most of the code is seldom used, (ie repeating),
 * so it should be much more efficient than it looks.
 *
 * The algoritm is changed: hopefully better, and an elusive bug removed.
 */
// The following macro is used to check the modified flag and lock flag of the block at the
// same time, and the weight of the modified flag is defined to be larger than the lock flag.
#define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)

//// Get the specified block in the buffer cache.
// Check if the buffer block of the specified (device no. & block no.) is already in the cache.
// If the specified block is already in the buffer cache, the corresponding buffer block header
// is returned and exited; if not, a new entry corresponding to the device no. and block no.
// needs to be set in the cache and the corresponding buffer header pointer is returned.
struct buffer_head * getblk(int dev,int block)
{
        struct buffer_head * tmp, * bh;

repeat:
// Search the hash table, if the specified block is already in the buffer cache, return the
// corresponding block head pointer and exit.
        if (bh = get_hash_table(dev,block))
                return bh;
// Otherwise, the free list is scanned to find the free block. First let 'tmp' point to the
// first free block header of the free list, and then loop to perform subsequent operations.
        tmp = free_list;
        do {
// If the buffer block is being used (the reference count is not equal to 0), continue scanning
// the next item. For a block with b_count=0, that is, a block that is not currently referenced
// in the cache, it is not necessarily clean (b_dirt=0) or unlocked (b_lock=0). Therefore, we
// still need to continue the following judgments and choices. For example, for a block with
```

```
    // b_count=0, when a task rewrites the content of a block and then releases it, the block b_count
    // is still 0, but b_lock is not equal to 0; When a task executes breada() to pre-read several
    // blocks, it will decrement b_count as soon as ll_rw_block() is called. However, at this time,
    // the hard disk access operation may still be in progress, so b_lock=1, but b_count=0.
215                     if (tmp->b_count)
216                             continue;
    // If the buffer block header pointer bh is empty, or the weight of the flag (modification,
    // lock) of the block referred to by the current tmp is less than the weight of the bh's flag,
    // then bh is pointed to the tmp's block header. If the buffer block header pointed by the tmp
    // indicates that the buffer block is neither modified nor the lock flag is set (ie, BADNESS()
    // = 0), it means that the corresponding cache block has been obtained on the specified device,
    // and the loop is exited. Otherwise we will continue to execute this loop and see if we can
    // find a buffer with the smallest BADNESS() value.
217                     if (!bh || BADNESS(tmp)<BADNESS(bh)) {
218                             bh = tmp;
219                             if (!BADNESS(tmp))
220                                     break;
221                     }
222 /* and repeat until we find something good */
223             } while ((tmp = tmp->b_next_free) != free_list);

    // If the loop check finds that all buffer blocks are being used (all buffer block header reference
    // counts are >0), sleep waits for free buffer blocks to be available. The process is explicitly
    // woken up when a free buffer block is available, and then we jump to the beginning of the
    // function to re-look for the free buffer block.
224             if (!bh) {
225                     sleep_on(&buffer_wait);
226                     goto repeat;                    // jump to line 210.
227             }
    // Execution here, we have found a suitable free buffer block. Then wait for the buffer to unlock
    // (if it has been locked). If the buffer is used by other tasks during our sleep phase, we
    // have to repeat the above search process again.
228             wait_on_buffer(bh);
229             if (bh->b_count)                        // occupied again ??
230                     goto repeat;
    // If the buffer has been modified, write the data to the disk and wait for the buffer to unlock
    // again. Similarly, if the buffer is used by other tasks again, the above search process is
    // repeated.
231             while (bh->b_dirt) {
232                     sync_dev(bh->b_dev);
233                     wait_on_buffer(bh);
234                     if (bh->b_count)                // occupied again ??
235                             goto repeat;
236             }
237 /* NOTE!! While we slept waiting for this block, somebody else might */
238 /* already have added "this" block to the cache. check it */
    // Check if the specified buffer block has been added to the hash table while we are sleeping.
    // If so, repeat the above search process again.
239             if (find_buffer(dev,block))
240                     goto repeat;
241 /* OK, FINALLY we know that this buffer is the only one of it's kind, */
242 /* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
    // So let us occupy this buffer block, set the reference count to 1, reset the modified flag
// and the valid (update) flag.
        bh->b_count=1;
        bh->b_dirt=0;
        bh->b_uptodate=0;
// The buffer block header is first removed from the hash queue and the free block list, and
// the buffer block is used for the specified device and the specified block thereon. The buffer
// block is then reinserted into the free list and the new location of the hash queue, and finally
// the buffer block header is returned.
        remove_from_queues(bh);
        bh->b_dev=dev;
        bh->b_blocknr=block;
        insert_into_queues(bh);
        return bh;
}

//// Release the specified buffer block.
// Wait for the buffer block to unlock. The reference count is then decremented by one, and
// the process waiting for the free buffer block is explicitly awake.
void brelse(struct buffer_head * buf)
{
        if (!buf)                      // Returns if the block head pointer is invalid.
                return;
        wait_on_buffer(buf);
        if (!(buf->b_count--))
                panic("Trying to free free buffer");
        wake_up(&buffer_wait);
}

/*
 * bread() reads a specified block and returns the buffer that contains
 * it. It returns NULL if the block was unreadable.
 */
//// Read data blocks from the device.
// This function first requests a buffer block in the cache based on the specified device number
// dev and block number block. If the buffer block already contains valid data, it directly
// returns the buffer block pointer; otherwise, the specified data block is read from the device
// into the buffer block and the buffer block pointer is returned.
struct buffer_head * bread(int dev,int block)
{
        struct buffer_head * bh;

// Apply a block of buffers in the cache. If the return value is NULL, it means that the kernel
// has an error and it stops. Then we determine if there is data available. Returns if the data
// in the buffer block is valid (updated) and can be used directly.
        if (!(bh=getblk(dev,block)))
                panic("bread: getblk returned NULL\n");
        if (bh->b_uptodate)
                return bh;
// Otherwise we call the read and write ll_rw_block() function of the underlying block device
// to generate a read device block request. Then we wait for the specified data block to be
// read in and wait for the buffer to unlock. After the sleep wakes up, if the buffer has been
// updated, the buffer head pointer is returned and exits. Otherwise, it indicates that the
// read device operation failed, so the buffer is released, NULL is returned, and exit.
```

```
        ll_rw_block(READ,bh);
        wait_on_buffer(bh);
        if (bh->b_uptodate)
                return bh;
        brelse(bh);
        return NULL;
}

//// Copy the memory block.
// Copy a block (1024 bytes) of data from the 'from' address to the 'to' position.
#define COPYBLK(from,to) \
__asm__("cld\n\t" \
        "rep\n\t" \
        "movsl\n\t" \
        ::"c" (BLOCK_SIZE/4),"S" (from),"D" (to) \
        :"cx","di","si")

/*
 * bread_page reads four buffers into memory at the desired address. It's
 * a function of its own, as there is some speed to be got by reading them
 * all at the same time, not waiting for one to be read, and then another
 * etc.
 */
//// Read the one page (4 buffer blocks) on the device to the specified memory address.
// The parameter 'address' is the address where the page data is saved; 'dev' is the specified
// device number; b[4] is an array containing 4 device data block numbers. This function is
// used in the do_no_page() function of the mm/memory.c file (line 428).
void bread_page(unsigned long address,int dev,int b[4])
{
        struct buffer_head * bh[4];
        int i;

// This function is executed 4 times. According to the 4 block numbers placed in the array b[],
// a page is read from the device dev and placed at the specified memory location 'address'.
// For the valid block number given by the parameter b[i], the function first gets the buffer
// block of the specified device and block number from the buffer cache. If the data in the
// buffer block is invalid (not updated) then a read device request is made to read the
// corresponding data block from the device. For the invalid block number b[i], we don't have
// to deal with it. Therefore, this function can freely read 1-4 data blocks according to the
// block number in the specified b[].
        for (i=0 ; i<4 ; i++)
                if (b[i]) {                      // If the block number is valid.
                        if (bh[i] = getblk(dev,b[i]))
                                if (!bh[i]->b_uptodate)
                                        ll_rw_block(READ,bh[i]);
                } else
                        bh[i] = NULL;
// The contents of the 4 buffer blocks are then sequentially copied to the specified address.
// Before we can copy (use) the buffer block, we need to sleep and wait for the buffer block
// to be unlocked (if it is locked). In addition, because we may have slept, we also need to
// check if the data in the buffer block is valid before copying. We also need to release the
// buffer block after copying.
        for (i=0 ; i<4 ; i++,address += BLOCK_SIZE)
```

```
309                 if (bh[i]) {
310                         wait_on_buffer(bh[i]);   // Wait for block to unlock (if it is locked).
311                         if (bh[i]->b_uptodate)   // If the data is valid, copy it.
312                                 COPYBLK((unsigned long) bh[i]->b_data,address);
313                         brelse(bh[i]);           // Release the buffer block.
314                 }
315 }
316
317 /*
318  * Ok, breada can be used as bread, but additionally to mark other
319  * blocks for reading as well. End the argument list with a negative
320  * number.
321  */
    //// Reads some of the specified blocks from the specified device.
    // The number of parameters is variable and is a series of specified block numbers. When the
    // function succeeds, it returns the block header of the first block, otherwise it returns NULL.
322 struct buffer_head * breada(int dev,int first, ...)
323 {
324         va_list args;
325         struct buffer_head * bh, *tmp;
326
    // First take the first parameter (block number) in the variable parameter list. Then get the
    // block of the specified device and block number from the cache. If the buffer block data is
    // invalid (the update flag is not set), a read device data block request is issued.
327         va_start(args,first);
328         if (!(bh=getblk(dev,first)))
329                 panic("bread: getblk returned NULL\n");
330         if (!bh->b_uptodate)
331                 ll_rw_block(READ,bh);
    // Then take the other pre-read block numbers in the variable parameter list in order and do
    // the same as above, but do not use them. Note that there is a bug on line 336. Where bh should
    // be tmp. This bug was not corrected until the 0.96 kernel code. In addition, because this
    // is a pre-read subsequent data block, it only needs to be read into the cache but not used
    // immediately, so the 337th line statement needs to decrement its reference count to release
    // the block (because the getblk() function will increase the block reference count).
332         while ((first=va_arg(args,int))>=0) {
333                 tmp=getblk(dev,first);
334                 if (tmp) {
335                         if (!tmp->b_uptodate)
336                                 ll_rw_block(READA,bh);     // here 'bh' should be 'tmp'.
337                         tmp->b_count--;                    // release the pre-read block.
338                 }
339         }
    // At this point, all parameters in the variable parameter table are processed. Then wait for
    // the first buffer block to be unlocked (if it has been locked). After the wait is over, if
    // the data in the buffer block is still valid, the buffer block header pointer is returned.
    // Otherwise the buffer is released and returns NULL.
340         va_end(args);
341         wait_on_buffer(bh);
342         if (bh->b_uptodate)
343                 return bh;
344         brelse(bh);
345         return (NULL);
```

```
}

//// Buffer initialization function.
// The parameter buffer_end is the end of the buffer cache memory. For systems with 16MB of
// memory, the buffer cache end is set to 4MB. For systems with 8MB of memory, the buffer end
// is set to 2MB. The function sets (initializes) the block header structure and the corresponding
// data block from the start position of the buffer cache at the start_buffer and the buffer_end
// at the end of the buffer respectively until all the memory in the buffer cache is allocated.
void buffer_init(long buffer_end)
{
        struct buffer_head * h = start_buffer;
        void * b;
        int i;

// First, determine the high-end position 'b' of the actual buffer based on the high-end position
// of the buffer provided by the parameter. If the high end of the buffer is equal to 1Mb, since
// the video memory and BIOS are used from 640KB - 1MB, the actual available buffer memory should
// be 640KB at the high end. Otherwise, the high end of the buffer cache memory must be greater
// than 1MB.
        if (buffer_end == 1<<20)
                b = (void *) (640*1024);
        else
                b = (void *) buffer_end;
// The following code is used to initialize the buffer cache, create a free block circular list,
// and get the number of blocks in the system. The operation process is to divide the buffer
// block of 1 KB size from the high end of the buffer, and at the same time, construct the
// buffer_head describing the buffer block at the low end of the buffer, and form the buffer_head
// into a doubly linked list.
// 'h' is a pointer to the block header structure, and 'h+1' is the next block header address
// that points to the memory address consecutively, or can be said to point to the end of the
// h block header. In order to ensure that there is enough memory to store a block header
// structure, the address of the memory block pointed to by 'b' needs to be greater than or
// equal to the end of the block header of 'h', that is, 'b >= h+1' is required.
        while ( (b -= BLOCK_SIZE) >= ((void *) (h+1)) ) {
                h->b_dev = 0;              // device no.
                h->b_dirt = 0;             // dirt flag (block modified flag).
                h->b_count = 0;            // block reference count.
                h->b_lock = 0;             // block lock flag.
                h->b_uptodate = 0;         // Block update flag (or data valid flag).
                h->b_wait = NULL;          // block waiting queue.
                h->b_next = NULL;      // points to next header with same hash value.
                h->b_prev = NULL;      // points to previous header with same hash value.
                h->b_data = (char *) b; // block data pointer.
                h->b_prev_free = h-1;  // point to the previous item in the free-list.
                h->b_next_free = h+1;  // point to the next item in the free-list.
                h++;                   // h points to next new buffer header location.
                NR_BUFFERS++;                  // counting buffer blocks.
// If b is decremented to 1MB, skip 384KB (video memory) and let b point to 0xA0000 (640KB).
                if (b == (void *) 0x100000)
                        b = (void *) 0xA0000;
        }
// Then let h point to the last valid buffer block header; let the free list header point to
// the first buffer block header; b_prev_free of the free list header points to the previous
// header (ie the last header); The next pointer of h points to the first header, so that the
// free list forms a bidirectional ring structure. Finally initialize the hash table, set all
// pointers in the table to NULL.
        h--;                          // h points to the last block header.
        free_list = start_buffer;     // free list header points to first block header.
        free_list->b_prev_free = h;
        h->b_next_free = free_list;
        for (i=0;i<NR_HASH;i++)
                hash_table[i]=NULL;
}

```

12.3 bitmap.c

Starting with this program, we begin to explore the second part of the file system composition, the underlying operational function part of the file system. This part consists of five files, super.c, bitmap.c, truncate.c, inode.c, and namei.c.

The super.c program mainly contains functions for accessing and managing file system superblocks; The bitmap.c program is used to process the logical block bitmap and the i-node bitmap of the file system; The truncate.c program only contains a function truncate() that cuts the file data length to zero; The inode.c program mainly involves the access and management of file system i-node information; The namei.c program is mainly used to complete the function of finding and loading its corresponding i-node information from a given file path name.

According to the order of the functional parts of a file system, we should describe them in the order of the programs given above, but because the super.c program also contains several high-level functions related to file system loading/unloading or system-calls, they need to use functions in several other programs, so we will explain them after introducing the inode.c program.

12.3.1 Function

The purpose and function of the bitmap.c program is simple and clear. It is mainly used to occupy/release the bits in the logical block bitmap or the i-node bitmap according to the usage of the logical block or the i-node structure in the file system. The operation functions of the logic block bitmap are free_block() and new_block(); the operation functions of the i-node bitmap are free_inode() and new_inode().

The function free_block() is used to release the logical block in the data area on the specified device. The specific operation is to reset the bit in the logical block bitmap corresponding to the specified logical block. It first takes the super block of the specified device, and judges the validity of the logical block number according to the range of the device data logic block given in the super block. Then look in the buffer cache to see if the specified logical block is now in the buffer cache. If yes, the corresponding buffer block is released. Next, calculate the logical block number of the data from the beginning of the data zone (counting from 1), and operate on the logical block bitmap to reset the corresponding bit. Finally, in the buffer block containing the corresponding logical block bitmap, we set the modified bit flag according to the logical block number.

The function new_block() is used to request a logical block from the block device, return the logical block number, and set the logical block bitmap bit corresponding to the block. It first gets the superblock of the specified device dev. Then, the entire logical block bitmap is searched for the first bit that is 0. If not found, the disk device space is exhausted and the function returns 0. Otherwise, the first 0 bit position to be found is set to 1, indicating that the corresponding data logic block is occupied. At the same time, the code sets the modified flag of the buffer block in which the logical block bitmap containing the bit is located. Then calculate the disk block number of the data logical block, and apply the corresponding buffer block in the buffer cache, and clear the buffer block. Then set the updated and modified flags for this buffer block. Finally, the buffer block is released for use by other programs and returns the block number (logical block number).

The function free_inode() is used to release the specified i-node and reset the corresponding i-node bitmap bit; New_inode() is used to create a new i-node for device and return a pointer to the new i-node. The main operation process is to obtain an idle i-node entry in the memory i-node table and find an idle i-node from the i-node bitmap. The processing of these two functions is similar to the above two functions, so it will not be described here.

12.3.2 **Code annotation**

Program 12-2 linux/fs/bitmap.c

```
1 /*
2  *  linux/fs/bitmap.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 /* bitmap.c contains the code that handles the inode and block bitmaps */
  // <string.h> String header file. Defines some embedded functions about string operations.
  // <linux/sched.h> The scheduler header file defines the task structure task_struct, the
  //     data of the initial task 0, and some embedded assembly function macro statements
  //     about the descriptor parameter settings and acquisition.
  // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the
  //     commonly used functions of the kernel.
8 #include <string.h>                  // The memset() function is used here.
9
10 #include <linux/sched.h>
11 #include <linux/kernel.h>
12
   //// Clear a block of 1024 bytes of memory at the given address (addr).
   // Input: eax = 0; ecx = length of the block in long words (BLOCK_SIZE/4); edi = specifies the
   // starting address addr.
13 #define clear_block(addr) \
14 __asm__("cld\n\t" \                  // clear direction.
15         "rep\n\t" \                  // repeated execution of stored data (0).
16         "stosl" \
17         ::"a" (0),"c" (BLOCK_SIZE/4),"D" ((long) (addr)):"cx","di")
18
   //// Set the bit at the nr-th bit offset starting at the given address.
   // The 'nr' can be greater than 32! This macro returns the original bit value.
   // Input: %0 -eax (return value); %1 -eax(0); %2 -nr, bit offset; %3 -(addr), content of addr.
   // Line 20 defines a local register variable 'res'. This variable will be saved in the specified
   // EAX register for efficient access and operation. This method of defining variables is mainly
   // used in inline assembly programs. The entire macro is a statement expression whose value
   // is the value of the last 'res'.
   // The BTSL instruction on line 21 is used to test and set the bit. It saves the bit value specified
   // by the base address (%3) and the bit offset (%2) to the carry flag CF, and then sets the
   // bit to 1. The SETB instruction is used to set the operand (%al) according to the carry flag
   // CF. If CF=1 then %al =1, otherwise %al =0.
19 #define set_bit(nr,addr) ({\
20 register int res __asm__("ax"); \
21 __asm__ __volatile__("btsl %2,%3\n\tsetb %%al": \
22 "=a" (res):"" (0),"r" (nr),"m" (*(addr))); \
23 res;})
24
   //// Resets the bit at the nr bit offset from the beginning of the specified address.
   // Returns the inverse of the original bit value.
   // Input: %0 -eax(return value); %1 -eax(0); %2 -nr, bit offset; %3 -(addr), content of addr.
   // The BTRL instruction on line 27 is used to test and reset the bit. The command SETNB is used
   // to set the operand (%al) according to the carry flag CF. If CF = 1, then %al = 0, otherwise %al
   // = 1.
25 #define clear_bit(nr,addr) ({\
26 register int res __asm__("ax"); \
27 __asm__ __volatile__("btrl %2,%3\n\tsetnb %%al": \
28 "=a" (res):"" (0),"r" (nr),"m" (*(addr))); \
29 res;})
30
   //// Look for the first zero bit from address 'addr', return the bit offset from 'addr'.
   // Input: %0 - ecx (return value); %1 - ecx(0); %2 - esi(addr).
   // Look for the first bit that is 0 in the bitmap starting with the address specified by'addr',
   // and return the bit offset from the address addr. 'addr' is the address of the data area of
   // the buffer block, and the range of the scan is 1024 bytes (8192 bits).
   // The BSFL instruction on line 36 is used to scan the first non-zero bit in the EAX register
   // and place its offset into EDX.
31 #define find_first_zero(addr) ({ \
32 int __res; \
33 __asm__("cld\n" \                            // clear direction flag.
34         "1:\tlodsl\n\t" \                    // obtain [esi] -> eax
35         "notl %%eax\n\t" \                   // inverse each bits in eax
36         "bsfl %%eax,%%edx\n\t" \             // offset of first non-zero bit -> edx
37         "je 2f\n\t" \                        // jump forward to label 2 if eax = 0.
38         "addl %%edx,%%ecx\n\t" \             // offset is added to ecx (first 0 bit in bitmap).
39         "jmp 3f\n" \                         // jump forward to label 3 (end).
40         "2:\taddl $32,%%ecx\n\t" \           // ecx adds 32 bit offset if bit 0 not found.
41         "cmpl $8192,%%ecx\n\t" \             // scanned 8192 bits (1024 bytes)?
42         "jl 1b\n" \                          // jump backward to label 1 if not yet.
43         "3:" \                               // end. At this time, ecx contains 0 bit offset.
44         :"=c" (__res):"c" (0),"S" (addr):"ax","dx","si"); \
45 __res;})
46
   //// Free the logic block in the data zone on the device.
   // Resets the bit in the logical block bitmap corresponding to the specified logical block.
   // Returns 1 if successful, otherwise returns 0.
47 int free_block(int dev, int block)
48 {
49         struct super_block * sb;
50         struct buffer_head * bh;
51
   // First, the super block information of the file system on the device dev is taken, and the
   // validity of the parameter block is checked according to the data block start logical block
// number and the total number of logical blocks in the file system. If the specified device
// super block does not exist, an error occurs. If the logical block number is smaller than
// the block number of the first logical block in the data zone on the disk or greater than
// the total number of logical blocks on the device, system stops too.
        if (!(sb = get_super(dev)))                 // in fs/super.c, line 56.
                panic("trying to free block on nonexistent device");
        if (block < sb->s_firstdatazone || block >= sb->s_nzones)
                panic("trying to free block not in datazone");
        bh = get_hash_table(dev,block);

// Then look for the block data from the hash table. If a buffer block is found, its validity
// is checked. At this time, if the number of references is greater than 1, it indicates that
// someone else is using the buffer block, so brelse() is called, and b_count is decremented
// by 1 and then exits. Otherwise, clear the modified and updated flags to release the data
// block. The main purpose of this piece of code is to detect that if the logic block currently
// exists in the buffer cache, the corresponding buffer block is released.
        if (bh) {
                if (bh->b_count > 1) {
                        brelse(bh);        // ret after b_count--, still used by others.
                        return 0;
                }
                bh->b_dirt=0;
                bh->b_uptodate=0;
                if (bh->b_count)           // if b_count=1, call brelse().
                        brelse(bh);
        }
// Then we reset the bit of the block in the logic block bitmap (set to 0). First calculate
// the logical block number of the block starting from the data zone (counting from 1). The
// logic block bitmap is then operated to reset the corresponding bit. If the bit is already
// 0, it means system is faulty and stops. Since a block has 1024 bytes, which is 8192 bits,
// so 'block / 8192 ' can calculate which block in the logical bitmap is the specified block,
// and 'block & 8191 ' can get the bit offset position of the block in the current block of
// the logic block bitmap.
        block -= sb->s_firstdatazone - 1 ;          // block = block - (s_firstdatazone -1);
        if (clear_bit(block&8191,sb->s_zmap[block/8192]->b_data)) {
                printk("block (%04x:%d) ",dev,block+sb->s_firstdatazone-1);
                printk("free_block: bit already cleared\n");
        }
// Finally, set the modified flag of the buffer block where the logic block bitmap is located.
        sb->s_zmap[block/8192]->b_dirt = 1;
        return 1;
}

//// Request a logical disk block from the device.
// The function first takes the device's superblock and looks for the first zero value bit
// (representing an free block) in the logical block bitmap. This bit is then set to indicate
// that the corresponding logical block is expected to be obtained. Then, a corresponding buffer
// block is obtained in the buffer for the logic block. Finally, the buffer block is cleared,
// its updated flag and modified flag are set, and the logical block number is returned. If
// the execution is successful, the function returns the logical block number (disk block
// number), otherwise it returns 0.
int new_block(int dev)
{
        struct buffer_head * bh;
        struct super_block * sb;
        int i,j;

// First get the super block of the device. Then scan the file system's 8 logical block bitmaps,
// look for the first 0 value bit, find the free logic block, and get the block number where
// the logical block is placed. If all the bits of the 8 block logic block bitmaps are scanned
// (i >= 8 or j >= 8192), the 0 value bit is not found or the buffer block pointer of the bitmap
// is invalid (bh = NULL), then exits with 0. (There is no free logic block).
        if (!(sb = get_super(dev)))
                panic("trying to get new block from nonexistant device");
        j = 8192;
        for (i=0 ; i<8 ; i++)
                if (bh=sb->s_zmap[i])
                        if ((j=find_first_zero(bh->b_data))<8192)
                                break;
        if (i>=8 || !bh || j>=8192)
                return 0;
// Next, the bits in the logical block bitmap corresponding to the found new logical block j
// are set. If the corresponding bit is already set, it indicates that the system has an error
// and stops. Otherwise set the modified flag of the corresponding buffer block that stores
// the bitmap. Because the logic block bitmap only indicates the occupancy of the logic block
// in the data zone on the disk, that is, the bit offset value in the logic block bitmap indicates
// the block number from the beginning of the data zone, so the logical block number of the
// first block of data zone needs to be added here to convert j to the logical block number.
// If the new logical block number is greater than the total number of logical blocks on the
// device, the block does not exist on the device. The application failed, return 0 & exit.
        if (set_bit(j,bh->b_data))
                panic("new_block: bit already set");
        bh->b_dirt = 1;
        j += i*8192 + sb->s_firstdatazone-1;
        if (j >= sb->s_nzones)
                return 0;
// Then, a buffer block is obtained in the buffer cache for the logical block number specified
// on the device, and the buffer block header pointer is returned. Because the logical block
// just obtained must have a reference count of 1 (set in getblk()), if it is not 1, it will
// stop. Finally, the new logical block is cleared and its updated flag and modified flag are
// set. Then release the corresponding buffer block and return the logical block number.
        if (!(bh=getblk(dev,j)))
                panic("new_block: cannot get block");
        if (bh->b_count != 1)
                panic("new block: count is != 1");
        clear_block(bh->b_data);
        bh->b_uptodate = 1;
        bh->b_dirt = 1;
        brelse(bh);
        return j;
}

//// Release the specified i-node.
// This function first determines the validity and releasability of the i-node given by the
// parameter. If the i-node is still in use, it cannot be released. Then, the i-node bitmap
// is operated by using the super block information, the bit in the i-node bitmap corresponding
// to the i-node number is reset, and the i-node structure is cleared.
void free_inode(struct m_inode * inode)
{
        struct super_block * sb;
        struct buffer_head * bh;

// First determine the validity or legitimacy of the i-node that needs to be released. If the
// i-node pointer = NULL, then exit. If the device number field on the i-node is 0, the node
// is not used. Then clear the memory area occupied by the corresponding i-node and return.
// Memset() is defined at line 395 in file include/string.h. This means that the memory area
// specified by the inode pointer is filled with 0, and the sizeof(*inode) bytes are filled.
        if (!inode)
                return;
        if (!inode->i_dev) {
                memset(inode,0,sizeof(*inode));
                return;
        }
// If there are other program references in this i-node, it cannot be released, indicating that
// there is a problem with the kernel code, and the system is down. If the number of file links
// is not 0, it means that there are other file directory entries that are using the node, so
// they should not be released, but should be put back.
        if (inode->i_count>1) {
                printk("trying to free inode with count=%d\n",inode->i_count);
                panic("free_inode");
        }
        if (inode->i_nlinks)
                panic("trying to free inode with links");
// After checking the validity of the i-node, we begin to operate on the i-node bitmap using
// its super-block information. First, take the super block of the device where the i-node is
// located, and test whether the device exists. Then determine whether the range of the i-node
// number is correct. If the i-node number is equal to 0 or greater than the total number of
// i-nodes on the device, an error occurs (the i-node of No. 0 is reserved). If the node bitmap
// corresponding to the i-node does not exist, an error occurs. Since the i-node bitmap of a
// buffer block has 8192 bits, i_num>>13 (ie, i_num/8192) can obtain the s_imap[] entry of the
// current i-node number, that is, the disk block.
        if (!(sb = get_super(inode->i_dev)))
                panic("trying to free inode on nonexistent device");
        if (inode->i_num < 1 || inode->i_num > sb->s_ninodes)
                panic("trying to free inode 0 or nonexistant inode");
        if (!(bh=sb->s_imap[inode->i_num>>13]))
                panic("nonexistent imap in superblock");
// Now we reset the bits in the node bitmap corresponding to the i-node. If the bit is already
// equal to 0, an error warning message is displayed. Finally, the buffer in which the i-node
// bitmap is located has been modified, and the memory area occupied by the i-node structure
// is cleared.
        if (clear_bit(inode->i_num&8191,bh->b_data))
                printk("free_inode: bit already cleared.\n\r");
        bh->b_dirt = 1;
        memset(inode,0,sizeof(*inode));
}

//// Create a new i-node for device, initialize and return a pointer to the new i-node.
// Obtain an free i-node entry in the memory i-node table and find an free i-node from the i-node
// bitmap.
struct m_inode * new_inode(int dev)
{
        struct m_inode * inode;
        struct super_block * sb;
        struct buffer_head * bh;
        int i,j;

// First, obtain a free i-node entry from the in-memory i-node table (inode_table) and read
// the super block structure of the specified device. Then scan the 8-block i-node bitmap in
// the super block, look for the first 0-bit (look for a free node), and obtain the node number
// where the i-node is placed. If the i-node is not found after the scan, or the buffer block
// where the bitmap is located is invalid (bh = NULL), then the i-node in the previously requested
// i-node table is returned, and the null pointer is returned and exited (indicating no Idle
// i-node available).
        if (!(inode=get_empty_inode()))         // fs/inode.c, line 197.
                return NULL;
        if (!(sb = get_super(dev)))             // fs/super.c, line 56.
                panic("new_inode with unknown device");
        j = 8192;
        for (i=0 ; i<8 ; i++)
                if (bh=sb->s_imap[i])
                        if ((j=find_first_zero(bh->b_data))<8192)
                                break;
        if (!bh || j >= 8192 || j+i*8192 > sb->s_ninodes) {
                iput(inode);
                return NULL;
        }
// Now that we have found the i-node number j that has not been used, we set the corresponding
// bit of the i-node bitmap corresponding to j (if it is already set, it indicates that the
// kernel is faulty). Then set the modified flag of the buffer block where the i-node bitmap
// is located. Finally, the i-node structure is initialized (i_ctime is the time when the content
// of the i-node is changed).
        if (set_bit(j,bh->b_data))
                panic("new_inode: bit already set");
        bh->b_dirt = 1;
        inode->i_count=1;                       // reference count.
        inode->i_nlinks=1;                      // number of file directory entry links.
        inode->i_dev=dev;                       // device number the i-node is located.
        inode->i_uid=current->euid;             // user id of the i-node.
        inode->i_gid=current->egid;             // group id.
        inode->i_dirt=1;                        // set the modified flag.
        inode->i_num = j + i*8192;              // Corresponds to i-node nr in the device.
        inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;
        return inode;
}
```

12.4 truncate.c

12.4.1 Function

The truncate.c program is used to release all logical blocks occupied by the specified i-node on the device, including direct blocks, primary indirect blocks, and secondary indirect blocks. Therefore, the file length corresponding to the node of the file is cut to 0, and the occupied device space is released. For the convenience of reading, the schematic diagram of the direct block and indirect block organization structures in the i-node is given again, as shown in Figure 12-23.

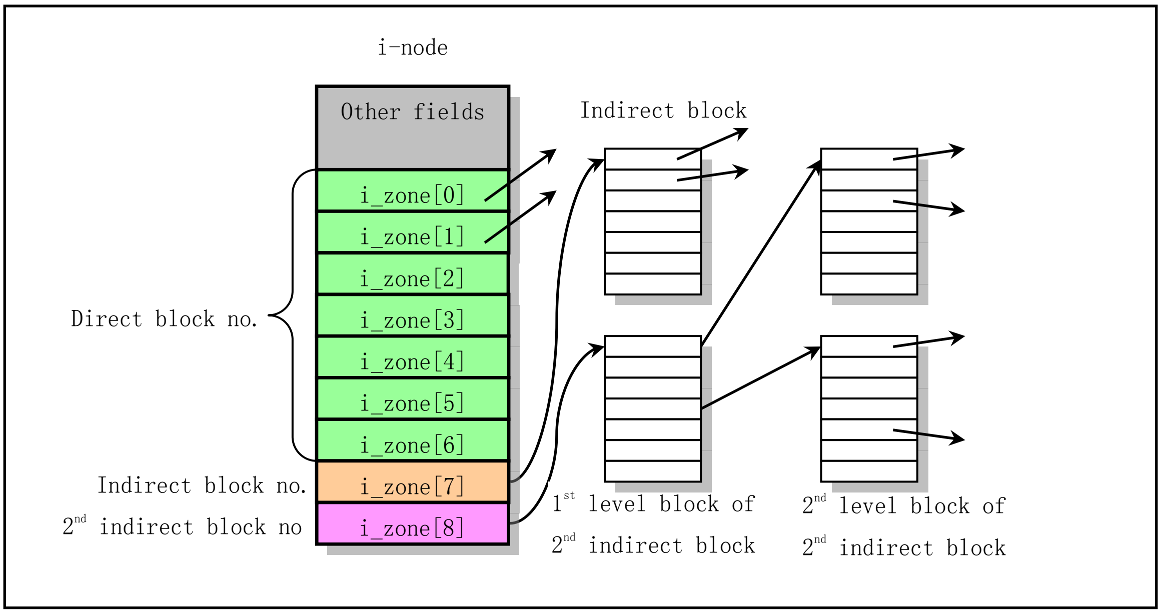


Figure 12-23 Schematic diagram of logical block connections of a i-node

The i_zone[] array in the i-node stores the disk block number of the logical block on the device. The first 7 items of the array (i_zone[0]--i_zone[6]) directly store the numbers of the first 7 data blocks in the relevant file. i_zone[7] stores the block number of the primary indirect block. Because the disk size is 1024 bytes, each disk block can store (1024 / 2) = 512 disk block numbers, that is, one indirect block number can address up to 512 device disk blocks. Accordingly, the secondary indirect block number i_zone[8] can addresses (512 * 512) = 261, 144 disk blocks.

12.4.2 Code annotation

Program 12-3 linux/fs/truncate.c

```
/*
 *  linux/fs/truncate.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//     and constants.
#include <linux/sched.h>

#include <sys/stat.h>

//// Release all one indirect blocks.
// The parameter dev is the device number of the file system; block is the logical block number.
// Returns 1 if successful, otherwise returns 0.
static int free_ind(int dev,int block)
{
        struct buffer_head * bh;
        unsigned short * p;
        int i;
        int block_busy;                          // The logic block has not been released.

// First determine the validity of the parameters. Returns if the logical block number is 0.
// Then the primary block is read, and all the logical blocks used on it are released, and then
// the buffer block of this primary indirect block is released. The function free_block() is
// used to release the disk block of the specified logical block number on the device (fs/bitmap.c
// line 47).
        if (!block)
                return 1;
        block_busy = 0;
        if (bh=bread(dev,block)) {
                p = (unsigned short *) bh->b_data;  // points to the block data area.
                for (i=0;i<512;i++,p++)             // a block can contains 512 blcok nr.
                        if (*p)
                                if (free_block(dev,*p)) {
                                        *p = 0;
                                        bh->b_dirt = 1;    // set the modified flag.
                                } else
                                        block_busy = 1;    // block is not released.
                brelse(bh);    // release the buffer block occupied by the indirect block.
        }
// Finally release an indirect block on the device. But if there is a logical block that is
// not released, it returns 0 (failure).
        if (block_busy)
                return 0;
        else
                return free_block(dev,block);      // 1 if successful, otherwise return 0.
}

//// Release all secondary indirect blocks.
// The parameter dev is the device number; 'block' is the logical block number of the secondary
// indirect block.
static int free_dind(int dev,int block)
{
        struct buffer_head * bh;
        unsigned short * p;
        int i;
        int block_busy;                              // The logic block has not been released.

// First determine the validity of the parameters. Returns if the logical block number is 0.
// The code then reads the first-level block of the secondary indirect block and releases all
// the logical blocks on it indicating that it has been used, and then releases the buffer block
// of the first-level block.
        if (!block)
                return 1;
        block_busy = 0;
        if (bh=bread(dev,block)) {
                p = (unsigned short *) bh->b_data;  // points to the block data area.
                for (i=0;i<512;i++,p++)              // 512 second-level blocks
                        if (*p)
                                if (free_ind(dev,*p)) {  // release primary indirect block.
                                        *p = 0;
                                        bh->b_dirt = 1;  // set modified flag.
                                } else
                                        block_busy = 1;  // block not released.
                brelse(bh);   // release buffer block occupied by secondary indirect block.
        }
// Finally, the secondary indirect block on the device is released. But if there is a logic
// block that is not released, it returns 0 (failure).
        if (block_busy)
                return 0;
        else
                return free_block(dev,block);
}

//// Truncates file data.
// The file length corresponding to the node is cut to 0, and the occupied device space is
// released.
void truncate(struct m_inode * inode)
{
        int i;
        int block_busy;

// First determine the validity of the specified i-node. Returns if it is not a regular file,
// a directory, or a link entry. Then release 7 direct blocks of the i-node and set all 7 logical
// block items to zero. If the block is busy and not released, the block_busy flag is set.
        if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode) ||
             S_ISLNK(inode->i_mode)))
                return;
repeat:
        block_busy = 0;
        for (i=0;i<7;i++)
                if (inode->i_zone[i]) {                // release if block nr is not zero.
                        if (free_block(inode->i_dev,inode->i_zone[i]))
                                inode->i_zone[i]=0;
                        else
                                block_busy = 1;        // set if not released.
                }
        if (free_ind(inode->i_dev,inode->i_zone[7]))   // release primory indirect blocks.
                inode->i_zone[7] = 0;
        else
                block_busy = 1;                       // set if not released.
        if (free_dind(inode->i_dev,inode->i_zone[8])) // release secondary indirect blocks
                inode->i_zone[8] = 0;
        else
                block_busy = 1;

// After that, the i-node modified flag is set, and if there is still a logical block that is
// not released due to "busy", the current process running time slice is set to 0 to switch
// to another process to run, and then wait for a while to re-execute the release operation.
// Finally, the file modification time and the i-node change time are set to the current time.
// The macro CURRENT_TIME is defined in line 142 of the header file linux/sched.h and is defined
// as (startup_time + jiffies/HZ) for the seconds value since 1970:0:0:0.
        inode->i_dirt = 1;
        if (block_busy) {
                current->counter = 0;           // current process time slice is set to 0.
                schedule();
                goto repeat;
        }
        inode->i_size = 0;                      // the file size is set to zero.
        inode->i_mtime = inode->i_ctime = CURRENT_TIME;
}


```

12.5 inode.c

12.5.1 Function

The inode.c program includes functions iget(), iput(), and block mapping function bmap() that process the i-node, as well as other auxiliary functions. The iget(), iput(), and bmap() functions are mainly used in the namei() mapping function of the namei.c program to find the corresponding i-node from the file path name.

1. iget() function

The iget() function is used to read the i-node of the specified node number nr from the device dev, and increment the reference count field value i_count of the node by one. The operation flow is shown in Figure 12-24. The function first determines the validity of the parameter dev, and takes an idle i-node from the i-node table, then scans the i-node table, finds the i-node with the specified node number nr, and increments the reference number of the i-node. If the device of the currently scanned is not the specified device or the node number is not equal to the specified node, continue scanning. Otherwise, the i-node with the specified device number and node number has been found, and the node is waiting to be unlocked (if If it is locked right now). While waiting for the i-node to be unlocked, the node table may change. So If the device number of the i-node is now not equal to the specified device number or the node number is not equal to the specified node number, the entire i-node table needs to be re-scanned again. Subsequently, the reference count value of the i-node is incremented by 1, and it is determined whether the i-node is a mount point of another file system.

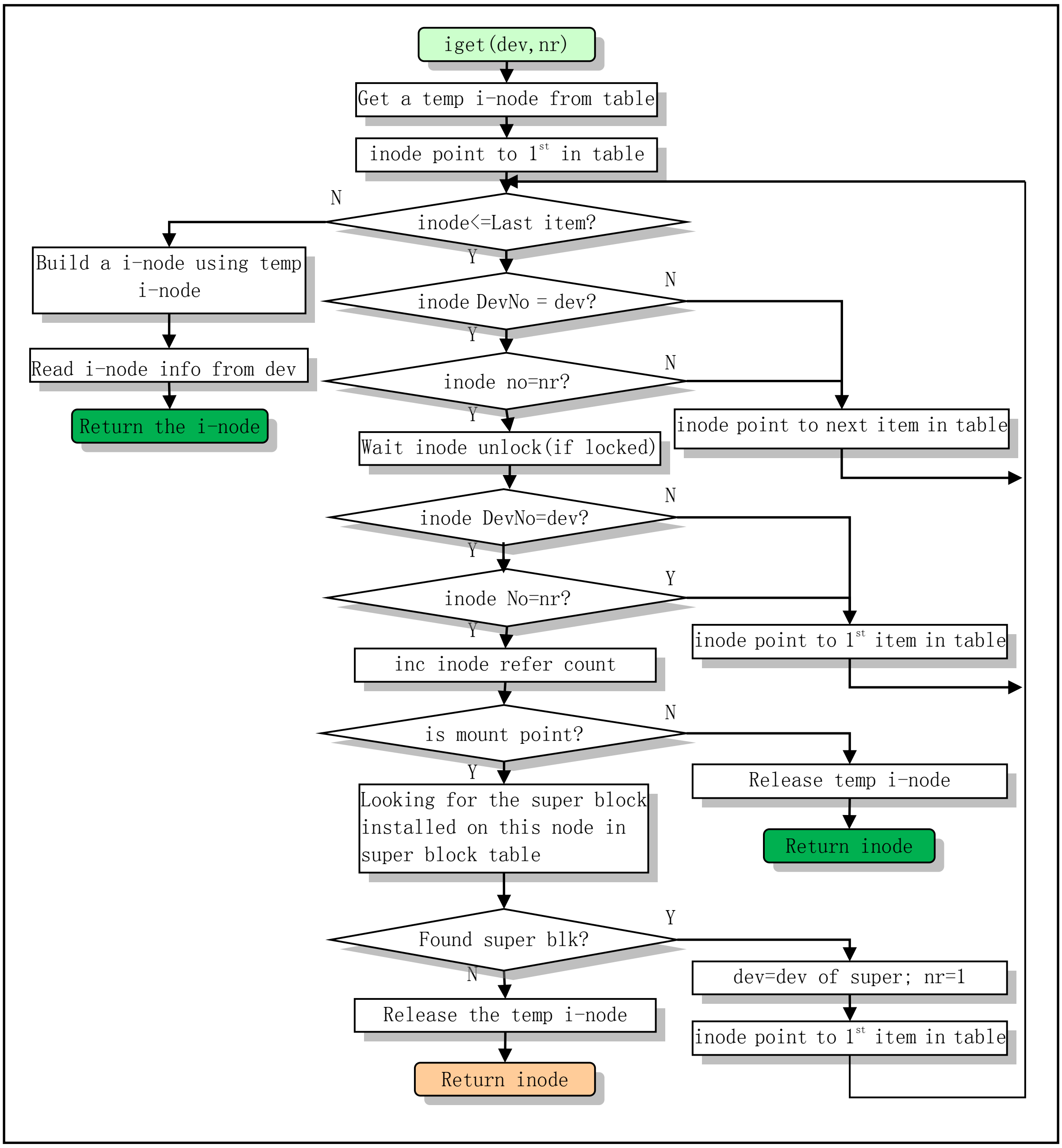

iget(dev,nr)
Get a temp i-node from table
inode point to 1st in table
inode<=Last item?
N
Y
Build a i-node using temp i-node
Read i-node info from dev
Return the i-node
inode DevNo = dev?
N
Y
inode no=nr?
N
Y
inode point to next item in table
Wait inode unlock(if locked)
inode DevNo=dev?
N
Y
inode No=nr?
Y
Y
inode point to 1st item in table
inc inode refer count
is mount point?
N
Y
Release temp i-node
Return inode
Looking for the super block installed on this node in super block table
Found super blk?
Y
N
dev=dev of super; nr=1
inode point to 1st item in table
Release the temp i-node
Return inode

Figure 12-24 iget() function operation flow chart

If the i-node is a mount point of a file system, the super block table is searched for the super block installed on the i-node. If the corresponding super block is not found, an error message is displayed, and the free node that the function starts to acquire is released, and the i-node pointer is returned. If the corresponding super block is found, the i-node is written to the disk. Then get the device number from the super block installed in the i-node file system, and let the i-node number be 1. Then scan the entire i-node table again to fetch the root node of the mounted file system.

If the i-node is not the installation point of other file systems, it indicates that the corresponding i-node has been found, so the idle i-node that is temporarily applied can be discarded at this time, and the found i-node pointer is returned.

If the specified i-node is not found in the i-node table, the node is established in the i-node table using the idle i-node of the previous application, and the i-node information is read from the corresponding device, and the i-node pointer is returned.

2. iput() function

The function performed by the iput() function is exactly the opposite of iget(). It is mainly used to decrement the i-node reference count value by 1, and if it is a pipe i-node, wake up the waiting process. If the i-node is the i-node of the block device file, the device is refreshed, and if the link count of the i-node is 0, all the disk logical blocks occupied by the i node are released, and are returned after the i-node is released. If the i-node reference count value i_count is 1, the number of links is not zero, and the content has not been modified, then the i-node reference count is decremented by one at this time, and then returned. Because if an i node has i_count=0, it means it has been released. The operation flow of this function is also similar to iget().

If the process does not need to continuously use an i-node at a certain time, the iput() function should be called to decrement the value of the reference count field i_count of the i-node, and also let the kernel perform some other processing. Therefore, after performing one of the following operations, the kernel code should normally call the iput() function:

- Increase the value of the i-node reference count field i_count by one;
- Called the namei(), dir_namei(), or open_namei() function;
- Called the iget(), new_inode(), or get_empty_inode() function;
- When closing a file, if no other process has used the file;
- When unmounting a file system (replace the device file i-node, etc.).

In addition, when a process is created, its current working directory pwd, current root directory root, and executable file directory executable fields are initialized to point to three i-nodes, and the reference count field of three i-nodes are also set accordingly. Therefore, when the process executes a system-call that changes the current working directory, the iput() function needs to be called in code of the system-call to first put back the i-node in use, and then let the pwd of the process point to the new i-node of the path name. Similarly, to modify the root and executable fields of a process, you also need to execute the iput() function.

3. bmap() function

The _bmap() function is used to map a file data block to the corresponding disk block. The parameter 'inode' is the i-node pointer of the file, 'block' is the data block number in the file, and 'create' is the creation flag, indicating whether the corresponding disk block needs to be established on the disk if the corresponding file data block does not exist. The return value of this function is the logical block number (disk block number) corresponding to the file data block on the device. When create=0, the function is the bmap() function. When create=1, it is the create_block() function.

The data in a regular file is placed in the data zone of the disk, and a file name is associated with these data disk blocks through the corresponding i-node. The numbers of these disk blocks are stored in the logical block array of the i-node. The _bmap() function mainly processes the logical block array i_zone[] of the i-node, and sets the occupancy of the logical block bitmap according to the logical block number in i_zone[]. See Figure 12-6 in Section 12.1. As mentioned earlier, i_zone[0] to i_zone[6] are used to store the direct logical block number of the file; i_zone[7] is used to store the indirect logical block number; and i_zone[8] is used to store the secondary indirect logical block number. When the file size is small (less than 7K), the disk block number used by the file can be directly stored in the 7 direct block items in the i-node; when the file is slightly larger (not more than 7K+512K), it needs to use primary indirect block item i_zone[7]; when the file is larger, the secondary indirect block item i_zone[8] is needed. Therefore, when the file is relatively small, the speed of the Linux addressing disk block is faster.

12.5.2 **Code annotation**

Program 12-4 linux/fs/inode.c

```
 1 /*
 2  *  linux/fs/inode.c
 3  *
 4  *  (C) 1991  Linus Torvalds
 5  */
 6
 // <string.h> String header file. Defines some embedded functions about string operations.
 // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
 //     and constants.
 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
 //     of the initial task 0, and some embedded assembly function macro statements about the
 //     descriptor parameter settings and acquisition.
 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
 //     used functions of the kernel.
 // <linux/mm.h> Memory management header file. Contains page size definitions and some page
 //     release function prototypes.
 // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
 //     descriptors/interrupt gates, etc. is defined.
 7 #include <string.h>
 8 #include <sys/stat.h>
 9
10 #include <linux/sched.h>
11 #include <linux/kernel.h>
12 #include <linux/mm.h>
13 #include <asm/system.h>
14
   // An array of pointers to the total number of device data blocks. Each pointer item points
   // to the total number of blocks of the given major device number, hd_sizes[]. Each item of
   // the total number of block arrays corresponds to the total number of data blocks owned by
   // a sub-device determined by the sub-device number.
15 extern int *blk_size[];
16
17 struct m_inode inode_table[NR_INODE]={{0,},};    // In-memory i-node table (NR_INODE=32).
18
19 static void read_inode(struct m_inode * inode);  // read i-node information, line 297.
20 static void write_inode(struct m_inode * inode); // write i-node info to cache, line 324.
21
   //// Wait for the specified i-node to be available.
   // If the i-node has been locked, the current task is placed in an uninterruptible wait state
   // and added to the i-wait queue until the i-node unlocks and explicitly wakes up the task.
22 static inline void wait_on_inode(struct m_inode * inode)
23 {
24         cli();
25         while (inode->i_lock)
26                 sleep_on(&inode->i_wait);        // kernel/sched.c, line 199.
27         sti();
28 }
29
   //// Lock the i-node (lock the given i-node).
   // If the i-node is locked, the current task is placed in an uninterruptible wait state and
   // added to the i-wait queue i_wait. Until the i-node unlocks and explicitly wakes up the task,
   // then locks it.
30 static inline void lock_inode(struct m_inode * inode)
31 {
32         cli();
33         while (inode->i_lock)
34                 sleep_on(&inode->i_wait);
35         inode->i_lock=1;                          // set locked flag.
36         sti();
37 }
38
   //// Unlock the specified i node.
   // Reset the lock flag of the i-node and explicitly wake up all processes waiting on the i-wait
   // queue i_wait.
39 static inline void unlock_inode(struct m_inode * inode)
40 {
41         inode->i_lock=0;
42         wake_up(&inode->i_wait);                 // kernel/sched.c, line 204.
43 }
44
   //// Release all i-nodes of device dev in the memory i-node table.
   // Scans the i-node table array in memory and releases it if an item is used by the specified
   // device.
45 void invalidate_inodes(int dev)
46 {
47         int i;
48         struct m_inode * inode;
49
   // First let the pointer point to the first item in the memory i-node table array, and then
   // scan all the i-nodes in it. For each of the i-nodes, wait for the i-node to unlock (if it
   // is currently locked), and then determine if it belongs to the specified device. If yes, it
   // is released, that is, the device number field i_dev of the i-node is set to 0, and the modified
   // flag is also reset. In the meantime, it will also check if it is still being used, that is,
   // whether its reference count is not 0, and if so, a warning message is displayed.
   // The pointer assignment sentence "0+inode_table" on line 50 is equivalent to "inode_table",
   // "&inode_table[0]", but this may be more straightforward.
50         inode = 0+inode_table;                 // points to the first item of i-node table.
51         for(i=0 ; i<NR_INODE ; i++,inode++) {
52                 wait_on_inode(inode);          // wait for i-node to be available(unlocked).
53                 if (inode->i_dev == dev) {
54                         if (inode->i_count)    // if references is not 0, error warning.
55                                 printk("inode in use on removed disk\n\r");
56                         inode->i_dev = inode->i_dirt = 0;     // release i-node.
57                 }
58         }
59 }
60
   //// Synchronize all i-nodes.
   // Synchronize all i-nodes in the memory i-node table with i-nodes on the device.
61 void sync_inodes(void)
62 {
63         int i;
```

```
        struct m_inode * inode;

// First, let the pointer of the memory i-node type point to the first item of the i-node table,
// and then scan all the nodes in the table. For each of the i-nodes, wait for the i-node to
// be unlocked (if it is currently locked), and then check if the i-node has been modified and
// is not a pipe node. If this is the case, the i-node is written to the buffer cache. The buffer
// manager program will write them to the disk at the appropriate time.
        inode = 0+inode_table;                   // points to the first item of the table.
        for(i=0 ; i<NR_INODE ; i++,inode++) {
                wait_on_inode(inode);                  // wait for i-node to be available.
                if (inode->i_dirt && !inode->i_pipe)   // modified and not a pipe node.
                        write_inode(inode);            // write to the buffer cache.
        }
}

//// File data block mapping to disk block (bmap - block map).
// Parameters: inode - i-node pointer of the file; block - data block number in the file;
// create - the block creation flag. This function maps the specified file data block to the
// logical block on the device and returns the logical block number. If the block creation flag
// is set, a new disk block is requested when the corresponding logical block does not exist
// on the device, and the logical block number (disk block number) corresponding to the file
// data block is returned. The function is processed in four parts: (1) parameters validity
// check; (2) direct block processing; (3) pimary indirect block processing; and (4) secondary
// indirect block processing.
static int _bmap(struct m_inode * inode,int block,int create)
{
        struct buffer_head * bh;
        int i;

// (1) First check the validity of the parameters. If the file data block number is less than
// 0, the kernel will stop. If the block number is greater than (direct block nr + indirect
// block nr + second indirect block nr), it is out of range of the file system.
        if (block<0)
                panic("_bmap: block<0");
        if (block >= 7+512+512*512)
                panic("_bmap: block>big");

// (2) Then, according to the size of the file block number and whether the creation flag is
// set, the processing is performed separately. If the block number is less than 7, it is
// represented by a direct block. At this time, if the creation flag is set and the logical
// block field in the i-node is 0, a disk block (logical block) is requested from the device,
// and the logical block number on the disk is filled in the logical block field. Then set
// i-node change time and modified flag, and finally return the logical block number. The function
// new_block() is defined at line 76 in the bitmap.c program.
        if (block<7) {
                if (create && !inode->i_zone[block])
                        if (inode->i_zone[block]=new_block(inode->i_dev)) {
                                inode->i_ctime=CURRENT_TIME;  // ctime - change time
                                inode->i_dirt=1;              // set modified flag.
                        }
                return inode->i_zone[block];
        }
```

```
// (3) If the block number is >= 7, and less than (7 + 512), it means that it uses an indirect
// block. Therefore, an indirect block is processed below. If it is a create operation, and
// the indirect block field i_zone[7] of the i-node is 0, it indicates that the file is the
// first time to use the indirect block. Therefore, it is necessary to apply for a disk block
// for storing the indirect block information, and fill the actual disk block number into the
// indirect block field. Then set the i-node modified flag and modification time. If the disk
// block fails to be created at the time of creation, the i-node indirect block field i_zone[7]
// is 0 at this time, and 0 is returned. Or the parameter creation flag is 0, but i_zone[7]
// is also 0, indicating that there is no indirect block in the i-node, so the mapping disk
// block fails, and returns 0 to exit.
        block -= 7;
        if (block<512) {
                if (create && !inode->i_zone[7])
                        if (inode->i_zone[7]=new_block(inode->i_dev)) {
                                inode->i_dirt=1;
                                inode->i_ctime=CURRENT_TIME;
                        }
                if (!inode->i_zone[7])
                        return 0;
// Now read an indirect block of the i-node on the device and get the logical block number i
// in the 'block' item on it (each block number occupies 2 bytes). If it is a create operation
// and the obtained logical block number is 0, then it is necessary to apply for a disk block
// and set the 'block' entry in the indirect block equal to the logical block number of the
// new disk block. Then set the modified flag of the indirect block. If it is not a create
// operation, then 'i' is the logical block number that needs to be mapped (find). Finally,
// the buffer block occupied by the indirect block is released, and the logical block number
// of the new application or the corresponding block on the disk is returned.
                if (!(bh = bread(inode->i_dev,inode->i_zone[7])))
                        return 0;
                i = ((unsigned short *) (bh->b_data))[block];
                if (create && !i)
                        if (i=new_block(inode->i_dev)) {
                                ((unsigned short *) (bh->b_data))[block]=i;
                                bh->b_dirt=1;
                        }
                brelse(bh);
                return i;
        }

// (4) If the program runs to this place, it indicates that the data block belongs to the secondary
// indirect block. Its processing is similar to the primary indirect block. The following is
// the processing of the secondary indirect block. First, the block is subtracted from the number
// of blocks (512) accommodated by the primary indirect block, and then created or searched
// according to whether the creation flag is set. If it is a create operation and the secondary
// indirect block field of the i node is 0, then a new disk block is required to store the primary
// block information of the secondary indirect block, and the actual disk block number is filled
// in the secondary indirect block field. After that, set the i-node modified flag and
// modification time. Similarly, if the disk block fails to be created at the time of creation,
// then the i-node secondary indirect block field i_zone[8] is 0, and 0 is returned. Or it is
// not a create operation, but i_zone[8] turns out to be 0, indicating that there is no indirect
// block in the i-node, so the mapped disk block fails, and returns 0 to exit.
        block -= 512;
        if (create && !inode->i_zone[8])
```

```
                if (inode->i_zone[8]=new_block(inode->i_dev)) {
                        inode->i_dirt=1;
                        inode->i_ctime=CURRENT_TIME;
                }
        if (!inode->i_zone[8])
                return 0;
// Now read the secondary indirect block of the i-node on the device and get the logical block
// number 'i' in the (block / 512) entry on the first-level block of the secondary indirect
// block. If it is a create operation, and the logical block number in the (block / 512) item
// on the first-level block of the secondary indirect block is 0, then it is necessary to apply
// for a disk block as the second-level block 'i' of the secondary indirect block, and let the
// (block / 512) item in the first-level block of the secondary indirect block to be equal to
// the block number 'i' of the second-level block. Then set the first-level block modified flag
// of the secondary indirect block and release this first-level block of the secondary indirect
// block. If it is not created, then 'i' is the logical block number that needs to be found.
        if (!(bh=bread(inode->i_dev,inode->i_zone[8])))
                return 0;
        i = ((unsigned short *)bh->b_data)[block>>9];
        if (create && !i)
                if (i=new_block(inode->i_dev)) {
                        ((unsigned short *) (bh->b_data))[block>>9]=i;
                        bh->b_dirt=1;
                }
        brelse(bh);
// If the second-level block number of the secondary indirect block is 0, it means that the
// application disk block fails or the original corresponding block number is 0, then 0 is
// returned. Otherwise, the second-level block of the secondary indirect block is read from
// the device, and the logical block number in the block item on the second-level block is taken.
        if (!i)
                return 0;
        if (!(bh=bread(inode->i_dev,i)))
                return 0;
        i = ((unsigned short *)bh->b_data)[block&511];   // limits to 511 (0x1ff).
// If the creation flag is set, and the logical block number in the block of the second-level
// block is 0, then apply a disk block as the block that finally stores the data information,
// and let the block item in the second-level block equal the new logical block number 'i'.
// Then set the modified flag of the second-level block. Finally, the second-level block of
// the secondary indirect block is released, and the logical block number of the newly applied
// or original corresponding block on the disk is returned.
        if (create && !i)
                if (i=new_block(inode->i_dev)) {
                        ((unsigned short *) (bh->b_data))[block&511]=i;
                        bh->b_dirt=1;
                }
        brelse(bh);
        return i;
}

//// Get the logical block number of the file data block on the device.
// Parameters: inode - i-node pointer of the file; block - the data block number in the file.
// If the operation is successful, the logical block number is returned, otherwise returns 0.
int bmap(struct m_inode * inode,int block)
{
```

```
144         return _bmap(inode,block,0);      // create flag = 0.
145 }
146
    //// Find or create a disk block on the device for the file block.
    // Find the corresponding disk block of the file data block on the device. If it does not exist,
    // create a block and return the corresponding disk block number.
    // Parameters: inode - i-node pointer of the file; block - the data block number in the file.
    // If the operation is successful, the logical block number is returned, otherwise returns 0.
147 int create_block(struct m_inode * inode, int block)
148 {
149         return _bmap(inode,block,1);
150 }
151
    //// Put back an i-node (write back to device).
    // This function is mainly used to decrement the i-node reference count by 1, and if it is a
    // pipe i-node, wake up the waiting process. If it is a block device file i-node, the device
    // is refreshed, and if the link count of the i-node is 0, all the disk logical blocks occupied
    // by the i-node are released, and the i-node is released.
152 void iput(struct m_inode * inode)
153 {
    // First check the validity of the i-node given by the parameter and wait for the i-node to
    // unlock (if it is locked). If the reference count of the i-node is 0, it means that the i-node
    // is already idle. The kernel then asks for a put back operation on it, indicating that there
    // is a problem with code in the kernel. The error message is then displayed and the machine
    // is shut down.
154         if (!inode)
155                 return;
156         wait_on_inode(inode);
157         if (!inode->i_count)
158                 panic("iput: trying to free free inode");
    // If it is a pipe i-node, wake up the process waiting for the pipe, the number of references
    // is decremented by 1, and if the reference count is still not 0, the function returned.
    // Otherwise, the memory page occupied by the pipeline is released, and the reference count,
    // modified flag, and pipe flag of the node are reset, then function returned. For the pipe
    // i-nodes, inode->i_size stores the memory page address. See get_pipe_inode(),lines 231,237.
159         if (inode->i_pipe) {
160                 wake_up(&inode->i_wait);
161                 wake_up(&inode->i_wait2);          //
162                 if (--inode->i_count)
163                         return;
164                 free_page(inode->i_size);
165                 inode->i_count=0;
166                 inode->i_dirt=0;
167                 inode->i_pipe=0;
168                 return;
169         }
    // If the device number field of the i-node is 0, the reference count for this node is decremented
    // by one and returned. For example, an i-node for pipeline operation has a device number of
    // 0 for the i-node. In addition, if it is the i-node of the block device file, then in the
    // logical block field 0 (i_zone[0]) is the device number, then refresh the device and wait
    // for the i-node to unlock.
170         if (!inode->i_dev) {
171                 inode->i_count--;
172                     return;
173             }
174             if (S_ISBLK(inode->i_mode)) {
175                     sync_dev(inode->i_zone[0]);
176                     wait_on_inode(inode);
177             }
    // If the reference count of the i-node is greater than 1, the count is decremented by 1 and
    // returned directly (because the i-node is still in use and cannot be released), otherwise
    // the reference count value of the i-node is 1 (because it has been checked on line 157 when
    // it is zero). If the number of links of the i-node is 0, it indicates that the corresponding
    // file is deleted. All logical blocks of the i-node are then released and the i-node is released.
    // The function free_inode() is used to actually release the i-node operation, that is, reset
    // the i-node bitmap bit, and clear the i-node structure content.
178 repeat:
179             if (inode->i_count>1) {
180                     inode->i_count--;
181                     return;
182             }
183             if (!inode->i_nlinks) {            // at this point, i_count = 1.
184                     truncate(inode);
185                     free_inode(inode);         // bitmap.c, line 108.
186                     return;
187             }
    // If the i-node has been modified, write back to update the i-node and wait for the i-node
    // to unlock (if locked). Since it may sleep when writing the i-node, other processes may modify
    // the i-node at this time, so the above-mentioned checking procedure (label repeat) needs to
    // be repeated again after the process is woken up.
188             if (inode->i_dirt) {
189                     write_inode(inode);        /* we can sleep - so do again */
190                     wait_on_inode(inode);
191                     goto repeat;
192             }
    // If the program is executed here, the i-node reference count i_count is 1, the number of links
    // is not zero, and the content has not been modified. Therefore, as long as the i-node reference
    // count is decremented by 1, it can be returned. At this time, i_count=0 indicates that the
    // i-node has been released.
193             inode->i_count--;
194             return;
195 }
196
    //// Obtain an idle i-node item from the i-node table.
    // Look for the i-node with a reference count of 0, clear and write it to disk, and return its
    // pointer. The reference count is now set to 1.
197 struct m_inode * get_empty_inode(void)
198 {
199             struct m_inode * inode;
200             static struct m_inode * last_inode = inode_table;   // point to the first item.
201             int i;
202
    // The entire i-node table is scanned after initializing the last_inode pointer to point to
    // the i-node table header. If the last_inode has already pointed to the last item of the i-node
    // table, let it re-point to the beginning of the i-node table to continue looping through the
    // table. If the reference count of the i-node pointed to by last_inode is 0, it indicates that
// an idle i-node item may be found, and then the inode is pointed to the i-node. If the modified
// flag and the lock flag of the i-node are both 0, then we can use the i-node and so we exit
// the for() loop.
        do {
                inode = NULL;
                for (i = NR_INODE; i ; i--) {                // NR_INODE = 32。
                        if (++last_inode >= inode_table + NR_INODE)
                                last_inode = inode_table;
                        if (!last_inode->i_count) {
                                inode = last_inode;
                                if (!inode->i_dirt && !inode->i_lock)
                                        break;
                        }
                }
// If no free i-node is found (inode = NULL), the i-node table is printed for debugging use
// and stops the system.
                if (!inode) {
                        for (i=0 ; i<NR_INODE ; i++)
                                printk("%04x: %6d\t",inode_table[i].i_dev,
                                        inode_table[i].i_num);
                        panic("No free inodes in mem");
                }
// Otherwise wait for the i-node to unlock (if it is locked again). If the i-node modified flag
// is set, the i-node is refreshed (synchronized). Because it may sleep when refreshing, you
// need to cycle again to wait for the i-node to unlock.
                wait_on_inode(inode);
                while (inode->i_dirt) {
                        write_inode(inode);
                        wait_on_inode(inode);
                }
        } while (inode->i_count);

// If the i-node is occupied by others (the reference count of the i-node is not 0), then the
// idle i-node needs to be searched again. Otherwise, it indicates that an idle i-node item
// that meets the requirements has been found. Then, the content of the i-node item is cleared,
// and the reference count is set to 1, and the i-node pointer is returned.
        memset(inode,0,sizeof(*inode));
        inode->i_count = 1;
        return inode;
}

//// Get the pipe i-node.
// First scan the i-node table, look for an idle i-node entry, and then get a page of free memory
// for the pipeline to use. Then the reference count of the obtained i-node is set to 2 (reader
// and writer), the pipe head and tail are initialized, and the pipe type representation of
// the i-node is set. Returns the pointer to the i-node and returns NULL if it fails.
struct m_inode * get_pipe_inode(void)
{
        struct m_inode * inode;

// First we get an free i-node from the memory i-node table and return NULL if we can't find
// one. Then apply for a page of memory for the i-node and have the node's i_size field point
// to the page. If there is no free memory, the i-node is released and NULL is returned.
```

```
        if (!(inode = get_empty_inode()))
                return NULL;
        if (!(inode->i_size=get_free_page())) {     // i_size points to a buffer page.
                inode->i_count = 0;
                return NULL;
        }
// Then set the reference count of the i-node to 2 and reset the pipe head and tail pointer.
// The i_zone[0] and i_zone[1] of the i-node logical block number array are used to store the
// pipe header and the pipe tail pointer, respectively. Finally, set the i-node flag to be the
// pipe i-node type and return the i-node.
        inode->i_count = 2;               /* sum of readers/writers */
        PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0;  // Reset the pipe head and tail.
        inode->i_pipe = 1;                          // pipe i-node.
        return inode;
}

//// Get an i-node from device.
// Parameters: dev - device number; nr - i-node number.
// The i-node structure content of the specified i-node number is read from the device into
// the memory i-node table, and the i-node pointer is returned. First search in the i-node table
// located in the buffer cache. If the i-node with the specified node number is found, the i-node
// pointer is returned after some check processing. Otherwise, the i-node is read from the device
// and placed in the i-node table, and the i-node pointer is returned.
struct m_inode * iget(int dev,int nr)
{
        struct m_inode * inode, * empty;

// First determine the validity of the parameters. If the device number is 0, it indicates a
// kernel code problem, displays an error message and stops. Otherwise, an idle i-node is taken
// in advance from the i-node table for backup. Then scan the entire i-node table, look for
// the i-node with the specified node number nr, and increment the reference number by one.
// However, if the device number of the currently scanned i-node is not equal to the specified
// device number or the node number is not equal to the specified node number, the scanning
// is continued.
        if (!dev)
                panic("iget with dev==0");
        empty = get_empty_inode();
        inode = inode_table;                       // points to the first item of table.
        while (inode < NR_INODE+inode_table) {
                if (inode->i_dev != dev || inode->i_num != nr) {
                        inode++;
                        continue;
                }
// If an i-node with the specified dev and nr is found, the node is awaited to be unlocked (if
// locked). The i-node table may change while waiting for the node to be unlocked. Therefore,
// the same check as above is required again when the code continues to execute. If a change
// has occurred, the entire i-node table is rescanned again.
                wait_on_inode(inode);
                if (inode->i_dev != dev || inode->i_num != nr) {
                        inode = inode_table;
                        continue;
                }
// This means that the corresponding i-node has been found. The i-node reference count is then
// incremented by one and then further checked to see if it is the mount point for another file
// system. If so, look for the root node of the mounted filesystem. If the i-node is indeed
// the mount point of other file systems, the super block table is searched for the super block
// installed on this i-node. If the super block is not found, an error message is displayed,
// and the idle node empty obtained at the beginning of the function is put back, and the i-node
// pointer is returned.
        inode->i_count++;
        if (inode->i_mount) {
            int i;

            for (i = 0 ; i<NR_SUPER ; i++)
                if (super_block[i].s_imount==inode)
                    break;
            if (i >= NR_SUPER) {
                printk("Mounted inode hasn't got sb\n");
                if (empty)
                    iput(empty);
                return inode;
            }
// The code execution to this point indicates that the file system superblock installed to the
// inode has been found. Then, we write the i-node to the disk to put it back, and replace the
// device  number with the one taken from the super block installed on the i-node, and the i-node
// number needed is re-set to ROOT_INO, which is 1. The entire i-node table is then rescanned
// to obtain the root i-node information of the mounted file system.
            iput(inode);
            dev = super_block[i].s_dev;
            nr = ROOT_INO;
            inode = inode_table;
            continue;
        }
// Finally we found the corresponding i-node. Therefore, you can discard the idle i-node
// temporarily applied at the beginning of this function and return the found i-node.
        if (empty)
            iput(empty);
        return inode;
    }
// If we do not find the specified i-node in the table, the i-node is created in the table by
// using the idle i-node 'empty' of the previous application, and the i-node information is
// read from the corresponding device, and the i-node pointer is returned.
    if (!empty)
        return (NULL);
    inode=empty;
    inode->i_dev = dev;                     // set the device of the new i-node.
    inode->i_num = nr;                      // set the i-node number.
    read_inode(inode);
    return inode;
}

//// Read the specified i-node information.
// The disk block containing the specified i-node information is read from the device and then
// copied to the specified i-node structure. In order to determine the logical block number
// of the disk block where the i-node is located, the super block on the corresponding device
// must first be read to obtain the information of INODES_PER_BLOCK for calculating the logical
// block number. After calculating the logical block number where the i-node is located, the
// logical block is read into a buffer block, and then the contents of the i-node at the
// corresponding position in the buffer block are copied to the specified position.
static void read_inode(struct m_inode * inode)
{
        struct super_block * sb;
        struct buffer_head * bh;
        int block;

// First lock the i-node and get the super-block of the device where the node is located.
        lock_inode(inode);
        if (!(sb=get_super(inode->i_dev)))
                panic("trying to read inode without dev");
// The logical block number of the device where the i-node is located = (boot block + super
// block) + i-node bitmap blocks + logical block bitmap blocks + (i node number - 1) / i-nodes
// per block, refer to Figure 12-1. Although the i-node number is numbered from 0, the first
// node 0 is not used, and the corresponding node 0 structure is not saved on the disk. Therefore,
// the first disk block storing the i-node holds the i-node structure with the i-node number
// 1--32 instead of 0--31. Therefore, it is necessary to decrement by one when calculating the
// disk block number of the i-node corresponding to the i-node number. Here we read the logical
// block where the i-node is located from the device and copy the contents of the i-node to
// the location indicated by the inode pointer.
        block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
                (inode->i_num-1)/INODES_PER_BLOCK;
        if (!(bh=bread(inode->i_dev,block)))
                panic("unable to read i-node block");
        *(struct d_inode *)inode =
                ((struct d_inode *)bh->b_data)
                        [(inode->i_num-1)%INODES_PER_BLOCK];
// Finally, the buffer block read in is released and the i-node is unlocked. For block device
// files, you also need to set the maximum length of the file for the i-node.
        brelse(bh);
        if (S_ISBLK(inode->i_mode)) {
                int i = inode->i_zone[0];   // i_zone[0] contains dev nr for device file.
                if (blk_size[MAJOR(i)])
                        inode->i_size = 1024*blk_size[MAJOR(i)][MINOR(i)];
                else
                        inode->i_size = 0x7fffffff;
        }
        unlock_inode(inode);
}

//// Write the i-node information to the buffer block.
// This function writes the specified i-node to the buffer block of the cache, and writes it
// to the disk when the cache is refreshed. In order to determine the logical block number of
// the device where the i-node is located, you need to first read the super-block on the device.
// After calculating the logical block number where the i-node is located, the logical block
// is read into a buffer block, and then the contents of the i-node are copied to the corresponding
// positions of the buffer block.
static void write_inode(struct m_inode * inode)
{
        struct super_block * sb;
        struct buffer_head * bh;
        int block;

// The i-node is first locked. If the i-node has not been modified or the device number of the
// i-node is equal to zero, the i-node is unlocked and exits. For an i-node that has not been
// modified, its content is the same as in the buffer or in the device. Then get the super-block
// of the i-node.
        lock_inode(inode);
        if (!inode->i_dirt || !inode->i_dev) {
                unlock_inode(inode);
                return;
        }
        if (!(sb=get_super(inode->i_dev)))
                panic("trying to write inode without device");
// Then we read the logical block where the i-node is located from the device, and copy the
// i-node information to the logical block corresponding to the location of the i-node item.
        block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
                (inode->i_num-1)/INODES_PER_BLOCK;
        if (!(bh=bread(inode->i_dev,block)))
                panic("unable to read i-node block");
        ((struct d_inode *)bh->b_data)
                [(inode->i_num-1)%INODES_PER_BLOCK] =
                        *(struct d_inode *)inode;
// Then we set the buffer modified flag, and the i-node content is already consistent with the
// buffer, so the i-node dirty flag is set to zero. Finally, the buffer containing the i-node
// is released and the i-node is unlocked.
        bh->b_dirt=1;
        inode->i_dirt=0;
        brelse(bh);
        unlock_inode(inode);
}

```

12.5.3 Information

None :)

12.6 super.c

12.6.1 Function

The super.c program contains functions that operate on the superblocks in the file system. These functions belong to the file system low-level functions and are used by the upper-level functions that deal with the filename and directories or pathes, mainly get_super(), put_super(), and read_super(). There are also file system load/unload system calls sys_umount() and sys_mount(), and the root file system load function mount_root(). Some other auxiliary functions are similar to those in buffer.c.

The super block mainly stores information about the entire file system. For its structure, see Figure 12-3 in Section 12.1, “Overall Function Description”.

The get_super() function is used to search the corresponding super block in the memory super block array and return the pointer of the super block under the condition of the specified device. Therefore, when the function is called, the corresponding file system must have been mounted, or at least the superblock has taken up an entry in the superblock array, otherwise it returns NULL.

The put_super() function is used to release the super block of the specified device. It releases the buffer block occupied by the i-node bitmap and the logical block bitmap of the file system corresponding to the super block, and releases the corresponding operation block item in the super block table (array) super_block[]. This function is called when the user calls umount() to unload a file system or replace a disk.

The read_super() function is used to read the super block of the file system of the specified device into the buffer and register it into the super block table. At the same time, the i-node bitmap and the logical block bitmap of the file system are also read into the super block structure in the memory array. Finally return a pointer to the super block structure.

The sys_umount() system-call is used to unmount a file system with a specified device file name, while sys_mount() is used to load a file system to a directory name. The last function in the program, mount_root(), is the root filesystem used to install the system and will be called when the system is initialized. The specific operation process is shown in Figure 12-25.

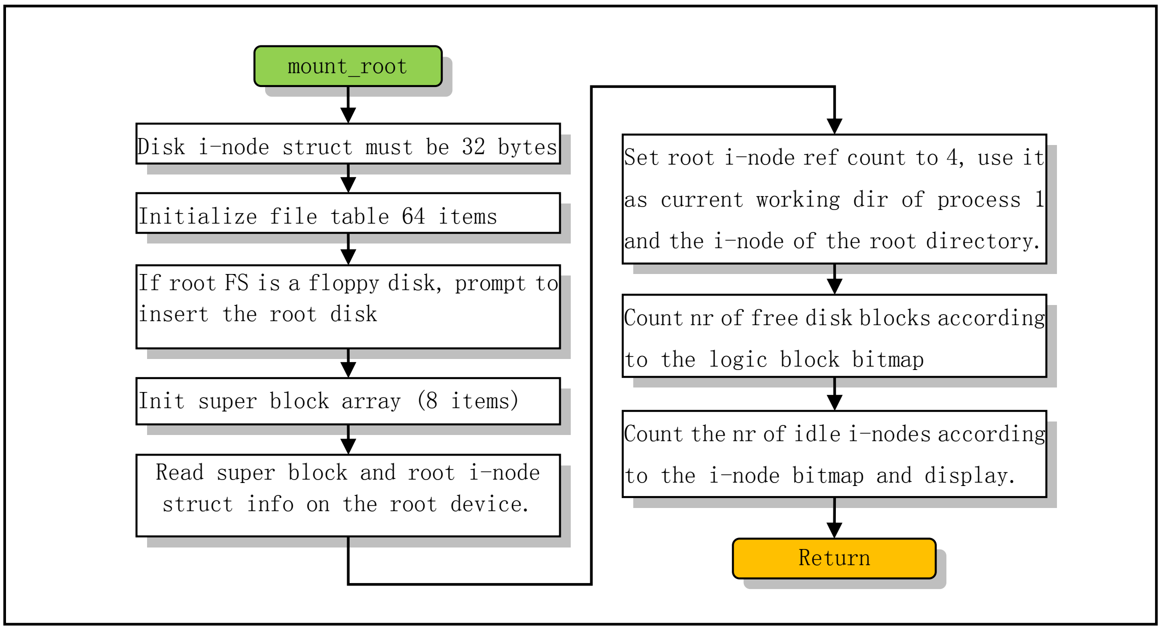

mount_root
Disk i-node struct must be 32 bytes
Initialize file table 64 items
If root FS is a floppy disk, prompt to insert the root disk
Init super block array (8 items)
Read super block and root i-node struct info on the root device.
Set root i-node ref count to 4, use it as current working dir of process 1 and the i-node of the root directory.
Count nr of free disk blocks according to the logic block bitmap
Count the nr of idle i-nodes according to the i-node bitmap and display.
Return

Figure 12-25 The purpose of the mount_root() function

In addition to installing the root file system, this function also initializes the kernel using the file system. It initializes the in-memory superblock array, initializes the file descriptor array table file_table[], and displays the number of free disk blocks and idle i-nodes in the root file system.

The mount_root() function is called in the system initialization file main.c, after process 0 creates the first child process (process 1), and the system only calls it once. The specific location to be called is in the setup() function of the initialization function init() in the main.c program. setup() is actually a system-call whose implementation code starts at line 74 of /kernel/blk_drv/hd.c.

12.6.2 Code annotation

Program 12-5 linux/fs/super.c

```
/*
 *  linux/fs/super.c
```

```
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   * super.c contains code to handle the super-block tables.
9   */
   // <linux/config.h> Kernel configuration header file. Define keyboard language and hard disk
   //     type (HD_TYPE) options.
   // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
   //     of the initial task 0, and some embedded assembly function macro statements about the
   //     descriptor parameter settings and acquisition.
   // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
   //     used functions of the kernel.
   // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
   //     descriptors/interrupt gates, etc. is defined.
   // <errno.h> Error number header file. Contains various error numbers in the system.
   // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
   //     and constants.
10 #include <linux/config.h>
11 #include <linux/sched.h>
12 #include <linux/kernel.h>
13 #include <asm/system.h>
14
15 #include <errno.h>
16 #include <sys/stat.h>
17
   // Perform a buffer cache synchronization with the data in the device (fs/buffer.c, line 59).
18 int sync_dev(int dev);
19 void wait_for_keypress(void);       // kernel/chr_drv/tty_io.c, line 140.
20
21 /* set_bit uses setb, as gas doesn't recognize setc */
   //// Tests the bit at the specified bit offset and returns the bit value.
   // Similar to the inline assembly macro defined on line 19 of the bitmap.c program, but this
   // macro only tests and returns the bit settings, without any changes to the bits (so the name
   // test_bit() may be more appropriate).
   // Input: %0 - ax(__res); %1 - eax(0); %2 - bitnr, bit offset value; %3 - (addr), starting address.
   // Line 23 defines a local register variable '__res'. This variable will be saved in the eax
   // register for efficient access and operation. The entire macro definition is a statement
   // expression whose value is the value of the last '__res'. The BT instruction on line 24 is
   // used to test the bit. It puts the value of the bit specified by the address addr (%3) and
   // the bit offset bitnr (%2) into the carry flag CF. The SETB instruction is used to set the
   // operand %al according to the carry flag CF. If CF = 1, then %al = 1, otherwise %al = 0.
22 #define set_bit(bitnr,addr) ({ \
23 register int __res __asm__("ax"); \
24 __asm__("bt %2,%3;setb %%al":"=a" (__res):"a" (0),"r" (bitnr),"m" (*(addr))); \
25 __res; })
26
27 struct super_block super_block[NR_SUPER];   // Super block table array (NR_SUPER = 8).
28 /* this is initialized in init/main.c */
29 int ROOT_DEV = 0;                           // Root file system device number.
30
   //// Lock the super block.
```

```
// If the superblock has been locked, the current task is placed in an uninterruptible wait
// state and added to the superblock wait queue s_wait until the superblock is unlocked and
// the task is explicitly woken up. Then lock it.
// The following three functions (lock_super(), free_super(), and wait_on_super()) have the
// same effect as the first three functions in the inode.c file, except that the object being
// manipulated here is replaced by a super block.
static void lock_super(struct super_block * sb)
{
        cli();                               // disable interrupt
        while (sb->s_lock)                   // sleep if super block is already locked.
                sleep_on(&(sb->s_wait));     // kernel/sched.c, line 199.
        sb->s_lock = 1;                      // lock it!
        sti();                               // enable interrupt
}

//// Unlock the specified super block.
// Resets the lock flag of the super block and explicitly wakes up all processes waiting on
// this wait queue s_wait (If we use the name unlock_super(), it might be more appropriate).
static void free_super(struct super_block * sb)
{
        cli();
        sb->s_lock = 0;                      // reset the lock.
        wake_up(&(sb->s_wait));              // kernel/sched.c, line 188.
        sti();
}

//// Sleep waiting for the super block to unlock.
// If the superblock has been locked, the current task is placed in an uninterruptible wait
// state and added to the wait queue s_wait of the superblock. Until the super block unlocks
// and explicitly wakes up the task.
static void wait_on_super(struct super_block * sb)
{
        cli();
        while (sb->s_lock)                   // sleep if super block is already locked.
                sleep_on(&(sb->s_wait));
        sti();
}

//// Get super block of the specified device.
// Search for the super block structure of the specified device dev in the super block table
// (array). Returns a pointer to the superblock if found, otherwise returns a null pointer.
struct super_block * get_super(int dev)
{
        struct super_block * s;              // superblock data structure pointer.

// First check the validity of the device given by the parameter. If the device number is 0,
// the null is returned. Then let s point to the beginning of the superblock array and start
// searching the entire superblock array to find the superblock of the specified device dev.
        if (!dev)
                return NULL;
        s = 0+super_block;
        while (s < NR_SUPER+super_block)
// If the current search term is a superblock of the specified device, ie the device number
```

```
   // field value of the superblock is the same as specified by the function parameter, then the
   // superblock is awaited to be unlocked (if it has been locked by another process). During the
   // waiting period, the super block item may be used by other devices, so it is necessary to
   // check again after the sleep returns whether it is still the super block of the specified
   // device. If it is, return the pointer of the super block, otherwise search the super block
   // array again, so s needs to point to the beginning of the super block array again.
64                 if (s->s_dev == dev) {
65                         wait_on_super(s);
66                         if (s->s_dev == dev)
67                                 return s;
68                         s = 0+super_block;
   // If the current search term is not, check the next item. Finally, if the specified superblock
   // is not found, a null pointer is returned.
69                 } else
70                         s++;
71         return NULL;
72 }
73
   //// Release (put back) super block of the specified device.
   // Releases the superblock array item used by the device (sets its s_dev=0) and releases the
   // buffer cache blocks occupied by the device's inode bitmap and logic block bitmap. If the
   // file system corresponding to the super block is the root file system, or another file system
   // is already installed on one of its i nodes, the super block cannot be released.
74 void put_super(int dev)
75 {
76         struct super_block * sb;
77         int i;
78
   // First check the validity and legality of the parameters. If the specified device is a root
   // file system device, the warning message is displayed and returned. Then look for the file
   // system superblock of the specified device in the superblock table. If the super block indicates
   // that the i-node to which the file system is installed has not been processed, a warning message
   // is displayed and returned. In the file system unmount operation, s_imount will be set to
   // null before calling this function, see line 192.
79         if (dev == ROOT_DEV) {
80                 printk("root diskette changed: prepare for armageddon\n\r");
81                 return;
82         }
83         if (!(sb = get_super(dev)))
84                 return;
85         if (sb->s_imount) {
86                 printk("Mounted disk changed - tssk, tssk\n\r");
87                 return;
88         }
   // After finding the super block of the specified device, we first lock it, and then set its
   // device number field s_dev to 0, that is, release the file system super block on the device.
   // Then release the other kernel resources occupied by the super block, that is, release the
   // buffer blocks occupied by the file system i-node bitmap and the logical block bitmap in the
   // buffer cache. The following constant symbols I_MAP_SLOTS and Z_MAP_SLOTS are both equal to
   // 8, which are used to indicate the number of disk logical blocks occupied by the i-node bitmap
   // and the logical block bitmap, respectively. Note that if the contents of these buffer blocks
   // have been modified, a synchronization operation is required to write the data in the buffer
   // block to the device. The function finally unlocks the super block and returns.
```

```
        lock_super(sb);
        sb->s_dev = 0;                        // Set the super block to be idle.
        for(i=0;i<I_MAP_SLOTS;i++)
                brelse(sb->s_imap[i]);
        for(i=0;i<Z_MAP_SLOTS;i++)
                brelse(sb->s_zmap[i]);
        free_super(sb);
        return;
}

//// Read the super block of the specified device.
// If the file system superblock on the specified device dev is already in the superblock table,
// the pointer to the superblock entry is returned directly. Otherwise, the super block is read
// from the device dev into the buffer block and copied into the super block table. Finally
// return the super block pointer.
static struct super_block * read_super(int dev)
{
        struct super_block * s;
        struct buffer_head * bh;
        int i,block;

// First check the validity of the parameter and then check if the device has replaced the disc
// (ie whether it is a floppy device). If the disk is replaced, all the buffer blocks in the
// buffer cache are invalid and need to be invalidated, that is, the original loaded file system
// is released.
        if (!dev)
                return NULL;
        check_disk_change(dev);
// If the device's superblock is already in the superblock table, the pointer to the superblock
// is returned directly. Otherwise, find a free item in the super block array (that is, the
// item with the field s_dev=0). Returns a null pointer if the array is already full.
        if (s = get_super(dev))
                return s;
        for (s = 0+super_block ;; s++) {
                if (s >= NR_SUPER+super_block)
                        return NULL;
                if (!s->s_dev)
                        break;
        }
// After a free entry is found in the superblock array, it is used to specify the file system
// on device dev. The memory fields in the super block structure are then partially initialized.
        s->s_dev = dev;
        s->s_isup = NULL;
        s->s_imount = NULL;
        s->s_time = 0;
        s->s_rd_only = 0;
        s->s_dirt = 0;
// The superblock is then locked and its information is read from the device into the buffer
// block pointed to by bh. The super block is located in the second logical block (block 1)
// of the block device. If the read superblock operation fails, the item in the super block
// array selected above is released (ie, s_dev=0), and the item is unlocked, and a null pointer
// is returned. Otherwise, the read super block is copied from the buffer block data area to
// the corresponding item of the super block array, and the buffer block storing the read
// information is released.
        lock_super(s);
        if (!(bh = bread(dev,1))) {
                s->s_dev=0;
                free_super(s);
                return NULL;
        }
        *((struct d_super_block *) s) =
                *((struct d_super_block *) bh->b_data);
        brelse(bh);
// Now that we have obtained the superblock of the file system from device dev, we start checking
// the validity of this superblock and read the i-node bitmap and logical block bitmap from
// the device. If the file system magic number field of the read super block is incorrect, it
// means that the file system is not the correct one. Therefore, as in the above, the item in
// the super block array selected above is released, and the item is unlocked, and the empty
// pointer is returned. For this version of the Linux kernel, only the MINIX file system version
// 1.0 is supported, and the magic number is 0x137f.
        if (s->s_magic != SUPER_MAGIC) {
                s->s_dev = 0;
                free_super(s);
                return NULL;
        }

// The following begins to read the i-node bitmap and logic block bitmap data on the device.
// First initialize the bitmap space in the memory superblock structure. Then, the i-node bitmap
// and the logical block bitmap information are read from the device and stored in the
// corresponding field of the super block. The i-node bitmap is stored in the logical block
// starting from block 2 on the device, occupying a total of s_imap_blocks blocks. The logic
// block bitmap occupies s_zmap_blocks blocks in subsequent blocks.
        for (i=0;i<I_MAP_SLOTS;i++)                // initialize.
                s->s_imap[i] = NULL;
        for (i=0;i<Z_MAP_SLOTS;i++)
                s->s_zmap[i] = NULL;
        block=2;
        for (i=0 ; i < s->s_imap_blocks ; i++)     // read i-node bitmap in the device.
                if (s->s_imap[i]=bread(dev,block))
                        block++;
                else
                        break;
        for (i=0 ; i < s->s_zmap_blocks ; i++)   // Read logic block bitmap in the device.
                if (s->s_zmap[i]=bread(dev,block))
                        block++;
                else
                        break;
// If the number of bitmap blocks read is not equal to the number of logical blocks that the
// bitmap should occupy, it indicates that there is a problem with the file system bitmap, and
// the super block initialization fails. Therefore, all resources that were previously applied
// and occupied can be released, that is, the cache block occupied by the i-node bitmap and
// the logical block bitmap is released, the super block array item selected above is released,
// the super block item is unlocked, and a null pointer is returned. .
        if (block != 2+s->s_imap_blocks+s->s_zmap_blocks) {
                for(i=0;i<I_MAP_SLOTS;i++)       // release buffer blocks used by bitmap.
                        brelse(s->s_imap[i]);
                for(i=0;i<Z_MAP_SLOTS;i++)
                        brelse(s->s_zmap[i]);
                s->s_dev=0;                      // releases the selected super block item.
                free_super(s);                   // unlock the super block item.
                return NULL;
        }
// Otherwise everything is OK. In addition, for a function that requests an idle i-node, if
// all i-nodes on the device have been used, the lookup function will return a value of zero.
// Therefore, the 0th i-node is not available, so the lowest bit of the 1st block in the bitmap
// is set to 1 to prevent the file system from assigning the 0th i node. For the same reason,
// the lowest bit of the logic block bitmap is also set to 1. The last function unlocks the
// superblock and returns the superblock pointer.
        s->s_imap[0]->b_data[0] |= 1;
        s->s_zmap[0]->b_data[0] |= 1;
        free_super(s);
        return s;
}

//// Unmount file system system-call.
// The parameter 'dev_name' is the file name of the device where the file system is located.
// The function first obtains the device number based on the given block device file name, then
// resets the corresponding field in the file system super block, releasing the buffer block
// occupied by the super block and the bitmap. Finally, the buffer cache is synchronized with
// the data on the device. Returns 0 if the unmount operation succeeds, otherwise returns an
// error code.
int sys_umount(char * dev_name)
{
        struct m_inode * inode;
        struct super_block * sb;
        int dev;

// First find its i-node according to the device file name, and get the device number. The
// device number defined by the device file is stored in i_zone[0] of its i-node. See line 445
// of the system call sys_mknod() in the fs/namei.c program. In addition, since the file system
// needs to be stored on the block device, if it is not a block device file, the i-node dev_i
// just returned is put back, and the error code is returned.
        if (!(inode=namei(dev_name)))
                return -ENOENT;
        dev = inode->i_zone[0];
        if (!S_ISBLK(inode->i_mode)) {
                iput(inode);                            // fs/inode.c, line 150.
                return -ENOTBLK;
        }
// OK, now the i-node obtained to get the device number has completed its mission, so put back
// the i-node here. Then we check if the conditions for unmounting the file system are met.
// If the device is a root file system, it cannot be unmounted, and a busy error is returned.
// If the super block of the file system on the device is not found in the super block table,
// or the file system on the device is not installed, an error code is returned. If the i-node
// to which it is mounted does not set its flag i_mount, a warning message is displayed. Then
// look up the i-node table to see if any processes are using the files on the device, and if
// so, return a busy error code.
        iput(inode);
        if (dev==ROOT_DEV)
```

```
                return -EBUSY;
        if (!(sb=get_super(dev)) || !(sb->s_imount))
                return -ENOENT;
        if (!sb->s_imount->i_mount)
                printk("Mounted inode has i_mount=0\n");
        for (inode=inode_table+0 ; inode<inode_table+NR_INODE ; inode++)
                if (inode->i_dev==dev && inode->i_count)
                                return -EBUSY;
// The unmounting conditions for the file system on the device are now met, so we can start
// implementing the actual unmount operation. First reset the mount flag of the i-node to which
// it is mounted, release the i-node, then set the mounted i-node field in the superblock to
// be NULL, and put back the root i-node of the device file system. Then set the pointer of
// the root i node of the mounted file system in the super block to be NULL.
        sb->s_imount->i_mount=0;
        iput(sb->s_imount);
        sb->s_imount = NULL;
        iput(sb->s_isup);
        sb->s_isup = NULL;
// Finally, we release the superblock on the device and the buffer block occupied by the bitmap,
// and perform a synchronization operation between the cache and the data on the device, and
// then return 0 (unmounted successfully).
        put_super(dev);
        sync_dev(dev);
        return 0;
}

//// Mount (new) file system system-call.
// The parameter dev_name is the device file name, dir_name is the directory name to be mounted,
// and rw_flag is the read/write flag of the mounted file system. The location to be mounted
// must be a directory name, and the corresponding i-node is not occupied by other programs.
// Returns 0 if successful, otherwise returns an error number.
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
{
        struct m_inode * dev_i, * dir_i;
        struct super_block * sb;
        int dev;

// First, find the corresponding i-node according to the device file name to get the device
// number. For block special device files, the device number is in i_zone[0] of its i-node.
// In addition, since the file system must be in the block device, if not, the i-node dev_i
// just obtained is put back and the error code is returned.
        if (!(dev_i=namei(dev_name)))
                return -ENOENT;
        dev = dev_i->i_zone[0];
        if (!S_ISBLK(dev_i->i_mode)) {
                iput(dev_i);
                return -EPERM;
        }
// OK, now the device file i-node dev_i obtained in order to get the device number has completed
// its mission, so put it back here. Then let's check if the directory name to which the file
// system will be mounted on is valid. Then obtaining the corresponding i-node dir_i according
// to the given directory file name, and if the reference count of the i-node is not 1 (only
// referenced here), or the node number is 1 of the root file system, then putting back the
// i node and returns an error code. In addition, if the node is not a directory file node,
// the i-node is also put back, and the error code is returned because the new file system can
// only be mounted on a directory name.
        iput(dev_i);
        if (!(dir_i=namei(dir_name)))
                return -ENOENT;
        if (dir_i->i_count != 1 || dir_i->i_num == ROOT_INO) {
                iput(dir_i);
                return -EBUSY;
        }
        if (!S_ISDIR(dir_i->i_mode)) {      // mount point needs to be a directory entry.
                iput(dir_i);
                return -EPERM;
        }
// Now that the mount point is checked, we start reading the super block of the new file system.
// A file system's superblock will first search from the superblock table, and read from the
// device if it is not in the superblock table. After getting the super block of the new file
// system, we check it first. We want to make sure that the new file system has not been mounted
// elsewhere, and that the i-node to be mounted is not occupied (by other file system), otherwise
// error occurred and returned.
        if (!(sb=read_super(dev))) {
                iput(dir_i);
                return -EBUSY;
        }
        if (sb->s_imount) {                 // if new fs has been mounted elsewhere.
                iput(dir_i);
                return -EBUSY;
        }
        if (dir_i->i_mount) {               // if the mount point is occupied.
                iput(dir_i);
                return -EPERM;
        }
// Finally, we set the "mounted i-node" field 's_imount' of the new file system superblock to
// point to the i-node of the directory to which it is mounted, and set the mount flag and node
// modified flag of the mount location i-node, and then return 0 (installation is successful).
        sb->s_imount=dir_i;
        dir_i->i_mount=1;
        dir_i->i_dirt=1;                /* NOTE! we don't iput(dir_i) */
        return 0;                       /* we do that in umount */
}

//// Mount root file system.
// This function is part of the system initialization operation. It first initializes the file
// table file_table[] and the super block table, then reads the root file system super block
// and gets the root i node. Finally, the available resources (the number of free blocks and
// the number of idle i nodes) on the root file system are counted and displayed. This function
// is called at system initialization (sys_setup() in file blk_drv/hd.c, line 157).
void mount_root(void)
{
        int i,free;
        struct super_block * p;
        struct m_inode * mi;

```

```
// First check if the size of the disk i-node structure meets the requirements (32 bytes) to
// prevent inconsistencies when modifying the code. Then initialize the file table (64 items)
// and the super block table (8 items). Here, the reference count in all file structures is
// set to 0 (indicating idle), and the device field of each structure in the superblock table
// is initialized to 0 (also indicating idle). If the device where the root file system is located
// is a floppy disk, it prompts "Insert root floppy and press ENTER" and wait for the key.
        if (32 != sizeof (struct d_inode))
                panic("bad i-node size");
        for(i=0;i<NR_FILE;i++)                     // initialize file table (64 items).
                file_table[i].f_count=0;
        if (MAJOR(ROOT_DEV) == 2) {
                printk("Insert root floppy and press ENTER");
                wait_for_keypress();
        }
        for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {
                p->s_dev = 0;                      // initialize super block table (8).
                p->s_lock = 0;
                p->s_wait = NULL;
        }
// After doing the "extra" initialization work above, we started to mount the root file system.
// The file system superblock is then read from the root device and a pointer to the root i-node
// (node 1) of the file system in the memory i-node table is obtained. If the superblock on
// the device fails or the root node fails, the message is displayed and the device is down.
        if (!(p=read_super(ROOT_DEV)))
                panic("Unable to mount root");
        if (!(mi=iget(ROOT_DEV,ROOT_INO)))        // ROOT_INO is defined as 1 in fs.h.
                panic("Unable to read root i-node");
// Now we setup the super block and the root i-node, and increment the reference number of the
// root i-node by 3 times. Because the i-node is also referenced in the next few lines of code,
// in addition, the i-node reference count in the iget() function has already been set to 1.
// Then, the mounted file system root i-node field (s_isup) and the i-node field (s_imount)
// of the super block are set to this i-node, and then the current working directory and the
// root i-node of the current process are set too. At this point, the current process is the
// number 1 process (init process).
        mi->i_count += 3 ;      /* NOTE! it is logically used 4 times, not 1 */
        p->s_isup = p->s_imount = mi;
        current->pwd = mi;
        current->root = mi;
// Then we count the resources on the root file system and count the number of free blocks and
// the number of idle i-nodes on the device. First, let 'i' be equal to the total number of
// logical blocks of the device indicated in the super block, and then count the number of free
// blocks according to the occupancy of the corresponding bits in the logical block bitmap.
// Here the macro function set_bit() is only used to test the bits. "i&8191" is used to obtain
// the corresponding bit offset value of the i-node number in the current bitmap block. "i>>13"
// divides i by 8192, which is the number of bits contained in a disk block.
        free=0;
        i=p->s_nzones;
        while (-- i >= 0)
                if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
                        free++;
// After displaying the number of free logical blocks on the device, we then count the number
// of free i-nodes on the device. First, let 'i' be equal to the total number of i-nodes on
// the device indicated in the super block +1 (add 1 to count the 0 nodes as well), and then
// calculate the number of idle i-nodes according to the occupancy of the corresponding bits
// in the i-node bitmap. Finally, the number of free i-nodes and the total number of i-nodes
// available on the device are displayed.
        printk("%d/%d free blocks\n\r",free,p->s_nzones);
        free=0;
        i=p->s_ninodes+1;
        while (-- i >= 0)
                if (!set_bit(i&8191,p->s_imap[i>>13]->b_data))
                        free++;
        printk("%d/%d free inodes\n\r",free,p->s_ninodes);
}

```

12.7 namei.c

12.7.1 Function

The namei.c program is the longest file in the Linux 0.12 kernel, but it only has about 900 lines of code :). This file mainly implements the function namei() which finds the corresponding i-node according to the directory name or file name, as well as some operation functions and system calls about the creation and deletion of the directory, the creation and deletion of the directory entries.

The Linux 0.12 system uses the MINIX file system version 1.0. Its directory entry structure is the same as that of a traditional UNIX file, and is defined in the include/linux/fs.h file. In a directory of the file system, the directory entries corresponding to all file names are stored in the data block of the directory file name. For example, the directory entry of all file names under the directory name home/ is stored in the data block of the home/ directory name file; and all files under the file system root directory are stored in the data block of the specified i-node (ie node 1). Each directory entry includes only a file name string of 14 bytes in length and a 2-byte i-node number corresponding to the file name, as shown below.

```
// Defined in the include/linux/fs.h file.
#define NAME_LEN 14                             // file name maximum length.
#define ROOT_INO 1                              // root i-node number.

// File directory entry structure.
struct dir_entry {
        unsigned short inode;                   // i-node number.
        char name[NAME_LEN];                    // filename string
};
```

Other information about the file is saved in the i-node structure specified by the i-node number. The i-node structure mainly includes information such as file access attributes, host, length, access save time, and disk block. The i-node of each inode number is located at a fixed location on the disk.

When a file is opened, the file system finds its i-node number based on the given file name, thereby finding the disk block location where the file is located. For example, to find the i-node number of the file '/usr/bin/vi', the file system first starts from the root directory with a fixed i-node number (1). That is, from the data block of the i-node number 1, the directory entry with the file name 'usr' is found, thereby obtaining the i-node number of the file '/usr'. According to the i-node number file system, the directory '/usr' can be successfully obtained, and the directory entry of the file name 'bin' can be found therein. This also knows the i-node number of '/usr/bin', so we can know the location of the directory '/usr/bin' and find the directory entry for the 'vi' file in that directory. Finally, we obtain the i-node number of the file path name '/usr/bin/vi', so that the i-node structure information of the i-node number can be obtained from the disk.

There are also two special file directory entries in each directory, whose names are fixed to '.' and '..' respectively. The '.' directory entry gives the i-node number of the current directory, and the '..' directory entry gives the i-node number of the direct parent directory of the current directory. Therefore, when a relative path name is given, the file system can use these two special directory entries for lookup operations. For example, to find '../kernel/Makefile', you can first get the i-node number of the parent directory according to the '..' directory entry of the current directory, and then perform the lookup operation according to the process described above.

Since several main functions in the program have detailed comments in front of them, and the usages of each function and system-calls are clear, they will not be described here.

12.7.2 **Code annotation**

Program 12-6 linux/fs/namei.c

```
/*
 *  linux/fs/namei.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * Some corrections by tytso.
 */

// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined
//     for segment register operations.
// <string.h> String header file. Defines some embedded functions about string operations.
// <fcntl.h> File control header file. The definition of the operation control constant symbol
//     used for the file and its descriptors.
// <errno.h> Error number header file. Contains various error numbers in the system.
// <const.h> The constant symbol file currently defines the flags of i_mode field in i-node.
// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//     and constants.
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>

#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <const.h>
```

```
#include <sys/stat.h>

// Internal function that finds the corresponding i-node by file name.
static struct m_inode * _namei(const char * filename, struct m_inode * base,
        int follow_links);

// The right expression in the macro below is a special way to access an array. It is based
// on the fact that the value of an array item (such as a[b]) represented by an array name and
// an array subscript is equivalent to the value of the array pointer (address) plus the offset
// address* ( a + b), at the same time, it can be known that the term a[b] can also be expressed
// in the form of b[a]. So for the character array item form "LoveYou"[2] (or 2["LoveYou"])
// is equivalent to *("LoveYou" + 2). In addition, the position where the string "LoveYou" is
// stored in the memory is its address, so the value of the array item "LoveYou"[2] is the
// character "v" whose index value is 2 in the string, and its ASCII code value is 0x76, or
// 0166 in octal notation. In C, characters can also be represented by their ASCII value by
// adding a backslash to the ASCII value of the character. For example, the character "v" can
// be expressed as "\x76" or "\166". Therefore, characters that cannot be displayed (for example,
//control characters with an ASCII code value in range of 0x00--0x1f) can be represented by
// their ASCII code values.
//
// Below is the file access mode macro. Where 'x' is the file access (open) mode flag defined
// in line 7 of the header file include/fcntl.h. This macro indexes the corresponding value
// in the double quoted string based on the value of the file access token x. The double quotation
// marks have four octal control characters: "\004\002\006\377", which indicate the read, write,
// and execute permissions: r, w, rw, and wxrwxrwx, respectively, and correspond to the index
// value of x: 0--3. For example, if x is 2, the macro returns an octal value of 006, indicating
// readability and writability (rw). In addition, where O_ACCMODE = 00003, is the mask of the
// index value x.
#define ACC_MODE(x) ("\004\002\006\377"[(x)&O_ACCMODE])

/*
 * comment out this line if you want names > NAME_LEN chars to be
 * truncated. Else they will be disallowed.
 */
/* #define NO_TRUNCATE */

#define MAY_EXEC 1
#define MAY_WRITE 2
#define MAY_READ 4

/*
 *      permission()
 *
 * is used to check for read/write/execute permissions on a file.
 * I don't know if we should look at just the euid or both euid and
 * uid, but that should be easily changed.
 */
//// Detect file access permissions.
// Parameters: inode - the i-node pointer of the file; mask - access attribute mask.
// Returns: access allowed returns 1, otherwise returns 0.
static int permission(struct m_inode * inode,int mask)
{
        int mode = inode->i_mode;                   // File access mode.
```

```
46
47 /* special case: not even root can read/write a deleted file */
   // If the i-node has a corresponding device, but the link count is equal to 0, indicating that
   // the file has been deleted, then returns. Otherwise, if the effective user id (euid) of the
   // process is the same as the user id of the i-node, the access permission of the file owner
   // is taken. Otherwise, if the effective group id (egid) of the process is the same as the group
   // id of the i-node, the access rights of the group user is taken.
48         if (inode->i_dev && !inode->i_nlinks)
49                 return 0;
50         else if (current->euid==inode->i_uid)
51                 mode >>= 6;
52         else if (in_group_p(inode->i_gid))
53                 mode >>= 3;
   // Finally, it is judged that if the access right is the same as the mask code, or is a super
   // user, it returns 1; otherwise, it returns 0.
54         if (((mode & mask & 0007) == mask) || suser())
55                 return 1;
56         return 0;
57 }
58
59 /*
60  * ok, we cannot use strncmp, as the name is not in our data space.
61  * Thus we'll have to use match. No big problem. Match also makes
62  * some sanity tests.
63  *
64  * NOTE! unlike strncmp, match returns 1 for success, 0 for failure.
65  */
   //// String matching and comparison function.
   // Parameters: len - the length of the string to compare; name - pointer to the file name;
   // de - the structure of the directory entry.
   // Returns: Returns 1 if they are the same, otherwise returns 0.
66 static int match(int len,const char * name,struct dir_entry * de)
67 {
68         register int same __asm__("ax");          // register variable.
69
   // First determine the validity of the function parameters. Returns 0 if the directory entry
   // pointer is empty, or if the directory entry i-node is null, or if the length to be compared
   // exceeds the file name length NAME_LEN. If the length of the comparison is equal to 0, but
   // the file name in the directory entry is '.', then we consider the same, so return 1 (match).
   // If the given length 'len' to be compared is less than NAME_LEN, but the file name length
   // exceeds the len, then 0 (no match) is also returned.
   // The method for judging whether the file name length exceeds 'len' on line 75 is to detect
   // whether name[len] is NULL. If the length exceeds len, then name[len] is a normal character
   // that is not NULL. For a string name of length len, the character name[len] should be NULL.
70         if (!de || !de->inode || len > NAME_LEN)
71                 return 0;
72         /* "" means "." ---> so paths like "/usr/lib//libc.a" work */
73         if (!len && (de->name[0]=='.') && (de->name[1]=='\0'))
74                 return 1;
75         if (len < NAME_LEN && de->name[len])      // the length of the given name > len.
76                 return 0;
   // Then use the embedded assembly statement for a quick comparison. It performs string
   // comparisons in the user data space (fs segment).
```

```
    // %0 - eax (comparison result, same); %1 - eax (eax initial value 0); %2 - esi (name);
    // %3 - edi (directory name pointer);  %4 - ecx (compared bytes) len).
77          __asm__("cld\n\t"                   // clear direction.
78                  "fs ; repe ; cmpsb\n\t"     // compare [esi++] and [edi++] in user space.
79                  "setz %%al"                 // set al = 1 if identical (same = eax).
80                  :"=a" (same)
81                  :"" (0), "S" ((long) name), "D" ((long) de->name), "c" (len)
82                  :"cx", "di", "si");
83          return same;                        // return the result.
84  }
85
86  /*
87   *      find_entry()
88   *
89   * finds an entry in the specified directory with the wanted name. It
90   * returns the cache buffer in which the entry was found, and the entry
91   * itself (as a parameter - res_dir). It does NOT read the inode of the
92   * entry - you'll have to do that yourself if you want to.
93   *
94   * This also takes care of the few special cases due to '..'-traversal
95   * over a pseudo-root and a mount point.
96   */
    //// Finds the directory entry for the specified file name in the specified directory.
    // Parameters: *dir - directory i-node pointer; name - file name; namelen - filename length.
    // This function searches the data (file) of the specified directory for the directory entry
    // of the given file name, and performs special processing according to the current related
    // settings for the case where the file name is '..'. See the comments before line 151 of
    // linux/sched.c for the role of pointers to pointers in function arguments.
    // Returns: the cache block pointer, if successful, and the directory entry pointer returned
    // at *res_dir. Failure returns a null pointer.
97  static struct buffer_head * find_entry(struct m_inode ** dir,
98          const char * name, int namelen, struct dir_entry ** res_dir)
99  {
100         int entries;
101         int block,i;
102         struct buffer_head * bh;
103         struct dir_entry * de;
104         struct super_block * sb;
105
    // Similarly, this function also needs to judge and verify the validity of the function
    // parameters. If we define the symbol constant NO_TRUNCATE on line 30 above, then if the file
    // name length exceeds the maximum length NAME_LEN, it will not be processed. If NO_TRUNCATE
    // is not defined, it is truncated when the file name length exceeds the maximum length NAME_LEN.
106 #ifdef NO_TRUNCATE
107         if (namelen > NAME_LEN)
108                 return NULL;
109 #else
110         if (namelen > NAME_LEN)
111                 namelen = NAME_LEN;
112 #endif

    // First calculate the number of directory entries in this directory. The i_size field of the
    // i-node of the directory contains the size of the data contained in this directory, so it
    // is divided by the size of a directory entry (16 bytes) to get the number of directory entries.
    // Then null the pointer to the directory entry to be returned.
            entries = (*dir)->i_size / (sizeof (struct dir_entry));
            *res_dir = NULL;

    // Next, we specialize in the case where the directory entry file name is ’..’. If the root
    // i-node specified by the current process is the same directory given by the function parameter,
    // it means that for this process, this directory is its pseudo root directory, that is, the
    // process can only access the items in the directory and cannot fall back to its parent directory.
    // That is, for this process, this directory is now the root directory of the file system. So
    // we need to change the file name to ’.’.
    // Otherwise, if the i-node number of the directory is equal to ROOT_INO (number 1), it is indeed
    // the root i-node of the file system, so the superblock of the file system is taken. If the
    // i-node (s_imount) to which the file system is mounted in the superblock exists, the current
    // operation is in the file system mounted on the s_imount node, then the original i-node is
    // put back first, and then the i-node to be mounted is dealed with. So we let ’*dir’ point
    // to the i-node to which it is mounted; and the number of references to this i-node is incremented
    // by one. That is to say, in response to this situation, we quietly carried out the "stealing
    // the column" project :)
    /* check for '..', as we might have to do some "magic" for it */
            if (namelen==2 && get_fs_byte(name)=='.' && get_fs_byte(name+1)=='.') {
    /* '..' in a pseudo-root results in a faked '.' (just change namelen) */
                    if ((*dir) == current->root)
                            namelen=1;
                    else if ((*dir)->i_num == ROOT_INO) {
    /* '..' over a mount-point results in 'dir' being exchanged for the mounted
       directory-inode. NOTE! We set mounted, so that we can iput the new dir */
                            sb=get_super((*dir)->i_dev);
                            if (sb->s_imount) {             // mounted point.
                                    iput(*dir);
                                    (*dir)=sb->s_imount;
                                    (*dir)->i_count++;
                            }
                    }
            }

    // Now let’s get started and find out where the directory entry for the specified filename is.
    // Therefore, we need to read the data of the directory, that is, take out the data block (logical
    // block) in the data zone of the block device corresponding to the directory i-node. The block
    // numbers of these logical blocks are stored in the i_zone[] array of the i-node structure.
    // We first take the first direct block number saved in it, and then read the specified directory
    // item data block from the device.
            if (!(block = (*dir)->i_zone[0]))
                    return NULL;
            if (!(bh = bread((*dir)->i_dev,block)))
                    return NULL;

    // At this point we can search for the directory entry matching the given file name in the read
    // data block. First let ’de’ point to the data block part of the buffer block, and loop the
    // search without exceeding the number of directory entries in the directory. Where i is the
    // index of the directory entry.
            i = 0;
            de = (struct dir_entry *) bh->b_data;
```

```
137         while (i < entries) {
    // If the current directory entry data block has been searched and no matching entry has been
    // found, the current directory entry data block is released and then read into the next logical
    // block of the directory. If the block is empty, then as long as all the entries in the directory
    // have not been searched, the block is skipped and the next logical block of the directory
    // is read. If the block is not empty, let de point to the data block and continue searching
    // in it. On the 141th line, i/DIR_ENTRIES_PER_BLOCK can get the data block number in the
    // directory file where the directory item currently searched, and the bmap() function (inode.c,
    // line 142) can calculate the corresponding logical block number on the device.
138                 if ((char *)de >= BLOCK_SIZE+bh->b_data) {
139                         brelse(bh);
140                         bh = NULL;
141                         if (!(block = bmap(*dir,i/DIR_ENTRIES_PER_BLOCK)) ||
142                             !(bh = bread((*dir)->i_dev,block))) {
143                                 i += DIR_ENTRIES_PER_BLOCK;
144                                 continue;
145                         }
146                         de = (struct dir_entry *) bh->b_data;
147                 }
    // If a matching directory entry is found, the directory entry structure pointer 'de' and the
    // directory entry i-node pointer '*dir' and the directory entry data block pointer 'bh' are
    // returned, and the function is exited. Otherwise continue to compare the next directory entry
    // in the directory entry data block.
148                 if (match(namelen,name,de)) {
149                         *res_dir = de;
150                         return bh;
151                 }
152                 de++;
153                 i++;
154         }
    // If all the directory entries in the specified directory have been searched and the
    // corresponding directory entry has not been found, the data block of the directory is released,
    // and finally NULL is returned (failed).
155         brelse(bh);
156         return NULL;
157 }
158
159 /*
160  *      add_entry()
161  *
162  * adds a file entry to the specified directory, using the same
163  * semantics as find_entry(). It returns NULL if it failed.
164  *
165  * NOTE!! The inode part of 'de' is left at 0 - which means you
166  * may not sleep between calling this and putting something into
167  * the entry, as someone else might have used it while you slept.
168  */
    //// Adds a file entry to the specified directory.
    // Parameters: dir - i-node of the directory; name - file name; namelen - file name length.
    // Returns: buffer block pointer; res_dir - the pointer to the returned directory entry.
169 static struct buffer_head * add_entry(struct m_inode * dir,
170         const char * name, int namelen, struct dir_entry ** res_dir)
171 {
```

```
172         int block,i;
173         struct buffer_head * bh;
174         struct dir_entry * de;
175 
    // Similarly, this function also needs to judge and verify the validity of the function
    // parameters. If we define the symbol constant NO_TRUNCATE on line 30 above, then if the file
    // name length exceeds the maximum length NAME_LEN, it will not be processed. If NO_TRUNCATE
    // is not defined, it is truncated when the file name length exceeds the maximum length NAME_LEN.
176         *res_dir = NULL;                // the result directory entry pointer.
177 #ifdef NO_TRUNCATE
178         if (namelen > NAME_LEN)
179                 return NULL;
180 #else
181         if (namelen > NAME_LEN)
182                 namelen = NAME_LEN;
183 #endif
    // Now let's get started and add a directory entry with the given filename to the specified
    // directory. Therefore, we need to read the data of the directory, that is, take out the data
    // block (logical block) in the data zone of the block device corresponding to the directory
    // i-node. The block numbers of these logical blocks are stored in the i_zone[] array of the
    // i-node structure. We first take the first direct block number saved in it, and then read
    // the specified directory item data block from the device. In addition, if the file name length
    // provided by the parameter is equal to 0, it returns with a NULL.
184         if (!namelen)
185                 return NULL;
186         if (!(block = dir->i_zone[0]))
187                 return NULL;
188         if (!(bh = bread(dir->i_dev,block)))
189                 return NULL;
    // At this point, we loop through the data blocks of the i-node of this directory to find the
    // last unused empty directory entry.First, let the directory entry pointer point to the data
    // block part of the buffer block, that is, the first directory entry. Where i is the index
    // of the directory entry in the directory.
190         i = 0;
191         de = (struct dir_entry *) bh->b_data;
192         while (1) {
    // If the current directory entry data block has been searched but the required empty directory
    // entry has not been found, the current directory entry data block is released and then the
    // next logical block of the directory is read. Create a block if the corresponding logical
    // block does not exist. Returns null if the read or create operation fails. If the buffer block
    // pointer returned by the disk logical block data read this time is empty, it indicates that
    // the logical block may be a newly created empty block because it does not exist. So we add
    // the directory entry index value to the number of directory entries DIR_ENTRIES_PER_BLOCK
    // that a logical block can hold, to skip the block and continue searching. Otherwise, the newly
    // read block has directory entry data, so the directory entry structure pointer points to the
    // buffer block data portion of the block, and then continues the search there.
    // The i/DIR_ENTRIES_PER_BLOCK on line 196 can be used to calculate the block number in the
    // directory file where the currently searched directory entry i is located, and the function
    // create_block()(inode.c, line 147) can be used to read or created logic block on the device.
193                 if ((char *)de >= BLOCK_SIZE+bh->b_data) {
194                         brelse(bh);
195                         bh = NULL;
196                         block = create_block(dir,i/DIR_ENTRIES_PER_BLOCK);
```

```
                        if (!block)
                                return NULL;
                        if (!(bh = bread(dir->i_dev,block))) {      // skip if empty.
                                i += DIR_ENTRIES_PER_BLOCK;
                                continue;
                        }
                        de = (struct dir_entry *) bh->b_data;
                }
// If the size of the directory entry number i currently multiplied by the directory entry size
// exceeds the directory data size i_size indicated by the i-node of the directory, it means
// that there is no empty entries left due to the deletion of file. So we can only append new
// directory entry that need to be added to the end of the directory data file. Therefore, we
// need to set the directory entry for this directory, set the i-node number of the directory
// entry to be empty, and update the size of the directory file (plus the length of a directory
// entry), and then set the i-node of the directory to be modified, and then update the change
// time of this directory to the current time.
                if (i*sizeof(struct dir_entry) >= dir->i_size) {
                        de->inode=0;
                        dir->i_size = (i+1)*sizeof(struct dir_entry);
                        dir->i_dirt = 1;
                        dir->i_ctime = CURRENT_TIME;
                }
// If the i-node number of the currently searched directory entry 'de' is empty, it means that
// a free directory entry that has not been used or a new directory entry added is found.
// Therefore, we update the directory modification time to the current time, and copy the file
// name from the user data space to the file name field of the directory entry, and set the
// corresponding cache buffer block modified flag of the directory entry. Returns with a pointer
// to the directory entry and a pointer to the cache buffer block.
                if (!de->inode) {
                        dir->i_mtime = CURRENT_TIME;
                        for (i=0; i < NAME_LEN ; i++)
                                de->name[i]=(i<namelen)?get_fs_byte(name+i):0;
                        bh->b_dirt = 1;
                        *res_dir = de;
                        return bh;
                }
                de++;           // if entry is in use, continue to check the next entry.
                i++;
        }
// This function cannot be executed here. This may be because Mr. Linus copied the code of the
// find_entry() function above and then modified it into this function :).
        brelse(bh);
        return NULL;
}

//// Find the i-node of the symbolic link.
// Parameters: dir - directory i-node; inode - directory entry i-node.
// Returns: i-node pointer to the file to which the symbolic link is located. NULL if error.
static struct m_inode * follow_link(struct m_inode * dir, struct m_inode * inode)
{
        unsigned short fs;                // Used to temporarily save FS segment register.
        struct buffer_head * bh;

```

```
// First determine the validity of the function parameters. If the directory i-node is not given,
// we use the root i-node set in the process task structure and increase the number of links
// by one. If the directory entry i-node is not given, the directory i-node is put back and
// NULL is returned. If the specified directory entry is not a symbolic link, we directly return
// the i-node corresponding to the directory entry.
        if (!dir) {
                dir = current->root;
                dir->i_count++;
        }
        if (!inode) {
                iput(dir);
                return NULL;
        }
        if (!S_ISLNK(inode->i_mode)) {
                iput(dir);
                return inode;
        }
// Then get the FS segment register value. The FS usually holds the selector 0x17 pointing to
// the task (user) data segment. If FS does not point to the user data segment, or if the first
// direct block number of the given directory entry i-node is equal to 0, or an error occurs
// when reading the first direct block, put back the dir and inode nodes and return NULL.
// Otherwise, the FS is now pointing to the user data segment, and we have successfully read
// the file contents of this symbolic link directory entry, and the file content is already
// in the buffer block data area pointed to by bh. In fact, this buffer block data area contains
// only one file path name string pointed to by the link.
        __asm__("mov %%fs,%0":"=r" (fs));              // get FS contents.
        if (fs != 0x17 || !inode->i_zone[0] ||
           !(bh = bread(inode->i_dev, inode->i_zone[0]))) {
                iput(dir);
                iput(inode);
                return NULL;
        }
// At this point we can use the data content of the symbolic link file to find the i-node of
// the file to which it is linked. Now we no longer need the i-node information of the symbolic
// link directory entry, so we put it back. Now we have a problem, that is, the user data processed
// by the kernel function should be stored in the user data space by default, and use the FS
// segment register to transfer data from the user space to the kernel space. But the data that
// needs to be processed here is in kernel space. Therefore, in order to correctly process the
// user data located in the kernel space, we need to temporarily point the FS segment register
// to the kernel space, that is, let FS = 0x10, and restore the original FS after the function
// namei() is called. Finally, the buffer block is released, and the file i-node pointed to
// by the symbol link obtained by _namei() is returned.
        iput(inode);
        __asm__("mov %0,%%fs"::"r" ((unsigned short) 0x10));
        inode = _namei(bh->b_data,dir,0);
        __asm__("mov %0,%%fs"::"r" (fs));
        brelse(bh);
        return inode;
}

/*
 *      get_dir()
 *
 * Getdir traverses the pathname until it hits the topmost directory.
 * It returns NULL on failure.
 */
//// Search for the i-node of the topmost directory of the given pathname.
// Parameters: pathname - the path name; inode - the i-node specifying the starting directory.
// Returns: the i-node pointer of the directory, or NULL if it fails.
static struct m_inode * get_dir(const char * pathname, struct m_inode * inode)
{
        char c;
        const char * thisname;
        struct buffer_head * bh;
        int namelen,inr;
        struct dir_entry * de;
        struct m_inode * dir;

// First determine the validity of the parameters. If the i-node pointer inode of the given
// directory is empty, the current working directory i-node of the current process is used.
// If the user specifies that the first character of the path name is '/', the path name is
// an absolute path name, and then we should start from the root (or pseudo root) i-node set
// in the current process task structure. So we need to put back the directory i-node specified
// or set by the parameter and get the root i-node used by the process, then increment the
// reference count of the i-node and delete the first character '/' of the path name. This ensures
// that the current process can only use the root i-node it sets as the starting point for the
// search.
        if (!inode) {
                inode = current->pwd;                // i-node of the working directory.
                inode->i_count++;
        }
        if ((c=get_fs_byte(pathname))=='/') {
                iput(inode);                         // put back the original i-node.
                inode = current->root;          // root i-node specified for the process.
                pathname++;
                inode->i_count++;
        }
// Then loop through the various directory names and file name in the path name. In the loop
// processing, we first determine the validity of the i-node of the directory name currently
// being processed, and point the variable 'thisname' to the part of the directory name currently
// being processed. If the i-node indicates that the currently processed directory name portion
// is not a directory type, or there is no access permission to enter the directory, then the
// i-node is put back and a NULL is returned. Of course, when entering the loop, the i-node
// 'inode' of the current directory is the process root i-node or the i-node of the current
// working directory, or the i-node of a search starting directory specified by the parameter.
        while (1) {
                thisname = pathname;
                if (!S_ISDIR(inode->i_mode) || !permission(inode,MAY_EXEC)) {
                        iput(inode);
                        return NULL;
                }
// In each loop we process one directory name in the pathname. So in each loop we have to separate
// a directory name from the pathname string. The method is to search for the detected character
// from the current pathname pointer 'pathname' until the character is a trailing character
// (NULL) or a '/' character. At this point the variable 'namelen' is exactly the length of
// the currently processed directory name part, and the variable 'thisname' is pointing to the
// beginning of the directory name part. At this time, if the character is the end character
// NULL, it indicates that the end of the path name has been searched, and the last specified
// directory name or file name has been reached, then the i-node pointer is returned.
// NOTE: If the last name in the path name is also a directory name, but the '/' character is
// not appended to it, the function will not return the i-node of the last directory name!
// For example: For the path name '/usr/src/linux', this function will only return the i-node
// of the 'src/' directory name.
                for(namelen=0;(c=get_fs_byte(pathname++))&&(c!='/');namelen++)
                        /* nothing */ ;
                if (!c)
                        return inode;
// After getting the current directory name part (or file name), we call the lookup directory
// entry function find_entry() to find the directory entry with the specified name in the
// currently processed directory. (If not found, put back the i-node and return NULL.) Then,
// the i-node number 'inr' and the device number 'idev' are taken out in the found directory
// entry, the cache block containing the directory entry is released, and the i-node is put
// back. Then take the i-node 'inode' of the node number 'inr', and continue to loop the next
// directory name part in the path name with the directory entry as the current directory. If
// the currently processed directory entry is a symbolic link name, then follow_link() is used
// to get the i-node of the directory entry name it points to.
                if (!(bh = find_entry(&inode,thisname,namelen,&de))) {
                        iput(inode);
                        return NULL;
                }
                inr = de->inode;       // i-node number of the current directory name part.
                brelse(bh);
                dir = inode;
                if (!(inode = iget(dir->i_dev,inr))) {      // get i-node content.
                        iput(dir);
                        return NULL;
                }
                if (!(inode = follow_link(dir,inode)))
                        return NULL;
        }
}

/*
 *      dir_namei()
 *
 * dir_namei() returns the inode of the directory of the
 * specified name, and the name within that directory.
 */
//// Gets the i-node of the specified directory name, and the top-level directory name.
// Parameters: pathname - the directory path name; namelen - the path name length; name - the
// topmost directory name returned; base - the i-node of the directory to start searching.
// Returns: the i-node and name and length of the top-level directory of the specified directory
// name. Returns NULL on error.
static struct m_inode * dir_namei(const char * pathname,
        int * namelen, const char ** name, struct m_inode * base)
{
        char c;
        const char * basename;
        struct m_inode * dir;
```

```

// First, get the i-node of the top-level directory of the specified path name, then search
// and detect the pathname, find the name string after the last '/' character, calculate its
// length, and return the i-node pointer of the top-level directory. Note that if the last
// character of the path name is the slash character '/', then the returned directory name is
// empty and the length is 0. However, the returned i-node pointer still points to the i-node
// of the directory name before the last '/' character.
        if (!(dir = get_dir(pathname,base)))      // base is the i-node of the starting dir.
                return NULL;
        basename = pathname;
        while (c=get_fs_byte(pathname++))
                if (c=='/')
                        basename=pathname;
        *namelen = pathname-basename-1;
        *name = basename;
        return dir;
}

//// Get the i-node of the specified pathname (internal function).
// Parameters: pathname - path name; base - search start directory i-node;
// follow_links - whether to follow the symbolic link, 1 - yes, 0 - no.
// Returns: the corresponding i-node.
struct m_inode * _namei(const char * pathname, struct m_inode * base,
        int follow_links)
{
        const char * basename;
        int inr,namelen;
        struct m_inode * inode;
        struct buffer_head * bh;
        struct dir_entry * de;

// First find the directory name of the topmost directory in the specified path name and get
// its i-node. If it does not exist, it returns NULL and exits. If the length of the topmost
// name returned is 0, it means that the path name is named after the last item in a directory.
// Therefore, we have found the i-node of the corresponding directory, so we can return the
// i-node directly. If the length of the returned name is not 0, then we call the dir_namei()
// function again to search for the top-level directory name with the newly specified start
// directory base, and make a similar judgment based on the returned information.
        if (!(dir = dir_namei(pathname,&namelen,&basename)))
                return NULL;
        if (!namelen)                   /* special case: '/usr/' etc */
                return dir;
        if (!(base = dir_namei(pathname,&namelen,&basename,base)))
                return NULL;
        if (!namelen)                   /* special case: '/usr/' etc */
                return base;
// Then look for the i-node of the specified file name directory entry in the returned top-level
// directory. Please pay attention again! If it is also a directory name at the end, but there
// is no '/' added after it, it will not return the i-node of the last directory! For example:
// '/usr/src/linux' will only return the i-node of the 'src/' directory name. Since the function
// dir_namei() treats the last name that does not end with '/' as a file name, it is necessary
// to use the find directory entry i-node function find_entry() to handle this situation
// separately. At this point, 'de' contains the directory entry pointer found, and 'dir' is
// the i-node pointer of the directory containing the directory entry.
        bh = find_entry(&base,basename,namelen,&de);
        if (!bh) {
                iput(base);
                return NULL;
        }
// Then take the i-node number of the directory entry, release the cache buffer block containing
// the directory entry, and put back the i-node of the directory. Then take the i-node
// corresponding to the node number, modify the access time to be the current time, and set
// the modified flag. Finally, the i-node pointer inode is returned. In addition, if the currently
// processed directory entry is a symbolic link name, then follow_link() is used to get the
// i-node of the directory entry it points to.
        inr = de->inode;
        brelse(bh);
        if (!(inode = iget(base->i_dev,inr))) {
                iput(base);
                return NULL;
        }
        if (follow_links)
                inode = follow_link(base,inode);
        else
                iput(base);
        inode->i_atime=CURRENT_TIME;
        inode->i_dirt=1;
        return inode;
}

//// Take the i-node of the specified path name without following the symbolic link.
// Parameters: pathname - the path name.
// Returns: the corresponding i-node.
struct m_inode * lnamei(const char * pathname)
{
        return _namei(pathname, NULL, 0);
}

/*
 *      namei()
 *
 * is used by most simple commands to get the inode of a specified name.
 * Open, link etc use their own routines, but this is enough for things
 * like 'chmod' etc.
 */
//// Take the i-node of the specified path name and follow the symbolic link.
// Parameters: pathname - the path name.
// Returns: the corresponding i-node.
struct m_inode * namei(const char * pathname)
{
        return _namei(pathname,NULL,1);
}

/*
 *      open_namei()
 *
```

```
386  * namei for open - this is in fact almost the whole open-routine.
387  */
    //// The namei function used to open the file.
    // Parameters: filename - is the file path name; flag - is the open file flag, which can take
    // values O_RDONLY (read only), O_WRONLY (write only) or O_RDWR (read and write), and O_CREAT
    // (create), O_EXCL (file must not exist), O_APPEND (add at the end of the file) and other
    // combinations of other flags; Mode - the permission attribute of the specified file when
    // creating a new file. These attributes are S_IRWXU (the file owner has read, write, and execute
    // permissions), S_IRUSR (user has read file permissions), S_IRWXG (group members have read,
    // write, and execute permissions), and so on. For newly created files, these attributes are
    // only used for future access to the file, and open calls that create read-only files will
    // also return a readable and writable file handle. See the include files sys/stat.h, fcntl.h.
    // Returns: 0 is returned successfully, otherwise the error code is returned; res_inode - returns
    // the i-node pointer of the corresponding file path name.
388 int open_namei(const char * pathname, int flag, int mode,
389         struct m_inode ** res_inode)
390 {
391         const char * basename;
392         int inr,dev,namelen;
393         struct m_inode * dir, *inode;
394         struct buffer_head * bh;
395         struct dir_entry * de;
396
    // First, the function parameters are handled reasonably. If the file access mode flag is
    // read-only (0), but the file truncate flag O_TRUNC is set, the write-only flag O_WRONLY is
    // added to the file open flag. The reason for this is because the truncate flag O_TRUNC is
    // valid only if the file is writable. Then use the file access permission mask of the current
    // process, mask the corresponding bit in the given mode, and add the normal file flag I_REGULAR.
    // This flag will be used as the default property for new files when the open file does not
    // exist and needs to be created. See the note on line 411 below.
397         if ((flag & O_TRUNC) && !(flag & O_ACCMODE))
398                 flag |= O_WRONLY;
399         mode &= 0777 & ~current->umask;
400         mode |= I_REGULAR;                  // regular file flag. See file include/const.h.

    // Then find the corresponding i-node according to the specified path name, and the top directory
    // name and its length. At this time, if the top directory name length is 0 (for example, '/usr/'),
    // then if the operation is not read/write, creation, and file truncation 0, it means that a
    // directory name file operation is open. So directly return the i-node of the directory, and
    // return 0 to exit. Otherwise, the process is illegal, so the i-node is put back and the error
    // code is returned.
401         if (!(dir = dir_namei(pathname,&namelen,&basename,NULL)))
402                 return -ENOENT;
403         if (!namelen) {                     /* special case: '/usr/' etc */
404                 if (!(flag & (O_ACCMODE|O_CREAT|O_TRUNC))) {
405                         *res_inode=dir;
406                         return 0;
407                 }
408                 iput(dir);
409                 return -EISDIR;
410         }
    // Then, according to the i-node of the top-level directory name obtained above, the directory
    // entry structure corresponding to the last file name in the path name is searched, and the
    // cache block of the directory entry is obtained at the same time. If the cache block pointer
    // is NULL, it means that the directory entry corresponding to the file name is not found, so
    // it is only possible to create a file. At this time, if the operation is not to create a file,
    // or the user does not have the right to write in the directory, put back the i-node of the
    // directory and return the corresponding error code to exit.
411         bh = find_entry(&dir,basename,namelen,&de);
412         if (!bh) {
413                 if (!(flag & O_CREAT)) {
414                         iput(dir);
415                         return -ENOENT;
416                 }
417                 if (!permission(dir,MAY_WRITE)) {
418                         iput(dir);
419                         return -EACCES;
420                 }
    // Now we have determined that it is a create operation and has a write permission. Therefore,
    // we apply for a new i-node on the device to use the file name specified on the path name,
    // and make initial settings for the new i-node: set the user id; access mode; set the modified
    // flag. Then add a new directory entry in the specified directory dir.
421                 inode = new_inode(dir->i_dev);
422                 if (!inode) {
423                         iput(dir);
424                         return -ENOSPC;
425                 }
426                 inode->i_uid = current->euid;
427                 inode->i_mode = mode;
428                 inode->i_dirt = 1;
429                 bh = add_entry(dir,basename,namelen,&de);
    // If the returned cache block pointer that should contain a new directory entry is NULL, it
    // means the add directory entry operation failed. Then the error rollback process is executed:
    // the reference connection count of the new i-node is decremented by 1, the i-nodes are put
    // back, and the error code exit is returned. Otherwise, the operation of adding a directory
    // entry is successful. So we set some initial values of the new directory entry: set the i-node
    // number to the number of the newly applied i-node; set the cache block modified flag. Then
    // release the cache block and the i-node of the directory is put back. Then the i-node pointer
    // of the new directory entry is returned and exits successfully.
430                 if (!bh) {
431                         inode->i_nlinks--;
432                         iput(inode);
433                         iput(dir);
434                         return -ENOSPC;
435                 }
436                 de->inode = inode->i_num;
437                 bh->b_dirt = 1;
438                 brelse(bh);
439                 iput(dir);
440                 *res_inode = inode;
441                 return 0;
442         }
    // If the above (line 411) succeeds in obtaining the directory entry corresponding to the file
    // name (ie, bh is not NULL), this indicates that the specified file already exists. Then, the
    // i-node number of the directory entry and the device number are taken, and the cache block
    // and the i-node of the directory are released. If the exclusive operation flag O_EXCL is set,
// but the file already exists, then the error code (file exists) is returned and exits.
        inr = de->inode;
        dev = dir->i_dev;
        brelse(bh);
        if (flag & O_EXCL) {
                iput(dir);
                return -EEXIST;
        }
// Then we read the i-node content of the directory entry. If the i-node is an i-node of a directory
// and the access mode is write-only or read-write, or has no access permission, then put back
// the i-node and return the access permission error code.
        if (!(inode = follow_link(dir,iget(dev,inr))))
                return -EACCES;
        if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
            !permission(inode,ACC_MODE(flag))) {
                iput(inode);
                return -EPERM;
        }
// Then we update the access time field of the i-node to the current time. If the truncate flag
// is set, the file size is truncated to zero. Finally returns a pointer to the i-node of the
// directory entry and returns 0 (success).
        inode->i_atime = CURRENT_TIME;
        if (flag & O_TRUNC)
                truncate(inode);
        *res_inode = inode;
        return 0;
}

//// Create a device special file or a regular file node.
// This function creates a file system node (common file, device special file, or named pipe)
// named filename, specified by mode and dev.
// Parameters: filename - the pathname of filename; mode - specifies the permission to use and
// the type of node created; dev - the device number.
// Returns: 0 if successful, otherwise returns an error code.
int sys_mknod(const char * filename, int mode, int dev)
{
        const char * basename;
        int namelen;
        struct m_inode * dir, * inode;
        struct buffer_head * bh;
        struct dir_entry * de;

// First check the validity of the operation permissions and parameters and get the i-node of
// the top-level directory in the path name. If it is not a superuser, return the access permission
// error code. If the i-node corresponding to the top-level directory in the path name is not
// found, an error code is returned. If the top file name length is 0, it means that the given
// path name does not specify a file name at the end or the process has no write permission
// in the directory, then we put back the i-node of the directory and return an error code.
        if (!suser())
                return -EPERM;
        if (!(dir = dir_namei(filename,&namelen,&basename, NULL)))
                return -ENOENT;
        if (!namelen) {
                iput(dir);
                return -ENOENT;
        }
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                return -EPERM;
        }
// Then we search for the file specified by the path name already exists, if it already exists,
// you can not create a file node with the same name. Therefore, if the directory entry of the
// last file name corresponding to the path name already exists, the buffer block containing
// the directory entry is released and the i-node of the directory is put back, and the error
// code that the file already exists is returned.
        bh = find_entry(&dir,basename,namelen,&de);
        if (bh) {
                brelse(bh);
                iput(dir);
                return -EEXIST;
        }
// If the directory entry for the file name is not found, then we apply for a new i-node and
// set the attribute mode of the i-node. If you are creating a block device file or a character
// device file, make the direct logical block pointer 0 of the i-node equal to the device number.
// That is, for the device file, the i_zone[0] of the i-node stores the device number of the
// device defined by the device file. Then set the modification time and access time of the
// i-node to the current time, and set the i-node modified flag.
        inode = new_inode(dir->i_dev);
        if (!inode) {           // put back inode and return error code if failed.
                iput(dir);
                return -ENOSPC;
        }
        inode->i_mode = mode;
        if (S_ISBLK(mode) || S_ISCHR(mode))
                inode->i_zone[0] = dev;
        inode->i_mtime = inode->i_atime = CURRENT_TIME;
        inode->i_dirt = 1;
// Then add a new directory entry for this new i-node. If it fails (the cache block pointer
// containing the directory entry is NULL), put back the i-node of the directory; reset the
// requested i-node reference connection count, put back the i-node, and return the error code.
        bh = add_entry(dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                inode->i_nlinks=0;
                iput(inode);
                return -ENOSPC;
        }
// Now adding the directory entry operation is also successful, so we set the contents of this
// directory item. Let the i-node field of the directory entry be equal to the new i-node number,
// and set the cache block modified flag, put back the directory and the new i-node, release
// the cache block, and finally return 0 (success).
        de->inode = inode->i_num;
        bh->b_dirt = 1;
        iput(dir);
        iput(inode);
        brelse(bh);
        return 0;
}

////  Make a directory (used in system-call).
// Parameters: pathname - the path name; mode - permission attribute used by the directory.
// Returns: 0 if successful, otherwise returns an error code.
int sys_mkdir(const char * pathname, int mode)
{
        const char * basename;
        int namelen;
        struct m_inode * dir, * inode;
        struct buffer_head * bh, *dir_block;
        struct dir_entry * de;

// First check the validity of the operation permissions and parameters and get the i-node of
// the top-level directory in the path name. If the i-node corresponding to the top-level
// directory in the path name is not found, an error code is returned. If the top file name
// length is 0, it means that the given path name does not specify a file name at the end or
// the process has no write permission in the directory, then we put back the i-node of the
// directory and return an error code.
        if (!(dir = dir_namei(pathname,&namelen,&basename, NULL)))
                return -ENOENT;
        if (!namelen) {
                iput(dir);
                return -ENOENT;
        }
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                return -EPERM;
        }
// Then we search for the directory name specified by the path name already exists, if it already
// exists, we can not create a directory node with the same name. Therefore, if the directory
// entry of the last directory name on the path name already exists, the buffer block containing
// the directory entry is released and the i-node of the directory is put back, and the error
// code that the file already exists is returned. Otherwise, we apply for a new i-node and set
// the attribute mode of the i-node: set the file size corresponding to the new i-node to 32
// bytes (the size of two directory entries), set the modified flag of the node, and the node's
// modify time and access time. Two directory entries are used for the '.' and '..' directories
// respectively.
        bh = find_entry(&dir,basename,namelen,&de);
        if (bh) {
                brelse(bh);
                iput(dir);
                return -EEXIST;
        }
        inode = new_inode(dir->i_dev);
        if (!inode) {                // put back inode and return error code if failed.
                iput(dir);
                return -ENOSPC;
        }
        inode->i_size = 32;
        inode->i_dirt = 1;
        inode->i_mtime = inode->i_atime = CURRENT_TIME;
```

```
// Next, we apply for a disk block for storing the directory entry data for the new i-node,
// and let the first direct block pointer of the i-node be equal to the block number. If the
// application fails, put back the i-node of the directory; reset the newly requested i-node
// link count; put back the new i-node, and return no space error code. Otherwise, set the new
// i-node modified flag.
        if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
                iput(dir);
                inode->i_nlinks--;
                iput(inode);
                return -ENOSPC;
        }
        inode->i_dirt = 1;
// Now we read the newly requested disk block from the device (the purpose is to put the
// corresponding block into the buffer cache). Similarly, if an error occurs, put back the i-node
// of the directory; release the requested disk block; reset the newly requested i-node link
// count; put back the new i-node, and return no space error code to exit.
        if (!(dir_block=bread(inode->i_dev,inode->i_zone[0]))) {
                iput(dir);
                inode->i_nlinks--;
                iput(inode);
                return -ERROR;
        }
// Then we create two default new directory entries ('.' and '..') data in the created directory
// file in the buffer block. First, let 'de' point to the data block that holds the directory
// entry, and then set the i-node number field of the directory entry equal to the newly applied
// i-node number, and the name field is set to ".". Then 'de' points to the next directory entry
// structure, and stores the i-node number of the parent directory and name ".." in the structure.
// Then set the cache block modified flag and release the buffer block. Re-initializes the mode
// field of the new i-node and sets the i-node modified flag.
        de = (struct dir_entry *) dir_block->b_data;
        de->inode=inode->i_num;                     // set the '.' directory entry.
        strcpy(de->name,".");
        de++;
        de->inode = dir->i_num;                     // set the '..' directory entry.
        strcpy(de->name,"..");
        inode->i_nlinks = 2;
        dir_block->b_dirt = 1;
        brelse(dir_block);
        inode->i_mode = I_DIRECTORY | (mode & 0777 & ~current->umask);
        inode->i_dirt = 1;
// Now we add a new directory entry in the specified directory to store the i-node and directory
// name of the newly created directory. If the failure (including the buffer block pointer of
// the directory entry is NULL), put back the i-node of the directory; the requested i-node
// reference link count is reset and put back to the i-node. Return error code to exit.
        bh = add_entry(dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                inode->i_nlinks=0;
                iput(inode);
                return -ENOSPC;
        }
// Finally, the i-node field of the new directory entry is equal to the new i-node number, the
// cache block modified flag is set, the directory and the new i-node are put back, the buffer
// block is released, and finally 0 (success) is returned.
        de->inode = inode->i_num;
        bh->b_dirt = 1;
        dir->i_nlinks++;
        dir->i_dirt = 1;
        iput(dir);
        iput(inode);
        brelse(bh);
        return 0;
}

/*
 * routine to check that the specified directory is empty (for rmdir)
 */
//// Check if the directory is empty.
// Parameters: inode - the i-node pointer of the specified directory.
// Returns: 1 - the directory is empty; 0 - not empty.
static int empty_dir(struct m_inode * inode)
{
        int nr,block;
        int len;
        struct buffer_head * bh;
        struct dir_entry * de;

// First calculate the number of existing directory entries in the specified directory and check
// if the information in the two specific directory entries is correct. There should be at least
// 2 directory entries in a directory: the entries whose names are "." and "..". If the number
// of directory entries is less than 2 or the first direct block of the i-node of the directory
// does not point to any disk block number, or the direct block cannot be read, then the warning
// message is displayed, and 0 is returned. (failure).
        len = inode->i_size / sizeof (struct dir_entry);   // number of directory entries.
        if (len<2 || !inode->i_zone[0] ||
            !(bh=bread(inode->i_dev,inode->i_zone[0]))) {
                printk("warning - bad directory on dev %04x\n",inode->i_dev);
                return 0;
        }
// At this time, the buffer block indicated by 'bh' contains directory entry data. We let the
// directory entry pointer 'de' point to the first directory entry in the buffer block. For
// the first directory entry ("."), its i-node number field should be equal to the i-node number
// of the current directory; for the second directory entry (".."), its i-node number field
// should be equal to the one of up layer directory, and will not be 0. Therefore, if the i-node
// number of the first directory entry is not equal to the that of the current directory, or
// the i-node number of the second directory entry is zero, or the name fields of the two directory
// entries are not equal to ". "and"..", an error warning message is displayed and returns 0.
        de = (struct dir_entry *) bh->b_data;
        if (de[0].inode != inode->i_num || !de[1].inode ||
            strcmp(".",de[0].name) || strcmp("..",de[1].name)) {
                printk("warning - bad directory on dev %04x\n",inode->i_dev);
                return 0;
        }
// Then we make 'nr' equal to the directory entry number (counted from 0); 'de' points to the
// third directory entry, and loop through all the remaining (len - 2) directory entries in
// the directory to see if the i-node number field is not 0 (used).
```

```
    nr = 2;
    de += 2;
    while (nr<len) {
// If the directory entries in the disk block have been completely checked, and the directory
// entry being used is not found, the buffer block of the disk block is released, and the next
// disk block containing the directory entries in the directory data file is read in. The reading
// method is to calculate the data block number (nr/DIR_ENTRIES_PER_BLOCK) of the corresponding
// directory entry item in the directory file according to the currently detecting directory
// item number 'nr', and then use the bmap() function to obtain the block number, and then use
// the block device function bread() reads the disk block into the buffer cache and returns
// a pointer to the buffer block. If the disk block is not used (or has not been used, such
// as the file has been deleted, etc.), continue to read the next block, if it cannot be read
// out, then error returns 0. Otherwise let 'de' point to the first directory entry in the block.
        if ((void *) de >= (void *) (bh->b_data+BLOCK_SIZE)) {
            brelse(bh);
            block=bmap(inode,nr/DIR_ENTRIES_PER_BLOCK);
            if (!block) {
                nr += DIR_ENTRIES_PER_BLOCK;
                continue;
            }
            if (!(bh=bread(inode->i_dev,block)))
                return 0;
            de = (struct dir_entry *) bh->b_data;
        }
// For the current directory entry, if its i-node number field is not equal to 0, it means that
// the directory item is being used before, indicating that the directory is not empty, then
// the buffer block is released, and 0 is returned. Otherwise, if all the directory entries
// in the directory have not been queried, the directory entry number 'nr' is incremented, and
// 'de' is pointed to the next directory entry, and the detection continues.
        if (de->inode) {
            brelse(bh);
            return 0;
        }
        de++;
        nr++;
    }
// If the code is executed here, it means that the used directory entry is not found in the
// directory (except for the first two, of course), then the buffer block is released and 1
// is returned.
    brelse(bh);
    return 1;          // directory is empty!
}

//// Remove the directory.
// Parameters: name - the directory name (path name).
// Returns: Returns 0 for success, otherwise returns an error number.
int sys_rmdir(const char * name)
{
    const char * basename;
    int namelen;
    struct m_inode * dir, * inode;
    struct buffer_head * bh;
    struct dir_entry * de;
```

```

// First check the validity of the operation permissions and parameters and get the i-node of
// the top-level directory in the path name. If the i-node corresponding to the top-level
// directory in the path name is not found, an error code is returned. If the top file name
// length is 0, it means that the given path name does not specify a file name at the end or
// the process has no write permission in the directory, then we put back the i-node of the
// directory and return an error code.
        if (!(dir = dir_namei(name,&namelen,&basename, NULL)))
                return -ENOENT;
        if (!namelen) {
                iput(dir);
                return -ENOENT;
        }
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                return -EPERM;
        }
// Then, according to the i-node and directory name of the specified directory, the function
// find_entry() is used to find the directory entry, and the buffer block 'bh' containing the
// directory entry, the i-node 'dir' of the directory, and the directory entry 'de' are returned.
// Then, according to the i-node number in the directory entry 'de', the corresponding i-node
// is obtained by using the iget() function. In the course of specific operations, if the directory
// entry of the last directory name in the path name does not exist, the buffer block containing
// the directory entry is released, and the i-node of the directory is put back, the error code
// of file already exists is returned. If the operation of obtaining the directory entry's i-node
// is wrong, the i-node of the directory is put back, and the buffer block containing the directory
// entry is released, and the error code is returned.
        bh = find_entry(&dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                return -ENOENT;
        }
        if (!(inode = iget(dir->i_dev, de->inode))) {
                iput(dir);
                brelse(bh);
                return -EPERM;
        }
// At this point we have the directory i-node 'dir' , the directory entry 'de' to be deleted
// and its corresponding i-node. Below we verify the feasibility of the deletion by checking
// the information in these three objects.
//
// If the directory has the restricted delete flag set and the effective user id (euid) of the
// process is not root, and the effective user id (euid) of the process is not equal to the
// user id of the i-node, it means that the current process does not have permission to remove
// the directory, so we put back the directory i-node and the entry's i-node, then release the
// buffer block and return the error code.
        if ((dir->i_mode & S_ISVTX) && current->euid &&
            inode->i_uid != current->euid) {
                iput(dir);
                iput(inode);
                brelse(bh);
                return -EPERM;
        }
```

```
// If the device number of the directory entry i-node is not equal to the device number of the
// directory containing this entry, or the referenced link count of the removing directory is
// greater than 1 (indicating there is symbol link, etc.), the directory cannot be removed.
// So the directory i-node containing the directory name to be deleted and the entry's i-node
// are then released, the buffer block is released, and an error code is returned.
        if (inode->i_dev != dir->i_dev || inode->i_count>1) {
                iput(dir);
                iput(inode);
                brelse(bh);
                return -EPERM;
        }
// If the directory entry i-node is equal to the i-node of the directory, indicating that the
// "." directory is attempted to be deleted, this is not allowed. Then the directory entry i-node
// and the i-node of the directory to be deleted are put back, the buffer block is released,
// and the error code is returned.
        if (inode == dir) {     /* we may not delete ".", but "../dir" is ok */
                iput(inode);
                iput(dir);
                brelse(bh);
                return -EPERM;
        }
// If the attribute of the i-node of the directory to be deleted indicates that this is not
// a directory, the premise of this deletion operation does not exist at all. Then, the i-node
// of the directory entry to be deleted and the i-node of its directory are put back, the buffer
// block is released, and an error code is returned.
        if (!S_ISDIR(inode->i_mode)) {
                iput(inode);
                iput(dir);
                brelse(bh);
                return -ENOTDIR;
        }
// If the directory to be deleted is not empty, it cannot be deleted. Then, the directory i-node
// containing the directory name to be deleted and the i-node of the directory to be deleted
// are put back, the buffer block is released, and an error code is returned.
        if (!empty_dir(inode)) {
                iput(inode);
                iput(dir);
                brelse(bh);
                return -ENOTEMPTY;
        }
// For an empty directory, the number of directory entry links should be 2 (link to the upper
// directory and itself). If the number of links of the i-node to be deleted is not equal to
// 2, a warning message is displayed, but the deletion operation continues. Then, the i-node
// number field of the directory entry of the directory to be deleted is set to 0, indicating
// that the directory entry is no longer used, and the buffer block modified flag is set, and
// the buffer block is released. Then set the number of links of the i-node of the deleted
// directory to 0 (indicating idle), and set the i-node modified flag.
        if (inode->i_nlinks != 2)
                printk("empty directory has nlink!=2 (%d)",inode->i_nlinks);
        de->inode = 0;
        bh->b_dirt = 1;
        brelse(bh);
        inode->i_nlinks=0;
        inode->i_dirt=1;
// Then reduce the i-node link count of the directory containing the deleted directory name
// by 1, modify the change time and modification time to the current time, and set the modified
// flag of the node. Finally, the directory i-node containing the directory name to be deleted
// and the i-node of the directory to be deleted are put back, and 0 is returned (the deletion
// operation is successful).
        dir->i_nlinks--;
        dir->i_ctime = dir->i_mtime = CURRENT_TIME;
        dir->i_dirt=1;
        iput(dir);
        iput(inode);
        return 0;
}

//// Delete (release) the directory entry corresponding to a file name.
// Remove a name from the file system. If it is the last link to the file and no process is
// opening the file, the file will also be deleted and the occupied device space will be freed.
// Parameters: name - the file name (path name).
// Returns: 0 if successful, otherwise returns an error code.
int sys_unlink(const char * name)
{
        const char * basename;
        int namelen;
        struct m_inode * dir, * inode;
        struct buffer_head * bh;
        struct dir_entry * de;

// First check the validity of the operation permissions and parameters and get the i-node of
// the top-level directory in the path name. If the i-node corresponding to the top-level
// directory in the path name is not found, an error code is returned. If the top file name
// length is 0, it means that the given path name does not specify a file name at the end or
// the process has no write permission in the directory, then we put back the i-node of the
// directory and return an error code.
        if (!(dir = dir_namei(name,&namelen,&basename, NULL)))
                return -ENOENT;
        if (!namelen) {
                iput(dir);
                return -ENOENT;
        }
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                return -EPERM;
        }
// Then, according to the i-node and directory name of the specified directory, the function
// find_entry() is used to find the directory entry, and the buffer block 'bh' containing the
// directory entry, the i-node 'dir' of the directory, and the directory entry 'de' are returned.
// Then, according to the i-node number in the directory entry 'de', the corresponding i-node
// is obtained by using the iget() function.
        bh = find_entry(&dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                return -ENOENT;
        }
```

```
    if (!(inode = iget(dir->i_dev, de->inode))) {
        iput(dir);
        brelse(bh);
        return -ENOENT;
    }
// At this point we have the directory i-node 'dir' , the directory entry 'de' to be deleted
// and its corresponding i-node. Below we verify the feasibility of the deletion by checking
// the information in these three objects.
//
// If the directory has the restricted deletion flag set and the valid user id (euid) of the
// process is not root, and the euid of the process is not equal to the user id of the i-node,
// and the euid of the process is not equal to the user id of the directory i-node, Indicates
// that the current process does not have permission to delete the directory. Then put back
// the directory i-node containing the directory name to be deleted and the i-node of the
// directory to be deleted, then release the buffer block and return the error code.
    if ((dir->i_mode & S_ISVTX) && !suser() &&
        current->euid != inode->i_uid &&
        current->euid != dir->i_uid) {
        iput(dir);
        iput(inode);
        brelse(bh);
        return -EPERM;
    }
// If the specified file name is a directory, it cannot be deleted. Then the i-node of the
// directory i-node and the file name directory entry is put back, the buffer block containing
// the directory entry is released, and the error code is returned.
    if (S_ISDIR(inode->i_mode)) {
        iput(inode);
        iput(dir);
        brelse(bh);
        return -EPERM;
    }
// If i-node's link count is already 0, a warning message is displayed and corrected to 1.
    if (!inode->i_nlinks) {
        printk("Deleting nonexistent file (%04x:%d), %d\n",
            inode->i_dev,inode->i_num,inode->i_nlinks);
        inode->i_nlinks=1;
    }
// Now we can delete the directory entry corresponding to the file name. Then, the i-node number
// field in the file name directory entry is set to 0, indicating that the directory entry is
// released, and the buffer block modified flag is set, and the buffer block is released.
    de->inode = 0;
    bh->b_dirt = 1;
    brelse(bh);
// Then, the number of links of the i-node corresponding to the file name is decremented by
// 1, the modified flag is set, and the update time is set to the current time. Finally, put
// back the i-node and directory's i-node, and return 0 (success). If it is the last link of
// the file, that is, the number of i-node links minus 1 is equal to 0, and no process is opening
// the file at this time, the file will also be deleted when iput() is called to put back the
// i-node and the occupied device space is released. See fs/inode.c, line 183.
    inode->i_nlinks--;
    inode->i_dirt = 1;
    inode->i_ctime = CURRENT_TIME;
```

```
        iput(inode);
        iput(dir);
        return 0;
}

//// Create a symbolic link.
// Create a symbolic link (also known as a soft link) for an existing file.
// Parameters: oldname - the original path name; newname - the new path name.
// Returns: 0 if successful, otherwise returns an error code.
int sys_symlink(const char * oldname, const char * newname)
{
        struct dir_entry * de;
        struct m_inode * dir, * inode;
        struct buffer_head * bh, * name_block;
        const char * basename;
        int namelen, i;
        char c;

// First find the i-node 'dir' of the top-level directory of the new pathname and return the
// last file name and its length. If the i-node of the directory is not found, an error code
// is returned. If the file name is not included in the new path name, the i-node of the new
// path name directory is put back and the error code is returned. In addition, if the user
// does not have permission to write in the new directory, then the link cannot be established.
// Then put back the i-node of the new path name directory and return the error code.
        dir = dir_namei(newname,&namelen,&basename, NULL);
        if (!dir)
                return -EACCES;
        if (!namelen) {
                iput(dir);
                return -EPERM;
        }
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                return -EACCES;
        }
// Now we apply for a new i-node on the device specified by the directory, and set the i-node
// mode to the symbolic link type and the mode mask code specified by the process, and set the
// i-node modified flag.
        if (!(inode = new_inode(dir->i_dev))) {
                iput(dir);
                return -ENOSPC;
        }
        inode->i_mode = S_IFLNK | (0777 & ~current->umask);
        inode->i_dirt = 1;
// In order to save the symbolic link path name string, we need to apply for a disk block for
// the i-node, and let the first direct block number i_zone[0] equal to the obtained logical
// block number, and then set the i-node modified flag. If the application fails, put back the
// i-node of the corresponding directory; reset the newly requested i-node link count; put back
// the new i-node, and return no space error code.
        if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
                iput(dir);
                inode->i_nlinks--;
                iput(inode);
                return -ENOSPC;
        }
        inode->i_dirt = 1;
// The newly requested disk block is then read from the device (the purpose is to place the
// block in the buffer cache). If there is an error, put back the i-node of the directory; reset
// the newly requested i-node link count; put back the new i-node, and return error code.
        if (!(name_block=bread(inode->i_dev,inode->i_zone[0]))) {
                iput(dir);
                inode->i_nlinks--;
                iput(inode);
                return -ERROR;
        }
// Now we can put the symbolic link name string into this disk block. The disk block length
// is 1024 bytes, so the default symbol link name maximum length can only be 1024 bytes. We
// copy the symbolic link name string in the user space to the buffer block where the disk block
// is located, and set the buffer block modified flag. To prevent the user-supplied string from
// ending without a null, we need to put a NULL at the last byte of the buffer block data area.
// Then release the buffer block, and set the size of the data in the corresponding file of
// the i-node to be equal to the length of the symbolic link name string, and set the i-node
// modified flag.
        i = 0;
        while (i < 1023 && (c=get_fs_byte(oldname++)))
                name_block->b_data[i++] = c;
        name_block->b_data[i] = 0;
        name_block->b_dirt = 1;
        brelse(name_block);
        inode->i_size = i;
        inode->i_dirt = 1;
// Then we search for the name of the symbolic link file specified by the path name. If it already
// exists, you cannot create a directory entry i-node with the same name. If the corresponding
// symbolic link file name already exists, the buffer block containing the directory entry is
// released, the newly requested i-node link count is reset, and the i-node of the directory
// is put back, and the error code that the file already exists is returned.
        bh = find_entry(&dir,basename,namelen,&de);
        if (bh) {
                inode->i_nlinks--;
                iput(inode);
                brelse(bh);
                iput(dir);
                return -EEXIST;
        }
// Now we add a new directory entry in the specified directory, which is used to store the i-node
// number and directory name of the newly created symbolic link file name. If it fails (the
// buffer block pointer containing the directory entry is NULL), the i-node of the directory
// is put back; the requested i-node reference link count is reset, and the i-node is put back,
// and the error code is returned.
        bh = add_entry(dir,basename,namelen,&de);
        if (!bh) {
                inode->i_nlinks--;
                iput(inode);
                iput(dir);
                return -ENOSPC;
        }
```

```
// Finally, the i-node field of the new directory entry is equal to the new i-node number, the
// buffer block modified flag is set, the buffer block is released, the directory and the new
// i-node are put back, and finally 0 (success) is returned.
        de->inode = inode->i_num;
        bh->b_dirt = 1;
        brelse(bh);
        iput(dir);
        iput(inode);
        return 0;
}

//// Create a file name directory entry for an existing file.
// Create a new link (also known as hard link) for an existing file.
// Parameters: oldname - the original path name; newname - the new path name.
// Returns: 0 if successful, otherwise returns an error code.
int sys_link(const char * oldname, const char * newname)
{
        struct dir_entry * de;
        struct m_inode * oldinode, * dir;
        struct buffer_head * bh;
        const char * basename;
        int namelen;

// First validate the original file name, it should exist and not a directory name. So we first
// take the i-node 'oldinode' corresponding to the original file path name. If it is 0, it means
// an error and returns the error code. If the original path name corresponds to a directory
// name, the i-node is put back and an error code is also returned.
        oldinode=namei(oldname);
        if (!oldinode)
                return -ENOENT;
        if (S_ISDIR(oldinode->i_mode)) {
                iput(oldinode);
                return -EPERM;
        }
// Then find the i-node 'dir' of the top-level directory of the new pathname and return the
// last file name and its length. If the i-node of the directory is not found, the i-node of
// the original path name is put back and the error code is returned. If the file name is not
// included in the new path name, the i-node of the original path name and the new path name
// directory are put back, and the error code is returned.
        dir = dir_namei(newname,&namelen,&basename, NULL);
        if (!dir) {
                iput(oldinode);
                return -EACCES;
        }
        if (!namelen) {
                iput(oldinode);
                iput(dir);
                return -EPERM;
        }
// We can't build hard links across devices. Therefore, if the device number of the top directory
// of the new path name is different from the device number of the original path name, the i-node
// of the new path name directory and the i-node of the original path name are put back, and
// the error code is returned. In addition, if the user does not have the right to write in
// the new directory, the link cannot be established, so the i-node of the new path name directory
// and the i-node of the original path name are put back, and the error code is returned.
        if (dir->i_dev != oldinode->i_dev) {
                iput(dir);
                iput(oldinode);
                return -EXDEV;
        }
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                iput(oldinode);
                return -EACCES;
        }
// Now check if the new pathname already exists and if it does, it will not be able to establish
// a link. Then, the buffer block containing the existing directory entry is released, and the
// i-node of the new path name directory and the i-node of the original path name are put back,
// and the error code is returned.
        bh = find_entry(&dir,basename,namelen,&de);
        if (bh) {
                brelse(bh);
                iput(dir);
                iput(oldinode);
                return -EEXIST;
        }
// Now that all the conditions are met, we add a directory entry in the new directory. If it
// fails, put back the i-node of the directory and the i-node of the original pathname, and
// return the error code. Otherwise, the i-node number of the directory entry is initially set
// equal to the i-node number of the original path name, and the buffer block modified flag
// containing the newly added directory entry is set, the buffer block is released, and the
// i-node of the directory is put back.
        bh = add_entry(dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                iput(oldinode);
                return -ENOSPC;
        }
        de->inode = oldinode->i_num;
        bh->b_dirt = 1;
        brelse(bh);
        iput(dir);
// Then increase the link count of the original node by 1, modify its change time to the current
// time, and set the i-node modified flag. Finally, put back the i-node of the original path
// name and return 0 (success).
        oldinode->i_nlinks++;
        oldinode->i_ctime = CURRENT_TIME;
        oldinode->i_dirt = 1;
        iput(oldinode);
        return 0;
}

```

12.8 file_table.c

12.8.1 Function

The file_table.c program is currently empty, only the file table array is defined.

12.8.2 Code annotation

Program 12-7 linux/fs/file_table.c

```
/*
 *  linux/fs/file_table.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <linux/fs.h> File system header file. Define the file table structure (file, buffer_head,
//     m_inode, etc.).
#include <linux/fs.h>

struct file file_table[NR_FILE];  // File table array (64 items in total).

```

12.9 block_dev.c

From here on is the third part of the file system program, which includes five programs: block_dev.c, char_dev.c, pipe.c, file_dev.c, and read_write.c. The first four programs provide services for read_write.c, which mainly implements data access operations of the file system. The read_write.c program mainly implements the system-calls sys_write() and sys_read(). These five programs can be thought of as interface drivers for system-calls with block devices, character devices, pipe "devices", and file system "devices." The relationship between them can be represented by Figures 12-26. The system-calls sys_write() or sys_read() will determine which type of file is based on the attributes of the file descriptor provided by the parameter, and then call the read/write functions in the corresponding device interface program, and these functions will execute driver codes accordingly.

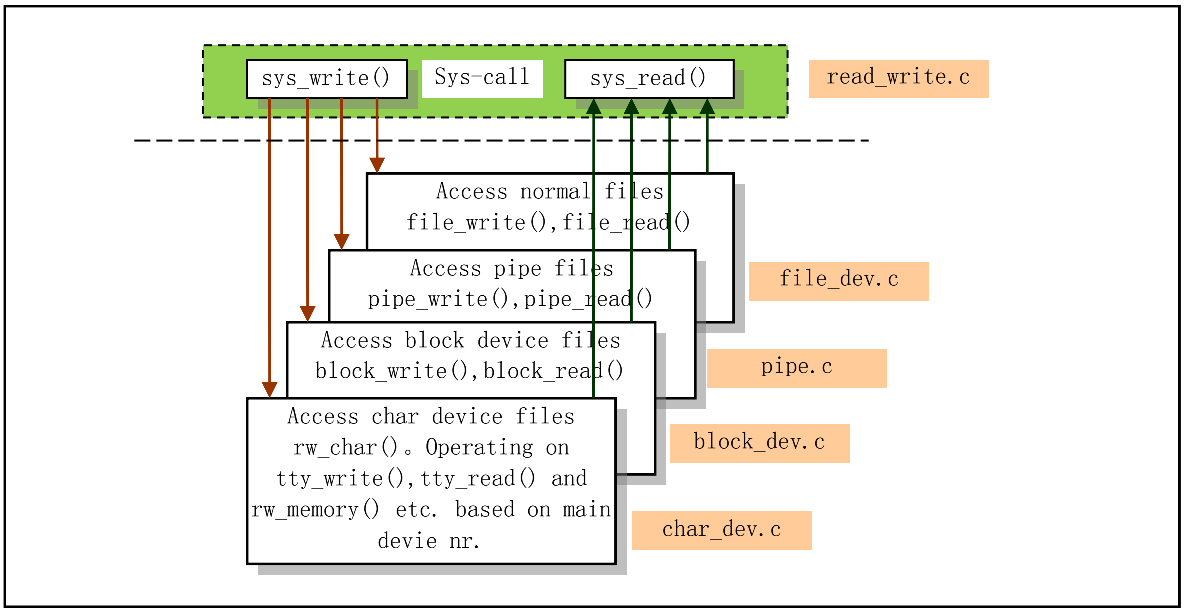


Figure 12-26 Interface functions for various types of files and system-calls

12.9.1 Function

The block_dev.c program belongs to the block device file data access operation program. The file includes two block device read and write functions, block_read() and block_write(), which are used to directly read and write the original data on the block device. These two functions are called by the system call function read() and write(), and they are not referenced elsewhere.

Since the block device reads and writes to the disk each time in units of disk blocks (same size as the buffer block), the function block_write() first maps the file pointer 'pos' position to the block number and the offset value in the block, and then use the block read function bread() or the block read-ahead function breada() to read the data block where the file pointer is located into the buffer block of the buffer cache. The data is then copied from the user data buffer to the offset position of the current buffer block according to the length 'chars' of the data to be written. If there is still data to be written, the next block is read into the buffer block in the cache, and the user data is copied into the buffer block. When the data is written for the second time and later, the offset offset is 0. See Figure 12-27.

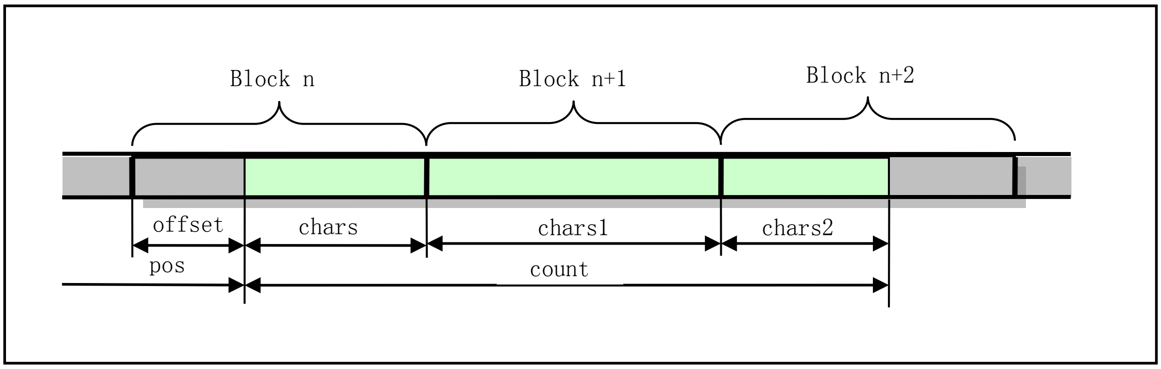


Figure 12-27 Block data read and write operation pointer position diagram

The user's buffer is allocated by the system when the user program starts executing, or dynamically applied during execution. Before calling this function, the system maps the virtual linear address used by the user buffer to the corresponding memory page in the main memory area.

The function block_read() operates in the same way as block_write() except that the data is copied from the buffer to the location specified by the user.

12.9.2 Code annotation

Program 12-8 linux/fs/block_dev.c

```
/*
 *  linux/fs/block_dev.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <errno.h> Error number header file. Contains various error numbers in the system.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//     segment register operations.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//     descriptors/interrupt gates, etc. is defined.
#include <errno.h>

#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>
#include <asm/system.h>

// An array of pointers to the total number of device data blocks. Each of its pointers points
// to an array of total blocks of the specified major device number, hd_sizes[]. Each item of
// the total number of block arrays corresponds to the total number of data blocks owned by
// a sub-device determined by the sub-device number.
extern int *blk_size[];                    // blk_drv/ll_rw_blk.c, line 49.

//// Block write function - Writes data of given size to the specified offset on a device.
// Parameters: dev - device number; pos - offset pointer in the device file;
// buf - buffer in user space; count - number of bytes to transfer.
// Returns: number of bytes written. If no data is written or errors, returns error code.
// For the kernel, the write operation is to write data to the cache. When the data is finally
// written to the device is determined and processed by the cache manager. In addition, since
// the block device reads and writes in units of blocks, when the write position is not at the
// edge of the block, the entire block in which the data is located needs to be read out first,
// and then the data is filled from the write position. Then write a complete block of data
// to the disk (ie, handle it to the buffer cache).
int block_write(int dev, long * pos, char * buf, int count)
{
// First, the position 'pos' in the file is converted into a block number 'block' in which the
// disk block is started to be read and written, and the offset position 'offset' at which the
// first byte is to be written in the block is obtained.
        int block = *pos >> BLOCK_SIZE_BITS;     // the block number where pos is located.
        int offset = *pos & (BLOCK_SIZE-1);      // offset value in the data block.
        int chars;
        int written = 0;
        int size;
        struct buffer_head * bh;
        register char * p;                       // local register variable.

// When writing a block device file, the total number of data blocks required to be written
// must of course not exceed the maximum number of data blocks allowed on the specified device.
// Therefore, the total number of blocks of the specified device is first taken out to compare
// and limit the write data length given by the function parameter. If no length is specified
// for the device in the system, the default length of 0x7fffffff (2GB blocks) is used.
        if (blk_size[MAJOR(dev)])
                size = blk_size[MAJOR(dev)][MINOR(dev)];
        else
                size = 0x7fffffff;
// Then, for the number of bytes to be written 'count', the following operations are looped
// until the data is completely written. During the loop execution, if the block number of the
// currently written data is greater than or equal to the total number of blocks of the specified
// device, the number of bytes written is returned and exited. The number of bytes that can
// be written in the currently processed data block is then calculated. If the number of bytes
// to be written is less than one block, then we only need to write the count byte; if we just
// want to write 1 block of data, then directly apply for a buffer block and put the user data
// into it. Otherwise, we need to read in the data block that will be written to the partial
// data, and pre-read the next two blocks of data. Then increment the block number and prepare
// for the next operation. If the buffer block operation fails, the number of bytes written
// is returned, and if no bytes are written, an error code (negative number) is returned.
        while (count>0) {
                if (block >= size)
                        return written?written:-EIO;
                chars = BLOCK_SIZE - offset;     // bytes that can be written to this block.
                if (chars > count)
                        chars=count;
                if (chars == BLOCK_SIZE)
                        bh = getblk(dev,block);          // buffer.c, lines 206 and 322.
                else
                        bh = breada(dev,block,block+1,block+2,-1);
                block++;
                if (!bh)
                        return written?written:-EIO;
// Next, the pointer p is first pointed to the position in the buffer block where the data is
// to be written. If there is less than one block of data written in the last loop, we need
// to fill in (modify) the required bytes from the beginning of the block, so the offset must
// be set to zero by default. After that, the offset pointer 'pos' in the file is forwarded
// by the number of bytes to be written this time, and the number of bytes to be written is
// accumulated into the statistical value 'written'. Then subtract the number of bytes 'counts'
// that need to be written from the number of bytes (chars) to be written this time. Then we
// copy the 'chars' bytes from the user buffer to the location in the buffer block where the
// write begins. After the copy is completed, the buffer block modified flag is set and the
// buffer block is released (that is, the buffer block reference count is decremented by 1).
                p = offset + bh->b_data;
                offset = 0;
                *pos += chars;
                written += chars;                      // The total number of bytes written.
                count -= chars;
                while (chars-->0)
                        *(p++) = get_fs_byte(buf++);
                bh->b_dirt = 1;
                brelse(bh);
        }
        return written;                          // Returns the number of bytes written.
}

//// Block read function - Reads data from the specified device into the user buffer.
// Parameters: dev - device number; pos - offset in the device file; buf - buffer in user space;
// count - number of bytes to transfer.
// Returns: number of bytes read. If no bytes are read or error, an error code is returned.
int block_read(int dev, unsigned long * pos, char * buf, int count)
{
        int block = *pos >> BLOCK_SIZE_BITS;
        int offset = *pos & (BLOCK_SIZE-1);
        int chars;
        int size;
        int read = 0;
        struct buffer_head * bh;
        register char * p;                     // local register variable.

// When reading a block device file, the total number of blocks required to be read cannot exceed
// the maximum number of blocks allowed on the specified device. Therefore, the total number
// of blocks of the device is first taken out to compare and limit the read data length given
// by the function parameter. If no length is specified for the device in the system, the default
// length of 0x7fffffff (2GB blocks) is used.
        if (blk_size[MAJOR(dev)])
                size = blk_size[MAJOR(dev)][MINOR(dev)];
        else
                size = 0x7fffffff;
// Then, for the number of bytes to be read in 'count', the following operations are looped
// until the data is completely read. During the loop execution, if the block number of the
// currently read data is greater than or equal to the total number of blocks of the specified
// device, the number of read bytes is returned and exited. Then calculate the number of bytes
// to read in the currently processed data block. If the number of bytes that need to be read
// is less than one block, then just read 'count' bytes. Then invoke the read block function
// breada () to read in the required data block, and pre-read the next two blocks of data. If
// the read operation is in error, the number of bytes read is returned. If no bytes are read,
// an error code is returned. Then increment the block number by one to prepare for the next
// operation.
        while (count>0) {
                if (block >= size)
                        return read?read:-EIO;
                chars = BLOCK_SIZE-offset;
                if (chars > count)
                        chars = count;
                if (!(bh = breada(dev,block,block+1,block+2,-1)))
                        return read?read:-EIO;
                block++;
   // Next, the pointer p is first pointed to the position in the buffer block of the read disk
block where the data is read. If the data of the last loop read operation is less than one block,
the required byte needs to be read from the beginning of the block, so the offset must be set to
zero beforehand. After that, the offset pointer 'pos' in the file is moved forward by the number
of bytes 'chars' to be read, and the number of bytes to be read is accumulated into the statistical
value 'read'. Then subtract the count number 'chars' to be read from the count value 'count' that
needs to be read. Then we copy the 'chars' bytes from the starting point pointed to by the 'p' in
the buffer block to the user buffer, and move the user buffer pointer forward. The buffer block
is released after this copy is completed.
                p = offset + bh->b_data;
                offset = 0;
                *pos += chars;
                read += chars;                  // The total number of bytes read in.
                count -= chars;
                while (chars-->0)
                        put_fs_byte(*(p++),buf++);
                brelse(bh);
        }
        return read;                            // returns the number of bytes read.
}

```

12.10 file_dev.c

12.10.1 Function

The file_dev.c file contains two functions, file_read() and file_write(). They are also used by the system-call functions read() and write() to read and write ordinary files. Similar to the previous file block_dev.c, this file is also used to access file data, but the functions in this program are operated by specifying the file path name. The file i-node and file structure are given in the function parameters. Through the information in the i node, the corresponding device number can be obtained, and by the file structure, we can obtain the current read and write pointer position of the file. The function in the previous file block_dev.c specifies the device number and the read/write location in the file directly in the parameter. It is specifically used to operate on the block device file, such as the /dev/hd0 device file.

12.10.2 Code annotation

Program 12-9 linux/fs/file_dev.c

```
/*
 *  linux/fs/file_dev.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <errno.h> Error number header file. Contains various error numbers in the system.
// <fcntl.h> File control header file. The definition of the operation control constant symbol
//     used for the file and its descriptors.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//     segment register operations.
#include <errno.h>
#include <fcntl.h>

#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>

#define MIN(a,b) (((a)<(b))?(a):(b))            // get the minimum value in a, b.
#define MAX(a,b) (((a)>(b))?(a):(b))            // get the maximum value in a, b.

//// File Read Function - Reads data in the file based on the i-node and file structure.
// From the i-node we can know the device number, and the file structure can know the current
// read-write pointer position in the file. 'Buf' specifies the location of the buffer in user
// space, and 'count' is the number of bytes that need to be read. The return value is the number
// of bytes actually read, or the error code (less than 0).
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
        int left,chars,nr;
        struct buffer_head * bh;

// First determine the validity of the parameters. Returns 0 if the byte count to be read is
// less than or equal to zero. If the number of bytes to be read is not equal to 0, the following
// loop operation is performed until the data is completely read or a problem is encountered.
// During the read loop operation, we use bmap() to get the corresponding logical (disk) block
// number 'nr' of the data block on the device based on the i-node and file table structure
// information. If nr is not 0, the disk block is read from the device. Exit the loop if the
// read operation fails. If nr is 0, it means that the specified data block does not exist,
// so the buffer block pointer is set to NULL. (filp->f_pos)/BLOCK_SIZE is used to calculate
// the data block number where the current pointer of the file is located.
        if ((left=count)<=0)
                return 0;
        while (left) {
                if (nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE)) {     // inode.c, line 140.
                        if (!(bh=bread(inode->i_dev,nr)))
                                break;
                } else
                        bh = NULL;
// Then we calculate the offset value nr of the file read/write pointer in the data block, then
// the number of bytes we want to read in the data block is (BLOCK_SIZE - nr). Then compare
// with the number of bytes that need to be read now, 'left', where the small value is the number
// of bytes 'chars' to be read for this operation. If '(BLOCK_SIZE - nr) > left', the block
// is the last piece of data to be read, otherwise the next block of data needs to be read.
// Then adjust the read and write file pointers. The pointer advances the number of bytes 'chars'
// that will be read this time. The remaining byte count 'left' also needs to be subtracted
// by 'chars' bytes.
                nr = filp->f_pos % BLOCK_SIZE;
                chars = MIN( BLOCK_SIZE-nr , left );
                filp->f_pos += chars;
                left -= chars;
// If the above data is read from the device, point p to the location in the buffer block where
// the data is to be read, and copy the 'chars' bytes into the user buffer 'buf', otherwise
// fill in the user buffer with 'chars' bytes of zero.
                if (bh) {
                        char * p = nr + bh->b_data;
                        while (chars-->0)
                                put_fs_byte(*(p++),buf++);
                        brelse(bh);
                } else {
                        while (chars-->0)
                                put_fs_byte(0,buf++);
                }
        }
// Finally, the access time of the i-node is modified to the current time, and the number of
// bytes read is returned. If the number of read bytes is 0, the error code is returned.
// CURRENT_TIME is a macro defined on line 142 of include/linux/sched.h and is used to calculate
// UNIX time. That is, from 0:00 on January 1, 1970, to the current time. The unit is seconds.
        inode->i_atime = CURRENT_TIME;
        return (count-left)?(count-left):-ERROR;
}

//// File Write Function - Writes user data to a file based on the i-node and file structure.
// From the i-node we can know the device number, and from the file structure we can know the
// current read-write pointer position in the file. 'buf' specifies the location of the buffer
// in the user space, and 'count' is the number of bytes to be written. The return value is
// the number of bytes actually written, or the error code (less than 0).
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
{
        off_t pos;
        int block,c;
        struct buffer_head * bh;
        char * p;
        int i=0;

/*
 * ok, append may not work when many processes are writing at the same time
 * but so what. That way leads to madness anyway.
 */
// First determine where the data is written to the file. If you want to append data to the
// file, move the file write pointer to the end of the file, otherwise it will be written at
// the current read and write pointer of the file.
        if (filp->f_flags & O_APPEND)
                pos = inode->i_size;
        else
                pos = filp->f_pos;
// Then, when the number of bytes that have been written 'i' (zero at the beginning) is less
// than the specified number of write bytes 'count', the following operations are performed
// cyclically. In the loop operation, we first get the logical (disk) block number 'block'
// corresponding to the file data block (pos/BLOCK_SIZE) on the device. If the disk block does
// not exist, create one. If the resulting disk block number equal to 0, the creation failed
// and the loop is exited. Otherwise, we read the disk block on the device according to the
// disk block number, and exit the loop if an error occurs.
        while (i<count) {
                if (!(block = create_block(inode,pos/BLOCK_SIZE)))
                        break;
                if (!(bh=bread(inode->i_dev,block)))
                        break;
// At this time, the buffer block pointer 'bh' is pointing to the file data block just read.
// Now we can find the offset 'c' of the current read/write pointer of the file in the data
// block, and point the pointer 'p' to the position in the buffer block where the data is started
// to be written, and set the buffer block modified flag. For the current pointer in the block,
// a total of c = (BLOCK_SIZE - c) bytes can be written from the start of the write position
// to the end of the block. If c is greater than the remaining number of bytes to be written
// (count - i), then simply write c = (count - i) bytes this time.
                c = pos % BLOCK_SIZE;
                p = c + bh->b_data;
                bh->b_dirt = 1;
                c = BLOCK_SIZE-c;
                if (c > count-i) c = count-i;
// Before writing the data, we pre-set the position in the file to be written in the next loop
// operation. So we move the 'pos' pointer forward by the number of bytes that need to be written.
// If the value of the 'pos' position exceeds the current length of the file at this time, the
// file length field in the i-node is modified, and the i-node modified flag is set. Then, the
// number of bytes to be written this time is added to the written byte count 'i' for loop judgment.
// Then 'c' bytes are copied from the user buffer 'buf' to the start position pointed to by
// 'p' in the cache buffer block. The buffer block is released after copying.
                pos += c;
                if (pos > inode->i_size) {
                        inode->i_size = pos;
                        inode->i_dirt = 1;
                }
                i += c;
                while (c-->0)
                        *(p++) = get_fs_byte(buf++);
                brelse(bh);
        }
// The loop exits when the data has all been written to the file or if a problem occurs during
// the write operation. At this point we change the file modification time to the current time
// and adjust the file read and write pointer. That is, if the operation does not append data
// to the end of the file, the file read/write pointer is adjusted to the current read/write
// position 'pos', and the modification time of the file i-node is changed to the current time.
// Finally, the number of bytes written is returned. If the number of bytes written is 0, the
// error code -1 is returned.
        inode->i_mtime = CURRENT_TIME;
        if (!(filp->f_flags & O_APPEND)) {
                filp->f_pos = pos;
                inode->i_ctime = CURRENT_TIME;
        }
        return (i?i:-1);
}

```

12.11 pipe.c

12.11.1 Function

Pipeline operations are the most basic way of communicating between processes. The pipe.c program includes the pipe file read and write operation functions read_pipe() and write_pipe(), and implements the pipe system-call sys_pipe(). These two functions are also low-level implementation functions for system-calls read() and write(), and are only used in read_write.c.

When creating and initializing a pipeline, the program specifically requests a pipeline i-node and allocates a page buffer (4KB) to the pipeline. The i_size field of the pipe i-node is set to point to the pipe buffer, the pipe data header pointer is stored in the i_zone[0] field, and the pipe data tail pointer is stored in the i_zone[1] field. For read pipeline operation, the data is read from the end of the pipeline and the pipe tail pointer is moved forward a number of bytes read; for write to the pipeline operation, the data is written to the pipe header, and the head pointer is moved forward a number of locations (pointing to a null byte), as shown in Figure 12-28.

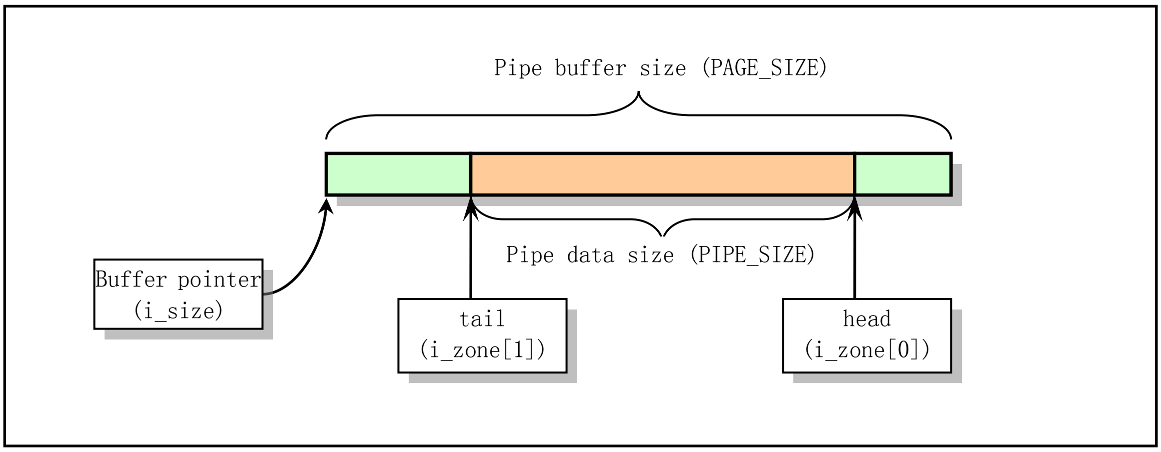


Figure 12-28 pipe buffer operation diagram

read_pipe() is used to read the data in the pipeline. If there is no data in the pipeline, the process of writing the pipeline is awakened, and itself goes to sleep. If the data is read, the pipe head pointer is adjusted accordingly and the data is passed to the user buffer. When all data in the pipeline are taken away, the process waiting for the write pipeline is also woken up, and the number of bytes of read data is returned. When the pipeline write process has exited the pipeline operation, the function exits immediately and returns the number of bytes read.

The write_pipe() function operates like a read pipeline function.

The system-call sys_pipe() is used to create an unnamed pipe. It first obtains two entries in the system's file table, and then looks for two unused descriptor entries in the file descriptor table of the current process to save the corresponding file structure pointer. Then apply for an idle i-node in the system and get a buffer block for the pipeline. The corresponding file structure is then initialized, one file structure is set to read-only mode, and the other is set to write-only mode. Finally, the two file descriptors are passed to the user.

In addition, several macros related to pipeline operations (such as PIPE_HEAD(), PIPE_TAIL(), etc.) used in the above functions are defined on lines 57--64 of the include/linux/fs.h file.

12.11.2 Code annotation

Program 12-10 linux/fs/pipe.c

```
/*
 *  linux/fs/pipe.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
//     manipulation function prototypes.
// <errno.h> Error number header file. Contains various error numbers in the system.
// <termios.h> Terminal input and output function header file. It mainly defines the terminal
//     interface that controls the asynchronous communication port.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/mm.h> Memory management header file. Contains page size definitions and some page
//     release function prototypes.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//     segment register operations.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
#include <signal.h>
#include <errno.h>
#include <termios.h>

#include <linux/sched.h>
#include <linux/mm.h>             /* for get_free_page */
#include <asm/segment.h>
#include <linux/kernel.h>

//// Pipe read function.
// The parameter 'inode' is the i-node of the pipe, 'buf' is the user data buffer pointer, and
// 'count' is the number of bytes read.
int read_pipe(struct m_inode * inode, char * buf, int count)
{
        int chars, size, read = 0;

// If the byte count that needs to be read is greater than 0, we loop through the following
// operations. During the loop operation, if there is no data in the current pipe (size=0),
// the process waiting for the node is woken up, which is usually a write pipeline process.
// If the pipe writer is no longer there, that is, the i-node reference count value is less
// than 2, the number of read bytes is returned. If there is a non-blocking signal currently
// received, it will immediately return the number of bytes read and exit; if no data has been
// received yet, it will return to restart the system-call number and exit. Otherwise, let the
// process sleep on the pipe to wait for the arrival of information. The macro PIPE_SIZE is
// defined in include/linux/fs.h. For the meaning of "restart system-call", see the
// kernel/signal.c program.
        while (count>0) {
                while (!(size=PIPE_SIZE(*inode))) {      // get data length in the pipe.
                        wake_up(& PIPE_WRITE_WAIT(*inode));
                        if (inode->i_count != 2)    /* are there any writers? */
                                return read;
                        if (current->signal & ~current->blocked)
                                return read?read:-ERESTARTSYS;
                        interruptible_sleep_on(& PIPE_READ_WAIT(*inode));
                }
// At this point, there is data in the pipeline (buffer), so we get the number of 'chars' in
// the pipe from the tail pointer to the end of the buffer. If it is greater than the number
// of bytes that need to be read, 'count', make it equal to 'count'. If 'chars' is greater than
// the 'size' of the data in the current pipeline, it is equal to 'size'. Then, the number of
// bytes to be read 'count' is subtracted from the number of bytes readable by this time, and
// added to the already read number of bytes 'read'.
                chars = PAGE_SIZE-PIPE_TAIL(*inode);
                if (chars > count)
                        chars = count;
                if (chars > size)
                        chars = size;
                count -= chars;
                read += chars;
// Then let 'size' point to the pipe tail pointer and adjust the current pipe tail pointer (forward
// 'chars' bytes). Wraps around if the tail pointer exceeds the end of the pipe. The data in
// the pipe is then copied into the user buffer. For a pipe i-node, the pipe buffer block pointer
// is in the i_size field.
                size = PIPE_TAIL(*inode);
                PIPE_TAIL(*inode) += chars;
                PIPE_TAIL(*inode) &= (PAGE_SIZE-1);
                while (chars-->0)
                        put_fs_byte(((char *)inode->i_size)[size++],buf++);
        }
// When the read pipe operation ends, the process waiting for the pipeline is woken up and the
// number of bytes read is returned.
        wake_up(& PIPE_WRITE_WAIT(*inode));
        return read;
}

//// Pipe write function.
// The parameter 'inode' is the i-node of the pipe, 'buf' is the data buffer pointer, and 'count'
// is the number of bytes that will be written to the pipe.
int write_pipe(struct m_inode * inode, char * buf, int count)
{
        int chars, size, written = 0;

// If the number of bytes to be written 'count' is still greater than 0, then we loop through
// the following operations. During the loop operation, if the current pipe is full (free space
// size = 0), the process waiting for the pipeline is woken up, usually the read pipe process
// is woken up. If there is no pipe reader, that is, the i-node reference count value is less
// than 2, the SIGPIPE signal is sent to the current process, and the number of bytes written
// is returned. If 0 bytes are written, -1 is returned. Otherwise, let the current process sleep
// on the pipe, waiting for the pipe process to read the data, thus making the pipe free the
// space. Macros PIPE_SIZE(), PIPE_HEAD(), etc. are defined in the file include/linux/fs.h.
        while (count>0) {
                while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) {
                        wake_up(& PIPE_READ_WAIT(*inode));
                        if (inode->i_count != 2) { /* no readers */
                                current->signal |= (1<<(SIGPIPE-1));
                                return written?written:-1;
                        }
                        sleep_on(& PIPE_WRITE_WAIT(*inode));
                }
// The program executes here, indicating that there is a writable space 'size' in the pipeline
// buffer. So we take the number of bytes 'chars' from the pipe head pointer to the end of the
// buffer. The write pipe operation begins from the pipe head pointer. If 'chars' is greater
// than the number of bytes that need to be written, 'count', make it equal to 'count'. If 'chars'
// is greater than the free space 'size' in the current pipeline, it is equal to 'size'. Then
// we let the number of bytes to be written 'count' minus the number of bytes written to this
// time 'chars', and add the number of bytes written this time to 'written'.
                chars = PAGE_SIZE-PIPE_HEAD(*inode);
                if (chars > count)
                        chars = count;
                if (chars > size)
                        chars = size;
                count -= chars;
                written += chars;
// Then let 'size' point to the pipe head pointer and adjust the current pipe data head pointer
// (forward chars bytes). Wraps around if the head pointer exceeds the end of the pipe. Then
// copy the 'chars' bytes from the user buffer to the beginning of the pipe head pointer. For
// a pipe i-node, the pipe buffer block pointer is in the i_size field.
                size = PIPE_HEAD(*inode);
                PIPE_HEAD(*inode) += chars;
                PIPE_HEAD(*inode) &= (PAGE_SIZE-1);
                while (chars-->0)
                        ((char *)inode->i_size)[size++]=get_fs_byte(buf++);
        }
// When the write pipe operation ends, the process waiting for the pipe is woken up, and the
// number of bytes written is returned.
        wake_up(& PIPE_READ_WAIT(*inode));
        return written;
}

//// Create pipe system-call.
// Create a pair of file handles (descriptors) in the fildes array. This pair of file handles
// point to a pipe i-node. Parameters: filedes - an array of file handles. Fildes[0] is used
// to read pipe data, and fildes[1] writes data to pipe. Returns 0 on success and -1 on error.
int sys_pipe(unsigned long * fildes)
{
        struct m_inode * inode;
        struct file * f[2];                     // array of file structures.
        int fd[2];                              // array of file handles.
        int i,j;

// First, take two free items from the system file table (items with a reference count field
// of 0), and set the reference count to 1. If there is only one free entry, the item is released
// (reset reference count). Returns -1 if no two free items are found.
        j=0;
        for(i=0;j<2 && i<NR_FILE;i++)
                if (!file_table[i].f_count)
```

```
                        (f[j++]=i+file_table)->f_count++;
        if (j==1)
                f[0]->f_count=0;
        if (j<2)
                return -1;
// For each of the two file table structure items obtained, a file handle number is assigned,
// and two items of the process file structure array are respectively pointed to the two file
// structures, and the file handle is the index number of the array. Similarly, if there is
// only one free file handle, the handle is released. If no two free handles are found, the
// two file structure items obtained above are released and -1 is returned.
        j=0;
        for(i=0;j<2 && i<NR_OPEN;i++)
                if (!current->filp[i]) {
                        current->filp[ fd[j]=i ] = f[j];
                        j++;
                }
        if (j==1)
                current->filp[fd[0]]=NULL;
        if (j<2) {
                f[0]->f_count=f[1]->f_count=0;
                return -1;
        }
// Then use the function get_pipe_inode() to request an i-node used by the pipe and allocate
// a page of memory as a buffer for the pipe. If unsuccessful, the two file handles and file
// structure items are released accordingly, and -1 is returned.
        if (!(inode=get_pipe_inode())) {                // fs/inode.c, line 231.
                current->filp[fd[0]] =
                        current->filp[fd[1]] = NULL;
                f[0]->f_count = f[1]->f_count = 0;
                return -1;
        }
// If the pipe i-node application is successful, the two file structures are initialized so
// that they both point to the same pipe i-node and set the read and write pointers to zero.
// The file mode of the first file structure is set to read, and the file mode of the second
// file structure is set to write. Finally, the file handle array is copied to the user space
// array. If it succeeds, it returns 0 and exits.
        f[0]->f_inode = f[1]->f_inode = inode;
        f[0]->f_pos = f[1]->f_pos = 0;
        f[0]->f_mode = 1;               /* read */
        f[1]->f_mode = 2;               /* write */
        put_fs_long(fd[0],0+fildes);
        put_fs_long(fd[1],1+fildes);
        return 0;
}

//// Pipe io control function.
// Parameters: pino - pipe i node pointer; cmd - control command; arg - arguments.
// The function returns 0 for success, otherwise it returns an error code.
int pipe_ioctl(struct m_inode *pino, int cmd, int arg)
{
// If the command is to get the current readable data length in the pipeline, put the pipe data
// length value into the location specified by the user argument and return 0. Otherwise an
// invalid command error code is returned.
```

```
        switch (cmd) {
                case FIONREAD:
                        verify_area((void *) arg,4);
                        put_fs_long(PIPE_SIZE(*pino),(unsigned long *) arg);
                        return 0;
                default:
                        return -EINVAL;
        }
}

```

12.12 char_dev.c

12.12.1 Function

The char_dev.c program includes access functions for character device files, mainly rw_ttyx(), rw_tty(), rw_memory(), and rw_char(). There is also a device read/write function pointer table whose item number represents the major device number.

rw_ttyx() is a serial terminal device read/write function whose main device number is 4. It implements read and write operations on the serial terminal by calling the tty driver.

rw_tty() is the console terminal read/write function, and the major device number is 5. The implementation principle is the same as rw_ttyx(), but it is limited to whether the process can perform console operations.

rw_memory() is a memory device file read/write function, and the major device number is 1. It implements byte operations on the memory image, but the Linux 0.12 kernel has not implemented the operation of the minor device number 0, 1, and 2, until the 0.96 version begins to implement the read and write operations of the minor device numbers 1 and 2.

rw_char() is an interface function for character device read and write operations. The other character device performs the operation of the corresponding character device on the character device read/write function pointer table through this function. The file system's operation functions open(), read(), etc. operate on all character device files.

12.12.2 Code annotation

Program 12-11 linux/fs/char_dev.c

```
/*
 *  linux/fs/char_dev.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <errno.h> Error number header file. Contains various error numbers in the system.
// <sys/types.h> Type header file. The basic system data types are defined.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//     segment register operations.
// <asm/io.h> io header file. Defines the function that operates on the io port in the form of
//     a macro’s embedded assembly language.
#include <errno.h>
#include <sys/types.h>

#include <linux/sched.h>
#include <linux/kernel.h>

#include <asm/segment.h>
#include <asm/io.h>

extern int tty_read(unsigned minor,char * buf,int count);     // terminal read.
extern int tty_write(unsigned minor,char * buf,int count);    // terminal write.

// Defines the character device read/write function pointer type.
typedef (*crw_ptr)(int rw,unsigned minor,char * buf,int count,off_t * pos);

//// Serial terminal read/write operation function.
// Parameters: rw - read/write command; minor - terminal subdevice number; buf - buffer;
// cout - number of read/write bytes; pos - read/write operation current pointer, this pointer
// is useless for terminal operations.
// Returns: the number of bytes actually read or written. If it fails, it returns an error code.
static int rw_ttyx(int rw,unsigned minor,char * buf,int count,off_t * pos)
{
        return ((rw==READ)?tty_read(minor,buf,count):
                tty_write(minor,buf,count));
}

//// Terminal read/write operation functions.
// Same as rw_ttyx(), but it add detection of whether the process has a control terminal.
static int rw_tty(int rw,unsigned minor,char * buf,int count, off_t * pos)
{
// If the process does not have a corresponding control terminal, an error code is returned.
// Otherwise, the terminal invokes the read/write function rw_ttyx() and returns the actual
// number of read or write bytes.
        if (current->tty<0)
                return -EPERM;
        return rw_ttyx(rw,current->tty,buf,count,pos);
}

//// Read/write memory data. Not implemented.
static int rw_ram(int rw,char * buf, int count, off_t *pos)
{
        return -EIO;
}

//// Physical memory data read/write function. Not implemented.
static int rw_mem(int rw,char * buf, int count, off_t * pos)
{
        return -EIO;
}

//// Kernel virtual memory data read/write function. Not implemented.
static int rw_kmem(int rw,char * buf, int count, off_t * pos)
{
        return -EIO;
}

// Port read/write operation function.
// Parameters: rw - read or write commands; buf - buffer; cout - number of read/write bytes;
// pos - port address. Returns: the number of bytes actually read or written.
static int rw_port(int rw,char * buf, int count, off_t * pos)
{
        int i=*pos;

// For the number of bytes required to be read or written, and the port address is less than
// 64k, a single byte read or write operation is performed cyclically. If the command is read,
// one byte of content is read from port i and placed in the user buffer. If the command is
// written, one byte is taken from the user data buffer and output to port i.
        while (count-->0 && i<65536) {
                if (rw==READ)
                        put_fs_byte(inb(i),buf++);
                else
                        outb(get_fs_byte(buf++),i);
                i++;                                    // increase port address. [??]
        }
// Then count the number of bytes read/written, adjust the corresponding read and write pointers,
// and return the number of bytes read/written.
        i -= *pos;
        *pos += i;
        return i;
}

//// Memory read/write operation functions. The memory master number is 1. Only the processing
// of the 0-5 subdevices is implemented here.
static int rw_memory(int rw, unsigned minor, char * buf, int count, off_t * pos)
{
// According to the memory subdevice number, different memory read/write functions are called.
        switch(minor) {
                case 0:                 // device file name is /dev/ram0 or /dev/ramdisk.
                        return rw_ram(rw,buf,count,pos);
                case 1:                 // device file name is /dev/ram1 or /dev/mem or ram.
                        return rw_mem(rw,buf,count,pos);
                case 2:                 // device file name is /dev/ram2 or /dev/kmem.
                        return rw_kmem(rw,buf,count,pos);
                case 3:                 // device file name is /dev/null.
                        return (rw==READ)?0:count;      /* rw_null */
                case 4:                 // device file name is /dev/port
                        return rw_port(rw,buf,count,pos);
                default:
                        return -EIO;
        }
```

```
}

// Define the number of devices in the system.
#define NRDEVS ((sizeof (crw_table))/(sizeof (crw_ptr)))

// Character device read/write function pointer table.
static crw_ptr crw_table[]={
        NULL,           /* nodev */
        rw_memory,      /* /dev/mem etc */
        NULL,           /* /dev/fd */
        NULL,           /* /dev/hd */
        rw_ttyx,        /* /dev/ttyx */
        rw_tty,         /* /dev/tty */
        NULL,           /* /dev/lp */
        NULL};          /* unnamed pipes */

//// Character device read/write functions.
// Parameters: rw - read or write commands; dev - device number; buf - buffer; count - number
// of read or write bytes; pos - read or write pointer.
// Returns: the actual number of read/write bytes.
int rw_char(int rw,int dev, char * buf, int count, off_t * pos)
{
        crw_ptr call_addr;

// If the device number exceeds the number of system devices, an error code is returned. If
// the device does not have a corresponding read/write function, an error code is also returned.
// Otherwise, the read/write operation function of the corresponding device is called, and the
// actual number of bytes read/written is returned.
        if (MAJOR(dev)>=NRDEVS)
                return -ENODEV;
        if (!(call_addr=crw_table[MAJOR(dev)]))
                return -ENODEV;
        return call_addr(rw,MINOR(dev),buf,count,pos);
}

```

12.13 read_write.c

12.13.1 Function

The read_write.c program implements the file operating system-calls read(), write(), and lseek(). Read() and write() will call the corresponding read or write functions implemented in the previous 4 files according to different file types. Therefore, this program is the upper interface implementation of the functions in the previous four files. lseek () is used to set the file read/write pointer.

The read() system-call first determines the validity of the given parameter, and then checks the type of the file based on the i-node information of the file. If it is a pipe, call the read function in the program pipe.c; if it is a character device file, call the rw_char() character read function in char_dev.c; if it is a block device file, execute the block device read operation in the block_dev.c and return the number of bytes read; if it is a directory file or a normal formal file, call the file read function file_read() in file_dev.c. The implementation of the write() system-call is similar to read().

The lseek() system-call modifies the current read/write pointers in the file structure corresponding to the file handle. For files and pipe files where the read/write pointers cannot be moved, an error code will be given and returned immediately.

12.13.2 Code annotation

Program 12-12 linux/fs/read_write.c

```
/*
 *  linux/fs/read_write.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//     and constants.
// <errno.h> Error number header file. Contains various error numbers in the system.
// <sys/types.h> Type header file. The basic system data types are defined.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//     segment register operations.
#include <sys/stat.h>
#include <errno.h>
#include <sys/types.h>

#include <linux/kernel.h>
#include <linux/sched.h>
#include <asm/segment.h>

// Character device read/write functions. fs/char_dev.c, line 95.
extern int rw_char(int rw,int dev, char * buf, int count, off_t * pos);
// Read pipe operation function. fs/pipe.c, line 16.
extern int read_pipe(struct m_inode * inode, char * buf, int count);
// Write pipe operation function. fs/pipe.c, line 46.
extern int write_pipe(struct m_inode * inode, char * buf, int count);
// Block device read operation function. fs/block_dev.c, line 56.
extern int block_read(int dev, off_t * pos, char * buf, int count);
// Block device write operation function. fs/block_dev.c, line 16.
extern int block_write(int dev, off_t * pos, char * buf, int count);
// Read file manipulation function. fs/file_dev.c, line 17.
extern int file_read(struct m_inode * inode, struct file * filp,
                char * buf, int count);
// Write a file manipulation function. fs/file_dev.c, line 48.
extern int file_write(struct m_inode * inode, struct file * filp,
                char * buf, int count);

//// Relocate file read/write pointer system-call.
```

```
// The parameter fd is the file handle, offset is the new file read or write pointer offset,
// and origin is the starting position of the offset. There are three options: SEEK_SET (0,
// from the beginning of the file), SEEK_CUR (1, from the current read/ write location), SEEK_END
// (2, from the end of the file).
int sys_lseek(unsigned int fd,off_t offset, int origin)
{
        struct file * file;
        int tmp;

// First determine the validity of the parameters provided by the function. If the file handle
// value is greater than the maximum number of files opened by the program NR_OPEN(20), or the
// file structure pointer of the handle is empty, or the i-node field of the corresponding file
// structure is empty, or the specified device file pointer is not positionable, an error code
// is returned and quit. If the i-node corresponding to the file is a pipe node, an error code
// is returned to exit. Because the pipe head and tail pointers are not free to move!
        if (fd >= NR_OPEN || !(file=current->filp[fd]) || !(file->f_inode)
           || !IS_SEEKABLE(MAJOR(file->f_inode->i_dev)))
                return -EBADF;
        if (file->f_inode->i_pipe)
                return -ESPIPE;
// Then relocate the file read/write pointer according to the set positioning flag.
// Origin = SEEK_SET, which means that the file read and write pointer is required to be set
// as the origin at the beginning of the file. If the offset value is less than zero, an error
// code is returned, otherwise we set the file read/write pointer equal to offset.
        switch (origin) {
                case 0:
                        if (offset<0) return -EINVAL;
                        file->f_pos=offset;
                        break;
// Origin = SEEK_CUR, which means that the read/write pointer is required to be relocated as
// the origin of the file's current read/write pointer. If the current pointer of the file plus
// the offset value is less than 0, an error code is returned. Otherwise, an offset value is
// added to the current read/write pointer.
                case 1:
                        if (file->f_pos+offset<0) return -EINVAL;
                        file->f_pos += offset;
                        break;
// Origin = SEEK_END, which means that the read/write pointers are required to be relocated
// from the end of the file. At this time, if the file size plus the offset value is less than
// zero, the error code is returned. Otherwise relocate the read/write pointer to the file length
// plus the offset value.
                case 2:
                        if ((tmp=file->f_inode->i_size+offset) < 0)
                                return -EINVAL;
                        file->f_pos = tmp;
                        break;
// If the origin setting is invalid, an error code is returned. Finally, the file read/write
// pointer value after relocation is returned.
                default:
                        return -EINVAL;
        }
        return file->f_pos;
}
```

```
//// Read file system-call.
// The parameter 'fd' is the file handle, 'buf' is the user buffer, and 'count' is the number
// of bytes to read.
int sys_read(unsigned int fd,char * buf,int count)
{
        struct file * file;
        struct m_inode * inode;

// The function first determines the validity of the parameter. If the file handle value is
// greater than the program maximum open file number NR_OPEN, or the byte count value to be
// read is less than 0, or the file structure pointer of the handle is null, an error code is
// returned and exits. If the number of bytes to be read is equal to 0, then 0 is returned.
        if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
                return -EINVAL;
        if (!count)
                return 0;
// Next verify the buffer memory limit used to hold the data. Then take the i-node of the file
// and call the corresponding read operation function according to the attribute (mode) of the
// i-node. If it is a pipe file, and it is in the read pipe mode, the read pipe operation is
// executed. If it succeeds, the number of bytes read is returned. Otherwise, the error code
// is returned. If it is a character file, we perform a read character device operation and
// return the number of characters read. If it is a block device file, a block device read
// operation is performed and the number of bytes read is returned.
        verify_area(buf,count);
        inode = file->f_inode;
        if (inode->i_pipe)
                return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
        if (S_ISCHR(inode->i_mode))
                return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
        if (S_ISBLK(inode->i_mode))
                return block_read(inode->i_zone[0],&file->f_pos,buf,count);
// If it is a directory file or a regular file, first verify the validity of the read byte 'count'
// and adjust it: if the number of read bytes plus the file current read/write pointer value
// is greater than the file size, set the number of read bytes to be the the file size subtracts
// the current read/write pointer value; if the read number is equal to 0, 0 is returned. Then
// perform the file read operation, return the number of bytes read and exit.
        if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
                if (count+file->f_pos > inode->i_size)
                        count = inode->i_size - file->f_pos;
                if (count<=0)
                        return 0;
                return file_read(inode,file,buf,count);
        }
// If executed here, it means that we can't judge the attributes of the file. Then print the
// node file mode and return an error code.
        printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
        return -EINVAL;
}

//// Write file system-call.
// The parameter 'fd' is the file handle, 'buf' is the user buffer, and 'count' is the number
// of bytes to be written.
```

```
int sys_write(unsigned int fd,char * buf,int count)
{
        struct file * file;
        struct m_inode * inode;

// Similarly, we first check the validity of the function parameters. If the file handle value
// is greater than the program maximum open file number NR_OPEN, or the byte count value to
// be read is less than 0, or the file structure pointer of the handle is null, an error code
// is returned and exits. If the number of bytes to be read is equal to 0, then 0 is returned.
        if (fd>=NR_OPEN || count <0 || !(file=current->filp[fd]))
                return -EINVAL;
        if (!count)
                return 0;
// Then we get the i-node of the file, which is used to call the corresponding write operation
// function according to the mode of the i-node. If it is a pipe file and is in the write pipe
// mode, the write pipe operation is executed. If it succeeds, the number of bytes written is
// returned, otherwise the error code is returned; If it is a character device file, write a
// character device operation and return the number of characters written; If it is a block
// device file, a block device write operation is performed and the number of bytes written
// is returned; If it is a regular file, it performs a file write operation and returns the
// number of bytes written, and exits.
        inode=file->f_inode;
        if (inode->i_pipe)
                return (file->f_mode&2)?write_pipe(inode,buf,count):-EIO;
        if (S_ISCHR(inode->i_mode))
                return rw_char(WRITE,inode->i_zone[0],buf,count,&file->f_pos);
        if (S_ISBLK(inode->i_mode))
                return block_write(inode->i_zone[0],&file->f_pos,buf,count);
        if (S_ISREG(inode->i_mode))
                return file_write(inode,file,buf,count);
// If executed here, it means that we can’t judge the attributes of the file. Then print the
// i-node file attribute and return an error code to exit.
        printk("(Write)inode->i_mode=%06o\n\r",inode->i_mode);
        return -EINVAL;
}

```

12.13.3 User Program Read/Write Operation

After reading the above program, we should be able to understand how a read/write operation in a user program is performed. Below we take the read function as an example to illustrate how a read file function call in the user program is executed and completed.

Usually, applications do not directly invoke Linux system-calls, but instead call subroutines in a function library (such as libc.a). But if you want to improve some efficiency, you can of course call it directly. For a basic library, you usually need to provide a collection of the following basic functions or subroutines:

A. System call interface functions;
B. Memory allocation management functions;
C. Set of signal processing functions;
D. String processing functions;
E. Standard input/output functions;

F. Other function sets, such as bsd functions, encryption and decryption functions, arithmetic operations functions, terminal operation functions, and network socket function sets.

In these function sets, the system-call is the underlying interface function of the operating system. Many functions that involve system calls will invoke system functions with standard names in the system-call interface function set instead of directly using the Linux system-call interface. This can greatly make a function library independent of the operating system it is in, making the function library highly portable and allowing the user program to have higher system independence. For a new library source code, simply replace the part of the system-call (system interface part) with the one of the new operating system, you can basically complete the porting of the function library.

Subroutines in the library can be thought of as an intermediate layer between the application and the kernel system. Their main role is to provide "wrapper functions" for application executing system-calls, in addition to providing functional functions such as computational functions that are not part of the kernel. In this way, the calling interface can be simplified, the interface is more simple and easy to remember, and some parameter verification and error processing can be performed in these wrapping functions, thereby making the program more reliable and stable.

For Linux systems, all input and output operations are done by reading and writing files. Because all peripherals are presented as files in the system, all access between the program and the peripherals can be handled using a uniform file handle. Under normal circumstances, before we can read or write a file, we need to first use the open file operation to notify the operating system of the action to be started. If you want to perform a write operation on a file, you may first need to create this file or delete the previous content in the file. The operating system also needs to check if you have permission to perform these operations. If everything is ok, the open operation will return a file descriptor to the program. The file descriptor will replace the file name to determine the file being accessed. It works just like the file handle in MS-DOS. At this point, all information about an open file is maintained by the system, and the user program only needs to use the file descriptor to access the file.

File read and write operations use read and write system-calls, respectively, and user programs typically execute these two system-calls by accessing the read and write functions in the library. The definition of these two functions are as follows:

```
    int read(int fd, char *buf, int n);
    int write(int fd, char *buf, int n);
```

The first parameter of these two functions is the file descriptor; the second parameter is a character buffer array for storing data that is read or written; the third parameter is the number of data bytes that need to be read or written. The function return value is the byte count transmitted at the time of one call. For read file operations, the returned value may be smaller than the data you want to read. If the return value is 0, it means that it has reached the end of the file; if it returns -1, it means that the read operation encountered an error. For write operations, the value returned is the number of bytes actually written. If the value is not equal to the value specified by the third argument, this indicates that the write operation encountered an error. For a read function, its implementation in the library is as follows:

```
#define __LIBRARY__
#include <unistd.h>
```

```
_syscall3(int,read,int,fd,char *,buf,off_t,count)
```

Where _syscall3() is a macro defined at line 189 of the include/unistd.h header file. If the macro is expanded with the specific parameters above, we can get the following code:

```
int read(int fd, char *buf, off_t count)
{
    long __res;
    __asm__ volatile (
        "int $0x80"
        : "=a" (__res)
        : "" (__NR_read), "b" ((long)(fd)), "c" ((long)(buf)), "d" ((long)(count)));
    if (__res>=0)
        return int __res;
    errno=-__res;
    return -1;
}
```

It can be seen that this expanded macro is a concrete implementation of a read operation function, from which the program enters the system kernel for execution. It uses the embedded assembly statement to execute the Linux system-call interrupt 0x80 with the function number __NR_read(3). This interrupt call returns the actual number of bytes read in the eax(__res) register. If the returned value is less than 0, it means that the read operation error occurs, so the error number is inverted and stored in the global variable errno, and the value of -1 is returned to the calling program.

In the Linux kernel, read operations are implemented in the file system's read_write.c file. When the above system-call interrupt is executed, the sys_read() function starting from line 55 in the read_write.c file is called. The prototype of the sys_read() function is defined as follows:

```
    int sys_read(unsigned int fd, char *buf, int count)
```

This function first determines the validity of the parameter. If the file descriptor value is greater than the maximum number (20) of files opened by the system at the same time, or the size to be read is less than 0, or the file has not been opened yet (the file structure pointer indexed by the file descriptor is null), returns a negative error code. The kernel then verifies that the buffer size that will hold the read data is appropriate. During the verification process, the kernel program verifies the size of the buffer buf according to the specified number of read bytes. If the buf is too small, the system will expand it. Therefore, if the memory buffer opened by the user program is too small, it is possible to destroy the latter data.

Then the kernel code obtains the i-node of the file from the internal file table corresponding to the file descriptor, and classifies and judges the file according to the flag information in the node, calls the following type of read operation function, and returns the actual read number of bytes.

- If the file is a pipe file, call the read pipe function read_pipe() (implemented in fs/pipe.c).
- If it is a character device file, the read character device function rw_char() is called (implemented in fs/char_dev.c). The function then calls the character device driver or operates on the memory character device based on the specific character device subtype.
- If it is a block device file, the block device read function block_read() is called (implemented in

fs/block_dev.c). This function calls the read block function bread() in the buffer cache manager fs/buffer.c, and finally calls the ll_rw_block() function in the block device driver to perform the actual block device read operation.

- If the file is a normal regular file, the regular file read function file_read() (implemented in fs/file_read.c) is called to perform the read data operation. This function is similar to the operation of the read block device. Finally, it also calls the underlying driver access function ll_rw_block() of the block device where the file system is located, but file_read() also needs to maintain information about the internal file table structure, such as moving current pointers of the file.

When the system-call of the read operation returns, the read() function in the library can determine whether the operation is correct according to the return value of the system-call. If the returned value is less than 0, it means that the read error occurs, so the error number is inverted and stored in the global variable errno, and the value of -1 is returned to the application. See Figure 12-29 for the entire process from the execution of the read() function by the user to the actual operation in the kernel.

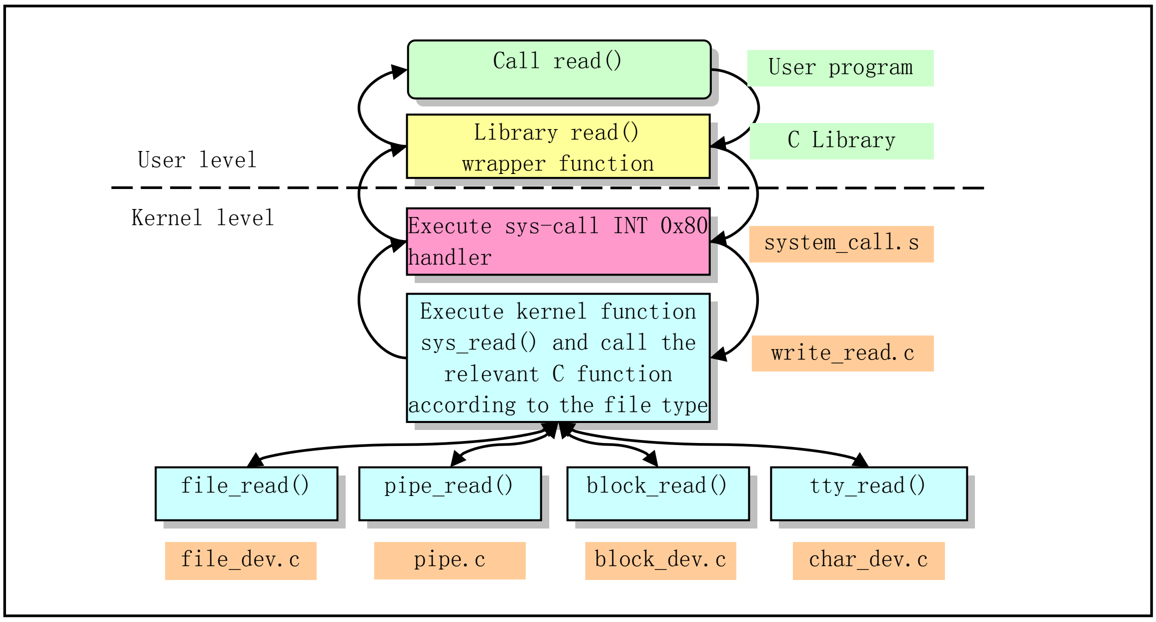


Figure 12-29 Read() function call execution procedure

12.14 open.c

From the beginning of this section, all the procedures described are part of the high-level operations and management part of the file system, which is the fourth part of the procedure in this chapter. This section includes five programs, open.c, exec.c, stat.c, fcntl.c, and ioctl.c.

The open.c program mainly includes file access operating system-calls; The open.c program mainly includes file access operating system calls; exec.c mainly contains program loading and execution functions execve(); stat.c program is used to obtain status information of a file; fcntl.c program implements file access control management; ioctl.c program is used to control access to the device.

12.14.1 Function

The open.c program implements many system calls related to file operations, including file creation, opening and closing, file host and attribute modification, file access permission modification, file operation time modification, and system file system root change.

12.14.2 Code annotation

Program 12-13 linux/fs/open.c

```
/*
 *  linux/fs/open.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <string.h> String header file. Defines some embedded functions about string operations.
// <errno.h> Error number header file. Contains various error numbers in the system.
// <fcntl.h> File control header file. The definition of the operation control constant symbol
//     used for the file and its descriptors.
// <sys/types.h> Type header file. The basic system data types are defined.
// <utime.h> User time header file. The access and modification time structures and the utime()
//     prototype are defined.
// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//     and constants.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
//     communication.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//     segment register operations.
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <utime.h>
#include <sys/stat.h>

#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/kernel.h>

#include <asm/segment.h>

//// Get file system status information.
// The parameter dev is the device number containing the installed file system; ubuf is a ustat
// structure buffer pointer used to store the file system information returned. This system-call
// is used to return statistical status information for a mounted file system. Returns 0 on
// success, and the total number of free blocks and idle i nodes of the file system are added
// to the ustate structure pointed to by ubuf. The ustat structure is defined in
// include/sys/types.h.
int sys_ustat(int dev, struct ustat * ubuf)
{
        return -ENOSYS;                 // Return error code, not been implemented yet.
}

//// Set file access and modification time.
// The parameter filename is the file name, and the times is the access and modification time
// that needs to be set. If the times pointer is not NULL, the time information in the utimbuf
// structure is used to set the access and modification time of the file; if the times pointer
// is NULL, the current time of the system is taken to set the access and modification time
// domain of the specified file.
int sys_utime(char * filename, struct utimbuf * times)
{
        struct m_inode * inode;
        long actime,modtime;

// The file's time is saved in its i-node, so we first get the corresponding i-node based on
// the file name. If the supplied access and modification time structure pointer times are not
// NULL, the time value set by the user is read from the structure. Otherwise, the current time
// of the system is used to set the access and modification time of the file.
        if (!(inode=namei(filename)))
                return -ENOENT;
        if (times) {
                actime = get_fs_long((unsigned long *) &times->actime);
                modtime = get_fs_long((unsigned long *) &times->modtime);
        } else
                actime = modtime = CURRENT_TIME;
// Then modify the access time field and the modification time field in the i-node, set the
// i-node modified flag, put back the i-node, and return 0.
        inode->i_atime = actime;
        inode->i_mtime = modtime;
        inode->i_dirt = 1;
        iput(inode);
        return 0;
}

/*
 * XXX should we use the real or effective uid?  BSD uses the real uid,
 * so as to make this call useful to setuid programs.
 */
//// Check the access rights of the file.
// The parameter mode is the checked access attribute. It consists of three valid bits: R_OK
// (value 4), W_OK(2), X_OK(1), and F_OK(0), which indicate that the file being checked is
// readable, writable, executable, or file exists. Returns 0 if access is allowed, otherwise
// returns an error code.
int sys_access(const char * filename,int mode)
{
        struct m_inode * inode;
        int res, i_mode;

// The access permission of the file is also stored in the i-node of the file, so we must first
```

```
   // obtain the i-node corresponding to the file name. The detected access attribute mode consists
   // of the lower 3 bits, so "AND" octal 07 is required to clear all high bits. If the i-node
   // corresponding to the file name does not exist, an error code with no permission is returned.
   // If the i-node exists, the file attribute code, and the i-node is put back. In addition, the
   // statement "iput(inode);" on line 57 is preferably placed after line 61.
53         mode &= 0007;
54         if (!(inode=namei(filename)))
55                 return -EACCES;                     // error code: No access rights.
56         i_mode = res = inode->i_mode & 0777;
57         iput(inode);
   // If the current process user is the owner of the file, the file owner attribute is taken;
   // otherwise, if the current process user belongs to the same group as the file owner, the file
   // group attribute is taken; otherwise, the lowest 3 bits of the res is 'Others' access rights
   // of the file.
58         if (current->uid == inode->i_uid)
59                 res >>= 6;
60         else if (current->gid == inode->i_gid)
61                 res >>= 6;                          // Here should be "res >>= 3;" [??]
   // At this time, the lowest 3 bits of res are the access attribute bits selected according to
   // the relationship between the current process and the file. Now let's check these 3 bits.
   // If the file has the attribute bit mode queried by the parameter, the access is granted and
   // returns 0.
62         if ((res & 0007 & mode) == mode)
63                 return 0;
64         /*
65          * XXX we are doing this test last because we really should be
66          * swapping the effective with the real user id (temporarily),
67          * and then calling suser() routine.  If we do call the
68          * suser() routine, it needs to be called last.
69          */
   // If the current user ID is 0 (super user) and the mask execution bit is 0 or the file can
   // be executed and searched by anyone, it returns 0, otherwise it returns an error code.
70         if ((!current->uid) &&
71             (!(mode & 1) || (i_mode & 0111)))
72                 return 0;
73         return -EACCES;                             // error code: No access rights.
74 }
75
   //// Change the current working directory.
   // The parameter filename is the directory name.
   // Returns 0 if the operation is successful, otherwise it returns an error code.
76 int sys_chdir(const char * filename)
77 {
78         struct m_inode * inode;
79
   // Changing the current working directory requires that the current working directory field
   // of the process task structure be pointed to the i-node of the given directory name. So we
   // first take the i-node of the given name. If the i-node is not a directory i-node, the i-node
   // is put back and an error code is returned.
80         if (!(inode = namei(filename)))
81                 return -ENOENT;             // error code: file or directory does not exist.
82         if (!S_ISDIR(inode->i_mode)) {
83                 iput(inode);
```

```
                return -ENOTDIR;          // error code: not a directory name.
        }
// Then release the i-node of the original working directory of the process and point it to
// the newly set working directory i-node and return 0.
        iput(current->pwd);
        current->pwd = inode;
        return (0);
}

//// Change the root directory.
// Set the specified directory name to be the root directory '/' of the current process.
// Returns 0 if the operation is successful, otherwise returns an error code.
int sys_chroot(const char * filename)
{
        struct m_inode * inode;

// This system-call is used to change the root field in the current process task structure to
// point to the i-node of the given directory name. If the i-node of the given name does not
// exist, an error code is returned. If the i-node is not a directory i-node, the i-node is
// put back and an error code is returned.
        if (!(inode=namei(filename)))
                return -ENOENT;
        if (!S_ISDIR(inode->i_mode)) {
                iput(inode);
                return -ENOTDIR;
        }
// Then release the original root i-node of the current process and reset it to the i-node of
// the specified directory name and return 0.
        iput(current->root);
        current->root = inode;
        return (0);
}

//// Modify file attribute (mode).
// The parameter mode is the new file mode.
// Returns 0 if the operation is successful, otherwise returns an error code.
int sys_chmod(const char * filename,int mode)
{
        struct m_inode * inode;

// This function sets a new access mode for the specified file. Because the mode of the file
// is in the i-node of the file, we need to first get the i-node corresponding to the file name.
// If the effective user id of the current process is different from the user id of the file
// i-node and is not a superuser, then the file i-node is put back and an error code is returned
// (no access rights).
        if (!(inode=namei(filename)))
                return -ENOENT;
        if ((current->euid != inode->i_uid) && !suser()) {
                iput(inode);
                return -EACCES;
        }
// Otherwise, the given mode is used to modify the file mode of the i-node, and the i-node modified
// flag is set, the i-node is put back, and 0 is returned.
```

```
        inode->i_mode = (mode & 07777) | (inode->i_mode & ~07777);
        inode->i_dirt = 1;
        iput(inode);
        return 0;
}

//// Modify the file owner.
// The parameter uid is the user identifier (user ID) and the gid is the group ID.
// Returns 0 if the operation is successful, otherwise returns an error code.
int sys_chown(const char * filename,int uid,int gid)
{
        struct m_inode * inode;

// This call is used to set the user and group IDs in the file i-node, so we first need to get
// the i-node of the given file name. If the current process is not a superuser, put back the
// i-node and return an error code (no access rights).
        if (!(inode=namei(filename)))
                return -ENOENT;
        if (!suser()) {
                iput(inode);
                return -EACCES;
        }
// Otherwise, we use the value provided by the parameter to set the user ID and group ID of
// the file i-node, and set the i-node to modify the flag, put back the i-node, and return 0.
        inode->i_uid=uid;
        inode->i_gid=gid;
        inode->i_dirt=1;
        iput(inode);
        return 0;
}

//// Check and set terminal character device properties.
// This auxiliary function is only used for the following system-call sys_open(). It is used
// to check the modification and setting of the current process attributes and system tty table
// when the open file is a tty terminal character device.
// Returns 0 if the operation is successful. A return of -1 indicates a failure, and the
// corresponding character device cannot be opened.
static int check_char_dev(struct m_inode * inode, int dev, int flag)
{
        struct tty_struct *tty;
        int min;                                      // subdevice number.

// This function only handles situations where the major device number is 4 (/dev/ttyxx file)
// or 5 (/dev/tty file). The subdevice number of /dev/tty is 0. In fact, if a process has a
// controlling terminal, /dev/tty is the synonymous name of the process control terminal device,
// that is, the /dev/tty device is a virtual device, which corresponds to one of the /dev/ttyxx
// devices actually used by the process. For a process, if it has a control terminal, the tty
// field in its task structure will be a sub-device number of device No.4.
// Now let's get the control terminal device of the current process and temporarily store it
// in the min variable for later use. If the parameter specifies device 5 (/dev/tty), then let
// min be equal to the tty field value in the process structure, that is, get the sub-device
// number of device No.4. Otherwise, if a sub-device of a device No. 4 is given in the parameter,
// the sub-device number is directly taken. If the obtained sub-device number of device No.
// 4 is less than 0, then the process does not have a control terminal, or the device number
// is incorrect, then -1 is returned.
        if (MAJOR(dev) == 4 || MAJOR(dev) == 5) {
                if (MAJOR(dev) == 5)
                        min = current->tty;
                else
                        min = MINOR(dev);
                if (min < 0)
                        return -1;
// The primary pseudo terminal device can only be used exclusively by a process. Therefore,
// if the sub-device number indicates a primary pseudo terminal and the open file i-node reference
// count is greater than 1, it indicates that the device has been used by other processes. so
// the character device can no longer be opened again, and -1 is returned. Otherwise, we let
// the pointer tty point to the corresponding structure item in the tty table.
// If the open file operation flag does not contain the flag O_NOCTTY, and the process
// is the group leader, and has no control terminal, and the session field in the tty structure
// is 0 (indicating that the terminal is not yet the control terminal of any process group),
// then It is allowed to set this terminal device min for the process as its control terminal.
// Therefore, we set the terminal device number field tty value in the process task structure
// to be equal to min, and set the session number and group number of the tty structure to be
// equal to the session number and group number of the process, respectively.
                if ((IS_A_PTY_MASTER(min)) && (inode->i_count>1))
                        return -1;
                tty = TTY_TABLE(min);
                if (!(flag & O_NOCTTY) &&
                    current->leader &&
                    current->tty<0 &&
                    tty->session==0) {
                        current->tty = min;
                        tty->session= current->session;
                        tty->pgrp = current->pgrp;
                }
// If the open file operation flag contains the O_NONBLOCK flag, we need to make the relevant
// settings for the character terminal device: To satisfy the read operation, the minimum number
// of characters to be read is 0, the timeout timing value is set to 0, and the terminal device
// is set to the non-canonical mode. The non-blocking mode works only in non-canonical mode.
// In this mode, when both VMIN and VTIME are set to 0, no matter how many characters are in
// the auxiliary queue, the process reads how many characters and returns immediately.
                if (flag & O_NONBLOCK) {
                        TTY_TABLE(min)->termios.c_cc[VMIN] =0;
                        TTY_TABLE(min)->termios.c_cc[VTIME] =0;
                        TTY_TABLE(min)->termios.c_lflag &= ~ICANON;
                }
        }
        return 0;
}

//// Open (or create) a file.
// The parameter 'flag' is the open file flag, which can take values: O_RDONLY, O_WRONLY or
// O_RDWR, and O_CREAT, O_EXCL, O_APPEND and other combinations of flags. If this call creates
// a new file, mode is used to specify the permission properties of the file. These attributes
// are S_IRWXU (the file owner can read, write, and execute), S_IRUSR (user can read), S_IRWXG
// (group members can read, write, and execute), and so on. For newly created files, these
```

```
    // properties apply only to future access to the file. An open call that creates a read-only
    // file will also return a readable and writable file handle. If the call operation succeeds,
    // the file handle (file descriptor) is returned, otherwise an error code is returned. See
    // the files sys/stat.h, fcntl.h.
171 int sys_open(const char * filename,int flag,int mode)
172 {
173         struct m_inode * inode;
174         struct file * f;
175         int i,fd;
176 
    // The parameters are processed first. We combine the file mode set by the user with the process
    // mode mask to generate the allowed file mode. In order to create a file handle for an open
    // file, we need to search the array of file structure pointers in the process structure to
    // find a free entry. The index number fd of the free item is the handle value. If there is
    // no free item, an error code is returned (invalid parameter).
177         mode &= 0777 & ~current->umask;
178         for(fd=0 ; fd<NR_OPEN ; fd++)
179                 if (!current->filp[fd])                 // free item found.
180                         break;
181         if (fd>=NR_OPEN)
182                 return -EINVAL;
    // Then we set the current process's close_on_exec file handle bitmap and reset the corresponding
    // bit. close_on_exec is a bitmap flag for all file handles of a process. Each bit represents
    // an open file descriptor that determines the file handle that needs to be closed when the
    // system-call execve() is called. When a program creates a child process using the fork()
    // function, it usually calls the execve() function in the child process to load another new
    // program, and the new program starts executing in the child process. If the bit in the
    // close_on_exec of a file handle is set, the corresponding file handle will be closed when
    // execve() is executed, otherwise the file handle will always be open.
    // When a file is opened, the file handle is also open by default in the child process, so the
    // corresponding bit needs to be reset here. Then we look for a free entry in the file table
    // (the entry with a reference count of 0) for the open file, and we let f point to the file
    // table array, and start searching. In addition, the pointer assignment "0+file_table" on line
    // 184 is equivalent to "file_table" and "&file_table[0]", but this may be more clear.
183         current->close_on_exec &= ~(1<<fd);
184         f=0+file_table;
185         for (i=0 ; i<NR_FILE ; i++,f++)
186                 if (!f->f_count) break;                 // idle item found.
187         if (i>=NR_FILE)
188                 return -EINVAL;
    // At this point we let the file structure pointer of the process's file handle fd point to
    // the searched file structure, increment the file reference count by 1, and then call the
    // function open_namei() to perform the open operation. If the return value is less than 0,
    // it indicates an error, so the file structure just applied is released, and the error code
    // is returned. If the file open operation is successful, the inode is the i-node pointer of
    // the opened file.
189         (current->filp[fd]=f)->f_count++;
190         if ((i=open_namei(filename,flag,mode,&inode))<0) {
191                 current->filp[fd]=NULL;
192                 f->f_count=0;
193                 return i;
194         }
    // Based on the mode field of the opened file i-node, we can know the type of the file. For
// different types of files, we need to do some special processing. If the character device
// file is opened, then we need to call the check_char_dev() function to check if the current
// process can open the file. If allowed (the function returns 0), then in the check_char_dev(),
// the control terminal will be set for the process according to the file open flag; if the
// character device file is not allowed to be opened, then we can only release the file item
// and handle resource requested above, and returns the error code.
/* ttys are somewhat special (ttyxx major==4, tty major==5) */
        if (S_ISCHR(inode->i_mode))
                if (check_char_dev(inode,inode->i_zone[0],flag)) {
                        iput(inode);
                        current->filp[fd]=NULL;
                        f->f_count=0;
                        return -EAGAIN;            // resource is temporarily unavailable.
                }
// If a block device file is open, check if the disc has been replaced. If it is replaced, all
// the buffer blocks of the device in the cache buffer need to be invalidated.
/* Likewise with block-devices: check for floppy_change */
        if (S_ISBLK(inode->i_mode))
                check_disk_change(inode->i_zone[0]);
// Now we initialize the file structure of the opened file. Set the file structure properties
// and flags, set the handle reference count to 1, and set the i-node field to the i-node of
// the open file, the file read-write pointer to 0, and finally the file handle is returned.
        f->f_mode = inode->i_mode;
        f->f_flags = flag;
        f->f_count = 1;
        f->f_inode = inode;
        f->f_pos = 0;
        return (fd);
}

//// Create file system call.
// The parameter pathname is the path name, and the mode is the same as the sys_open() above.
// If successful, the file handle is returned, otherwise the error code is returned.
int sys_creat(const char * pathname, int mode)
{
        return sys_open(pathname, O_CREAT | O_TRUNC, mode);
}

// Close file system call.
// The parameter fd is the file handle.
// Returns 0 if successful, otherwise returns an error code.
int sys_close(unsigned int fd)
{
        struct file * filp;

// First check the validity of the parameters. If the given file handle value is greater than
// the number of files NR_OPEN that the program can open at the same time, an error code (invalid
// parameter) is returned. Then reset the corresponding bit of the file handle in the
// close_on_exec bitmap of the process. In addition, if the file structure pointer of the file
// handle is NULL, an error code is returned.
        if (fd >= NR_OPEN)
                return -EINVAL;
        current->close_on_exec &= ~(1<<fd);
        if (!(filp = current->filp[fd]))
                return -EINVAL;
// Now we set the file structure pointer of the file handle to NULL. If the handle reference
// count in the file structure is already 0 before the file is closed, the kernel error occurs
// and the machine is stopped. Otherwise, the reference count of the file structure is decremented
// by 1. At this point if it is not yet 0, then there are other processes that are using the
// file, so it returns 0 (success). If the reference count is equal to 0 now, the file has no
// process references and the file structure has become free. Then release the file i-node and
// return 0.
        current->filp[fd] = NULL;
        if (filp->f_count == 0)
                panic("Close: file count is 0");
        if (--filp->f_count)
                return (0);
        iput(filp->f_inode);
        return (0);
}

```

12.15 exec.c

12.15.1 Function

The exec.c program implements the loading and execution of binary executables and shell script files. The main function is do_execve(), which is the C handler for the system-call interrupt (int 0x80) function number __NR_execve(), and the kernel implementation function for the exec() function cluster. The other five related exec functions are generally implemented in the library function, and eventually this function is called.

The Linux kernel version 0.12 only supports executable files in the a.out format. This format is simple and straightforward and is suitable for introductory learning. The execution file header of the a.out format holds an exec data structure, as shown below. It describes the basic format and content of the executable file. See the include/a.out.h header file in Chapter 14 for a detailed description of the a.out format.

```
// The header structure of the executable.
struct exec {
  unsigned long a_magic;        /* Use macros N_MAGIC, etc for access */
  unsigned a_text;              /* length of text, in bytes */
  unsigned a_data;              /* length of data, in bytes */
  unsigned a_bss;               /* length of uninitialized data area for file, in bytes */
  unsigned a_syms;              /* length of symbol table data in file, in bytes */
  unsigned a_entry;             /* start address */
  unsigned a_trsize;            /* length of relocation info for text, in bytes */
  unsigned a_drsize;            /* length of relocation info for data, in bytes */
};
```

When a program creates a child process using the fork(), one of the exec() cluster functions is usually called in the child process to load and execute another new program. At this point, the code, data segment (including heap, stack content) of the child process will be completely replaced by the new program, and the new program will be executed in the child process. The main uses of the execve() function are:

- Perform initialization operations on command line parameters and environment parameter space pages -- set the initial space start pointer; initialize the space page pointer array to (NULL); get the i-node of the file according to the execution file name; calculate the number of parameters and environment variables number; check file type and execute permission;
- According to the header data structure at the beginning of the executable file, the information is processed therein - the file header is read according to the executed file i-node; If it is a shell script (the first line starts with #!), analyze the shell program name and its parameters, and execute the shell program with the executable file as a parameter; check whether it is executable according to the magic number of the file and the length of the segment;
- The initialization operation before running the new file on the current calling process -- point to the i-node of the new executable file; reset the signal processing handle; set the local descriptor base address and segment length according to the header information; set the parameter and environment parameter page pointer; modify the contents of each field of the process.
- Replace the return address of the original call execve() program on the stack with the new executable running address and run the newly loaded executable program.

During the execution of execve(), the kernel clears the page directory and page table entries of the original program copied by fork() and releases the corresponding pages. The kernel only re-sets the information in the process structure for the newly loaded program code, and requests and maps the memory pages occupied by the command line parameters and environment parameter blocks, and sets the code execution point. At this point, the kernel does not immediately load the code and data of the new program from the block device where the executable file is located. When execve() exits and returns, the new program starts executing, but the initial execution will definitely cause a page fault exception to occur. Because the code and data have not yet been read into memory from the block device. At this time, the page fault exception processing procedure requests a memory page (memory frame) for the new program in the main memory area according to the linear address causing the exception, and reads the specified page from the block device, and also set corresponding memory page directory entries and page table entries for the linear address. This method of loading executable files is called load on demand, as described in the memory management chapter.

In addition, since the new program is executed in the child process, the child process is the process of the new program, and the process ID of the new program is the process ID of the child process. Similarly, the properties of the child process become properties of the new program process, and the processing of the open file is related to the close on exec flag of each file descriptor, refer to description in the file linux/fs/fcntl.c. Each open file descriptor in the process has an close- on- execution flag. In the process control structure, the flag is represented by an unsigned long integer close_on_exec, each bit of which represents the flag for each file descriptor. If a file descriptor is set in the corresponding bit in close_on_exec, the descriptor will be closed when execve() is executed, otherwise the descriptor will always be open. Unless we use the file control function fcntl() to specifically set this flag, the kernel default operation will keep the descriptor open after execve() execution.

The meaning of the command line parameters and environment parameters is explained below. When a user types a command at the command prompt, the program specified to execute accepts the typed command line arguments from that command line. For example, when a user types the following file name list command:

```
ls -l /home/john/
```

The shell process creates a new process and executes the /bin/ls command there. The three parameters ls, -l, and /home/john/ on the command line when the /bin/ls executable is loaded will be inherited by the new process. In an environment that supports C, when calling the main function main() of the program, it takes two arguments.

```
int main(int argc, char *argv[])
```

The first is the number of arguments on the command line when the program is executed, usually argc(argument count); the second is an array of pointers to the string arguments (argv -- argument vector). Each string represents a parameter, and the end of the argv array always ends with a null. Usually, argv[0] is the name of the program being executed, so the value of argc is at least 1. For the above example, argc=3, argv[0], argv[1], and argv[2] are 'ls', '-l', and '/home/john/', respectively, and argv[3] = NULL, as shown in Figure 12-30.

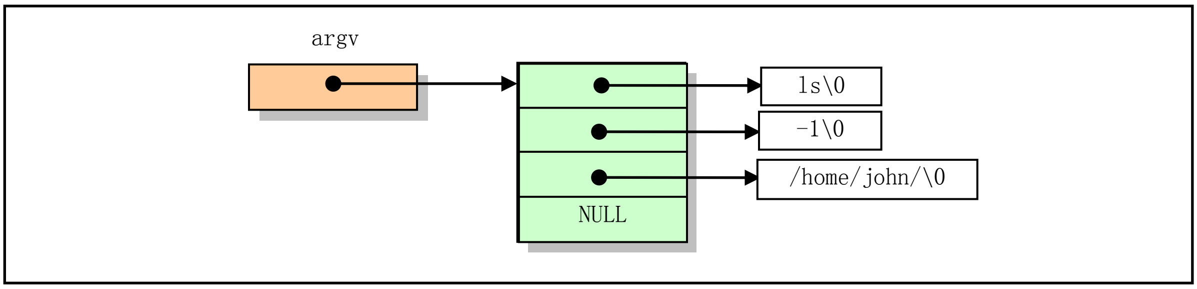


Figure 12-30 Command line argument pointer array argv[]

Main() also has an optional third parameter that contains environment variable parameters that are used to customize the environment settings of the executable and provide environment setting parameter values for it. It is also an array of pointers to string arguments and ends with NULL, except that they are environment variable values. When a program needs to explicitly use an environment variables, main() can be declared as:

```
int main(int argc, char *argv[], char *envp[])
```

The environment string is in the form:

```
VAR_NAME=somevalue
```

Where VAR_NAME represents the name of an environment variable, and the string after the equal sign represents the value assigned to this environment variable. At the command line prompt, type the shell internal command 'set' to display a list of environment parameters in the current environment. Command line parameters and environment strings are placed at the top of the user stack in the program memory area before the program begins execution, as explained below.

The execve() function has a lot of processing operations on command line arguments and environment space. The parameters and environment space of each process or task can have a total of MAX_ARG_PAGES pages, and the total size can reach 128kB bytes. The way data is stored in this space is similar to the stack operation, where parameters or environment variable strings are stored backwards from the end of the assumed 128kB space. Initially, the program defines an offset value p in the space pointing to the end of the space (128kB-4 bytes). This offset will fall back as the data is stored, as shown in Figure 12-31. As can be seen from the figure, p clearly indicates how much free space is left in the current parameter environment space. The copy_string() function is used to copy command line arguments and environment strings from the user memory space to the kernel free page. When analyzing copy_string(), we can refer to this figure.

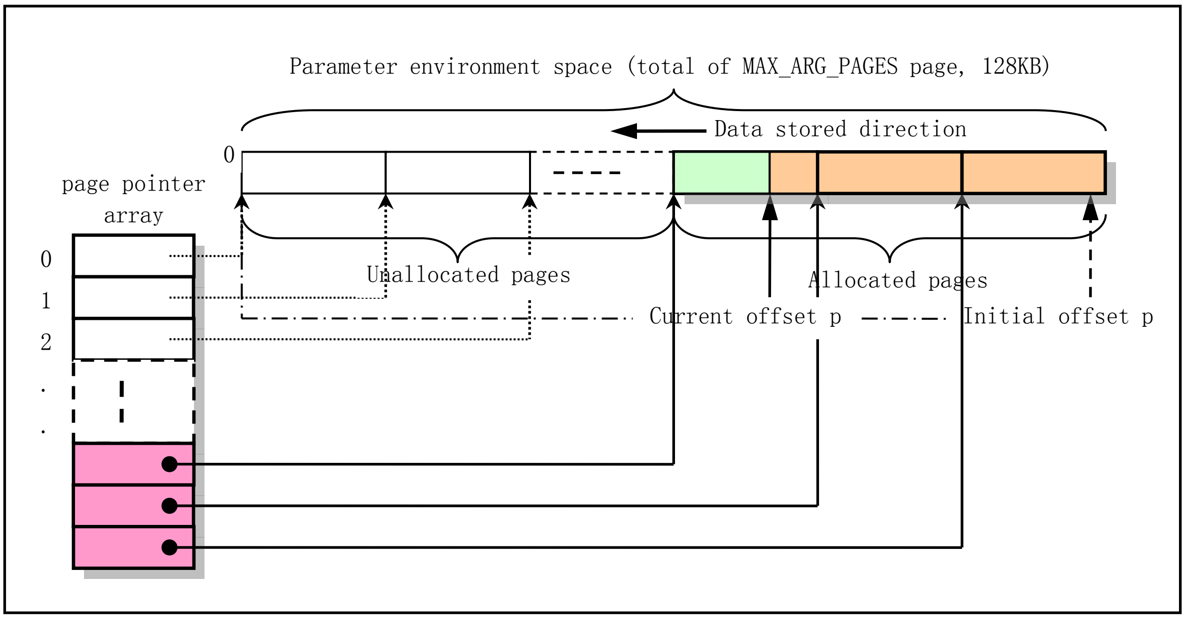


Figure 12-31 Parameter and environment variable string space

After copy_string() is executed, the kernel code will adjusts p to be the parameter and environment variable pointers from the beginning of the process's logical address space, as shown in Figure 12-32. The method is to subtract the size occupied by the parameters and environment variables (128KB - p) from the maximum logical space size of 64MB occupied by a process. The left part of p1 will also use the create_tables() function to hold a pointer table of parameters and environment variables, and p1 will be adjusted to the left again to point to the beginning of the pointer table. Then the resulting pointer is page aligned, and finally the initial stack pointer sp is obtained.

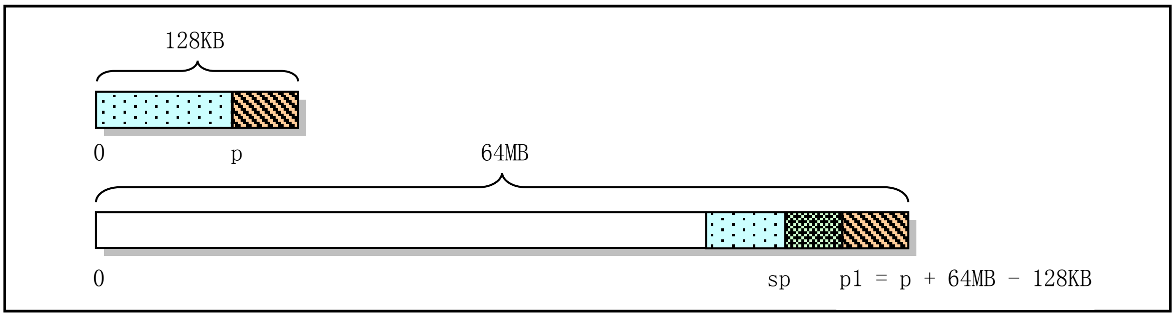


Figure 12-32 Method for converting p to the initial stack pointer

The create_tables() function is used to create an environment and parameter variable pointer table in the new program stack based on the given current stack pointer value p and the parameter variable value argc and the number of environment variables envc, and return the adjusted stack pointer value. Then the pointer is page aligned, and finally the initial stack pointer sp is obtained. The form of the stack pointer table after creation is shown in Figure 12-33.

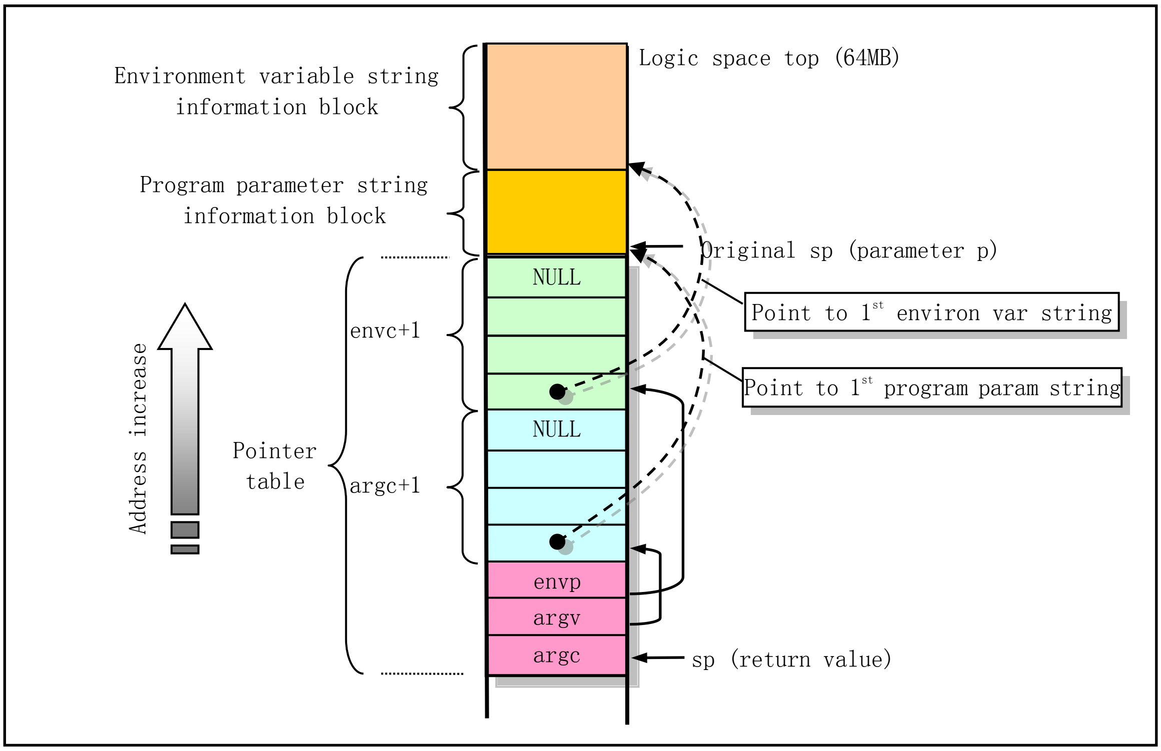

Logic space top (64MB)
Environment variable string information block
Program parameter string information block
Original sp (parameter p)
Point to 1st environ var string
Point to 1st program param string
NULL
envc+1
Address increase
Pointer table
NULL
argc+1
envp
argv
argc
sp (return value)

Figure 12-33 Schematic diagram of the pointer table in the new program stack

When the function do_execve() returns last, it replaces the code pointer eip on the stack of the original call system interrupt program with the code entry point pointing to the new executable program, and replaces the stack pointer with the stack pointer esp of the new execution file. Thereafter, the return instruction of this system-call will eventually pop up the data in the stack and cause the CPU to execute the new executable file, as shown in Figure 12-34.

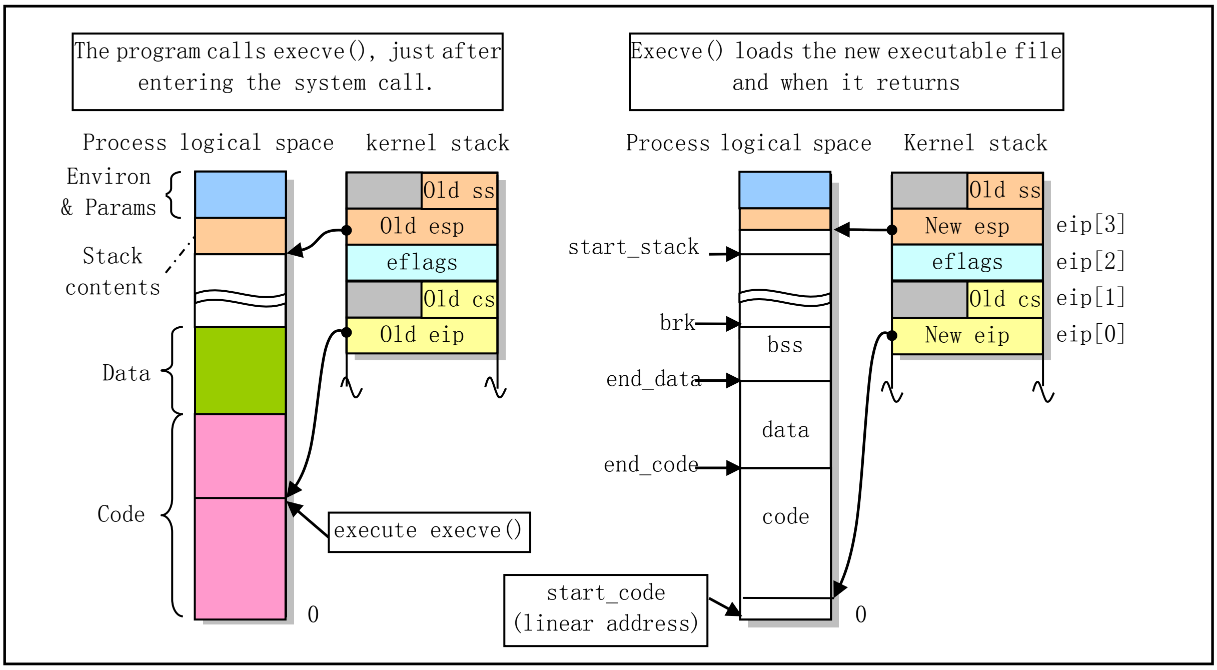


Figure 12-34 Changes in the esp and eip in the stack during loading a execution file

The left half of the figure is the case when the process logic 64MB space still contains the original execution program; the right half is the case when the original execution code and data are released and the stack and code pointers are updated. The shaded (color) section of the figure contains code or data information. The start_code in the process task structure is the address in the linear space of the CPU, and the remaining variable values are the addresses in the process logical space.

12.15.2 Code annotation

Program 12-14 linux/fs/exec.c

```
/*
 *  linux/fs/exec.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * #!-checking implemented by tytso.
 */

/*
 * Demand-loading implemented 01.12.91 - no need to read anything but
 * the header into memory. The inode of the executable is put into
 * "current->executable", and page faults do the actual loading. Clean.
 *
 * Once more I can proudly say that linux stood up to being changed: it
 * was less than 2 hours work to get demand-loading completely implemented.
 */

  // <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
```

```
   //     manipulation function prototypes.
   // <errno.h> Error number header file. Contains various error numbers in the system.
   // <string.h> String header file. Defines some embedded functions about string operations.
   // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
   //     and constants.
   // <a.out.h> The a.out header file defines the a.out executable file format and some macros.
   // <linux/fs.h> File system header file. Define the file table structure (file,buffer_head,
   //     m_inode, etc.).
   // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
   //     of the initial task 0, and some embedded assembly function macro statements about the
   //     descriptor parameter settings and acquisition.
   // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
   //     used functions of the kernel.
   // <linux/mm.h> Memory management header file. Contains page size definitions and some page
   //     release function prototypes.
   // <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
   //     segment register operations.
#include <signal.h>
#include <errno.h>
#include <string.h>
#include <sys/stat.h>
#include <a.out.h>

#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <asm/segment.h>

extern int sys_exit(int exit_code);        // kernel/exit.c, line 365.
extern int sys_close(int fd);              // fs/open.c, line 219.

/*
 * MAX_ARG_PAGES defines the number of pages allocated for arguments
 * and envelope for the new program. 32 should suffice, this gives
 * a maximum env+arg of 128kB !
 */
#define MAX_ARG_PAGES 32

//// Use the library file.
// Parameters: library - the name of the library file.
// Select a library file for the process and replace the i-node field of the current library
// file of the process with the i-node of the library file specified here. If the given library
// of the parameter is empty, the current library file of the process is released.
// Returns: 0 if successful, otherwise returns an error code.
int sys_uselib(const char * library)
{
        struct m_inode * inode;
        unsigned long base;

// First determine if the current process is a normal process. This is done by looking at the
// size of the current process space, because the space size of the normal process is set to
// TASK_SIZE (64MB). Therefore, if the size of the process logical address space is not equal
// to TASK_SIZE, an error code (invalid parameter) is returned, otherwise the library file inode
// is taken. If the library file name pointer is empty, set the inode equal to NULL.
        if (get_limit(0x17) != TASK_SIZE)
                return -EINVAL;
        if (library) {
                if (!(inode=namei(library)))            /* get library inode */
                        return -ENOENT;
        } else
                inode = NULL;
/* we should check filetypes (headers etc), but we don't */
// Then put back the process original library file i-node, and preset the process library i-node
// field to be null. Then get the location of the library code of the process, and release the
// page table of the original library code and the memory page occupied. Finally, let the process
// library i-node field point to the new library i-node and return 0 (success). Similar to loading
// an executable file, the actual library code will be loaded into memory when it is actually
// used. In addition, the library file code is placed at the end of the process space, the size
// is a multiple of 4MB, see the linux/sched.h file.
        iput(current->library);
        current->library = NULL;
        base = get_base(current->ldt[2]);
        base += LIBRARY_OFFSET;                         // linux/sched.h, line 26.
        free_page_tables(base,LIBRARY_SIZE);
        current->library = inode;
        return 0;
}

/*
 * create_tables() parses the env- and arg-strings in new user
 * memory and creates the pointer tables from them, and puts their
 * addresses on the "stack", returning the new stack pointer value.
 */
//// Create a parameter and environment variable pointer table in the new task stack.
// See Figure 12-31 through Figure 12-33 above.
// Parameters: p - the parameter and environment information offset pointer in the data segment;
// argc - the number of parameters; envc - the number of environment variables.
// This function returns the stack pointer.
static unsigned long * create_tables(char * p,int argc,int envc)
{
        unsigned long *argv,*envp;
        unsigned long * sp;

// The stack pointer is addressed with a 4-byte boundary, so here we need to make sp a multiple
// of 4. At this point sp is at the end of the parameter environment table. Then we first move
// sp down (low address direction), empty the space occupied by the environment variable pointers
// on the stack, and let the environment pointer envp point to it. One more place here is used
// to store a NULL value at the end. Move sp down again, leaving room for the command line argument
// pointer and pointing the argv pointer to it. Similarly, a more reserved location is used
// to hold a NULL value. At this point sp points to the beginning of the parameter pointer block,
// and then we push the environment parameter block pointer envp and the command line parameter
// block pointer, as well as the number of command line parameters, onto the stack.Note that
// the following pointer plus 1 operation causes sp to increment 4 bytes.
        sp = (unsigned long *) (0xfffffffc & (unsigned long) p);
        sp -= envc+1;                                // That is sp = sp - (envc+1);
        envp = sp;
        sp -= argc+1;
        argv = sp;
        put_fs_long((unsigned long)envp,--sp);       // put in user space.
        put_fs_long((unsigned long)argv,--sp);
        put_fs_long((unsigned long)argc,--sp);
// Then put the pointers of the command line and the pointers of the environment variables into
// the corresponding places reserved above, and finally place a NULL pointer.
        while (argc-->0) {
                put_fs_long((unsigned long) p,argv++);
                while (get_fs_byte(p++)) /* nothing */ ; // p points to next param string.
        }
        put_fs_long(0,argv);
        while (envc-->0) {
                put_fs_long((unsigned long) p,envp++);
                while (get_fs_byte(p++)) /* nothing */ ; // p points to next param string.
        }
        put_fs_long(0,envp);
        return sp;                   // returns the current new stack pointer constructed.
}

/*
 * count() counts the number of arguments/envelopes
 */
//// Count the number of parameters.
// Parameters: argv - an array of parameter pointers, the last pointer is NULL.
// Counts the number of pointers in the pointer array, and then returns the count value.
static int count(char ** argv)
{
        int i=0;
        char ** tmp;

        if (tmp = argv)
                while (get_fs_long((unsigned long *) (tmp++)))
                        i++;

        return i;
}

/*
 * 'copy_string()' copies argument/envelope strings from user
 * memory to free pages in kernel mem. These are in a format ready
 * to be put directly into the top of new user memory.
 *
 * Modified by TYT, 11/24/91 to add the from_kmem argument, which specifies
 * whether the string and the string array are from user or kernel segments:
 *
 * from_kmem     argv *        argv **
 *    0          user space    user space
 *    1          kernel space  user space
 *    2          kernel space  kernel space
 *
 * We do this by playing games with the fs segment register.  Since it
 * it is expensive to load a segment register, we try to avoid calling
 * set_fs() unless we absolutely have to.
 */
//// Copy the parameter strings into the parameter and environment space of the process.
// Parameters: argc - the number of arguments to be added; argv - an array of argument pointers;
// page - an array of arguments to the environment space page pointer; p - an offset pointer
// in the space of the arguments table, always pointing to the header of the copied string;
// from_kmem - the character string source flag.
// In the do_execve() function, p is initialized to point to the last longword in the argument
// table (128kB) space, and the argument strings are reversely copied to it in the stack operation
// mode. Therefore, the p pointer will gradually decrease as the copy information increases,
// and always points to the head of the parameter string. The string source flag from_kmem should
// be a new parameter added by TYT(Tytso) to give execve() the ability to execute script files.
// When execve() does not have the ability to run script files, all parameter strings will be
// in the user data space.
// Returns: the current head pointer of the argument/environment space, or 0 if error.
static unsigned long copy_strings(int argc,char ** argv,unsigned long *page,
                unsigned long p, int from_kmem)
{
        char *tmp, *pag;
        int len, offset = 0;
        unsigned long old_fs, new_fs;

// First, the current segment register DS (pointing to the kernel data segment) and FS (user
// data segment) values are taken and stored in the variables new_fs and old_fs, respectively.
// If the string and its pointers are in kernel space, set fs to point to kernel space.
        if (!p)
                return 0;       /* bullet-proofing */  // offset pointer verification
        new_fs = get_ds();
        old_fs = get_fs();
        if (from_kmem==2)       // string and string pointers all in kernel space.
                set_fs(new_fs);
// Then loop through the parameters, start copying backwards from the last parameter, and copy
// to the specified offset address. In the loop, first take the current string pointer that
// needs to be copied. If the string is in user space and the string array (string pointer)
// is in kernel space, first set the FS segment register to point to the kernel data segment
// (DS), and restore the FS immediately after taking the string pointer tmp in the kernel data
// space, otherwise the string pointer is directly taken from the user space to tmp without
// modifying the FS value.
        while (argc-- > 0) {
                if (from_kmem == 1)     // FS points to kernel space if ponters in it.
                        set_fs(new_fs);
                if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc)))
                        panic("argc is wrong");
                if (from_kmem == 1)          // if string pointers in kernel.
                        set_fs(old_fs);      // FS points back to user space.
// The string is then taken from the user space and the length len of the parameter string is
// calculated, after which tmp points to the end of the string. If the string length exceeds
// the free length remaining in the parameter and environment space at this time, the space
// is not enough. The FS segment register value is then restored (if changed) and returns 0.
// However, this shouldn't happened because the parameters/environment space are 128KB.
                len=0;                         /* remember zero-padding */
                do {
```

```
                                len++;
                } while (get_fs_byte(tmp++));
                if (p-len < 0) {                /* this shouldn't happen - 128kB */
                        set_fs(old_fs);
                        return 0;
                }
// Then we reversely copy the string char by char to the end of the parameter and environment
// space. In the process of looping through the characters of a string, we first need to determine
// whether there is a memory page at the corresponding position in the parameter and environment
// space. If it does not exist, first apply for a page of memory. The 'offset' is used as the
// current pointer offset value in a page. Since the offset variable 'offset' is initialized
// to 0 at the beginning of this function, the following check for '(offset-1 < 0)' is definitely
// true, so that 'offset' is re-set to the current p pointer within the page range.
                while (len) {
                        --p; --tmp; --len;
                        if (--offset < 0) {
                                offset = p % PAGE_SIZE;
                                if (from_kmem==2)          // If string in kernel space,
                                        set_fs(old_fs);    // FS points back to user space.
// If the string space page pointer array item (page[p/PAGE_SIZE]) is zero, it means that the
// space memory page where the p pointer is located does not exist yet, then you need to apply
// for a free memory page and fill the page pointer into the array. At the same time, we also
// make the page pointer pag point to the new page. Returns 0 if no free page is available.
                                if (!(pag = (char *) page[p/PAGE_SIZE]) &&
                                    !(pag = (char *) page[p/PAGE_SIZE] =
                                      (unsigned long *) get_free_page()))
                                        return 0;
                                if (from_kmem==2)            // If string in kernel space,
                                        set_fs(new_fs);      // FS points to kernel space.

                        }
// Then copy one byte of the string from the FS segment to the offset of the parameter and
// environment space in the memory page 'pag'.
                        *(pag + offset) = get_fs_byte(tmp);
                }
        }
// Finally, if the string and string array are in kernel space, the original value of the FS
// segment register is restored, and then the header offset of the copied parameter in parameter
// and environment space is returned.
        if (from_kmem==2)
                set_fs(old_fs);
        return p;
}

//// Modify the local descriptor table of the task.
// Modify the segment base address and length limit of the descriptor in the local descriptor
// table LDT, and place the parameter and environment space page at the end of the data segment.
// Parameters: text_size - the length limit of the code segment given by the a_text field in
// the file header; page - an array of parameters and environment space page pointers.
// Returns: the data segment limit value (64MB).
static unsigned long change_ldt(unsigned long text_size,unsigned long * page)
{
        unsigned long code_limit,data_limit,code_base,data_base;
```

```
180         int i;
181
    // First set the code and data segment length limit to 64MB, and then take the code segment
    // base address in the code segment descriptor in the LDT of the current process. The code segment
    // base address is the same as the data segment base address. These new values are then used
    // to re-set the base and segment lengths in the code segment and data segment descriptors in
    // the LDT. Please note here that since the code and data segment base address of the new program
    // being loaded is the same as the original program, there is no need to repeat them. That is,
    // the two statements on the lines 186, 188 that set the base address of the segment are redundant
    // and can be omitted.
182         code_limit = TASK_SIZE;
183         data_limit = TASK_SIZE;
184         code_base = get_base(current->ldt[1]);       // include/linux/sched.h, line 277
185         data_base = code_base;
186         set_base(current->ldt[1],code_base);
187         set_limit(current->ldt[1],code_limit);
188         set_base(current->ldt[2],data_base);
189         set_limit(current->ldt[2],data_limit);
190 /* make sure fs points to the NEW data segment */
    // The selector of the local table data segment descriptor (0x17) is put in the FS register,
    // that is, by default, the FS points to the task data segment. Then put pages with data stored
    // in parameter and environment space (up to MAX_ARG_PAGES page, 128kB) at the end of the data
    // segment. The method is: from the beginning of AA, put it backwards page by page. The method
    // is: From the beginning of the library code location of the process space, put backwards into
    // the segment page by page. The library file code occupies the end of the process space and
    // is a multiple of 4MB. The function put_dirty_page() is used to map physical pages into the
    // process logic space. See the mm/memory.c file.
191         __asm__("pushl $0x17\n\tpop %%fs"::);
192         data_base += data_limit - LIBRARY_SIZE;
193         for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
194                 data_base -= PAGE_SIZE;
195                 if (page[i])                        // put the page if exists.
196                         put_dirty_page(page[i],data_base);
197         }
198         return data_limit;              // return the length limit of data segment(64MB).
199 }
200
201 /*
202  * 'do_execve()' executes a new program.
203  *
204  * NOTE! We leave 4MB free at the top of the data-area for a loadable
205  * library.
206  */
    //// Load and execute other programs. Called in execve() system-call.
    // This is a function called by the system call interrupt (int 0x80), function number __NR_execve.
    // The argument of the function is the value that is gradually pushed onto the stack after
    // the system-call is executed until the function is called (kernel/system_call.s, line 217).
    // These values (in system_call.s file) include:
    // (1) The edx, ecx, and ebx register values pushed onto the stack on line 89--91 correspond
    // to **envp, **argv, and *filename, respectively; (2) The return address (tmp) of the function
    // pushed onto the stack when the sys_execve function in sys_call_table is called on line 99;
    // (3) On line 216, the program code pointer eip that calls the system interrupt that is pushed
    // onto the stack before calling this function do_execve().
    // Parameters: eip - the program code pointer to call the system interrupt;
    // tmp - the return address of the system interrupt when calling _sys_execve, not used;
    // filename - the pointer to the executable file name;
    // argv - a pointer to an array of command line argument pointers;
    // envp - a pointer to an array of environment variable pointers.
    // Returns: if successful, it does not return; otherwise sets the error code and returns -1.
207 int do_execve(unsigned long * eip,long tmp,char * filename,
208         char ** argv, char ** envp)
209 {
    // The meaning of some of the following local variables is as follows:
    // Line 213 - An array of arguments & environmental string (A&E) space page pointers.
    // Line 217 - A flag to control if we will execute a shell script file.
    // Line 218 - A integer used to point to the tail of arguments & environmental space.
210         struct m_inode * inode;
211         struct buffer_head * bh;
212         struct exec ex;
213         unsigned long page[MAX_ARG_PAGES];        // array of page pointers in A&E space.
214         int i,argc,envc;
215         int e_uid, e_gid;                         // effective user and group ID.
216         int retval;
217         int sh_bang = 0;                          // control if we execute script file.
218         unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4; // p points to the end of the A&E space.
219
    // Before we officially set up the environment for the execution files, let's do some preparatory
    // work. The kernel prepares 128KB (32 pages) of space to store command line arguments and
    // environment string variables for the executable file. The previous line sets p to be initially
    // placed at the last long word of the 128KB space. During the initialization parameter and
    // environment space operations, p will be used to indicate the current location in 128KB space.
    // In addition, the parameter eip[1] is the original user program code segment register CS value
    // that calls this system call, and the segment selector must of course be the code segment
    // selector (0x000f) of the current task. If it is not the value, CS can only be the selector
    // 0x0008 of the kernel code segment. But this is absolutely not allowed, because the kernel
    // code is resident and cannot be replaced. Therefore, the following is based on the value of
    // eip[1] to confirm whether it is normal. Then we initialize the 128KB arguments & environment
    // string space, clear all bytes, and get the i-node of the execution file. Then according to
    // the function parameters, calculate the number of command line parameters argc and the number
    // of environment strings envc. In addition, the executable file must be a regular file.
220         if ((0xffff & eip[1]) != 0x000f)
221                 panic("execve called from supervisor mode");
222         for (i=0 ; i<MAX_ARG_PAGES ; i++)       /* clear page-table */
223                 page[i]=0;
224         if (!(inode=namei(filename)))           /* get executables inode */
225                 return -ENOENT;
226         argc = count(argv);                     // The number of command line arguments.
227         envc = count(envp);                     // The number of environment variables.
228
229 restart_interp:
230         if (!S_ISREG(inode->i_mode)) {          /* must be regular file */
231                 retval = -EACCES;
232                 goto exec_error2;               // if not regular file, jump to line 376.
233         }
    // Then we check whether the current process has the right to run the specified executable file,
    // that is, according to the mode in the execution file i-node, to see if the process has the
    // permission to execute it. We first check to see if the "set-user_id" flag and the
    // "set-group-id" flag are set in the mode. These two flags are mainly used to enable general
    // users to execute programs of privileged users (such as super user root), such as passwd,
    // which changes the password. If the set-user-id flag is set, the euid of the subsequent
    // execution process is set to the user ID of the execution file, otherwise it is set to the
    // euid of the current process. If the execution file set-group-id is set, the effective group
    // ID (egid) of the execution process is set to the group ID of the execution file, otherwise
    // it is set to the egid of the current process. Here, the two determined values are temporarily
    // stored in the variables e_uid and e_gid.
234         i = inode->i_mode;
235         e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
236         e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;

    // Now compare the euid and egid of the process with the access mode of the executable file.
    // If the execution file belongs to the user running the process, the file mode value is shifted
    // to the right by 6 bits, and the lowest 3 bits are the access permission flags of the file
    // owner. Otherwise, if the execution file belongs to the same group as the user of the current
    // process, the attribute is shifted to the right by 3 bits, so that the lowest 3 bits are the
    // access permission flags of the execution file group. Otherwise, the lowest 3 bits of the
    // mode at this time are the permissions of other users to access the executable file.
    // Then we determine whether the current process has permission to run the executable file based
    // on the lowest 3-bit value. If the selected user does not have the right to run the file (bit
    // 0 is the execution permission), and the other users do not have any rights, or the current
    // process user is not the super user, it indicates that the current process does not have the
    // authority to run the executable file. Then set the unexecutable error code, and jump to
    // exec_error2 to do the exit processing.
237         if (current->euid == inode->i_uid)
238                 i >>= 6;
239         else if (in_group_p(inode->i_gid))
240                 i >>= 3;
241         if (!(i & 1) &&
242             !((inode->i_mode & 0111) && suser())) {
243                 retval = -ENOEXEC;
244                 goto exec_error2;
245         }
    // If the code can be executed here, this means that the current process has permission to run
    // the specified executable file. So from here we need to extract the data from the execution
    // file header and analyze and set the runtime environment based on the information, or run
    // another shell program to execute this script file. First, the first block of the execution
    // file is read into the buffer block, and the buffer block data is copied into the ex structure.
    // If the first two bytes of the execution file are the characters '#!', it is a script file.
    // If we want to run a script file, we need to execute an interpreter for it (such as a shell
    // program). Usually the first line of the script file is "#!/bin/bash", which specifies the
    // interpreter needed to run the script file. The running method is to take the interpreter
    // name and the following parameters (if any) from the first line of the script file, and then
    // put these parameters and the script file (the executable file) name into the interpreter's
    // command line arguments space.
    // Before that, we of course need to put the original command line parameters and environment
    // strings specified by the function into the 128KB space, and the command line parameters
    // established here are placed in front of them (because they are reversed). Finally, let the
    // kernel execute the interpreter of the script file. Then, after setting the parameters such
    // as the script file name, the i-node of the interpreter is taken out and jumped to line 229
    // to execute the interpreter. Since we need to jump to the executed code line 229, we need
    // to set a flag sh_bang that prohibits the execution of the following script processing code
    // again after confirming and processing the script file. In the code that follows, this flag
    // is also used to indicate that we have set the command line arguments for the executable file,
    // and no need to repeat such settings.
            if (!(bh = bread(inode->i_dev,inode->i_zone[0]))) {
                    retval = -EACCES;
                    goto exec_error2;
            }
            ex = *((struct exec *) bh->b_data);     /* read exec-header */
            if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
                    /*
                     * This section does the #! interpretation.
                     * Sorta complicated, but hopefully it will work.  -TYT
                     */

                    char buf[128], *cp, *interp, *i_name, *i_arg;
                    unsigned long old_fs;

    // From here on, we extract the interpreter name and its parameters from the script file, and
    // put the interpreter name, its parameters, and script file name into the environment parameter
    // block. First copy the string after the character '#!' on the first line of the script file
    // into the buffer buf, which contains the script interpreter name (for example /bin/sh), and
    // possibly several parameters of the interpreter. Then process the contents of buf: replace
    // the first newline with NULL and remove the space tab.
                    strncpy(buf, bh->b_data+2, 127);
                    brelse(bh);                          // release the buffer block.
                    iput(inode);                         // put back the i-node of the script file.
                    buf[127] = '\0';
                    if (cp = strchr(buf, '\n')) {
                            *cp = '\0';                  // delete the space, tabs.
                            for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
                    }
                    if (!cp || *cp == '\0') {            // if the line contains nothing, error!
                            retval = -ENOEXEC;           /* No interpreter name found */
                            goto exec_error1;
                    }
    // At this point, we get a line of content (string) starting with the name of the script
    // interpreter. The line is analyzed below. First we get the first string, which should be the
    // interpreter name, at which point i_name points to the name. If there are characters after
    // the interpreter name, they should be the argument string, so that i_arg points to the string.
                    interp = i_name = cp;
                    i_arg = 0;
                    for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++) {
                            if (*cp == '/')
                                    i_name = cp+1;
                    }
                    if (*cp) {
                            *cp++ = '\0';            // add a NULL to the end of interpreter name.
                            i_arg = cp;              // i_arg points to the arguments.
                    }
                    /*
                     * OK, we've parsed out the interpreter name and
                     * (optional) argument.
```

```
285                  */
    // Now we need to put the parsed interpreter name i_name, its parameter i_arg and script file
    // name as parameters into the interpreter's environment and parameter block. But first we need
    // to put some of the original argument and environment strings provided by the function first,
    // and then put the contents of the first line parsed here. For example, if the command line
    // is "example.sh -arg1 -arg2", that is, the execution file is a script file. If the first line
    // of content is "#!/bin/bash -iarg1 -iarg2", then after placing the parameters in the first
    // line here, the new command line looks like this:
    //         "bash -iarg1 -iarg2 example.sh -arg1 -arg2"
    // Here we set the sh_bang flag, and then put the original parameters and environment strings
    // provided by the function parameters into the space. The number of environment strings and
    // arguments are envc and argc-1 respectively. One of the original arguments that are not copied
    // is the original executable file name, which is the name of the script file here, and will
    // be handled below.
    // Please pay attention here! The pointer p gradually moves toward the small address as the
    // copy information increases, so after the two copy string functions are executed, the
    // environment string block is located above the program command line argument string block,
    // and p points to the first of the program argument string. In addition, the last parameter
    // (0) of copy_strings() indicates that the parameter string is in user space.
286                 if (sh_bang++ == 0) {
287                         p = copy_strings(envc, envp, page, p, 0);
288                         p = copy_strings(--argc, argv+1, page, p, 0);
289                 }
290                 /*
291                  * Splice in (1) the interpreter's name for argv[0]
292                  *           (2) (optional) argument to interpreter
293                  *           (3) filename of shell script
294                  *
295                  * This is done in reverse order, because of how the
296                  * user environment and arguments are stored.
297                  */
    // Next we reverse copy the script file name, interpreter parameters, and interpreter file names
    // into the arguments and environment space. If an error occurs, the error code is set and jump
    // to exec_error1. In addition, since the script file name provided by this function parameter
    // is in user space, the pointer to the script file name given to copy_strings() is in kernel
    // space, so the last parameter of this copy string function (string source flag) needs to be
    // Set to 1. If the string is in kernel space, the last parameter of copy_strings() needs to
    // be set to 2, as shown in lines 301 and 304 below.
298                 p = copy_strings(1, &filename, page, p, 1);
299                 argc++;
300                 if (i_arg) {                  // copy multiple arguments of the interpreter.
301                         p = copy_strings(1, &i_arg, page, p, 2);
302                         argc++;
303                 }
304                 p = copy_strings(1, &i_name, page, p, 2);
305                 argc++;
306                 if (!p) {
307                         retval = -ENOMEM;
308                         goto exec_error1;
309                 }
310                 /*
311                  * OK, now restart the process with the interpreter's inode.
312                  */
```

```
    // Finally, we get the i-node pointer of the interpreter and then jump to line 229 to execute
    // the interpreter. In order to get the i-node of the interpreter, we need to use the namei()
    // function, but the parameters (filename) used by the function are obtained from the user data
    // space, that is, from the space pointed to by the segment register FS. So before calling the
    // namei() function, we need to temporarily let FS point to the kernel data space so that the
    // function can get the name of the interpreter from kernel space and restore the default settings
    // of FS after namei() returns. Then jump to restart_interp (line 229) to reprocess the new
    // executable file -- the script file's interpreter.
313                     old_fs = get_fs();
314                     set_fs(get_ds());
315                     if (!(inode=namei(interp))) {        /* get executables inode */
316                             set_fs(old_fs);
317                             retval = -ENOENT;
318                             goto exec_error1;
319                     }
320                     set_fs(old_fs);
321                     goto restart_interp;
322             }

    // At this point, the execution file header structure data in the buffer block has been copied
    // to ex. So we release the buffer block first and start checking the execution header information
    // in ex. For this kernel, it only supports the ZMAGIC executable file format, and the execution
    // file code is executed from logical address 0, so execution files containing code or data
    // relocation information are not supported. Of course, if the executable file is too large
    // or the executable file is incomplete, then we can't run it. Therefore, the program will not
    // be executed for the following situations: The executable file is not an executable file
    // (ZMAGIC), or the code and data relocation portion is not equal to 0, or (code section + data
    // section + heap) is longer than 50MB, or the execution file size is less than (code section
    // + data section + symbol table + execution header).
323             brelse(bh);
324             if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
325                     ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
326                     inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
327                     retval = -ENOEXEC;
328                     goto exec_error2;
329             }
    // In addition, if the code at the beginning of the execution file is not located at the boundary
    // of a page (1024 bytes), it cannot be executed. Because the demand paging technique requires
    // that the contents of the executable file be loaded in page units, the code and data in the
    // executable file image are required to start at the page boundary.
330             if (N_TXTOFF(ex) != BLOCK_SIZE) {
331                     printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
332                     retval = -ENOEXEC;
333                     goto exec_error2;
334             }
    // If the sh_bang flag is not set, copy the specified number of command line arguments and
    // environment strings into the parameter and environment space. If the sh_bang flag is already
    // set, it means that the script interpreter will be run, and the environment variable page
    // has been copied, no need to copy again. Similarly, if sh_bang is not set and needs to be
    // copied, then the pointer p gradually moves toward the small address as the copy information
    // increases. Therefore, after the two copy string functions are executed, the environment string
    // block is located above the program argument string block, and p points to the first argument
    // string of the program. In fact, p is the offset value in the 128KB parameter and environment
    // space, So if p=0, it means that the environment variable and the parameter space page are
    // already full.
            if (!sh_bang) {
                    p = copy_strings(envc,envp,page,p,0);
                    p = copy_strings(argc,argv,page,p,0);
                    if (!p) {
                            retval = -ENOMEM;
                            goto exec_error2;
                    }
            }
    /* OK, This is the point of no return */
    /* note that current->library stays unchanged by an exec */
    // In the previous section, we set the command line arguments and environment space for running
    // the execution file according to the information provided by the function parameters, but
    // we have not done any substantive work for it, that is, we have not done the initialization
    // of process task structure, creating the page table, and etc. Now let's do the job. Since
    // the execution file directly uses the "body" of the current process, that is, the current
    // process will be transformed into a process that executes the file, we need to first release
    // some system resources occupied by the current process, including closing the specified open
    // file and releasing occupied page table and memory pages, etc. Then, according to the execution
    // file header structure, the content of the descriptor in the local descriptor table LDT used
    // by the current process is modified, the length limit of the code segment and the data segment
    // descriptor is re-set, and the e_uid and e_gid obtained are used to set the related fields
    // in the process task. Finally, the return address eip[] of the program that executes this
    // system call is pointed to the beginning of the code in the execution file. This way, when
    // the system-call exits and returns, it will run the code of the new executable file.
    // Note that although the new executable file code and data have not yet been loaded into memory
    // from the file, its parameters and environment blocks have used get_free_page() in
    // copy_strings() to get the physical memory page to hold the data, and used put_page() in
    // change_ldt() to store the data at the end of the process logic space. In addition, in
    // create_tables(), a page fault exception is also caused by storing arguments and environment
    // pointer tables on the user stack, so that the memory manager also maps physical memory pages
    // for the user stack space.
    //
    // Here we first put back the i-node of the process's original execution program, and let the
    // process executable field point to the i-node of the new executable file. Then reset all signal
    // processing handles of the original process, but there is no need to reset for the SIG_IGN
    // handle. Then according to the close_on_exec bitmap flag, close the specified open file and
    // reset the flag.
            if (current->executable)
                    iput(current->executable);
            current->executable = inode;
            current->signal = 0;
            for (i=0 ; i<32 ; i++) {
                    current->sigaction[i].sa_mask = 0;
                    current->sigaction[i].sa_flags = 0;
                    if (current->sigaction[i].sa_handler != SIG_IGN)
                            current->sigaction[i].sa_handler = NULL;
            }
            for (i=0 ; i<NR_OPEN ; i++)
                    if ((current->close_on_exec>>i)&1)
                            sys_close(i);
            current->close_on_exec = 0;
```

```
// Then, according to the base address and the length limit specified by the current process,
// the physical memory page and the page table itself specified by the page table corresponding
// to the code and data segment of the original program are released. At this moment, the new
// execution file does not occupy any pages in the main memory area, so the page fault abort
// will occur when the processor actually runs the new execution file code. The memory management
// program then performs page fault processing to apply for a memory page and set related page
// table entries for the new execution file, and reads the relevant execution file page into
// the memory. Also, if the "last task used coprocessor" points to the current process, it is
// set to be NULL and the used math flag is reset.
        free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
        free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
        if (last_task_used_math == current)
                last_task_used_math = NULL;
        current->used_math = 0;
// Then we modify the descriptor base address and the segment length in the LDT according to
// the code length field a_text in the new execution header structure, and place the 128KB
// arguments and environment space page at the end of the data segment. After executing the
// following statement, p is now changed to be the offset starting at the beginning of the data
// segment, but still points to the beginning of the data in the arguments and environment space,
// that is, p has been converted to be the stack pointer value. Then call the internal function
// create_tables() to create an environment and parameter variable pointer table in the stack
// space for the program's main() as parameters and return the stack pointer.
        p += change_ldt(ex.a_text,page);
        p -= LIBRARY_SIZE + MAX_ARG_PAGES*PAGE_SIZE;
        p = (unsigned long) create_tables((char *)p,argc,envc);

// Then modify the process field values to become the information of the new executable file.
// That is, the process task structure code tail field end_code is equal to the code segment
// length a_text of the execution file; the data tail field end_data is equal to the code segment
// length of the execution file plus the data segment length (a_data + a_text); and the process
// heap end field brk = a_text + a_data + a_bss. Brk is used to indicate the end position of
// the current data segment (including the uninitialized portion) of the process, and the kernel
// specifies the starting position of the allocation when allocating memory for the process.
// Then set the process stack start field to the page where the stack pointer is located, and
// re-set the effective user id and effective group id of the process.
        current->brk = ex.a_bss +
                (current->end_data = ex.a_data +
                (current->end_code = ex.a_text));
        current->start_stack = p & 0xfffff000;
        current->suid = current->euid = e_uid;
        current->sgid = current->egid = e_gid;
// Eventually, the program pointer that originally invoked the system-call interrupt is replaced
// with the code pointer on the stack to point to the entry point of the new executable, and
// the stack pointer is replaced with the stack pointer of the new execution file. Thereafter,
// the return instruction will pop up the data on the stack and cause the CPU to execute the
// new execution file, so it will not return to the program that originally called the system
// interrupt.
        eip[0] = ex.a_entry;            /* eip, magic happens :-) */
        eip[3] = p;                     /* stack pointer */
        return 0;
exec_error2:
        iput(inode);                     // put back the i-node.
exec_error1:
        for (i=0 ; i<MAX_ARG_PAGES ; i++)
                free_page(page[i]);             // release memory pages if error occurs.
        return(retval);                         // return error code.
}

```

12.16 stat.c

12.16.1 Function

The stat.c program implements the system-call functions stat() and fstat() for obtaining file status information, and stores the obtained information in the user buffer. The file status information is stored in the stat structure as shown below, where all field information can be obtained from the i-node of the file. Stat() uses the file name to get the information, and fstat() uses the file handle (descriptor) to get the information. Please refer to the file include/sys/stat.h.

```
// File status structure. All fields are available from the file's i-node structure.
struct stat {
        dev_t   st_dev;         // device number that contains the file.
        ino_t   st_ino;         // file i-node number.
        umode_t st_mode;        // file type and mode.
        nlink_t st_nlink;       // number of links to the file.
        uid_t   st_uid;         // user identification number of the file.
        gid_t   st_gid;         // group number of the file.
        dev_t   st_rdev;        // device number (if the file is a char or block device file).
        off_t   st_size;        // file size (bytes) if the file is a regular file.
        time_t  st_atime;       // last access time.
        time_t  st_mtime;       // last modified time.
        time_t  st_ctime;       // i-node last changed time.
};
```

12.16.2 Code annotation

Program 12-15 linux/fs/stat.c

```
/*
 *  linux/fs/stat.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <errno.h> Error number header file. Contains various error numbers in the system.
// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//     and constants.
// <linux/fs.h> File system header file. Define the file table structure (file, buffer_head,
//     m_inode, etc.).
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//      of the initial task 0, and some embedded assembly function macro statements about the
//      descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//      used functions of the kernel.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//      segment register operations.
#include <errno.h>
#include <sys/stat.h>

#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>

//// Use the i-node to get file status information.
// The parameter inode is the file i-node, and the statbuf is the stat file state structure
// pointer in the user data space, which is used to store the obtained state information.
static void cp_stat(struct m_inode * inode, struct stat * statbuf)
{
        struct stat tmp;
        int i;

// First verify (or allocate) enough memory space to store the data, and then temporarily copy
// the information on the corresponding file i-node to tmp. Finally, these status information
// is copied into the user buffer.
        verify_area(statbuf,sizeof (struct stat));
        tmp.st_dev = inode->i_dev;          // device number that contains the file.
        tmp.st_ino = inode->i_num;          // file i-node number.
        tmp.st_mode = inode->i_mode;        // file type and mode.
        tmp.st_nlink = inode->i_nlinks;     // number of links to the file.
        tmp.st_uid = inode->i_uid;          // user id.
        tmp.st_gid = inode->i_gid;          // group id.
        tmp.st_rdev = inode->i_zone[0];     // device no.(if the file is a char/block file).
        tmp.st_size = inode->i_size;        // file size (bytes).
        tmp.st_atime = inode->i_atime;      // last access time.
        tmp.st_mtime = inode->i_mtime;      // last modified time.
        tmp.st_ctime = inode->i_ctime;      // i-node last changed time.
        for (i=0 ; i<sizeof (tmp) ; i++)
                put_fs_byte(((char *) &tmp)[i],i + (char *) statbuf);
}

//// File status system-call function.
// Get related file status information according to the given file name.
// The parameter statbuf is the user buffer pointer for storing state information.
// Returns 0 if successful, and returns an error code if an error occurs.
int sys_stat(char * filename, struct stat * statbuf)
{
        struct m_inode * inode;

// First, find the corresponding i-node according to the file name, then copy the file state
// information on the i-node into the user buffer, and finally put back the i-node.
        if (!(inode=namei(filename)))
                return -ENOENT;
        cp_stat(inode,statbuf);
        iput(inode);
        return 0;
}

//// Symbolic link file status system-call function.
// Get related file status information according to the given file name. If there is a symbolic
// link file name in the file path name, the status information of the symbol file itself is
// taken, and the link is not followed.
// The parameter statbuf is the user buffer pointer for storing file status information.
// Returns 0 if successful, and returns an error code if an error occurs.
int sys_lstat(char * filename, struct stat * statbuf)
{
        struct m_inode * inode;

// First find the corresponding i-node according to the file name. If it is a symbolic link
// file, it does not follow the link. The file status information on the i-node is then copied
// into the user buffer, and the i-node is put back.
        if (!(inode = lnamei(filename)))     // get the i-node without following the link.
                return -ENOENT;
        cp_stat(inode,statbuf);
        iput(inode);
        return 0;
}

//// Use the file handle to get the file state.
// Get status information about the file based on the given file handle.
// The parameter fd is the handle (descriptor) of the specified file, and the statbuf is the
// user buffer pointer that holds the status information.
// Returns 0 if successful, and returns an error code if an error occurs.
int sys_fstat(unsigned int fd, struct stat * statbuf)
{
        struct file * f;
        struct m_inode * inode;

// First take the file structure corresponding to the file handle, and then get the i-node of
// the file. The file status information on the i-node is then copied into the user buffer.
// If the file handle value is greater than the maximum number of files that can be opened by
// a program NR_OPEN, or the file structure pointer of the handle is NULL, or the i-node field
// of the corresponding file structure is NULL, an error occurs and an error code is returned.
        if (fd >= NR_OPEN || !(f=current->filp[fd]) || !(inode=f->f_inode))
                return -EBADF;
        cp_stat(inode,statbuf);
        return 0;
}

//// Read symbolic link file system-call.
// This function reads the contents of the symbolic link file (that is, the pathname string
// of the file pointed to by the symbolic link) and places it in the user buffer. If the buffer
// is too small, the contents of the symbolic link will be truncated.
// Parameters: path is the symbolic link file path name; buf is the user buffer; bufsiz is the
// buffer length.
// Returns: the number of characters in the buffer if successful, or error code if it fails.
```

```
int sys_readlink(const char * path, char * buf, int bufsiz)
{
        struct m_inode * inode;
        struct buffer_head * bh;
        int i;
        char c;

// First check and verify the validity of the function parameters and adjust them. The user
// buffer size bufsi must be between 1 and 1023. Then get the i-node of the symbolic link file
// and read the first block of data content of the file. Then put back the i-node.
        if (bufsiz <= 0)
                return -EBADF;
        if (bufsiz > 1023)
                bufsiz = 1023;
        verify_area(buf,bufsiz);
        if (!(inode = lnamei(path)))
                return -ENOENT;
        if (inode->i_zone[0])
                bh = bread(inode->i_dev, inode->i_zone[0]);
        else
                bh = NULL;
        iput(inode);
// If the file data content is successfully read, the most bufsiz characters are copied from
// the file into the user buffer , and NULL characters are not copied. Finally release the buffer
// block and return the number of bytes copied.
        if (!bh)
                return 0;
        i = 0;
        while (i<bufsiz && (c = bh->b_data[i])) {
                i++;
                put_fs_byte(c,buf++);
        }
        brelse(bh);
        return i;
}

```

12.17 fcntl.c

12.17.1 Function

The fcntl.c program implements the file control system call fcntl(), and the two file handle (descriptor) duplication system-calls dup() and dup2().Dup2() specifies the minimum value of the new handle, while dup() returns the unused handle with the smallest current value. Fcntl() is used to modify the state of an opened file, or to copy a file handle. Handle duplication operations are primarily used for standard input/output redirection and pipe operations of files. Some of the constant symbols used in this program are defined in the include/fcntl.h file. It is recommended that you also refer to this header file when reading this program.

The new file handle returned by the functions dup() and dup2() will use the same file table entry as the original handle being copied or duplicated. For example, when a process does not open any other file, if you use the dup() or dup2() function but specify a new handle of 3, the file handles after the function is executed are shown in Figure 12-35.

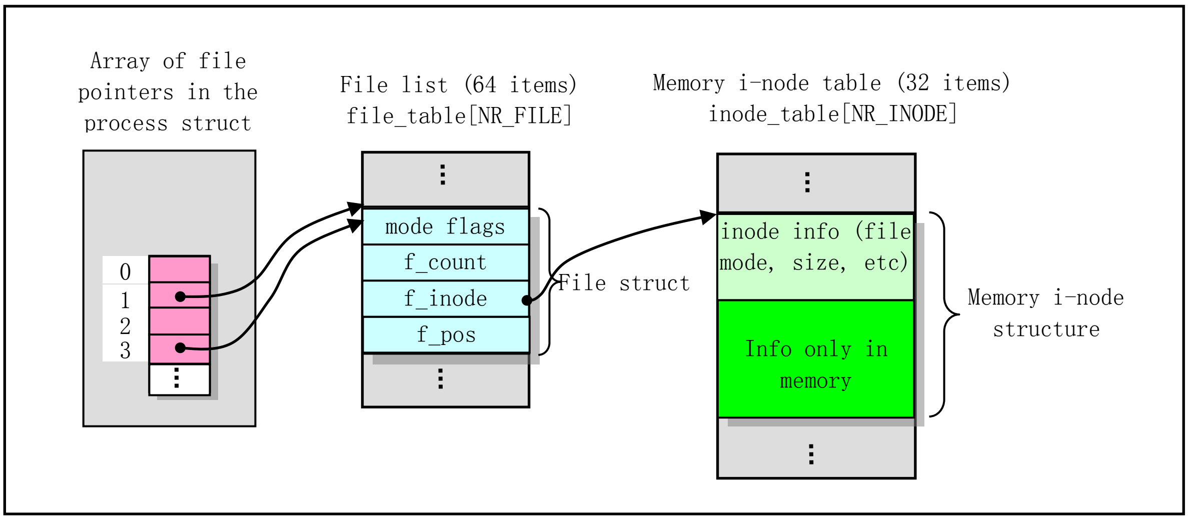

Array of file pointers in the process struct
File list (64 items) file_table[NR_FILE]
Memory i-node table (32 items) inode_table[NR_INODE]
0
1
2
3
mode flags
f_count
f_inode
f_pos
File struct
inode info (file mode, size, etc)
Info only in memory
Memory i-node structure

Figure 12-35 File structures after executing dup(1) or dup2(1,3) function

In addition, it can be seen from the internal function dupfd() in this program that for the file handle newly created by dup() or dup2() function, the flag close_on_exec will be cleared, that is, when the exec() class function is run, it will not close the file handle created with dup().

The fcntl() function, which was adopted by AT&T's System III, is mainly used to modify the properties of an open file. It integrates four functionalities in the function with the control command cmd in the parameter:

- cmd = F_DUPFD, duplicate the file handle. At this time, the return value of fcntl() is a new file handle whose value is greater than or equal to the value specified by the third parameter. The newly created file handle will use the same file entry as the original handle, but its execution close flag bit will be reset. For this command, the function dup(fd) is equivalent to fcntl(fd, F_DUPFD, 0); and the function dup2(fd, newfd) is equivalent to the statements "close(newfd); fcntl(fd, F_DUPFD, newfd); ”
- cmd = F_GETFD or F_SETFD。These two commands are used to read or set the corresponding bits in the flag close_on_exec of the file handle. When setting is flag, the third argument of the function is the new bit value of the flag.
- cmd = F_GETFL or F_SETFL. These two commands are used to read or set file operations and access flags, respectively. These flags are RDONLY, O_WRONLY, O_RDWR, O_APPEND, and O_NONBLOCK. For details, see the include/fcntl.h file. When setting the operation, the third parameter of the function is the new value of the file operation and access flags, and only the O_APPEND and O_NONBLOCK flags can be changed.
- cmd = F_GETLK, F_SETLK or F_SETLKW. These commands are used to read or set the file lock flag, but the file record lock function is not implemented in the Linux 0.12 kernel.

12.17.2 Code annotation

Program 12-16 linux/fs/fcntl.c

```
/*
```

```
 *  linux/fs/fcntl.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <string.h> String header file. Defines some embedded functions about string operations.
// <errno.h> Error number header file. Contains various error numbers in the system.
// <sys/types.h> Type header file. The basic system data types are defined.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
//     segment register operations.
// <fcntl.h> File control header file. The definition of the operation control constant symbol
//     used for the file and its descriptors.
// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//     and constants.
#include <string.h>
#include <errno.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/segment.h>

#include <fcntl.h>
#include <sys/stat.h>

extern int sys_close(int fd);          // close file system-call. (fs/open.c, 219)

//// Duplicate the file handle (file descriptor).
// The parameter fd is the file handle to be duplicated, and arg specifies the minimum number
// of the new file handle.
// Returns a new file handle or an error code.
static int dupfd(unsigned int fd, unsigned int arg)
{
// The function first checks the validity of the function parameters. If the file handle value
// is greater than the maximum number of open files NR_OPEN for a program, or if the file structure
// of the handle does not exist, an error code is returned; If the specified new handle value
// arg is greater than the maximum number of open files, an error code is also returned. Note
// that the file handle is actually the file structure pointer array item index number.
        if (fd >= NR_OPEN || !current->filp[fd])
                return -EBADF;
        if (arg >= NR_OPEN)
                return -EINVAL;
// Then we look in the process's file structure pointer array for an entry with an index number
// equal to or greater than arg and not yet used. If the new handle found is greater than the
// maximum number of open files NR_OPEN (ie, there are no free items), an error code is returned.
        while (arg < NR_OPEN)
                if (current->filp[arg])
                        arg++;
                else
                        break;
        if (arg >= NR_OPEN)
                return -EMFILE;
// Otherwise, for the found idle item (handle), the corresponding bit of the handle is reset
// in the flag close_on_exec. That is, when you run the exec() class function, the handle created
// with dup() will not be closed. The file structure pointer is then made equal to the pointer
// to the original handle fd, and the file reference count is incremented by one. Finally, the
// new file handle arg is returned.
        current->close_on_exec &= ~(1<<arg);
        (current->filp[arg] = current->filp[fd])->f_count++;
        return arg;
}

//// Duplicate the file handle system-call2.
// Duplicate the given file handle oldfd, the new file handle value is equal to newfd. If newfd
// is already open, close it first.
// Parameters: oldfd -- the original file handle; newfd - the new file handle.
// Returns the new file handle.
int sys_dup2(unsigned int oldfd, unsigned int newfd)
{
        sys_close(newfd);               // closed the newfd first if already opened.
        return dupfd(oldfd,newfd);      // duplicate and return the new handle.
}

//// Duplicate the file handle system-call.
// Duplicate the given file handle oldfd, the value of the new handle will be the current smallest
// unused handle value.
// Parameters: fildes -- the file handle to be duplicated.
// Returns the new file handle.
int sys_dup(unsigned int fildes)
{
        return dupfd(fildes,0);
}

//// The file control system-call.
// The parameter fd is the file handle; cmd is the control command (see include/fcntl.h, lines
// 23-30); arg has different meanings for different commands. For the duplicate handle command
// F_DUPFD, arg is the minimum value that can be taken for a new file handle; for the set file
// operation and the access flag command F_SETFL, arg is the new file operation and access mode.
// For the file lock commands F_GETLK, F_SETLK, and F_SETLKW, arg is a pointer to the flock
// structure. However, the file locking function is not implemented in this kernel.
// Returns: If an error occurs, all operations return -1. If successful, F_DUPFD returns a new
// file handle; F_GETFD returns the flag close_on_exec of the file handle; F_GETFL returns the
// file operation and access flags.
int sys_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
        struct file * filp;

// First check the validity of the given file handle, and then we will deal with them according
// to different commands cmd. If the file handle value is greater than the maximum number of
// open files NR_OPEN for a process, or if the file structure pointer for the handle is NULL,
// an error code is returned.
        if (fd >= NR_OPEN || !(filp = current->filp[fd]))
                return -EBADF;
        switch (cmd) {
                case F_DUPFD:           // duplicate file handle.
                        return dupfd(fd,arg);
                case F_GETFD:           // get the close_on_exec flag bit for the file.
                        return (current->close_on_exec>>fd)&1;
                case F_SETFD:           // set/reset close_on_exec flag. set if arg is 1.
                        if (arg&1)
                                current->close_on_exec |= (1<<fd);
                        else
                                current->close_on_exec &= ~(1<<fd);
                        return 0;
                case F_GETFL:           // get the file status flag and access mode.
                        return filp->f_flags;
                case F_SETFL:           // set status and access mode (append, non-block).
                        filp->f_flags &= ~(O_APPEND | O_NONBLOCK);
                        filp->f_flags |= arg & (O_APPEND | O_NONBLOCK);
                        return 0;
                case F_GETLK:   case F_SETLK:   case F_SETLKW:      // not implemented.
                        return -1;
                default:
                        return -1;
        }
}

```

12.18 ioctl.c

12.18.1 Function

The ioctl.c program implements the input/output control system-call ioctl().The ioctl() function can be thought of as an interface control function for each specific device driver. This function will call the IO control function in the driver of the device file specified by the file handle, mainly calling the tty_ioctl() function of the tty character device to control the I/O of the terminal. The tty device properties can usually be set in the user program when using the termios related function defined by the POSIX.1 standard, see the last part of the include/termios.h file. Those functions (such as tcflow()) are implemented in the compiled library libc.a, and the ioctl() function in the program is still executed through the system-call.

12.18.2 Code annotation

Program 12-17 linux/fs/ioctl.c

```
/*
 *  linux/fs/ioctl.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <string.h> String header file. Defines some embedded functions about string operations.
// <errno.h> Error number header file. Contains various error numbers in the system.
// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
//     and constants.
// <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
//     of the initial task 0, and some embedded assembly function macro statements about the
//     descriptor parameter settings and acquisition.
#include <string.h>
#include <errno.h>
#include <sys/stat.h>

#include <linux/sched.h>

extern int tty_ioctl(int dev, int cmd, int arg);        // chr_drv/tty_ioctl.c, line 133.
extern int pipe_ioctl(struct m_inode *pino, int cmd, int arg);    // fs/pipe.c, line 118.

// Define the input and output control (ioctl) function pointer type.
typedef int (*ioctl_ptr)(int dev,int cmd,int arg);

// Get the number of device types in the system.
#define NRDEVS ((sizeof (ioctl_table))/(sizeof (ioctl_ptr)))

// Ioctl operation function pointer table.
static ioctl_ptr ioctl_table[]={
        NULL,           /* nodev */
        NULL,           /* /dev/mem */
        NULL,           /* /dev/fd */
        NULL,           /* /dev/hd */
        tty_ioctl,      /* /dev/ttyx */
        tty_ioctl,      /* /dev/tty */
        NULL,           /* /dev/lp */
        NULL};          /* named pipes */


//// The input and output control system-call.
// The function first determines whether the file descriptor given by the parameter is valid,
// then checks the file type according to the file mode in the i-node, and invokes the relevant
// io processing function according to the specific file type.
// Parameters: fd - file descriptor (handle); cmd - command code; arg - parameter.
// Returns: 0 if successful, otherwise returns an error code.
int sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
        struct file * filp;
        int dev,mode;

// First determine the validity of the given file handle. If the file handle exceeds the number
// of openable files, or the file structure pointer of the corresponding handle is NULL, the
// error code is returned. If the file structure corresponds to the pipe i node, then we need
// to determine whether to execute the pipe IO according to whether the process has the right
// to operate the pipe. Pipe_ioctl() is called if there is permission to execute, otherwise
// an invalid file error code is returned.
        if (fd >= NR_OPEN || !(filp = current->filp[fd]))
                return -EBADF;
        if (filp->f_inode->i_pipe)
                return (filp->f_mode&1)?pipe_ioctl(filp->f_inode,cmd,arg):-EBADF;
// For other types of files, we take the mode of the corresponding file and determine the type
// of the file accordingly. If the file is neither a character device file nor a block device
// file, an error code is returned. If it is a character or block device file, the device number
// is taken from the i-node of the file. If the device number is greater than the number of
// devices in the system, an error number is returned.
        mode=filp->f_inode->i_mode;
        if (!S_ISCHR(mode) && !S_ISBLK(mode))
                return -EINVAL;
        dev = filp->f_inode->i_zone[0];           // device no. for device type file.
        if (MAJOR(dev) >= NRDEVS)
                return -ENODEV;
// Then, according to the IO control table ioctl_table, the ioctl function pointer of the
// corresponding device is found, and the function is called. If the device does not have a
// corresponding function in the table, an error code is returned.
        if (!ioctl_table[MAJOR(dev)])
                return -ENOTTY;
        return ioctl_table[MAJOR(dev)](dev,cmd,arg);
}

```

12.19 select.c

12.19.1 Function

Linux programmers often find it necessary to use multiple file descriptors at the same time to access I/O devices where the data stream is intermittently transmitted. If we only use multiple read(), write() calls to handle this situation, then one of the calls may block and let the program wait on a file descriptor. At the same time, other file descriptors may be able to read/write, but they are not processed in time.

There are many ways to solve this problem. One way is to set up a process for each file descriptor that needs to be accessed at the same time. However, this method requires coordination of communication between these processes, so this method is more complicated. Another method is to set all file descriptors to a non-blocking form, and cyclically detect whether each file descriptor has data readable or writable in the program. However, because this loop detection method consumes a lot of processor time, this method is not recommended in multitasking operating systems. The third method is to use asynchronous I/O technology. The principle is to let the kernel use signals to notify the process when a descriptor can be accessed. Since there is only one such "notification" signal for each process, if multiple file descriptors are used, it is still necessary to set each file descriptor to a non-blocking state, and when receiving such a signal, it is necessary to to test each descriptor to determine which one is ready.

Another good way to do this is to use the select()(sys_select()) function in the selcet.c program to handle this situation. The select() function originally appeared in the BSD 4.2 operating system and can be used in operating systems that support the BSD Socket network programming. It is mainly used to handle situations where multiple file descriptors (or socket handles) need to be accessed efficiently at the same time. The main working principle of this function is to let the kernel monitor multiple file descriptors provided by the user at the same time. If the state of the file descriptor has not changed, let the calling process go to sleep state; if one of the descriptors is ready to be accessed, This function returns to the process and tells the process which descriptor or descriptors are ready.

The select() function prototype is defined on line 277 of the include/unistd.h file, as shown below:

```
int select(int width, fd_set * readfds, fd_set * writefds, fd_set * exceptfds,
           struct timeval * timeout);
```

This function uses 5 parameters. The first parameter width is the value of the largest descriptor in the three descriptor sets given later plus one. This value is actually the range of values for the kernel code to check the number of descriptors. The next three parameters are pointers to the file descriptor set type fd_set, which point to the read operation descriptor set readfds, the write operation descriptor set writefds, and the descriptor set exceptfds where the exception condition occurs. Any of these 3 pointers can be NULL, indicating that we don't care about the corresponding set. If all three pointers are NULL, then the select() function can be used as a more accurate timer (the sleep() function can only provide second-level precision).

The file descriptor set type fd_set is defined in the include/sys/types.h file and is defined as an unsigned long word. Each bit of it represents a file descriptor, and the offset position value of the bit in the long word is the value of the file descriptor, as shown in the upper half of Figure 12-36.

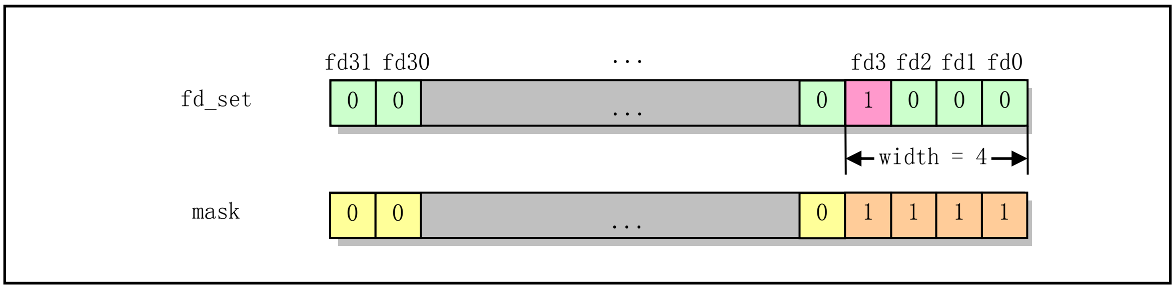


Figure 12-36 File descriptor set represents one descriptor per bit

If we are concerned with the descriptor fd3 of the read operation, then we need to set fd3 (bit 4) in the readfds set to 1; if we need to monitor the file descriptor fd1 that performs the write operation, we need to set fd1 in the writefds set to 1. If fd3 is the largest descriptor value in all descriptor sets, then the first parameter width is equal to 4. To facilitate the operation of the fd_set type variable, the Linux system provides four macros as shown below, defined in the file include/sys/time.h. They are used to clear a descriptor set, set/reset/detect a bit of a given descriptor in a descriptor set.

```
#define FD_ZERO(fdsetp)       (*(fdsetp) = 0)             // zero all bits of descriptor set.
#define FD_SET(fd,fdsetp)     (*(fdsetp) |= (1 << (fd)))  // set the bit of the descriptor.
#define FD_CLR(fd,fdsetp)     (*(fdsetp) &= ~(1 << (fd))) // reset the bit of the descriptor.
#define FD_ISSET(fd,fdsetp)   ((*(fdsetp) >> fd) & 1)     // test the bit of the descriptor.
```

After declaring a descriptor set variable, the user program should first clear it with FD_ZERO() and then use FD_SET() or FD_CLR() to set/reset the bit corresponding to the specified descriptor. FD_ISSET is used to test whether the specified bit of the descriptor set is still set when select() returns. When select() returns, the bits still set in the three descriptor sets indicate that the corresponding file descriptor is ready (read, write, or exception). Note that these macros need to use the descriptor set pointer as the second argument.

The last parameter of the select() function, timeout, is used to specify the maximum amount of time that a process will expect to wait for select() before any descriptor is ready. It is a pointer to the structure of type timeval (defined in the include/sys/time.h file), as shown below.

```
struct timeval {
        long    tv_sec;         /* seconds */
        long    tv_usec;        /* microseconds */
};
```

When the parameter timeout pointer is NULL, it means that we will wait indefinitely until one of the descriptors specified in the descriptor set is ready to operate. However, if the process receives a signal, the wait process will be interrupted, and select() will return -1, and the global variable errno will be set to EINTR.

When the timeout pointer is not NULL, but the value of both fields in the structure is 0, it means no waiting. At this point the select() function can be used to test the state of all specified descriptors and return immediately. When at least one of the two time field values is not 0, the select() function waits for a while before returning. If a descriptor is ready during the wait period, it returns directly, and at this time the two time field values are modified to indicate the remaining wait time value. If no descriptor is ready within the set time, select() returns 0. In addition, it can be interrupted by the signal during the waiting period and returns -1.

In general, when select() returns -1, it indicates an error; when select() returns a value of 0, it means that no descriptor is ready under the specified conditions; when select() returns a positive value, it indicates the number of file descriptors that are ready for access in the descriptor set. At this time, the descriptor corresponding to the bit still set in the three descriptor sets is the ready descriptor.

Since the Linux 0.12 kernel only provides system calls with up to 3 parameters, and select() has 5 parameters, when the user program calls the select() function, the select() in the library file (for example, libc.a) will pass the address of the first parameter as a pointer to the system-call sys_select() in the kernel. and the system-call will treat the pointer of the first argument as a "buffer" pointer holding all the arguments. It will first break the parameters in the "buffer" and then call the do_select() function to handle them. Then, when do_select() returns, the result is written to the user data "buffer". The following is the source code implementation of the select() function in the libc library of the Linux 0.1x system.

```
#define __LIBRARY__
#include <sys/time.h>
#include <unistd.h>

int select(int nd, fd_set * in, fd_set * out, fd_set * ex, struct timeval * tv)
{
 // First define the return result variable __res, and define the register variable __fooebx
 // as a pointer to the first parameter. Then use the system-call inline assembly code, set
 // eax = select system-call function number; ebx is the first parameter nd pointer.
     long __res;
     register long __fooebx __asm__ ("bx") = (long) &nd;
     __asm__ volatile ("int $0x80"
         : "=a" (__res)
         : "0" (__NR_select),"r" (__fooebx));
 // Finally, if the return value is greater than or equal to 0, the value is returned, otherwise
 // the global error number variable errno is set and then -1 is returned.
     if (__res >= 0)
         return (int) __res;
     errno = -__res;
     return -1;
16 }
```

In fact, the select.c program is more complicated. As Mr. Linus said in the 27th line of the program: "If you understand what I'm doing here, then you understand how the Linux sleep/wakeup mechanism works." Similar to the kernel/sched.c program, the main difficulty in this program is the understanding of the add_wait() and free_wait() functions. In order to understand the working principle of these two functions, we can refer to the sleep_on() function in the sched.c program, because these functions all involve the processing of the task waiting queue of a certain resource. Below we first explain the main working principle of the sys_select() system-call, and then detail how the select() handles the wait queues.

The code in the sys_select() function is mainly responsible for parameter copying and conversion before and after the select() function, and the main work of the select() operation is done in the do_select() function. Do_select() will first check the validity of each descriptor in the file descriptor set, then call the function check_XX() of the relevant descriptor set to check each descriptor, and also count the number of descriptors currently ready in the descriptor set. If any of the descriptors are ready, the function will return immediately, otherwise the process will invoke the add_wait() function to insert the current task into the corresponding wait queue and enter the sleep state in the do_select() function. If the process continues to run after a timeout has elapsed or because a process on the wait queue where a descriptor is located is awakened, the process will again check to see if a descriptor is ready. The do_select() function uses the free_wait() function to wake up the waiting tasks (if any) already on the queue before executing the repeat check operation.

The select.c program uses a wait table wait_table while processing the descriptor wait process, as shown in lines 37-45 in the program and Figure 12-37 below. The wait_table of type select_table contains a valid item count field nr and an array entry[NR_OPEN * 3], each array item is a wait_entry structure. The valid entry field nr of wait_table records the number of wait_entry entries waiting for the descriptor in the descriptor set to wait on the associated wait queue. The wait_entry structure contains two fields, where the wait_address pointer field is used to point to the task wait queue header corresponding to the descriptor currently being processed, and the old_task field is used to point to the wait task that the wait queue header pointer originally points to.

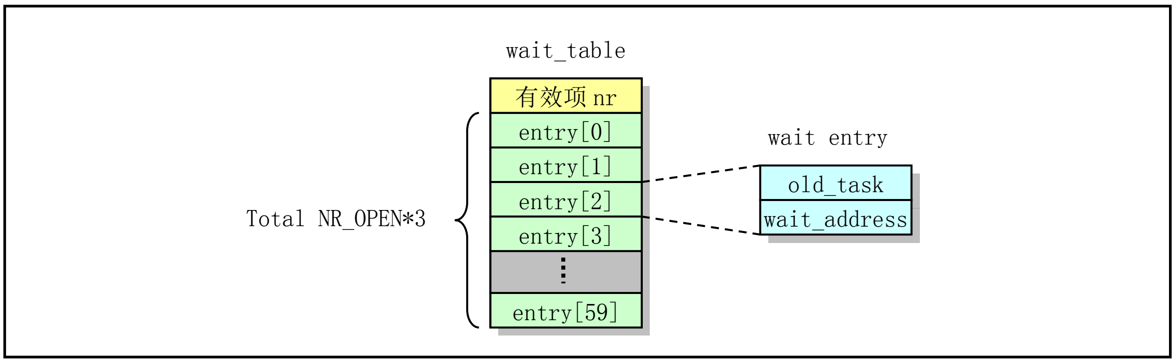


Figure 12-37 Wait table structure diagram

The wait table operates using the add_wait() and free_wait() functions. When a descriptor is not ready, add_wait() is used to add the current process to the task wait queue corresponding to the descriptor. Before adding an item to the wait list, it first searches for the wait queue header pointer field in the wait table that has the same wait item that you want to add. If it already exists, it will not be added to the wait table and will return directly (that is, only one waiting item will be inserted in the different waiting queue), otherwise the wait_address field of the wait table item points to the waiting queue head pointer, and the old_task field points to the task that the queue head pointer originally pointed to. Then let the wait queue head pointer point to the current task. Finally, the valid item count value nr of the wait table is incremented by one.

For example, for a descriptor whose read buffer queue is empty and waiting for the terminal tty to input characters, the corresponding terminal's read buffer queue 'secondary' is provided with a task wait queue header pointer proc_list that waits for a readable character in the buffer queue (see tty queue structure in file include/linux/tty.h, line 26). When no characters in the buffer secondary can be read, the select.c program will add the current task to the wait table using the add_wait() function. It will make the wait_entry field wait_address = proc_list and let the field old_task point to the task that proc_list originally pointed to. If proc_list did not originally point to any task, then old_task=NULL. Then let proc_list point to the current task. This process is shown in Figure 12-38. In the figure, (a) shows the original task waiting for the queue head pointer before calling the add_wait() function, and (b) shows the form of waiting for the entry after executing add_wait(). Note that only one wait_entry entry in the wait table is shown in the figure.

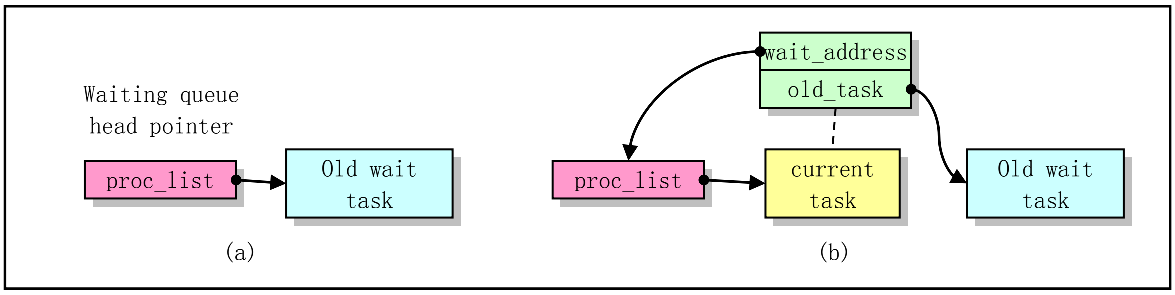


Figure 12-38 Add one wait item to the wait table

If the waiting tasks already in the waiting queue are inserted because the sleep_on() function is called, and after we insert the current task calling the select function into the waiting queue, another process inserts itself using the sleep_on() function. At this moment, the structure of the entire waiting queue is shown in Figure 12-39.

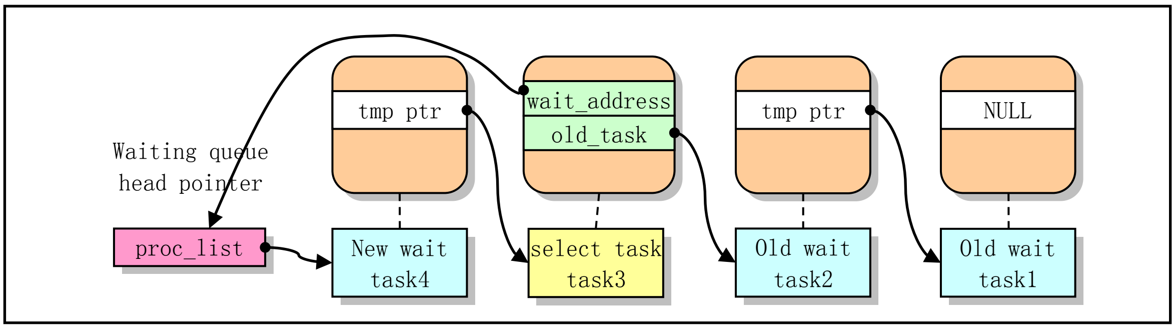


Figure 12-39 A new waiting task is inserted in the waiting queue

As can be seen from the figure, the wait table entry old_task pointer field is exactly the same as the tmp pointer in the sleep_on() function, and the wait_address field is only used for select to prevent the addition of an entry with the same wait queue pointer in the wait table wait_table. Therefore, when using the free_wait() function to clear items in the wait table, the algorithm used in free_wait() is exactly the same as in the sleep_on() function when the task is woken up.

When the waiting resource is available, for example, a character has been entered in the buffer secondary of the tty read buffer queue, the task pointed to by the head pointer in the waiting queue will be woken up. The task will then wake up the task pointed to by its tmp pointer. When the task executing select() is woken up, it immediately executes the free_wait() function (see line 204 of the code). If the task is waking up and waiting queue head pointer is pointing to this task (*wait_address == current), then the free_wait() function will immediately wake up the subsequent tasks pointed to by old_task. It can be seen that the functionality of free_wait() is exactly the same as the code of sleep_on() waking up tasks. If the task executing the select function is awakened and another process calls the sleep_on() function and sleeps on the wait queue, then the wait queue head pointer does not point to the current process (*wait_address != current) Then we need to wake up these tasks first. The operation method is to set the task pointed to by the queue head to the ready state (state = 0) and set itself to the non-interruptible wait state. That is, the task itself has to wait for these subsequent queued tasks to be awakened to start execution, then wakes up itself. Then re-execute the scheduler.

Also note that since select() implemented in the Linux kernel will modify (decrement) the field values in the structure pointed to by timeout during the run to reflect the remaining wait time, and the select() implementation in many other operating systems does not do this, so this will cause Linux programs that access the timeout structure value during select() runs into problems. Similarly, programs that do not initialize the timeout in the loop and use the select() function multiple times will encounter problems if they are ported to a Linux system. So when select() returns, the structure pointed to by timeout should be considered to be in an uninitialized state.

12.19.2 Code annotation

Program 12-18 linux/fs/select.c

```
1 /*
2  * This file contains the procedures for the handling of select
3  *
4  * Created for Linux based loosely upon Mathius Lattner's minix
5  * patches by Peter MacDonald. Heavily edited by Linus.
6  */
7
 // <linux/fs.h> File system header file. Define the file table structure (file, buffer_head,
 //     m_inode, etc.).
 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
 //     used functions of the kernel.
 // <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial
 //     communication.
 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
 //     of the initial task 0, and some embedded assembly function macro statements about the
 //     descriptor parameter settings and acquisition.
 // <asm/segment.h> Segment operation header file. An embedded assembly function is defined for
 //     segment register operations.
 // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
 //     descriptors/interrupt gates, etc. is defined.
 // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
 //     and constants.
 // <sys/types.h> Type header file. The basic system data types are defined.
 // <string.h> String header file. Defines some embedded functions about string operations.
 // <const.h> The constant file currently defines only the flags of i_mode field in the i-node.
 // <errno.h> Error number header file. Contains various error numbers in the system.
 // <sys/time.h> The timeval structure and the itimerval structure are defined.
 // <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
 //     manipulation function prototypes.
```

```
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/tty.h>
#include <linux/sched.h>

#include <asm/segment.h>
#include <asm/system.h>

#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
#include <const.h>
#include <errno.h>
#include <sys/time.h>
#include <signal.h>

/*
 * Ok, Peter made a complicated, but straightforward multiple_wait() function.
 * I have rewritten this, taking some shortcuts: This code may not be easy to
 * follow, but it should be free of race-conditions, and it's practical. If you
 * understand what I'm doing here, then you understand how the linux sleep/wakeup
 * mechanism works.
 *
 * Two very simple procedures, add_wait() and free_wait() make all the work. We
 * have to have interrupts disabled throughout the select, but that's not really
 * such a loss: sleeping automatically frees interrupts when we aren't in this
 * task.
 */
// Note that each process has its own EFLAGS flag register.

typedef struct {
        struct task_struct * old_task;
        struct task_struct ** wait_address;
} wait_entry;

typedef struct {
        int nr;
        wait_entry entry[NR_OPEN*3];
} select_table;

//// Adding the wait queue to the wait table.
// Add the wait queue pointer of the not ready descriptor to the wait_table. The parameter
// *wait_address is the wait queue header pointer associated with the descriptor. For example,
// the wait queue head pointer of the tty read buffer queue is proc_list. The parameter p is
// a pointer to the wait table structure defined in do_select().
static void add_wait(struct task_struct ** wait_address, select_table * p)
{
        int i;

// First check if the descriptor has a corresponding wait queue, and if not, return. Then in
// the wait table, search for the queue pointer given by the parameter to see if it has already
// been set in the wait table. If it was, the code will return immediately. This check is mainly
// for pipe file descriptors. For example, if a pipe is waiting to be read, it must be able
```

```
   // to write immediately.
51         if (!wait_address)
52                 return;
53         for (i = 0 ; i < p->nr ; i++)
54                 if (p->entry[i].wait_address == wait_address)
55                         return;
   // Then we store the header pointer of the descriptor's wait queue in wait_table, and let the
   // old_task field of the wait table entry points to the task pointed to by the wait queue header
   // pointer (NULL if none), then let the wait queue header points to the current task. Finally,
   // the wait table item count value nr is incremented by 1.
56         p->entry[p->nr].wait_address = wait_address;
57         p->entry[p->nr].old_task = * wait_address;
58         *wait_address = current;
59         p->nr++;
60 }
61
   //// Free each wait queue in the wait table.
   // The parameter p is a pointer to the wait table structure. This function (line 204, 207) is
   // called when the current process is woken up in the do_select() function, and is used to wake
   // up other tasks in the wait table that are on each wait queue. It is almost identical to the
   // second half of the sleep_on() function in kernel/sched.c, see the description of it.
62 static void free_wait(select_table * p)
63 {
64         int i;
65         struct task_struct ** tpp;
66
   // If the wait queue header of the entries in the wait table (total nr entry) indicates that
   // there are other wait tasks added later (for example, other processes call the sleep_on()
   // function and sleep on the wait queue), then the queue header does not point to the current
   // process, then we need to wake up these tasks first. The operation method is to set the task
   // pointed to by the queue header to the ready state (state = 0), and set itself to the
   // non-interruptible waiting state, that is, to wait for these subsequent queued tasks to be
   // awakened and execute again to wake up this task. So re-execute the scheduler.
67         for (i = 0; i < p->nr ; i++) {
68                 tpp = p->entry[i].wait_address;
69                 while (*tpp && *tpp != current) {
70                         (*tpp)->state = 0;
71                         current->state = TASK_UNINTERRUPTIBLE;
72                         schedule();
73                 }
   // Execution here, indicating that the wait queue header field wait_address in the current entry
   // of the wait table points to the current task. If it is NULL, it indicates that there is a
   // problem with the schedule code, and a warning message is displayed. Then we let the wait
   // queue header point to the task that entered the queue before us (line 76). If the head pointer
   // does point to a task instead of NULL at this time, then there is still a task in the queue
   // (*tpp is not empty), so the task is set to the ready state and wakes up. Finally, we set
   // the item count field nr of the wait list to zero and clear the wait table.
74                 if (!*tpp)
75                         printk("free_wait: NULL");
76                 if (*tpp = p->entry[i].old_task)
77                         (**tpp).state = 0;
78         }
79         p->nr = 0;
```

```
}

//// Get the tty according to the i-node.
// Check if the file is a character terminal device file according to the i-node. If it is,
// return its tty structure pointer, otherwise it returns NULL.
static struct tty_struct * get_tty(struct m_inode * inode)
{
        int major, minor;

// Returns NULL if it is not a character device file, or NULL if the major device number is
// not 5 (control terminal) or 4.
        if (!S_ISCHR(inode->i_mode))
                return NULL;
        if ((major = MAJOR(inode->i_zone[0])) != 5 && major != 4)
                return NULL;
// If the major device number is 5, then the tty field of the process is its terminal device
// number, otherwise it is equal to the minor device number of the character device number.
// If the terminal device number is less than 0, it indicates that the process has no control
// terminal, or does not use the terminal, so it returns NULL, otherwise it returns the
// corresponding tty structure pointer.
        if (major == 5)
                minor = current->tty;
        else
                minor = MINOR(inode->i_zone[0]);
        if (minor < 0)
                return NULL;
        return TTY_TABLE(minor);
}

/*
 * The check_XX functions check out a file. We know it's either
 * a pipe, a character device or a fifo (fifo's not implemented)
 */
//// Check read in ready.
// Check if the read file operation is ready, that is, whether the terminal read buffer queue
// secondary has characters to read, or whether the pipe file is not empty.
// The parameter wait is the wait table pointer; the inode is the file i-node pointer.
// Returns 1 if the descriptor can be read, otherwise returns 0.
static int check_in(select_table * wait, struct m_inode * inode)
{
        struct tty_struct * tty;

// First, according to the inode, call get_tty() to check if the file is a tty terminal (character)
// device file. If yes, check if there is any character in the secondary read buffer queue in
// the terminal, and return 1 if there is one. If secondary is empty at this time, the current
// task is added to the secondary wait queue proc_list and returns 0. If it is a pipe file,
// check if there is any character in the current pipe, if it is, return 1; if not (pipe empty),
// add the current task to the waiting queue of the pipe i-node and return 0. Note that the
// PIPE_EMPTY() macro uses the current head and tail pointer position of the pipe to determine
// if the pipe is empty. The i_zone[0] and i_zone[1] fields of the pipe i-node store the current
// head and tail pointers of the pipe.
        if (tty = get_tty(inode))
                if (!EMPTY(tty->secondary))
```

```
                        return 1;
                else
                        add_wait(&tty->secondary->proc_list, wait);
        else if (inode->i_pipe)
                if (!PIPE_EMPTY(*inode))
                        return 1;
                else
                        add_wait(&inode->i_wait, wait);
        return 0;
}

//// Check write out ready.
// Check whether the file write operation is ready, that is, whether there is any free location
// in the terminal write buffer queue write_q, or whether the pipe file is not full.
// The parameter wait is the wait table pointer; the inode is the file i-node pointer.
// Returns 1 if the descriptor can be written, otherwise returns 0.
static int check_out(select_table * wait, struct m_inode * inode)
{
        struct tty_struct * tty;

// First, according to the i-node, call get_tty() to check if the file is a tty terminal
// (character) device file. If yes, check if there is space in the write buffer queue write_q
// to write. If there is, return 1 if there is no empty space, then add the current task to
// the wait queue proc_list of write_q and return 0. If it is a pipe file, it is judged whether
// there is free space in the pipeline to write characters. If there is one, it returns 1; if
// not (the pipeline is full), the current task is added to the waiting queue of the pipeline
// i-node and returns 0.
        if (tty = get_tty(inode))
                if (!FULL(tty->write_q))
                        return 1;
                else
                        add_wait(&tty->write_q->proc_list, wait);
        else if (inode->i_pipe)
                if (!PIPE_FULL(*inode))
                        return 1;
                else
                        add_wait(&inode->i_wait, wait);
        return 0;
}

//// Check abnormal state.
// Check if the file is in an abnormal state. For terminal device files, the kernel always returns
// 0. For pipe files, return 1 if one or both of the two pipe descriptors have been closed at
// this time, otherwise add the current task to the wait queue of the i-node and return 0.
// The parameter wait is the wait table pointer; the inode is the file i-node pointer.
// Returns 1 if an exception condition occurs, otherwise returns 0.
static int check_ex(select_table * wait, struct m_inode * inode)
{
        struct tty_struct * tty;

        if (tty = get_tty(inode))
                if (!FULL(tty->write_q))
                        return 0;
                else
                        return 0;
        else if (inode->i_pipe)
                if (inode->i_count < 2)
                        return 1;
                else
                        add_wait(&inode->i_wait,wait);
        return 0;
}

//// select() internal function.
// do_select() is the actual handler for the kernel to execute the select() system-call. The
// function first checks the validity of each descriptor in the descriptor set, and then calls
// the function check_XX() to check the descriptor of each descriptor set, and counts the number
// of descriptors currently ready in the descriptor set. If any of the descriptors are ready,
// the function will return immediately, otherwise the process will go to sleep, and the process
// will continue running after the timeout expires or because the process on the waiting queue
// where a descriptor is located is awakened.
int do_select(fd_set in, fd_set out, fd_set ex,
        fd_set *inp, fd_set *outp, fd_set *exp)
{
        int count;                                // the number of ready descriptors
        select_table wait_table;
        int i;
        fd_set mask;

// First, the three descriptor sets are ORed, and the effective descriptor bit mask in the
// descriptor set is obtained in the mask. It then loops through the current process to see
// if each descriptor is valid and included in the descriptor set. In the loop, each time a
// descriptor is judged, the mask is shifted to the right by 1 bit. Therefore, based on the
// least significant bit of the mask, we can determine whether the corresponding descriptor
// is in the descriptor set given by the user. A valid descriptor should be a pipe file descriptor,
// either a character device file descriptor or a FIFO descriptor, and the rest of the types
// return an EBADF error as an invalid descriptor.
        mask = in | out | ex;
        for (i = 0 ; i < NR_OPEN ; i++,mask >>= 1) {
                if (!(mask & 1))                  // not in the descriptor set.
                        continue;
                if (!current->filp[i])            // file not opened.
                        return -EBADF;
                if (!current->filp[i]->f_inode)   // file i-node is null.
                        return -EBADF;
                if (current->filp[i]->f_inode->i_pipe) // valid: a pipe file.
                        continue;
                if (S_ISCHR(current->filp[i]->f_inode->i_mode)) // valid: a char dev file.
                        continue;
                if (S_ISFIFO(current->filp[i]->f_inode->i_mode)) // valid: a FIFO file.
                        continue;
                return -EBADF;                          // all the rest are invalid.
        }
// Let's start by looping to see if each descriptor in the three descriptor sets is ready
// (operational). At this point the 'mask' is used as the mask for the descriptor currently
// being processed. The three functions check_in(), check_out(), and check_ex() in the loop
```

```
    // are used to determine whether the descriptor is ready for read in, write out or in abnormal
    // condition. If a descriptor is ready to operate, the corresponding bit is set in the relevant
    // descriptor set, and the count number of counts of the ready descriptor is incremented by
    // one. On line 183, the statement "mask += mask" is equivalent to "mask << 1".
178 repeat:
179         wait_table.nr = 0;
180         *inp = *outp = *exp = 0;
181         count = 0;
182         mask = 1;
183         for (i = 0 ; i < NR_OPEN ; i++, mask += mask) {
    // If the descriptor checked at this time is in the read operation descriptor set, and the
    // descriptor is ready for a read operation, the corresponding bit in the descriptor set is
    // set to 1, and the count of ready descriptors is incremented by one.
184                 if (mask & in)
185                         if (check_in(&wait_table,current->filp[i]->f_inode)) {
186                                 *inp |= mask;           // set the bit in the set.
187                                 count++;                // ready number.
188                         }
    // If the descriptor checked at this time is in the write operation descriptor set, and the
    // descriptor is ready for a write operation, the corresponding bit in the descriptor set is
    // set to 1, and the count of ready descriptors is incremented by one.
189                 if (mask & out)
190                         if (check_out(&wait_table,current->filp[i]->f_inode)) {
191                                 *outp |= mask;
192                                 count++;
193                         }
    // If the descriptor checked at this time is in the abnormal descriptor set, and the descriptor
    // is in abnormal condition,the corresponding bit in the ex descriptor set is set to 1, and
    // the count of abnormal descriptors is incremented by one.
194                 if (mask & ex)
195                         if (check_ex(&wait_table,current->filp[i]->f_inode)) {
196                                 *exp |= mask;
197                                 count++;
198                         }
199         }
    // After all the descriptors of the process have been checked, if there is no ready descriptor
    // (count==0), and the process does not receive any non-blocking signal, and there are waiting
    // descriptors at this time, or if the wait time has not expired, then we set the current process
    // state to interruptable sleep state, and then execute the scheduler to perform other tasks.
    // When the kernel schedules this task again, it will call free_wait() to wake up the tasks
    // before and after the task on the relevant waiting queue, then jump to the repeat label (line
    // 178) and re-detect whether there are descriptors we are concerned about.
200         if (!(current->signal & ~current->blocked) &&
201             (wait_table.nr || current->timeout) && !count) {
202                 current->state = TASK_INTERRUPTIBLE;
203                 schedule();
204                 free_wait(&wait_table);      // the task is awakened and returned to here.
205                 goto repeat;
206         }
    // If count is not equal to 0 at this time, or if a signal is received, or the wait time is
    // up and there is no descriptor to wait for, then we call free_wait() to wake up the task on
    // the wait queue and then return the number of ready descriptors.
207         free_wait(&wait_table);
```

```
        return count;
}

/*
 * Note that we cannot return -ERESTARTSYS, as we change our input
 * parameters. Sad, but there you are. We could do some tweaking in
 * the library function ...
 */          // The parameter *timeout is changed during processing.
//// select() system-call function.
// The code in this function is mainly responsible for parameter copying and conversion before
// and after the select() function operation. The main job of select() is done by the do_select()
// function. Sys_select() will first decompose and copy the parameters of the select() function
// from the user data space into the kernel space according to the buffer pointer given by the
// parameter, then set the waiting timeout value, and then call do_select(). After returning,
// the result will be copied back into the user space.
// The parameter buffer points to the first parameter of the select() function in the user area.
// If the return value is less than 0, an error occurs during execution; if the return value
// is equal to 0, it means that no descriptor is ready for operation within the specified waiting
// time; if the return value is greater than 0, it indicates the number of ready descriptors.
int sys_select( unsigned long *buffer )
{
/* Perform the select(nd, in, out, ex, tv) system call. */
// First define several local variables that are used to decompose the select() function
// arguments passed by the pointer arguments.
        int i;
        fd_set res_in, in = 0, *inp;             // read in fd set.
        fd_set res_out, out = 0, *outp;          // write out fd set.
        fd_set res_ex, ex = 0, *exp;             // abnormal fd set.
        fd_set mask;                             // descriptor value range (nd) mask code.
        struct timeval *tvp;                     // wait time structure pointer.
        unsigned long timeout;

// Then, the parameters are isolated and copied from the user data area into the local pointer
// variables, and three descriptor sets in (read), out (write), and ex (exception or abnormal)
// are respectively obtained according to whether the descriptor set pointer is valid. Where
// mask is also a descriptor set variable. Based on the maximum descriptor value +1 in the three
// descriptor sets (ie, the value of the first parameter nd), it is set to the mask of all
// descriptors of interest to the user program. For example, if nd = 4, then mask = 0b00001111
// (32 bits total).
        mask = ~((~0) << get_fs_long(buffer++));
        inp = (fd_set *) get_fs_long(buffer++);
        outp = (fd_set *) get_fs_long(buffer++);
        exp = (fd_set *) get_fs_long(buffer++);
        tvp = (struct timeval *) get_fs_long(buffer);

        if (inp)                                   // read in set.
                in = mask & get_fs_long(inp);
        if (outp)                                  // write out set.
                out = mask & get_fs_long(outp);
        if (exp)                                   // abnormal set.
                ex = mask & get_fs_long(exp);
// Next we try to take the wait (sleep) time value 'timeout' from the time structure. First,
// the 'timeout' is initialized to the maximum (infinite) value, and then the time value in
```

```
    // the time structure is obtained from the user space, and the current tick value jiffies of
    // the system is converted and added to it, and finally the time ticking value 'timeout' that
    // needs to wait is obtained. We use this value to set the delay that the current process needs
    // to wait. In addition, the tv_usec field on line 241 is a microsecond value. Dividing it by
    // 1000000 gives the corresponding number of seconds, and multiplies it by the system ticks
    // per second HZ, which converts tv_usec into a tick value.
239         timeout = 0xffffffff;
240         if (tvp) {
241                 timeout = get_fs_long((unsigned long *)&tvp->tv_usec)/(1000000/HZ);
242                 timeout += get_fs_long((unsigned long *)&tvp->tv_sec) * HZ;
243                 timeout += jiffies;
244         }
245         current->timeout = timeout;      // set ticks that the current process need delay.
    // The main work of the select() is done in function do_select().The code after calling this
    // function is used to copy the processing result into the user data area and return it to the
    // user. To avoid race conditions, you need to disable the interrupt before calling do_select()
    // and then enable it after the function returns.
    // If after the do_select() returns, the process's wait delay field timeout is still greater
    // than the current system timing tick value jiffies, indicating that there is descriptors ready
    // before the timeout. So here we will record the time value remaining until the timeout, and
    // then we will return this value to the user. If the process's wait delay field timeout is
    // already less than or equal to the current system jiffies, it means that do_select() may be
    // returned due to a timeout, so the remaining time value is set to zero.
246         cli();                                       // disable int.
247         i = do_select(in, out, ex, &res_in, &res_out, &res_ex);
248         if (current->timeout > jiffies)
249                 timeout = current->timeout - jiffies;
250         else
251                 timeout = 0;
252         sti();                                       // enable int.
    // Next we clear the timeout field of the process. If the number of ready descriptors returned
    // by do_select() is less than 0, it indicates an execution error, so the error number is returned.
    // Then we write the processed descriptor set content and the delay time structure content back
    // to the user buffer space. When writing the time structure content, it is also necessary to
    // convert the remaining delay time represented by the tick time unit into seconds and microsecond
    // values.
253         current->timeout = 0;
254         if (i < 0)
255                 return i;
256         if (inp) {
257                 verify_area(inp, 4);
258                 put_fs_long(res_in,inp);         // readable descriptor set.
259         }
260         if (outp) {
261                 verify_area(outp,4);
262                 put_fs_long(res_out,outp);       // writable descriptor set.
263         }
264         if (exp) {
265                 verify_area(exp,4);
266                 put_fs_long(res_ex,exp);         // abnormal descriptor set.
267         }
268         if (tvp) {
269                 verify_area(tvp, sizeof(*tvp));
```

```
                put_fs_long(timeout/HZ, (unsigned long *) &tvp->tv_sec);  // seconds.
                timeout %= HZ;
                timeout *= (1000000/HZ);
                put_fs_long(timeout, (unsigned long *) &tvp->tv_usec);    // microseconds.
        }
// If there is no ready descriptor available at this time and a non-blocking signal is received,
// the interrupted error number is returned. Otherwise, the number of the ready descriptors
// is returned.
        if (!i && (current->signal & ~current->blocked))
                return -EINTR;
        return i;
}

```

12.20 Summary

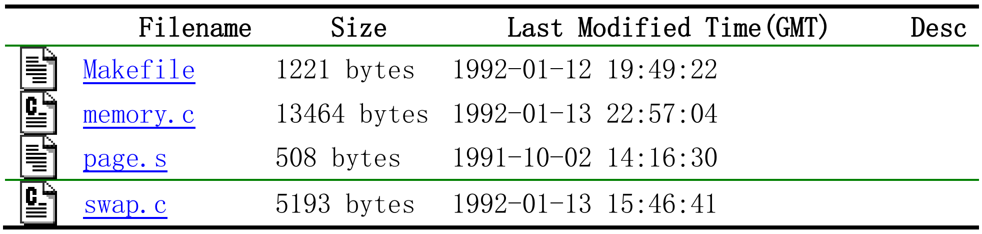
In this chapter, we first give and explain the structure and composition of the MINIX file system, and describe the directory structure, directory entries, and file path name structures in the file system. After that, the structure and usage of the high-speed buffer (buffer cache) in Linux are explained in detail, and the way in which other programs in the kernel access the block device by using the buffer block is explained. Then from the implementation of the file system code, the program code is detailed in three aspects: the underlying general file system functions, the file access operation code, and the file access and control system-call interfaces.

The next chapter mainly describes the specific method and code implementation of Linux using the segmentation and paging functionalities provided by Intel 80X86 to manage memory. It also explains the principle and implementation of the copy-on-write mechanism and the demond loading mechanism.

13 Memory Management (mm)

In the Intel 80X86 architecture system, the Linux kernel's memory management program uses paging management. It uses the page directory and page table structure to handle the application and release of memory from other parts of the kernel. Memory management is performed in units of memory pages, and a memory page refers to 4K bytes of physical memory with consecutive addresses. The page directory entry and page table entry allow you to address and manage the usage of the specified memory page. There are three files in the memory management directory of Linux 0.12, as shown in Listing 13-1:

List 13-1 Memory management subdirectory file list



Filename	Size	Last Modified Time(GMT)	Desc
Makefile	1221 bytes	1992-01-12 19:49:22	
memory.c	13464 bytes	1992-01-13 22:57:04	
page.s	508 bytes	1991-10-02 14:16:30	
swap.c	5193 bytes	1992-01-13 15:46:41	

Among them, the page.s assembly file is relatively short, it only contains the memory page exception interrupt service program (INT 14), mainly to achieve the processing of page fault and page write protection. Memory.c is the core program for memory page management. It is used for memory initialization operations, page directory and page table management, and other parts of the kernel for memory application processing. The swap.c program is used for memory page exchange management, which mainly includes exchange mapping bitmap management functions and switching device access functions.

13.1 Main Functionalities

In the Intel 80X86 CPU, the program uses an address consisting of segment and intra-segment offset during the addressing process. This address is not directly used to address physical memory addresses and is therefore referred to as a virtual address. In order to address physical memory, an address translation mechanism is needed to map or transform virtual addresses into physical memory addresses. This address translation mechanism is one of the main functionalities of memory management (Another main function is the memory addressing protection mechanism). The virtual address is first transformed into an intermediate address form by the segment management mechanism—the 32-bit linear address of the CPU, and then this linear address is mapped to the physical address using the paging mechanism.

In order to understand how the Linux kernel manages memory operations, we need to understand how memory paging management works and understand its addressing mechanism. The purpose of paging management is to map physical memory pages to a linear address. When analyzing the memory management program in this chapter, it is necessary to clearly distinguish whether the given address refers to a linear address or an actual physical memory address.

See Chapter 4 for a detailed description of memory management in Intel 80X86 CPU Protected Mode. For the convenience of reading, we will further explain the related contents of the memory paging management mechanism.

13.1.1 Memory paging mechanism

In the Intel 80X86 system, memory paging management is performed through a two-level tables consisting of a memory page directory table and page tables, as shown in Figure 13-1. The page directory table and the page table have the same structure, and the table item structure is also the same, as shown in Figure 13-4 below. Each entry in the page directory table (referred to as a page directory entry, 4 bytes) is used to address a page table, and each page table entry (4 bytes) is used to specify a page of physical memory pages. Therefore, when a page directory entry and a page table entry are specified, we can uniquely determine the corresponding physical memory page. The page directory table occupies one page of memory, so up to 1024 page tables can be addressed, and each page table also occupies one page of memory, so a page table can also address up to 1024 physical memory pages. Thus, in a 32-bit 80X86 CPU, all page tables addressed by a page directory table can address a total of 1024 X 1024 X 4096 = 4G memory space. In the Linux 0.12 kernel, all processes share a single page-directory table, and each process has its own page table. The kernel code and data segment length is specified as 16MB, using 4 page tables (ie 4 page directory entries). After segmentation mechanism transformation, the kernel code and data segment are located in the first 16MB range of the linear address space, and then transformed by the paging mechanism, which is directly mapped one by one to 16MB of physical memory. So for the kernel segment its linear address is the physical address.

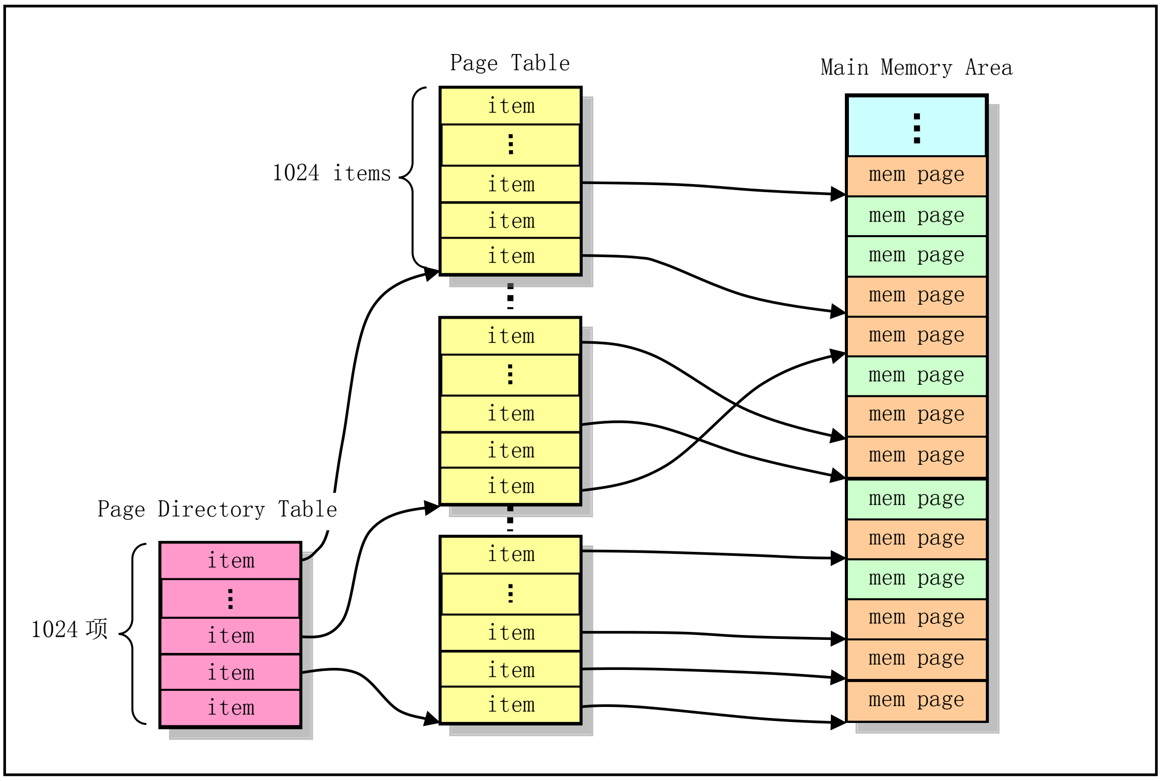


Figure 13-1 Page directory table and page table structure diagram

For user processes or other parts of the kernel, a linear address is used when applying for memory. So, how does a linear address use these two tables to map to a physical address? In order to use the paging mechanism, a 32-bit linear address is divided into three parts, which are used to specify a page directory entry, a page table entry, and an offset address on the corresponding physical memory page, so that it can be indirectly addressed the physical memory location specified by the linear address, as shown in Figure 13-2.

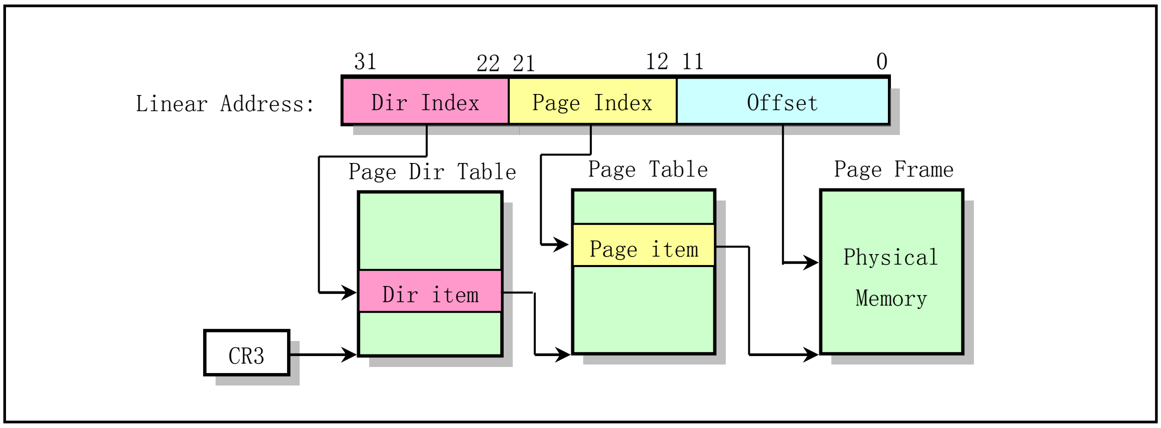


Figure 13-2 Schematic diagram of linear address transformation

Bits 31-22 of the linear address are used to determine the directory entries in the page directory; bits 21-12 are used to address the page table entries in the page table specified by the page directory entry, and the last 12 bits are used as the offset address in the physical memory page specified by the page entry.

In memory management functions, a large number of transform calculations from linear addresses to actual physical addresses are used. For the linear address of a given process, we can easily find the page directory entry corresponding to the linear address through the address translation relationship shown in Figure 13-2. If the directory entry is valid (used), the page frame address in the directory entry specifies the base address of a page table in physical memory. Then, in combination with the page table entry pointer in the linear address, if the page table entry is valid, based on the specified page frame address in the page table entry, we can finally determine the address of the actual physical memory page corresponding to the specified linear address. Conversely, if you need to find the corresponding linear address from a known physical memory page address, you need to search the entire page directory table and all page tables. If the physical memory page is shared, we may find multiple corresponding linear addresses. Figure 13-3 graphically shows how a given linear address is mapped onto a physical memory page. For the first process (task 0), its page table is after the page directory table, for a total of 4 pages. For an application's process, the memory used by its page table is applied to the memory manager when the process is created, so it is in the main memory area.

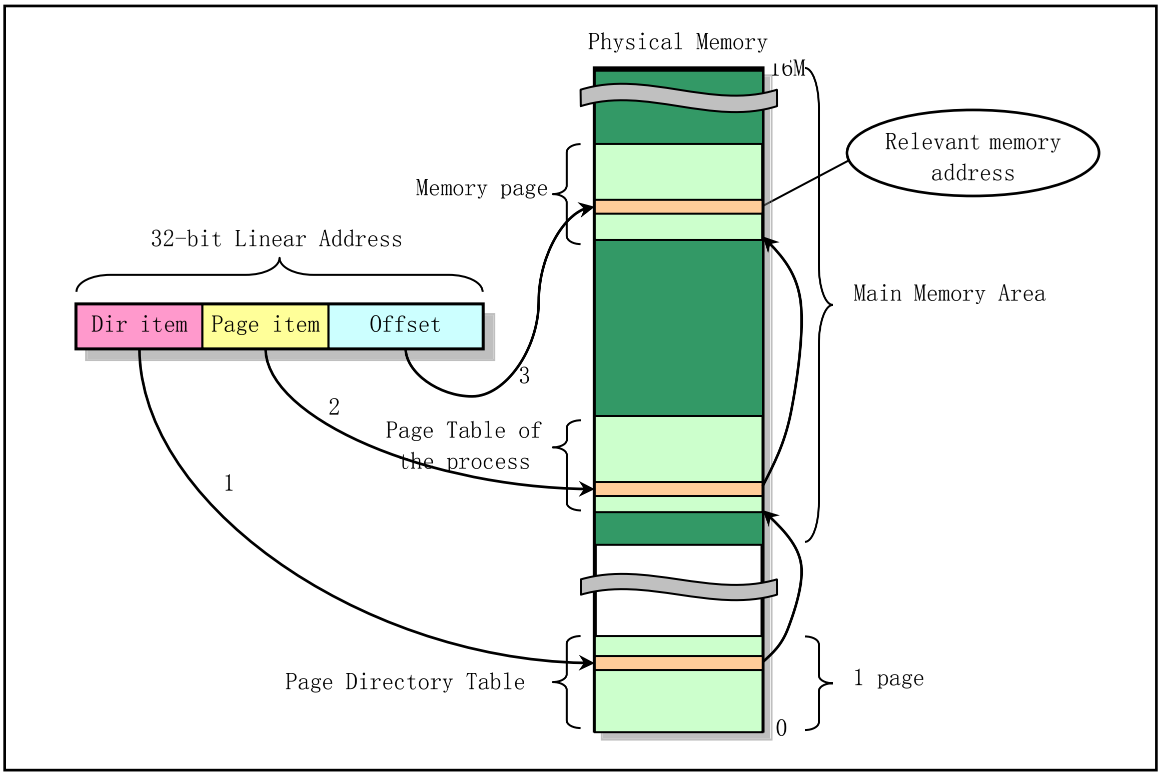


Figure 13-3 Physical address corresponding to the linear address

Multiple page directory tables can exist simultaneously in a system, and only one page directory table is available at a time. The currently used page directory table is determined by the CPU's register CR3, which stores the physical memory address of the current page directory table. But in the Linux kernel discussed in this book, the kernel code and all processes share a single page directory table.

In Figure 13-1, we see that the physical memory page corresponding to each page table entry is random within the 4G address range and is determined by the content of the page frame address in the page table entry set by the memory manager. Each page table entry consists of a page frame address, an access flag bit, a dirty (rewritten) flag bit, and a presence flag bit, as shown in Figure 13-4.

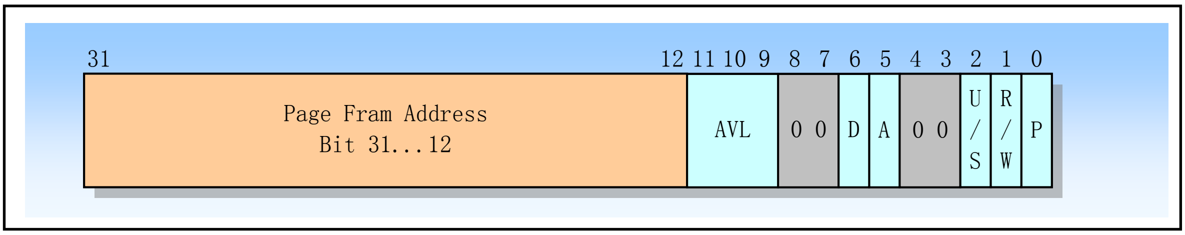


Figure 13-4 Page directory and page table entry structure

The page frame address specifies the physical start address of a page of memory. Because the memory page is on the 4K address boundary, its lower 12 bits are always 0, so the lower 12 bits of the entry can be used for other purposes. In a page directory table, the page frame address of the table entry is the start address of a page table; in the second level page table, the page frame address of the page table entry contains the physical memory page address of the desired memory operation.

The presence bit (P) in the figure determines whether a page table entry can be used for the address translation process. P=1 means the entry is available. When the directory entry or the second-level entry has P=0, the entry is invalid and cannot be used in the address translation process. At this moment, all other bits of the entry are available to the program; the processor does not test these bits.

When the CPU attempts to use a page table entry for address translation, if P=0 of any one of the page table entries at this time, the processor will issue a page exception interrupt signal. At this point, the page fault interrupt exception handler can map and load the requested page into physical memory, and the instruction that caused the exception will be re-executed.

The accessed (A) and modified or dirty (D) bits are used to provide information about the use of the page. These bits are set by hardware, but are not reset, except for the modified bits in the page directory entry. The small difference between a page directory entry and a page table entry is that the page table entry has a dirty bit (D), while the page directory entry does not.

Before a read/write operation is performed on a page of memory, the CPU sets the accessed bits of the associated directory entry and secondary page table entry. Before writing to the address covered by a secondary page table entry, the processor sets the modified bit (D) of the secondary page table entry, and the bits (D) in the page directory entry are not used. When the required memory exceeds the actual amount of physical memory, the memory manager can use these bits to determine which pages can be taken from memory to make room. The memory manager is also responsible for detecting and resetting these bits.

The read/write bit (R/W) and the user/supervisor bit (U/S) are not used for address translation, but the protection mechanism for paging level is performed by the CPU during the address translation process at the same time.

13.1.2 Physical Memory Allocation and Management

With the above basic concepts, we can explain how the Linux system manages memory, but we also need to first understand the use of memory space by the Linux kernel. For the Linux 0.12 kernel, it supports up to 16M of physical memory by default. In an 80X86 computer system with 16MB of memory, the Linux kernel occupies the foremost part of physical memory, as shown in Figure 13-5. The label 'end' in the figure indicates the location where the kernel module ends. This is followed by a cache buffer with a maximum memory address of 4M. The cache buffer is divided into two sections by the display memory and the ROM BIOS. The remaining memory portion is called the main memory area. The main memory area is allocated and managed by the procedures in this chapter. If there is a RAM virtual disk in the system, the memory space occupied by the virtual disk needs to be deducted from the front of the main memory area. When you need to use the main memory area, you need to apply to the memory management program in this chapter. The basic unit of application is the memory page. Figure 13-5 shows the function of each part of the physical memory.

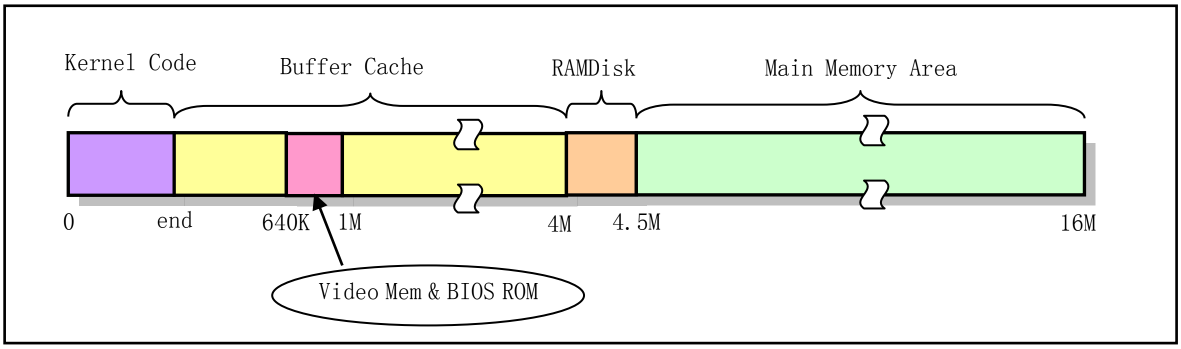


Figure 13-5 Main memory area schematic

In Chapter 6, Booting System, we already know that Linux's page directory and page table are set in the program head.s. The head.s program stores a page directory table at physical address 0, followed by four page tables. These four page tables will be used for mapping operations in the memory area occupied by the kernel. Since the code and data for task 0 are included in the kernel region, task 0 also uses these page tables. Other derived processes will request memory pages in the main memory area to store their own page tables. The two programs in this chapter are used to manage these tables to achieve the allocation of memory pages in the main memory area.

In order to save physical memory, when a new process is generated by calling fork(), the new process shares the same memory area with the original process. Only when one of the processes starts a write operation will the system allocate additional memory pages for it. This is the concept of copy-on-write.

The page.s program is used to implement the page fault or exception processing (INT 14). For interrupts caused by page faults and page write protection, the interrupt handler will call the do_no_page() and do_wp_page() functions in memory.c respectively. do_no_page() will take the required page from the block device to the memory specified location. In the case of shared memory pages, do_wp_page() copies the page being written (copy-on-write), which also cancels the sharing of the page.

13.1.3 **Linear address space allocation**

When reading the code in this chapter, we also need to understand the distribution of code and data in a program's logical address space, as shown in Figure 13-6.

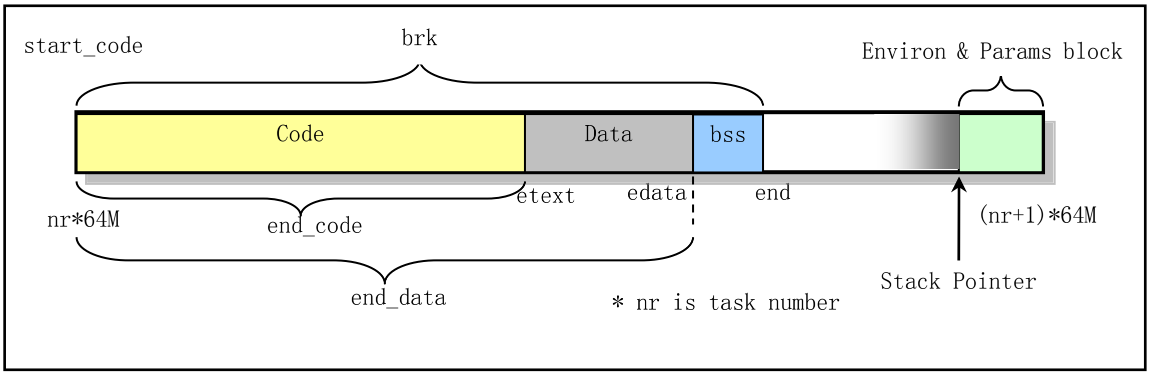


Figure 13-6 The distribution of processes in their logical address space

The logical address space occupied by each process starts from the location of nr*64MB in the linear address space (nr is the task number), and the logical address space occupies a range of 64MB. The last part of the environment parameter data block is up to 128K in length, and its left side is the starting stack pointer.另外，In the figure, bss is the data segment that is not initialized by the process. The first page of the bss segment is initialized to all zeros when the process is created.

13.1.4**Transform between logical, linear, and physical address**

In the kernel memory management code, the conversion operation between the program logical address (or virtual address), the linear address of the CPU, and the physical memory address is often encountered. For example, when copying a page table, we need to linearly translate a given page directory entry (PDE) to get the physical memory address of the corresponding page table (PT); when it comes to copy-on-write operations, it involves mapping of pages in linear address space to physical addresses; when attempting to share a page, it involves mapping operations that map program logical address pages into CPU linear address space. Below we explain the conversion operation between them separately. Since the memory management in the kernel is usually operated in units of 4KB memory pages, let us first give the conversion method of the address to the page start address, and then explain how the pages in these different address spaces are converted.

1. Conversion of address to corresponding page address

Since the page address starts on the 4KB memory address boundary (ie, the lower 12 bits of the page address is 0), the address 'Page_addr' of a memory page containing the specified address 'addr' is:

```
    Page_addr = addr & 0xfffff000;
```

In these three address spaces, their page address calculation process is the same. Below we explain the transformation calculation method for the page address.

2. Linear address and logical address decomposition

According to the paging mechanism of the CPU, a 32-bit linear address 'addr' can be decomposed into a page directory entry PDE number (bits 31-22), a page table entry PTE number (bits 21-12), and an in-page offset (bit 11- 0), see Figure 13-2. Therefore, the general formula for obtaining these three parts independently is as follows:

```
    Page directory entry number: PDE_No = (addr >> 22);
    Page table entry number: PTE_No = (addr >> 12) & 0x3ff;
    Offset within the page: Offset = (addr & 0xfff);
```

Similarly, when performing the address translation operation, we can also regard the logical address 'Vaddr' in the process logical address space as being composed of these three parts "logically". Just when mapping them to the CPU linear address space, you need to add the corresponding part of the program code base address 'Base' in the linear address space:

```
    Logic_PDE_No = (Vaddr >> 22);
    Logic_PTE_No = (Vaddr >> 12) & 0x3ff;
    Logic_Offset = (Vaddr & 0xfff);
```

Since in this kernel, each process allocates a linear address space in units of 64 MB in length, the page table entry number and the in-page offset value corresponding to the code base address of each process are both 0, so we only need to consider the page directory entry number of the code base address during the conversion process. In the actual code, the code base address 'Base' of the process 'p' is 'p->start_code', so when a logical address 'Vaddr' in the program corresponds to the linear address space, the three parts of the address are:

```
    PDE_No = (Base >> 22) + Logic_PDE_No;
           = (p->start_code >> 22) + (Vaddr >> 22);
    PTE_No = Logic_PTE_No
           = (Vaddr >> 12) & 0x3ff;
    Offset = Logic_Offset
           = (Vaddr & 0xfff);
```

3. Program page logical address to physical address conversion

The so-called logical page address means that the page address is the address calculated from the start address of the program process 'p' code, which is called the code base address, as shown in Figure 13-6. According to the structure of page directory entries and page table entries (see Figure 13-4), and the above decomposition of logical addresses, because each page directory entry and page table entry take up 4 bytes, we left shift the table entry number 2 bits to get the offset of entry in the table, plus the physical base address PDT_Base of the page directory table, we can get the directory entry address (pointer) PDE. For the Intel 80X86 CPU, its current page directory base address PDT_Base is stored in control register CR3. Because all programs in this kernel share only one page directory table, and the page directory table is stored at the beginning of physical memory 0, the page directory table base address PDT_Base=0. So we have the address (pointer) PDE of the page directory entry in physical memory:

```
    PDE = PDT_Base + (PDE_No << 2);
        = 0 + (PDE_No << 2);
        = (((p->start_code >> 22) + (Vaddr >> 22)) << 2);
        = ((p->start_code >> 20) & 0xffc) + ((Vaddr >> 20) & 0xffc);
```

Referring to Figure 13-4, we can get the physical address PT of the corresponding page table from the contents of the directory entry, and add the offset of the entry in the page table to obtain the physical address (pointer) of the page table entry PTE. :

```
    PT = (*PDE) & 0xfffff000;
    PTE = PT + (PTE_No << 2)
        = PT + (((Vaddr >> 12) & 0x3ff) << 2);
        = ((*PDE) & 0xfffff000) + ((Vaddr >> 10) & 0xffc);
```

Bits 31-12 of the page table entry are the physical page frame addresses. Therefore, the physical page address corresponding to the logical address ‘Vaddr’ of the final program is:

```
    PPaddr = (*PTE) & 0xfffff000;
```

4. Linear address to physical address conversion

According to the above decomposition of the linear address, for the linear address Laddr, its page directory entry number, page table entry number, and in-page offset value have been given at the beginning of the second point above. Correspondingly, the offset value PDE in the page directory table corresponding to the page directory entry number is:

```
PDE = (PDE_No << 2);
    = ((Laddr >> 22) << 2);
    = ((Laddr >> 20) & 0xffc);
```

We can get the physical address PT of the corresponding page table from the contents of the directory entry, and add the offset value of the entry in the page table to obtain the physical address PTE (pointer) of the page table entry:

```
PT = (*PDE) & 0xfffff000;
PTE = PT + (PTE_No << 2);
    = PT + ((Laddr >> 12) & 0x3ff) << 2);
    = (*PDE) & 0xfffff000 + ((Laddr >> 10) & 0xffc);
    = (*((Laddr >> 20) & 0xffc) & 0xfffff000) + ((Laddr >> 10) & 0xffc);
```

Therefore, the actual physical page address corresponding to the linear address 'Laddr' is 'PPaddr', and the corresponding physical address 'Paddr' is the physical page address plus the offset within the page, as shown below:

```
PPaddr = (*PTE) & 0xfffff000;
Paddr = PPaddr + Laddr & 0xfff;
   = (*PTE) & 0xfffff000 + Laddr & 0xfff;
   = *(((Laddr >> 10) & 0xffc) + (*((Laddr >> 20) & 0xffc) & 0xfffff000)) + Laddr & 0xfff;
```

13.1.5 Page-fault exception handling

In the state where the paging mechanism (PG=1) is enabled, if the CPU detects the following conditions during the conversion of the linear address to the physical address, it will cause a page-fault exception interrupt INT 14:

- The presence bit (P) in the page directory entry or page table entry used in the address translation process is equal to 0, indicating that the page table or the page containing the operand does not exist in physical memory;
- The current execution code does not have sufficient privileges to access the specified page, or the user mode code writes a read-only page, and so on.

The page exception handling procedure can recover and restart an interrupted program or process from the page-not-present conditions and does not affect the continuity of program execution. It can also restart a program or task after a privilege violation, but the problem that caused the privilege violation may not be corrected. At this point, the CPU provides the following two aspects to the page-fault exception handler to assist in diagnosing and correcting the error:

- An error code on the stack. The format of the error code is a 32-bit long word, but only the lowest 3 bits are useful. Their names are the same as the last three bits in the page table entry (U/S, W/R, P). Their meanings and roles are:
  - ◆ Bit 0 (P), the exception is caused by a not-present page or violating access privileges. P=0, indicating that the page does not exist; P=1 indicates that the page-level protection privilege is

violated.

- ◆ Bit 1 (W/R), the exception is due to a memory read or write operation. W/R=0, indicating that it is caused by a read operation; W/R=1, indicating that it is caused by a write operation.
- ◆ Bit 2 (U/S), the code level at which the CPU executes when an exception occurs. U/S=0, the CPU was executing at supervisor mode; U/S=1, CPU was executing at user mode.

- ■ The linear address in control register CR2. The CPU will store the linear address that generated the exception in CR2. The page fault exception handler can use this address to locate the relevant page directory and page table entry. If another page exception is allowed to occur during the execution of the page exception handler, the handler should push CR2 onto the stack.

The page.s program that will be described later uses the above information to distinguish between a page fault exception or a write protection exception, thereby determining whether to call the page fault processing function do_no_page() or the write protect function do_wp_page() in the memory.c program.

13.1.6 Copy-on-write mechanism

Copy-on-write is a way to defer or avoid copying data. At this point, the kernel does not copy the data in the entire address space of the process, but allows the parent and child processes to share the same copy. When process A creates a child process B using the fock(), since child process B is actually a copy of parent process A, it will have the same physical page as the parent process. That is, in order to save memory and speed up the process of creating a process, the fork() function will let the child process B share the physical page of the parent A in a read-only manner, and also set the access rights of the parent A to these physical pages to only read mode too (see the copy_page_tables() in the memory.c program). In this way, when either parent A or child B performs a write operation on these shared physical pages, a page-fault exception (INT 14) is generated. At this point, the CPU will execute the system-provided exception handler do_wp_page() to try to resolve the exception. This is the copy-on-write mechanism.

The do_wp_page() function un-shares the physical page that caused the write exception interrupt (by calling the un_wp_page() function) and copies a new physical page for the write process, thus making the parent A and the child B each has a physical page with the same content. And the physical page that will perform the write operation is marked as write-accessible, and then the copy operation is actually performed (only this physical page is copied). Finally, when returning from the exception handler, the CPU will re-execute the write operation that caused the exception, allowing the process to continue.

Therefore, when a process writes within its own virtual address range, the passive copy-on-write operation above is used, namely: write operation -> page fault exception -> handle write protection exception -> re-execute the write operation instruction. For kernel code, when a write operation is performed within the virtual address range of a process, for example, a process invokes a system-call, if the system-call copies data into the buffer of the process, the kernel will call function verify_area(). This function first actively calls the memory page verification function write_verify() to determine whether there is a page sharing condition. If there is, the copy-on-write operation of the page is performed.

In addition, it is worth noting that in the Linux 0.12 kernel, the copy-on-write technique was not used when executing the fork() to create the process in the kernel code address space (linear address <1MB) . So when process 0 (ie idle process) creates process 1 (init process) in kernel space, it will use the same piece of code and data segments. However, since the page table entry copied by process 1 is also read-only, when the process 1 needs to perform a stack (write) operation, it also causes a page exception, so in this case the memory manager will allocate memory for the process in the main memory area. .

It can be seen that copy-on-write will delay the copying operation of the memory page until the actual write operation, and the page copy operation may not be performed at all when the page is not written. For example, when fork() creates a process and immediately calls execve() to execute a new program. So this technique can avoid the overhead of unnecessary memory page copying.

13.1.7 Load on demand mechanism

When using the execve() system call to load an executable image file on the file system, the kernel allocates 64MB of contiguous space to the corresponding process in the 4G linear address space of the CPU, and applies and allocates a certain amount physical memory pages for its environment and command line parameters. Other than that, there is actually no other physical memory page allocated to the executable. Of course, it is also impossible to load the code and data from the execution image file from the file system. Therefore, once the program starts running from the entry execution point, it will immediately cause the CPU to generate a page fault exception (the memory page where the execution pointer is located does not exist). At this point, the kernel's page fault exception handler will load the relevant code page in the executable file from the file system into the physical memory page according to the specific linear address causing the page fault exception, and map to the page position specified in the process logical address. When the exception handler returns, the CPU re-executes the instruction that caused the exception, allowing the executable program to continue execution.

If the program needs to run to another page that has not been loaded during execution, or the code instruction needs to access data that has not been loaded, then the CPU will also generate a page fault exception interrupt, and the kernel will then load the other corresponding page content into memory and execute the program again. In this way, only the code or data pages that are run to (used) in the executable file are loaded into the physical memory by the kernel. This method of loading a page in an executable file only when it is actually needed is called a load on demand technique or a demand-paging technique.

One obvious advantage of using a demand-loading technique is that it allows the executable program to start running immediately after calling the execve() system-call, without having to wait for multiple block device I/O operations to load the entire executable file image into memory before it starts running. Therefore, the execution speed of the system to load the execution program will be greatly improved. However, this technique has certain requirements on the format in which the object file is loaded. It requires that the file object format being executed be of the ZMAGIC type, that is, the object file format of the demand paging format. In this object file format, the code segment and data segment of the program are stored from the page boundary to accommodate the kernel to read the code or data content in units of one page.

13.2 memory.c

13.2.1 Function

The memory.c program manages the paging of the memory, realizing the dynamic allocation and recycling of the memory pages in the main memory area. For physical memory areas other than the memory occupied by the kernel (above 1MB address), the kernel uses a byte array mem_map[] to indicate the state of the physical memory page. Each byte entry describes the occupancy status of a physical memory page. The value indicates the number of times occupied, and 0 indicates that the corresponding physical memory is idle. When applying a page of physical memory, the value of the corresponding byte is incremented by one. When the byte value is 100, it means that it is completely occupied and can no longer be allocated any more.

In the course of memory management initialization, the kernel code first calculates the number of memory pages (PAGING_PAGES) corresponding to the memory area above 1MB, and sets value of all items of mem_map[] to 100 (occupied), and then the mem_map[] items corresponding to the main memory area are all cleared (zeroed), as shown in Figure 13-7. Therefore, the buffer cache area above the 1MB address used by the kernel and the virtual disk area (if any) have been initialized to be in a full occupancy state. The items corresponding to the main memory area in mem_map[] are set or reset during system use. For example, for a machine with 16MB of physical memory and a 512KB virtual disk as shown in Figure 13-5, the mem_map[] array has a total of (16MB - 1MB) / 4KB = 3840 entries, which corresponds to 3840 pages. The number of pages in the main memory area is (16MB-4.5MB) / 4KB = 2944, corresponding to the last 2944 items of the mem_map[] array, while the first 896 items correspond to the physical memory occupied by the cache buffer and virtual disk above the 1MB memory. Therefore, in the memory management initialization process, the first 896 items of mem_map[] are set to the occupied state (value is 100) and can no longer be allocated for use. The value of the 2944 entry is cleared to 0 and can be allocated by the memory manager.

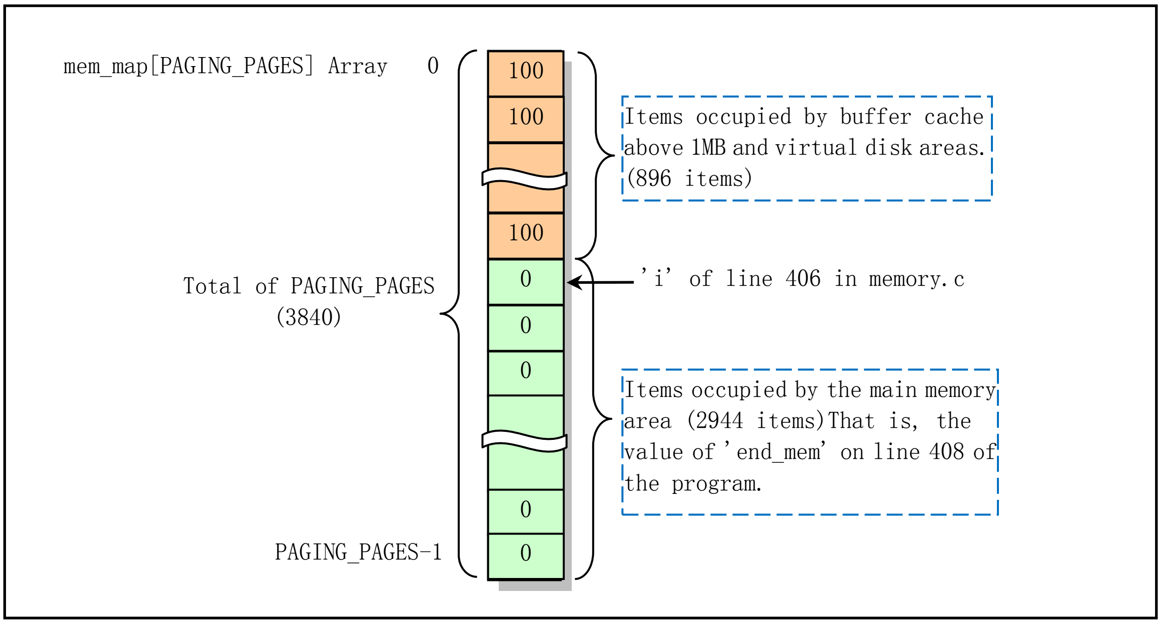

mem_map[PAGING_PAGES] Array
0
100
100
100
0
0
0
0
0
Items occupied by buffer cache above 1MB and virtual disk areas. (896 items)
'i' of line 406 in memory.c
Total of PAGING_PAGES (3840)
Items occupied by the main memory area (2944 items)That is, the value of 'end_mem' on line 408 of the program.
PAGING_PAGES-1

Figure 13-7 Mem_map[] array initialization with 16MB memory and 512KB vdisk

For the management of the process virtual address (or logical address), the kernel uses the segment management mechanism of the processor to implement, and the mapping relationship between the physical memory page and the linear address is handled by modifying the contents of the page directory and the page table entry. The following is a detailed description of several main functions provided in the program.

The get_free_page() and free_page() functions are specifically designed to manage the occupation and freeness of physical memory in the main memory area, regardless of the linear address of each process. The get_free_page() function is used to request a page of free memory in the main memory area and return the starting address of the physical memory page. It first scans the memory page bytemap array mem_map[], looking for a byte entry with a value of 0 (corresponding to a free page). If not, it returns 0, indicating that the physical memory has been used up. If a byte with a value of 0 is found, it is set to 1 and the starting address of the corresponding free page is calculated. Then the memory page is cleared, and finally the physical memory start address of the free page is returned.

free_page() is used to release a page of physical memory at the specified address. It first determines if the given memory address is <1M, and if so, returns, because within 1M is kernel-specific; If the specified physical memory address is greater than or equal to the actual memory highest address, an error message is displayed; Then, the page number is converted by the specified memory address: (addr - 1M) / 4K; then it is judged whether the mem_map[] byte item corresponding to the page number is 0, if not 0, then it is decremented by 1 and returned; Otherwise, the byte entry is cleared and an error message "Trying to release a free page" is displayed.

The free_page_tables() and copy_page_tables() functions release or copy the physical memory page blocks corresponding to the specified linear address and length (the number of page tables) in units of physical memory blocks (4M) corresponding to one page table. They not only modify the contents of the corresponding directory entries in the page directory and page table of the linear address, but also release or occupy the physical memory pages corresponding to all page table items in each page table.

The free_page_tables() function is used to release the physical memory page corresponding to the specified linear address and length (number of page tables). It first determines that the specified linear address should be on the 4M boundary and that the specified address value should be above the space occupied by the kernel and buffer cache. Then calculate the number of directory entries (ie, the number of page tables) occupied in the page directory table, and calculate the corresponding start directory entry number. Then, starting from the corresponding start directory entry, the occupied directory entry is released, and all page table entries and corresponding physical memory pages in the page table pointed to by the corresponding directory entry are released. Finally refresh the page transform cache.

The copy_page_tables() function is used to copy the page directory entry and page table corresponding to the specified linear address and length (page table count) memory, so that the copied page directory and the original physical memory area corresponding to the page table are shared. The function first verifies whether the specified source and destination linear addresses are both at the 4Mb memory boundary, and then caculates the corresponding start page directory entry (from_dir, to_dir) from the specified linear address; and calculates the number of page tables (page directory entries) occupied by the memory region to be copied. Then start copying the original directory entries and page table entries to the new free directory entries and page table entries, respectively. There is only one page directory table, and the page table of the new process needs to apply for a free memory page to store. Thereafter, the original and new page directories and page table entries are all set to read-only pages. When there is a write operation, the page exception handler is used to perform a copy-on-write operation. Finally, the byte array item corresponding to the shared physical memory page is incremented by one.

The put_page() function is used to map a specified physical memory page to a specified linear address. It first determines that the specified memory page address is within the range of 1M and the system's highest-end memory address, and then calculates the corresponding directory entry for the specified linear address in the page directory table. If the directory entry is valid, the address of the corresponding page table is taken, otherwise the free page is applied to the page table, and the attributes of the page table entry are set. Finally, the specified physical memory page address is still returned.

The do_wp_page() function is a page write protection procedure called in the page exception handler (implemented in mm/page.s). It first determines if the address is in the code area of the process and then performs a copy-on-write operation.

do_no_page() is a page fault function called during a page exception. It first determines the offset length value of the specified linear address relative to the process base address in the process space. If it is larger than the code plus data length, or the process is just starting to create, then apply for a page of physical memory and map it to the process linear address. Otherwise, try to perform a page sharing operation. If not, apply for a page of memory and read a page of content from the device. If the specified (linear address + 1 page) length exceeds the length of the process code plus data when the content of the page is added, the excess is cleared. The page is then mapped to the specified linear address.

The get_empty_page() function is used to get a page of free physical memory and map to the specified linear address. It mainly uses the get_free_page() and put_page() functions to implement this function.

13.2.2 Code annoation

Program 13-1 linux/mm/memory.c

```
/*
 *  linux/mm/memory.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * demand-loading started 01.12.91 - seems it is high on the list of
 * things wanted, and it should be easy to implement. - Linus
 */

/*
 * Ok, demand-loading was easy, shared pages a little bit tricker. Shared
 * pages started 02.12.91, seems to work. - Linus.
 *
 * Tested sharing by executing about 30 /bin/sh: under the old kernel it
 * would have taken more than the 6M I have free, but it worked well as
 * far as I could see.
 *
 * Also corrected some "invalidate()"s - I wasn't doing enough of them.
 */

/*
 * Real VM (paging to/from disk) started 18.12.91. Much more work and
 * thought has to go into this. Oh, well..
 * 19.12.91  -  works, somewhat. Sometimes I get faults, don't know why.
 *              Found it. Everything seems to work now.
 * 20.12.91  -  Ok, making the swap-device changeable like the root.
 */

  // <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
  //     manipulation function prototypes.
  // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
  //     descriptors/interrupt gates, etc. is defined.
  // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
  //     of the initial task 0, and some embedded assembly function macro statements about the
  //     descriptor parameter settings and acquisition.
  // <linux/head.h> Head header file. A simple structure for the segment descriptor is defined,
  //     along with several selector constants.
  // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
  //     used functions of the kernel.
#include <signal.h>

```

```
33 #include <asm/system.h>
34
35 #include <linux/sched.h>
36 #include <linux/head.h>
37 #include <linux/kernel.h>
38
   // CODE_SPACE(addr) ((((addr)+0xfff)&~0xfff) < current->start_code + current->end_code)
   // This macro is used to determine if a given linear address is within the code section of the
   // current process. "(((addr)+4095)&~4095)" is used to obtain the end address of the memory
   // page where the linear address 'addr' is located. See line 265.
39 #define CODE_SPACE(addr) ((((addr)+4095)&~4095) < \
40 current->start_code + current->end_code)
41
   // A global variable that holds the highest physical address of the actual physical memory.
42 unsigned long HIGH_MEMORY = 0;
43
   // Copy 1 page of memory from 'from' to 'to' (4K bytes).
44 #define copy_page(from,to) \
45 __asm__("cld ; rep ; movsl"::"S" (from),"D" (to),"c" (1024):"cx","di","si")
46
   // Physical memory mapped byte array (1 byte represents 1 page of memory). The corresponding
   // byte of each page is used to mark the number of times the page is currently referenced
   // (occupied). For machines with 16MB of physical memory, it can map up to 15Mb of memory. In
   // the initialization function mem_init(), the position that cannot be used as the main memory
   // area page is set to USED (100) in advance.
47 unsigned char mem_map [ PAGING_PAGES ] = {0,};
48
49 /*
50  * Free a page of memory at physical address 'addr'. Used by
51  * 'free_page_tables()'
52  */
   //// Free 1 page of memory starting with the physical address 'addr'.
   // The memory space below 1MB of physical address is used for kernel programs and buffers, and
   // cannot be used as memory space for allocation pages. Therefore the parameter 'addr' needs
   // to be greater than 1MB.
53 void free_page(unsigned long addr)
54 {
   // This function first determines the rationality of the physical address 'addr' given by the
   // parameter. If the physical address 'addr' is less than the low end of the memory (1MB), it
   // means it is in the kernel program or cache range, and will not be processed; if the 'addr'
   // is greater than or equal to the highest end of the physical memory in the system, it will
   // display an error message and the kernel stopped working.
55         if (addr < LOW_MEM) return;    // LOW_MEM = 1MB, defined in include/linux/mm.h, 30.
56         if (addr >= HIGH_MEMORY)
57                 panic("trying to free nonexistent page");
   // If the verification of the parameter 'addr' is passed, the memory page number counted from
   // the low end of the memory is calculated based on this physical address.
   //    "Page number = (addr - LOW_MEM) / 4096"
   // It can be seen that the page number starts from 0, and the page number is stored in 'addr'.
   // If the mapping byte corresponding to the page number is not equal to 0, it is returned after
   // decrementing by 1. At this point, the mapped byte value should be 0, indicating that the
   // page has been released. However, if the corresponding page byte is originally 0, it means
   // that the physical page is originally idle, indicating that the kernel code is faulty. The
// error message is then displayed and the kernel is stopped.
        addr -= LOW_MEM;
        addr >>= 12;                    // divided by 4096.
        if (mem_map[addr]--) return;
        mem_map[addr]=0;
        panic("trying to free free page");
}

/*
 * This function frees a continuos block of page tables, as needed
 * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
 */
//// According to the given linear address and limit length (number of page tables), free the
//// memory block and set the table entries to be empty.
// The page directory table is located at the beginning of physical address 0. There are 1024
// entries, each of which is 4 bytes, which together account for 4K bytes. Each directory entry
// corresponds to one page table. The kernel page table starts at the physical address 0x1000
// (following the directory table space) and has 4 page tables. Each page table has 1024 entries,
// each of which is 4 bytes, and therefore also accounts for 4KB (1 page) of memory. Except
// for processes 0 and 1 in the kernel code, the pages occupied by the page tables of other
// processes are requested by the kernel from the main memory area when the process is created.
// Each page table entry corresponds to 1 page of physical memory, so a page table can map up
// to 4MB of physical memory.
// Parameters: from - the linear base address; size - the length of bytes to be freed.
int free_page_tables(unsigned long from,unsigned long size)
{
        unsigned long *pg_table;
        unsigned long * dir, nr;

// First check if the linear base address given by the parameter 'from' is at the 4MB boundary,
// because this function can only handle this situation. If from = 0, an error occurs. This
// indicates that the function is trying to free up space occupied by the kernel and buffer.
// Then calculate the number of page directory entries corresponding to the length given by
// the parameter 'size' (multiple of 4MB with carry), that is, the number of page tables occupied.
// Since 1 page table can manage 4MB of physical memory, the value of the memory length to be
// copied is divided by 4MB by shifting 22 bits to the right. Add 0x3fffff (ie 4Mb -1) to get
// the result of the integer multiple with carry, that is, add 1 if there is a remainder in
// the operation. For example, if the original size = 4.01Mb, then the result size = 2 is obtained.
// Next, the starting directory entry corresponding to the given linear base address is
// calculated.
// The corresponding directory entry number is equal to "from >> 20". Since each entry occupies
// 4 bytes, and since the page directory table is stored from physical address 0, the actual
// directory entry pointer is "directory entry number << 2", which is "(from >> 20)". "& 0xffc"
// makes sure the directory entry pointer is within the valid range.
        if (from & 0x3fffff)
                panic("free_page_tables called with wrong alignment");
        if (!from)
                panic("Trying to free up swapper memory space");
        size = (size + 0x3fffff) >> 22;
        dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */

// At this point 'size' is the number of page tables that need to be released, ie the number
// of page directory entries, and 'dir' is the starting directory entry pointer. Now, start
// a loop operation for each page directory entry, and then release the entries in each page
// table in turn. If the current directory entry is invalid (P bit = 0), indicating that the
// directory entry is not used (the corresponding page table does not exist), the next directory
// entry continues to be processed. Otherwise, the page table address 'pg_table' is taken from
// the directory entry, and 1024 entries in the page table are processed, and the physical memory
// page corresponding to the valid page table entry (P bit=1) is released, or the invalid page
// table entry (P bit = 0) is released from the swap device, that is, release the corresponding
// memory page in the swap device (because the page may have been swapped out). Then clear the
// page table entry and continue processing the next page entry. When all the entries in a page
// table have been processed, the memory page occupied by the page table itself is freed, and
// the next page directory entry is processed. Finally refreshes the page transform cache and
// returns 0.
        for ( ; size-->0 ; dir++) {
                if (!(1 & *dir))
                        continue;
                pg_table = (unsigned long *) (0xfffff000 & *dir);  // get page table addr.
                for (nr=0 ; nr<1024 ; nr++) {
                        if (*pg_table) {
                                if (1 & *pg_table)   // free the page if valid.
                                        free_page(0xfffff000 & *pg_table);
                                else                 // free the page in swap device.
                                        swap_free(*pg_table >> 1);
                                *pg_table = 0;       // reset the page content.
                        }
                        pg_table++;                  // points to next page.
                }
                free_page(0xfffff000 & *dir);        // free the page of the page table.
                *dir = 0;                            // reset the directory entry.
        }
        invalidate();                                // refresh CPU page transform cache.
        return 0;
}

/*
 *  Well, here is one of the most complicated functions in mm. It
 * copies a range of linerar addresses by copying only the pages.
 * Let's hope this is bug-free, 'cause this one I don't want to debug :-)
 *
 * Note! We don't copy just any chunks of memory - addresses have to
 * be divisible by 4Mb (one page-directory entry), as this makes the
 * function easier. It's used only by fork anyway.
 *
 * NOTE 2!! When from==0 we are copying kernel space for the first
 * fork(). Then we DONT want to copy a full page-directory entry, as
 * that would lead to some serious memory waste - we just copy the
 * first 160 pages - 640kB. Even that is more than we need, but it
 * doesn't take any more memory - we don't copy-on-write in the low
 * 1 Mb-range, so the pages can be shared with the kernel. Thus the
 * special case for nr=xxxx.
 */
//// Copy page directory entries and page table entries.
// This function is used to copy the page directory entries and page table entries corresponding
// to the specified linear address and memory size, so that the physical memory area corresponding
// to them is shared by the two sets of page table mapping. When copying, we need to apply for
// a new page to store the new page table, the original physical memory area will be shared.
// After that, the two processes (the parent and its child processes) will share the memory
// area until the kernel performs a write operation, and the kernel allocates a new memory page
// (copy-on-write) for the write operation process. The parameters 'from' and 'to' are linear
// addresses, and 'size' is the length of memory that needs to be copied (shared) in bytes.
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
        unsigned long * from_page_table;
        unsigned long * to_page_table;
        unsigned long this_page;
        unsigned long * from_dir, * to_dir;
        unsigned long new_page;
        unsigned long nr;

// The code first detects the validity of the source address 'from' and the destination address
// 'to' given by the parameter. Both the source and destination addresses need to be on the
// 4Mb memory boundary address. This requirement is given because 1024 entries in a page table
// can manage 4 Mb of memory. The source address and the destination address only satisfy this
// requirement to ensure that the page table entry is copied from the first entry of a page
// table, and all the original entries of the new page table are valid. Then get the start
// directory entry pointers (from_dir and to_dir) of the source and destination addresses. Then
// calculate the number of page tables (ie, the number of directory entries) occupied by the
// memory block to be copied according to the 'size' given by the parameter.
        if ((from&0x3fffff) || (to&0x3fffff))
                panic("copy_page_tables called with wrong alignment");
        from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
        to_dir = (unsigned long *) ((to>>20) & 0xffc);
        size = ((unsigned) (size+0x3fffff)) >> 22;

// After obtaining the source start directory entry pointer from_dir and the destination start
// directory entry pointer to_dir and the number of page tables to be copied, the following
// begins to apply one page of memory to each page directory entry to save the corresponding
// page table, and start page table entry copy operation. If the page table specified by the
// destination directory entry already exists (P=1), the kernel crashes. If the source directory
// entry is invalid, that is, the specified page table does not exist (P=0), the next page
// directory entry continues to be looped.
        for( ; size-->0 ; from_dir++,to_dir++) {
                if (1 & *to_dir)
                        panic("copy_page_tables: already exist");
                if (!(1 & *from_dir))
                        continue;

// After verifying that the current source directory entry and destination entry are normal,
// we take the page table address from_page_table in the source directory entry. In order to
// save the page table corresponding to the destination directory entry, it is necessary to
// apply for one page of free memory page in the main memory area. If the function get_free_page()
// returns 0, it means that no free memory page is obtained, and there may be insufficient memory.
// Then return a value of -1 to exit.
                from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
                if (!(to_page_table = (unsigned long *) get_free_page()))
                        return -1;      /* Out of memory, see freeing */
```

```
    // Next, we set the destination directory entry information: the last 3 bits are set, that is,
    // the current destination directory entry "OR" 7, indicating that the memory page mapped to
    // the corresponding page table is user-level, and can read, write, and exists (Usr, R/W,
    // Present). (If the U/S bit is 0, then R/W has no effect. If U/S is 1, and R/W is 0, then the
    // code running at the user level can only read the page. If U/S and R /W are all set, there
    // is permission to read and write). Then set the number of page items to be copied for the
    // page table corresponding to the currently processed page directory entry. If it is in kernel
    // space, you only need to copy the corresponding page table entry of the first 160 pages (nr=160),
    // corresponding to the beginning of 640KB physical memory, otherwise you need to copy all 1024
    // entries in a page table (nr= 1024), which can map 4MB of physical memory.
140                 *to_dir = ((unsigned long) to_page_table) | 7;
141                 nr = (from==0)?0xA0:1024;                        // 0xA0 = 160

    // At this point, for the current page table, we start to cyclically copy the specified 'nr'
    // memory page table entries. We first take out the contents of the source page table entry.
    // If the current source page is not used (the content is 0), then the table item is not copied
    // and the next entry is processed.
142                 for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
143                         this_page = *from_page_table;
144                         if (!this_page)
145                                 continue;
    // If the entry has content, but its presence bit P=0, the page corresponding to the entry may
    // be in the swap device. So we apply for 1 page of memory and read the page from the swap device
    // (if there is one in the swap device), and then copy the page table entry to the destination
    // page table entry, and modify the source page table entry content to point to the new memory
    // page, and set the table entry flag to "page dirty" plus 7. Then continue processing the next
    // page entry. Otherwise, the R/W flag (bit 1 is set to 0) is reset in the page table entry,
    // that is, the memory page corresponding to the page table entry is set to read-only, and then
    // the page table entry is copied to the destination page table.
146                         if (!(1 & this_page)) {
147                                 if (!(new_page = get_free_page()))
148                                         return -1;
149                                 read_swap_page(this_page>>1, (char *) new_page);
150                                 *to_page_table = this_page;
151                                 *from_page_table = new_page | (PAGE_DIRTY | 7);
152                                 continue;
153                         }
154                         this_page &= ~2;
155                         *to_page_table = this_page;

    // If the address of the physical page indicated by the page table entry is above 1MB, you need
    // to set the memory page map array mem_map[].Then calculate the page number and use it as an
    // index to increase the number of references in the corresponding item of the page mapping
    // array. For pages below 1MB, it is a kernel page, so there is no need to set mem_map[]. Because
    // mem_map[] is only used to manage page usage in the main memory area. So for the kernel to
    // move to task 0 and call fork() to create task 1 (used to run init()), since the copied pages
    // are still in the kernel code area, the statements in the following judgment will not be
    // executed. The page for task 0 can still be read and written at any time. The judgment statement
    // will only be executed when the parent process code calling fork() is in the main memory area
    // (page position is greater than 1MB). This happens only when the process calls execve() and
    // loads the new program code.
    // The meaning of the line 157 statement is that the memory page pointed to by the source page
    // table entry is also read-only, because now two processes have shared memory areas. If one
    // of the processes needs to perform a write operation, the page for the write operation can
    // be allocated a new free page by the page exception write protection process, that is, a copy
    // on write operation.
                            if (this_page > LOW_MEM) {
                                    *from_page_table = this_page; // set source read-only too.
                                    this_page -= LOW_MEM;
                                    this_page >>= 12;
                                    mem_map[this_page]++;
                            }
                    }
            }
            invalidate();                                   // refresh CPU page transform cache.
            return 0;
    }

    /*
     * This function puts a page in memory at the wanted address.
     * It returns the physical address of the page gotten, 0 if
     * out of memory (either when trying to access page-table or
     * page.)
     */
    //// Map a physical memory page to a specified location in a linear address space.
    // Or, the page at the specified address in the linear address space is mapped to the main memory
    // area page. The main job of implementing this function is to set the information of the specified
    // page in the relevant page directory entry and page table entry. This function is called in
    // do_no_page() that handles page not present exceptions. For exceptions caused by page not
    // present, when the page table is modified due to any page absence, it is not necessary to
    // refresh the page translation buffer (or Translation Lookaside Buffer - TLB) of the CPU even
    // if the page table entry flag P was changed from 0 to 1. Because invalid page entries are
    // not buffered, there is no need to refresh when an invalid page table entry is modified, as
    // shown here without calling the Invalidate() function. The parameter 'page' is a pointer to
    // a page (page frame) in the allocated main memory area; 'address' is a linear address.
    static unsigned long put_page(unsigned long page,unsigned long address)
    {
            unsigned long tmp, *page_table;

    /* NOTE !!! This uses the fact that _pg_dir=0 */

    // Here first determine the validity of the physical memory page 'page' given by the parameter.
    // A warning message is issued if the page position is lower than LOW_MEM or if the system actually
    // contains the memory high-end HIGH_MEMORY. LOW_MEM=1MB, which is defined in the file
    // include/linux/mm.h file, which is the lowest possible starting position of the main memory
    // area. When the system physical memory is less than or equal to 6MB, the main memory area
    // starts directly at LOW_MEM. Then check whether the page is the one that has been applied
    // for, that is, whether the corresponding byte in the memory page map mem_map[] has been set.
    // If not, a warning is required.
            if (page < LOW_MEM || page >= HIGH_MEMORY)
                    printk("Trying to put page %p at %p\n",page,address);
            if (mem_map[(page-LOW_MEM)>>12] != 1)
                    printk("mem_map disagrees with %p at %p\n",page,address);

    // Then, according to the linear address given by the parameter, the corresponding entry pointer
    // in the page directory table is calculated, and the page table address is obtained therefrom.
```

```
    // If the directory entry is valid (P=1), that is, the specified page table is in memory, the
    // specified page table address is taken from it and placed in the page_table variable. Otherwise,
    // apply a free page to the page table, and set the corresponding flag (7 - User, U/S, R/W)
    // in the corresponding directory entry, and then put the page table address into the page_table
    // variable.
184         page_table = (unsigned long *) ((address>>20) & 0xffc);
185         if ((*page_table)&1)
186                 page_table = (unsigned long *) (0xfffff000 & *page_table);
187         else {
188                 if (!(tmp=get_free_page()))
189                         return 0;
190                 *page_table = tmp | 7;
191                 page_table = (unsigned long *) tmp;
192         }
    // Finally, the related entry content is set in the found page table page_table, that is, the
    // address of the physical page is filled in the entry, and three flags (U/S, W/R, P) are set
    // at the same time. The index value of the entry in the page table is equal to the 10-bit value
    // consisting of linear address bits 21 - bits 12. There are a total of 1024 entries (0 -- 0x3ff)
    // per page table.
193         page_table[(address>>12) & 0x3ff] = page | 7;
194 /* no need for invalidate */
195         return page;                    // Returns the physical page address.
196 }
197
198 /*
199  * The previous function doesn't work very well if you also want to mark
200  * the page dirty: exec.c wants this, as it has earlier changed the page,
201  * and we want the dirty-status to be correct (for VM). Thus the same
202  * routine, but this time we mark it dirty too.
203  */
    //// Map a physical memory page whose content has been modified to a specified location in the
    // linear address space. This function is almost exactly the same as the previous function
    // put_page(). In addition to setting the page table entry content on line 223, this function
    // also sets the page modified flag (bit 6, PAGE_DIRTY).
204 unsigned long put_dirty_page(unsigned long page, unsigned long address)
205 {
206         unsigned long tmp, *page_table;
207
208 /* NOTE !!! This uses the fact that _pg_dir=0 */
209
210         if (page < LOW_MEM || page >= HIGH_MEMORY)
211                 printk("Trying to put page %p at %p\n",page,address);
212         if (mem_map[(page-LOW_MEM)>>12] != 1)
213                 printk("mem_map disagrees with %p at %p\n",page,address);
214         page_table = (unsigned long *) ((address>>20) & 0xffc);
215         if ((*page_table)&1)
216                 page_table = (unsigned long *) (0xfffff000 & *page_table);
217         else {
218                 if (!(tmp=get_free_page()))
219                         return 0;
220                 *page_table = tmp|7;
221                 page_table = (unsigned long *) tmp;
222         }
```

```
        page_table[(address>>12) & 0x3ff] = page | (PAGE_DIRTY | 7);
/* no need for invalidate */
        return page;
}

//// Cancel write page protection. Used for the processing of write protection exception
// during page exceptions (copy-on-write).
// When the kernel creates a process, the new process and the parent process are set to share
// code and data memory pages, and all of these pages are set to read-only. When the new process
// or the original process needs to write data to the memory page, the CPU will detect this
// and generate a page write protection exception. So in this function the code will first
// determine if the page to be written is shared. If not, set the page to be writable and then
// exit; if the page is in a shared state, you need to re-apply a new page and copy the written
// page content for the writing process to use separately. The share was therefore cancelled.
// The input parameter is a page table entry pointer.
void un_wp_page(unsigned long * table_entry)
{
        unsigned long old_page,new_page;

// First take the physical page location (address) in the given page table entry of the parameter
// and determine whether the page is a shared page. If the original page address is greater
// than the low end of the memory LOW_MEM (in the main memory area), and its value in the page
// mapping byte array is 1 (indicating that the page is only referenced once, the page is not
// shared), then the R/W flag (writable) is set in the page table entry, and the page translation
// cache is refreshed and then returned. That is, if the memory page is only used by one process
// at this time, and is not a process in the kernel, the attribute can be directly changed to
// writable, and there is no need to re-apply a new page.
        old_page = 0xfffff000 & *table_entry;    // get physical page address in the entry.
        if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
                *table_entry |= 2;               // set R/W flag.
                invalidate();
                return;
        }
// Otherwise, it is necessary to apply for a free page in the main memory area for the process
// of performing the write operation and cancel the page sharing. If the original page position
// is greater than the low end of the memory (meaning mem_map[] > 1, the page is shared), the
// page byte array value of the original page is decremented by 1. Then update the contents
// of the specified page table entry to the new page address, and set the read and write flags
// (U/S, R/W, P). After refreshing the page transformation cache (TLB), the original page content
// is finally copied to the new page.
        if (!(new_page=get_free_page()))
                oom();                           // Out of Memory.
        if (old_page >= LOW_MEM)
                mem_map[MAP_NR(old_page)]--;
        copy_page(old_page,new_page);
        *table_entry = new_page | 7;
        invalidate();
}

/*
 * This routine handles present pages, when users try to write
 * to a shared page. It is done by copying the page to a new address
 * and decrementing the shared-page counter for the old page.
 *
 * If it's in code space we exit with a segment error.
 */
//// Perform write protection processing on the shared page.
// This is the C function called during the handling of page exception and will be called in
// the page.s program. The function parameters error_code and address are automatically
// generated by the CPU when the process writes a write-protected page. The parameter error_code
// indicates the type of error; address is the page linear address that generated the exception.
// The page needs to be copied (copy-on-write) when writing a shared page.
void do_wp_page(unsigned long error_code,unsigned long address)
{
// The function first determines in what range the linear address given by the CPU control
// register CR2 that caused the page exception. If address is less than TASK_SIZE (0x4000000,
// or 64MB), it means that the exception page location is within the linear address range of
// the kernel or task 0 or task 1, then issued a warning message; If (address - current process
// code start address) is greater than the size of one process (64MB), indicating that the linear
// address is not within the space of the process causing the exception, exit after issuing
// an error message.
        if (address < TASK_SIZE)
                printk("\n\rBAD! KERNEL MEMORY WP-ERR!\n\r");
        if (address - current->start_code > TASK_SIZE) {
                printk("Bad things happen: page error in do_wp_page\n\r");
                do_exit(SIGSEGV);
        }
#if 0
/* we cannot do this yet: the estdio library writes to code space */
/* stupid, stupid. I really want the libc.a from GNU */
// If the linear address is in the code space of the process, the execution program needs to
// be terminated. Because the code is read-only.
        if (CODE_SPACE(address))
                do_exit(SIGSEGV);
#endif
// Then we call the above function un_wp_page() to handle the cancel page protection. But first
// we need to prepare the parameters for it. Its parameter is the entry pointer in the page
// table for the given linear address "address", which is calculated as:
// (1) "((address>>10) & 0xffc)": Calculates the offset address of the page table entry in the
// specified linear address in the page table. Because according to the linear address structure,
// "(address>>12)" is the index of the page table entry, but each entry occupies 4 bytes, so
// after multiplying 4: "(address>>12)<<2 = (address>>10)&0xffc" , you can get the offset address
// of the page table entry in the table. The "AND" operation &0xffc is used to limit the address
// range within one page. Also, since only 10 bits are shifted, the last 2 bits are the highest
// 2 of the lower 12 bits of the linear address and should be masked out. Therefore, the more
// obvious way to find the offset of the page table entry in the linear address is:
//     "(((address>>12) & 0x3ff)<<2)".
// (2) "(0xfffff000 & *((address>>20) &0xffc))": Used to get the address of the page table in
// the directory entry. Where "((address>>20) &0xffc)" is used to get the offset of the entry
// index in the directory table in the linear address. Because "address>>22" is the directory
// entry index and each item is 4 bytes, multiply by 4: "(address>>22)<<2 = (address>>20)",
// this is the offset address of the specified entry in the directory table. "&0xffc" is used
// to mask out the last 2 bits of the directory entry index value, and "*((address>>20) &0xffc)"
// gets the physical address of the corresponding page table in the contents of the specified
// directory entry. Finally, "AND" with 0xffffff000 is used to mask some of the flag bits in
// the contents of the page directory entry (lower 12 bits of the directory entry). This can
```

```
    // be more intuitively represented as:
    //     "(0xfffff000 & *((unsigned long *) (((address>>22) & 0x3ff)<<2))))".
    // (3) A pointer (physical address) of the page table entry can be obtained by the offset address
    // in the page table of (1), plus the address of the corresponding page table in the content
    // of the directory entry in (2). Then we use it here to copy the shared page.
268             un_wp_page((unsigned long *)
269                     (((address>>10) & 0xffc) + (0xfffff000 &
270                     *((unsigned long *) ((address>>20) &0xffc)))));
271
272 }
273
    //// Write page verification.
    // If the page is not writable, copy the page. This function will be called by the memory-verified
    // generic function verify_area() on line 34 in fork.c.
    // The parameter 'address' is the linear address of the specified page.
274 void write_verify(unsigned long address)
275 {
276         unsigned long page;
277
    // First, take the page directory entry corresponding to the specified linear address, and judge
    // whether the page table corresponding to the directory entry exists according to the existing
    // bit (P) in the directory entry (bit P=1?), if not (P=0), return . This is because there is
    // no sharing and copy-on-write for pages that do not exist, and if the program performs a write
    // operation on a page that does not exist, the system will execute do_no_page() due to a page
    // fault exception , and use the put_page() function to map a physical page for this place.
    // Then the code fetches the page table address from the directory entry, plus the page table
    // entry offset value of the specified page in the page table, and obtains the page table entry
    // pointer corresponding to the address. The physical page corresponding to the given linear
    // address is included in the entry.
278         if (!( (page = *((unsigned long *) ((address>>20) & 0xffc)) )&1))
279                 return;
280         page &= 0xfffff000;
281         page += ((address>>10) & 0xffc);
    // Then, the bit 1 (R/W) and bit 0 (P) flags in the page table entry are checked. If the page
    // is not writable (R/W = 0) and exists, then the share checking and page copy operations
    // (copy-on-write) are performed, otherwise do nothing and quit directly.
282         if ((3 & *(unsigned long *) page) == 1)      /* non-writeable, present */
283                 un_wp_page((unsigned long *) page);
284         return;
285 }
286
    //// Get a free memory page and map it to the specified linear address.
    // The function get_free_page() is just an application to get a page of physical memory in the
    // main memory area. This function not only gets a page of physical memory, but also calls
    // put_page() to map the physical page to the specified linear address.
    // The parameter 'address' is the linear address of the specified page.
287 void get_empty_page(unsigned long address)
288 {
289         unsigned long tmp;
290
    // If a free page cannot be obtained, or if the page taken cannot be placed at the specified
    // address, a message indicating that the memory is insufficient is displayed. The meaning of
    // the comment on line 292 is: the parameter 'tmp' of the free_page() function can be 0, it
// doesn't matter, the function will ignore it and return normally.
        if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
                free_page(tmp);           /* 0 is ok - ignored */
                oom();
        }
}

/*
 * try_to_share() checks the page at address "address" in the task "p",
 * to see if it exists, and if it is clean. If so, share it with the current
 * task.
 *
 * NOTE! This assumes we have checked that p != current, and that they
 * share the same executable or library.
 */
//// Try to share the page at the given address of the current process with the given process.
// The current process has the same execution code as the process 'p'. It can also be considered
// that the current process is a child process generated by the 'p' process executing the fork()
// operation, so their code content is the same at the beginning. If the content of the data
// segment has not been modified, their content should be the same too.
// The parameter 'address' is the logical address in the process, which is the logical page
// address that the current process wants to share with the 'p' process, and p is the process
// that will be shared. If the page at the 'address' of the 'p' process exists and has not been
// modified, let the current process share it with the p process. At the same time, we also
// need to verify whether the specified address has been applied for and got a page, and if
// so, the kernel goes wrong and crash. Returns 1 if page sharing succeeded, 0 failed.
static int try_to_share(unsigned long address, struct task_struct * p)
{
        unsigned long from;
        unsigned long to;
        unsigned long from_page;        // page directory entry of process 'p'
        unsigned long to_page;          // page directory entry of current process.
        unsigned long phys_addr;

// First, the page directory entry location corresponding to the logical address 'address' in
// the given process p and the current process is respectively obtained. For the convenience
// of calculation, we first find out the "logical" page directory entry at the given address
// 'address', that is, the page directory entry locations calculated in the process space
// (0 - 64MB). This logical entry location plus the page directory entries corresponding to
// the start addresses of process p and current process in the CPU 4G linear address space can
// obtain the actual page directory entry addresses 'from_page' and 'to_page' corresponding
// to the page at the address 'address' respectively.
        from_page = to_page = ((address>>20) & 0xffc);
        from_page += ((p->start_code>>20) & 0xffc);        // page dir entry of process p.
        to_page += ((current->start_code>>20) & 0xffc);    // page dir entry of current.

// After obtaining the directory entries corresponding to the two processes, they are processed
// separately. The following is the operation of the entry of the p process. The purpose is
// to obtain the physical page address corresponding to the 'address' in the p process, and
// determine whether the physical page exists and is clean (not modified, not dirty). The method
// is to first take the content of the directory entry, and then get the physical address of
// the page table in it, thereby calculating the page table entry pointer corresponding to the
// logical address 'address', and extracting the content of the page table entry and temporarily
// saved in phys_addr.
/* is there a page-directory at from? */
        from = *(unsigned long *) from_page;           // get content of page dir entry.
        if (!(from & 1))                               // page table not exist ?
                return 0;
        from &= 0xfffff000;                            // page table address.
        from_page = from + ((address>>10) & 0xffc);    // page table entry pointer.
        phys_addr = *(unsigned long *) from_page;      // content of the entry.

// Then look at the physical page of the page table entry mapped, whether it exists and is clean.
// The value '0x41' corresponds to the D (Dirty) and P (Present) flags in the page table entry.
// Returns if the page is dirty or not present. Then we get the physical page address from the
// entry and save it back to phys_addr. Finally, we need to check the validity of this physical
// page address. It should not exceed the machine's maximum physical address value, nor should
// it be less than the low end of the memory (1MB).
/* is the page clean and present? */
        if ((phys_addr & 0x41) != 0x01)
                return 0;
        phys_addr &= 0xfffff000;                       // physical page address.
        if (phys_addr >= HIGH_MEMORY || phys_addr < LOW_MEM)
                return 0;

// Similarly, the following code operates on the entry of the current process. The purpose is
// to obtain the address of the page table entry corresponding to 'address' in the current
// process, and it is determined that the page table entry has not been mapped to any physical
// page, that is, its P=0. First we take the current process page directory entry content to
// 'to'. If the entry is invalid (P=0), that is, the page table corresponding to the directory
// entry does not exist, then a free page is requested to store the new page table. And update
// the contents of the directory entry 'to_page' to point to the new page.
        to = *(unsigned long *) to_page;               // content of the entry of current.
        if (!(to & 1))
                if (to = get_free_page())
                        *(unsigned long *) to_page = to | 7;
                else
                        oom();
// Then we take the page table address in the directory entry to 'to', plus the offset address
// of the entry in the table, and get the page table entry address into 'to_page'. For this
// page table entry, if we check that the corresponding physical page already exists (P=1),
// it means that we want to share the corresponding physical page in process p, but now we have
// it. Then the kernel is wrong, crash.
        to &= 0xfffff000;                              // page table address.
        to_page = to + ((address>>10) & 0xffc);        // page table entry pointer.
        if (1 & *(unsigned long *) to_page)
                panic("try_to_share: to_page already exists");

// After finding the clean and existing physical page corresponding to the address 'address'
// in process p, and also determining the page table entry address in the current process, we
// now start to share them. The method used is very simple, that is, first modify the page table
// entry of the p process, set its write protection (R/W=0, read only) flag, and then let the
// current process copy the entry of the p process. At this point, the page at the current process
// address 'address' is mapped to the physical page mapped at the logical address 'address'
// of the p process.
/* share them: write-protect */
```

```
        *(unsigned long *) from_page &= ~2;
        *(unsigned long *) to_page = *(unsigned long *) from_page;
// We then refresh the page transform cache, calculate the page number of the physical page
// being manipulated, and increment the reference in the map byte array entry by one. Finally,
// return 1 if the sharing process is successful.
        invalidate();
        phys_addr -= LOW_MEM;
        phys_addr >>= 12;                    // obtain the page numberin main mem area.
        mem_map[phys_addr]++;
        return 1;
}

/*
 * share_page() tries to find a process that could share a page with
 * the current one. Address is the address of the wanted page relative
 * to the current data space.
 *
 * We first check if it is at all feasible by checking executable->i_count.
 * It should be >1 if there are other tasks sharing this inode.
 */
//// Look for the process running the same executable and try to share the page with it.
// When a page fault exception occurs, first check to see if you can share the page with other
// processes running the same executable file. The function first checks if there is another
// process in the system that is also running the same execution file as the current process.
// If so, look for such a task in all current tasks in the system. If we find such a task, try
// to share the page at the given address with it. If no other tasks in the system are running
// the same executable file as the current process, the preconditions for the shared page
// operation do not exist, so the function exits immediately.
// The way to determine if another process is executing the same executable file in the system
// is to use the executable field (or library field) in the process task structure, which points
// to the executable file's i-node in memory. We can make this judgment based on the reference
// number field i_count of the i-node. If the i_count of the node is greater than 1, it means
// that two or more processes in the system are running the same executable file, so we can
// compare executable field of all tasks in the task structure array to see if there is the
// processes running the same executable file.
// The parameter 'inode' is the i-node of the executable file of the process that wants to share
// the page. 'address' is the logical address of the page that the current process wants to
// share with a process. Returns 1 if the sharing operation is successful, 0 if it fails.
static int share_page(struct m_inode * inode, unsigned long address)
{
        struct task_struct ** p;

// First check the memory i-node reference count value given by the parameter. If the i-node
// reference count value is equal to 1 (executable->i_count =1) or the i-node pointer is empty,
// it means that only one process in the current system is running the execution file, or the
// provided i-node is invalid. So there is no sharing at all, just exit the function directly.
        if (inode->i_count < 2 || !inode)
                return 0;
// Otherwise, search all tasks in the task array, find the process that can share the page with
// the current process, that is, another process running the same executable file, and try to
// share the page with the specified address. If the process logical address 'address' is smaller
// than the start address LIBRARY_OFFSET of the process library file, it indicates that the
// shared page is within the logical address space corresponding to the process execution file.
// Then check if the given i-node is the same as the executable file i-node of the process
// (that is, the executable field of the process). If it is not the same, continue to search.
// If the logical address 'address' is greater than or equal to the start address LIBRARY_OFFSET
// of the process library file, it indicates that the page to be shared is in the library file.
// So we check if the specified 'inode' is the same as the library file i-node of the process,
// if it is not the same, continue to search.
// If a process 'p' is found whose 'executable' or 'library' field is the same as the specified
// 'inode', the function try_to_share() is called to try page sharing. The function returns
// 1 if the sharing operation is successful, otherwise it returns 0.
        for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
                if (!*p)                           // continue search if the item is null.
                        continue;
                if (current == *p)                 // continue search if it's the current.
                        continue;
                if (address < LIBRARY_OFFSET) {
                        if (inode != (*p)->executable)   // i-node of the executable file.
                                continue;
                } else {
                        if (inode != (*p)->library)      // i-node of the library file.
                                continue;
                }
                if (try_to_share(address,*p))
                        return 1;
        }
        return 0;
}

//// Page-not-present processing.
// This function is called during page exception interrupt processing and is called in the page.s
// program. The parameters 'error_code' and 'address' are automatically generated by the CPU
// when the process accesses the page due to a page-not-present fault. 'error_code' indicates
// the type of error; 'address' is the page linear address that generated the exception.
// The function first checks to see if the missing page is in the swap device, and if so, swaps
// in. Otherwise, try to share the page with the same file that has already been loaded, or
// just map a page of physical memory pages simply because the process dynamically requests
// the memory page. If the sharing operation is unsuccessful, the missing data page can only
// be read from the corresponding file to the specified linear address.
void do_no_page(unsigned long error_code,unsigned long address)
{
        int nr[4];
        unsigned long tmp;
        unsigned long page;
        int block,i;
        struct m_inode * inode;

// The function first determines in what range the linear address given by the CPU control
// register CR2 that caused the page exception. If address is less than TASK_SIZE (0x4000000,
// or 64MB), it means that the exception page location is within the linear address range of
// the kernel or task 0 or task 1, then issued a warning message; If (address - current process
// code start address) is greater than the size of one process (64MB), indicating that the linear
// address is not within the space of the process causing the exception, exit after issuing
// an error message.
        if (address < TASK_SIZE)
```

```
                printk("\n\rBAD!! KERNEL PAGE MISSING\n\r");
        if (address - current->start_code > TASK_SIZE) {
                printk("Bad things happen: nonexistent page error in do_no_page\n\r");
                do_exit(SIGSEGV);
        }
// Then, according to the specified linear address 'address', the corresponding secondary page
// table entry pointer is obtained, and according to the content of the entry, it is determined
// whether the page at the 'address' is in the swap device. If yes, swap in the page and exit.
// The method is to first obtain the content of the page directory item corresponding to the
// specified linear address 'address', and then take out the address of the page table therein,
// and add the page table entry offset to obtain the corresponding page table entry pointer,
// thereby obtaining the content of the page table entry. If the content of the entry is not
// 0, but bit P=0, it indicates that the physical page specified by the page table entry should
// be in the swap device. The function then exits after the specified page is loaded from the
// swap device.
        page = *(unsigned long *) ((address >> 20) & 0xffc); // content of page dir entry.
        if (page & 1) {
                page &= 0xfffff000;                           // page table address.
                page += (address >> 10) & 0xffc;              // page table entry pointer.
                tmp = *(unsigned long *) page;                // content of page entry.
                if (tmp && !(1 & tmp)) {
                        swap_in((unsigned long *) page);      // read in from swap device.
                        return;
                }
        }
// Otherwise, we take the page address at the given linear address 'address' and calculate the
// offset 'tmp' of the address in the process space relative to the process base address, ie
// the corresponding logical address. Thus we can calculate the specific starting block number
// of the missing page in the execution file image or in the library file.
        address &= 0xfffff000;                            // page address
        tmp = address - current->start_code;              // logical address of the page.

// If the logical address 'tmp' is greater than the starting position of the library image file
// in the process logical space, the missing page is in the library image file. Therefore, the
// i-node 'library' of the library image file can be obtained from the current process task
// structure, and the starting block number 'block' of the missing page in the library file
// is calculated. If the logical address 'tmp' is smaller than the end of the execution image
// file of the process, then the missing page is in the process execution file image, so the
// i-node number 'executable' of the execution file can be obtained from the current process
// task structure, and calculate the starting block number 'block' of the missing page in the
// execution file image. If the logical address 'tmp' is neither in the address range of the
// execution file nor in the library file space, the page not present is caused by the process
// accessing the dynamically requested memory page data, so there is no corresponding i-node
// and data block number (both set to be NULL).
// Since the first block of the image file stored on the block device is the program header
// structure, the first block of data needs to be skipped when the file is read. Because each
// block of data has a length of BLOCK_SIZE = 1KB, one page of memory can store 4 blocks of
// data. The process logical address 'tmp' is divided by the data block size plus one to get
// the starting block number 'block' of the missing page in the execution image file.
        if (tmp >= LIBRARY_OFFSET ) {
                inode = current->library;           // inode & block no. of library file.
                block = 1 + (tmp-LIBRARY_OFFSET) / BLOCK_SIZE;
        } else if (tmp < current->end_data) {
                inode = current->executable;       // inode & block no of executable file.
                block = 1 + tmp / BLOCK_SIZE;
        } else {
                inode = NULL;                      // for dynamically applied page.
                block = 0;
        }
// If the process accesses the page of its dynamic application or the page fault exception caused
// by storing the stack information, then we directly apply for a page of physical memory page
// and map it to the linear address 'address'. Otherwise, the missing page is within the scope
// of the process execution file or library file, so try to share the page operation, and exit
// if successful. If it is unsuccessful, you need to apply for a page of physical memory page,
// then read the corresponding page in the execution file from the device and place (map) it
// to the process page logical address 'tmp'.
        if (!inode) {                              // it's a dynamic applied page.
                get_empty_page(address);
                return;
        }
        if (share_page(inode,tmp))                 // try to share the page at 'tmp'.
                return;
        if (!(page = get_free_page()))             // apply for a free page.
                oom();
/* remember that 1 block is used for header */
// According to this block number and the i-node of the execution file, we can find the device
// logic block number (stored in nr[] array) in the corresponding block device from the mapping
// bitmap, and use the bread_page() to read the 4 logical blocks into the physical page.
        for (i=0 ; i<4 ; block++,i++)
                nr[i] = bmap(inode,block);
        bread_page(page,inode->i_dev,nr);

// When reading a device logic block, there may be a situation where the read page position
// may be less than 1 page long from the end of the file. Therefore, some useless information
// may be read. The following operation is to clear this part of the part beyond the execution
// file 'end_data'. Of course, if the page is more than 1 page from the end, it is not read
// from the file in the executable file image, but is read from the library file, so there is
// no need to perform the clear operation.
        i = tmp + 4096 - current->end_data;          // the excess byte length.
        if (i>4095)                                  // more than 1 page from the end.
                i = 0;
        tmp = page + 4096;                           // points to the end of the page.
        while (i-- > 0) {                         // i bytes cleared start from page end.
                tmp--;
                *(char *)tmp = 0;
        }
// Finally, a page causing the page fault exception is mapped to the specified linear address
// address. If the operation is successful, it will return, otherwise the memory page will be
// released, indicating that the memory is not enough.
        if (put_page(page,address))
                return;
        free_page(page);
        oom();
}

//// Memory management initialization.
```

```
// This function initializes the physical memory area above the 1MB address. The kernel manages
// and accesses memory in pages, with a page length of 4KB. This function divides all physical
// memory above 1MB into pages and manages these pages using a page-mapped byte array
// mem_map[].For machines with 16MB of memory, the array has 3840 items ((16MB - 1MB) / 4KB)
// and can manage 3840 physical pages. Whenever a memory page is occupied, the corresponding
// byte item in mem_map[] is incremented by one; if a page is released, the corresponding byte
// value is decremented by one. If the byte item is 0, it indicates that the corresponding page
// is idle; if the byte value is greater than or equal to 1, it indicates that the page is occupied
// or shared by multiple processes.
// Since the kernel buffer cache and some devices need to use a certain amount of memory, the
// amount of memory that the system actually allocates for use is reduced. We refer to the memory
// area that can be actually allocated for use by the kernel as the "main memory area"(MMA),
// and its start position is represented by the variable 'start_mem', and the end address is
// represented by 'end_mem'. For a PC system with 16 MB of memory, 'start_mem' is typically
// 4 MB and 'end_mem' is 16 MB. Therefore, the main memory area is [4MB-16MB] at this time,
// and a total of 3072 physical pages are available for allocation. The range 0 - 1MB memory
// area is reserved for the kernel.
// The parameter 'start_mem' is the starting address of the main memory area that can be used
// for page allocation (the memory space occupied by RAMDISK has been removed). 'end_mem' is
// the actual physical memory maximum address. The address range [start_mem, end_mem] is the
// main memory area.
void mem_init(long start_mem, long end_mem)
{
        int i;

// The function first sets the memory mapped byte array items corresponding to all pages in
// the range of 1MB to 16MB to the occupied state, that is, all bytes are set to USED (100).
// PAGING_PAGES is defined as (PAGING_MEMORY>>12), which is the number of all physical memory
// pages above 1MB (15MB/4KB = 3840).
        HIGH_MEMORY = end_mem;                  // set the memory top (16MB).
        for (i=0 ; i<PAGING_PAGES ; i++)        // PAGING_PAGES = 3840.
                mem_map[i] = USED;
// Then find the item number 'i' in the byte array corresponding to the page at the start address
// 'start_mem', and calculate the number of pages in the main memory area. At this point, the
// i-th item of the mem_map[] corresponds to the first page in the main memory area. Finally,
// the array item corresponding to the pages in the main memory area are cleared (indicating
// idle). For systems with 16MB of physical memory, the bytes corresponding 4MB - 16MB main
// memory area in mem_map[] is cleared.
        i = MAP_NR(start_mem);                // page number at the beginning of the MMA.
        end_mem -= start_mem;
        end_mem >>= 12;                       // total number of pages in MMA.
        while (end_mem-->0)
                mem_map[i++]=0;               // bytes of MMA pages are reset.
}

// Display system memory information.
// The number of memory pages used in the system and the total number of physical memory pages
// in the main memory area are counted according to the information in the mem_map[], and the
// contents of page directory and page table. This function is called on line 186 of the
// chr_drv/keyboard.S program, which displays system memory statistics when the "Shift + Scroll
// Lock" key is pressed.
void show_mem(void)
{
```

```
459         int i,j,k,free=0,total=0;
460         int shared=0;
461         unsigned long * pg_tbl;
462
    // First, according to the byte array mem_map[], we count the total number of pages in the main
    // memory area 'total', and the number of free pages 'free' and the number of shared pages
    // 'shared', and display these information.
463         printk("Mem-info:\n\r");
464         for(i=0 ; i<PAGING_PAGES ; i++) {
465                 if (mem_map[i] == USED)                  // pages not for allocation.
466                         continue;
467                 total++;
468                 if (!mem_map[i])
469                         free++;                          // free pages in the main mem ares.
470                 else
471                         shared += mem_map[i]-1;          // shared pages (byte value > 1).
472         }
473         printk("%d free pages of %d\n\r",free,total);
474         printk("%d pages shared\n\r",shared);

    // Then we count the number of paging management logic pages of the CPU. The first four items
    // in the page directory table are used by the kernel code and are not listed as statistical
    // ranges. The method is to loop through all page directory entries starting with item 5. If
    // the corresponding secondary page table exists, the memory page occupied by the secondary
    // page table itself is first counted (line 484), and then the physical memory page corresponding
    // to all page entries in the page table is counted.
475         k = 0;                                    // statistics of pages occupied by a process.
476         for(i=4 ; i<1024 ;) {
477                 if (1&pg_dir[i]) {
    //(If the address of the corresponding page table in the page directory entry is greater than
    // the highest physical memory address of the machine, HIGH_MEMORY, there is a problem with
    // the directory entry. The directory entry information is displayed and the next directory
    // entry continues to be processed.)
478                         if (pg_dir[i]>HIGH_MEMORY) {          // content is abnormal.
479                                 printk("page directory[%d]: %08X\n\r",
480                                         i,pg_dir[i]);
481                                 continue;                     // needs "i++;" before it.
482                         }
    // If the "address" of the page table in the page directory entry is greater than LOW_MEM (that
    // is, 1MB), the physical memory page statistics value 'k' of one process is incremented by
    // one, and the statistics value 'free' of all physical memory pages occupied by the system
    // is incremented by one. Then take the corresponding page table address 'pg_tbl' and count
    // the contents of all page table items in the page table. If the physical page indicated by
    // the current page table entry exists and the physical page "address" is greater than LOW_MEM,
    // then the corresponding page of the page table entry is included in the statistical value.
483                         if (pg_dir[i]>LOW_MEM)
484                                 free++,k++;             // pages occupied in the page table.
485                         pg_tbl=(unsigned long *) (0xfffff000 & pg_dir[i]);
486                         for(j=0 ; j<1024 ; j++)
487                                 if ((pg_tbl[j]&1) && pg_tbl[j]>LOW_MEM)
    // (If the physical page address is greater than the highest physical memory address of the
    // machine, HIGH_MEMORY, it indicates that there is a problem with the content of the page table
    // item, so the content of the page table item is displayed. Otherwise, the corresponding page
    // of the page table item is included in the statistical value.)
                                            if (pg_tbl[j]>HIGH_MEMORY)
                                                    printk("page_dir[%d][%d]: %08X\n\r",
                                                            i,j, pg_tbl[j]);
                                            else
                                                    k++,free++;   // pages of page table items.
                    }
    // Since the linear space size of each task is 64MB, one task occupies 16 page directory entries.
    // Therefore, every 16 directory entries are counted here, the page table occupied by the process
    // task structure is counted. If k=0 at this time, it means that the process corresponding to
    // the current 16 directory entries does not exist in the system (not created or terminated).
    // Then, after the corresponding process number and the physical page statistics value k are
    // displayed, k is cleared to count the pages occupied by the next process.
                    i++;
                    if (!(i&15) && k) {         // k !=0 indicates that the process exists.
                            k++,free++;         /* one page/process for task_struct */
                            printk("Process %d: %d pages\n\r",(i>>4)-1,k);
                            k = 0;
                    }
            }
    // Finally, the memory page being used in the system and the total number of pages in the main
    // memory area are displayed.
            printk("Memory found: %d (%d)\n\r",free-shared,total);
    }

```

13.3 page.s

13.3.1 Function

The page.s file includes page fault exception interrupt handler (interrupt 14), which is handled primarily in two cases. First, due to page fault exceptions caused by page not present, this needs to be handled by calling do_no_page(error_code, address); second, page exceptions caused by page write protection. At this point, the page write protection handler do_wp_page(error_code, address) is called for processing. The error code (error_code) in the function parameter is automatically generated by the CPU and pushed onto the stack. The linear address (address) accessed when an exception occurs is taken from the control register CR2. CR2 is specifically used to store linear addresses when a page fails.

13.3.2 Code annotation

Program 13-2 linux/mm/page.s

```
/*
 *  linux/mm/page.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
```

```
 * page.s contains the low-level page-exception code.
 * the real work is done in mm.c              // memory.c
 */

// This variable will be used in kernel/traps.c to set the page exception descriptor.
.globl _page_fault          # declared as a global variable.

_page_fault:
        xchgl %eax,(%esp)    # take the error code to EAX.
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%edx      # set the kernel data segment selector.
        mov %dx,%ds
        mov %dx,%es
        mov %dx,%fs
        movl %cr2,%edx       # get the linear address that caused the page exception.
        pushl %edx           # the linear address and error code are pushed onto the stack
        pushl %eax           # as arguments to the function to be called.
// Test the "page present" flag P (bit 0), and call the do_no_page() function if it is an exception
// caused by a page fault. Otherwise, the page write protection function do_wp_page() is called.
        testl $1,%eax        # Test flag P (bit 0), jump if it is set.
        jne 1f
        call _do_no_page     # mm/memory.c, line 381.
        jmp 2f
1:      call _do_wp_page     # mm/memory.c, line 254.
// Discard the two arguments pushed onto the stack, pop the registers and exit the interrupt.
2:      addl $8,%esp
        pop %fs
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret
```

13.4 swap.c

13.4.1 Function

Starting with version 0.12, Linux has added virtual memory swapping functionality to the kernel. This function is mainly implemented by this program. When the physical memory capacity is limited and the usage is tight, the program saves the temporarily unused memory page contents to the disk (switching device) to free up memory space for the programs that are in urgent need. If we later need to use the memory page content already stored on the swap device again in the future, the program is responsible for taking them back and putting them back into memory. Memory swap management uses a mapping technique similar to that of main memory area management, using bit maps to determine the specific save location and map location of the swapped memory pages.

When compiling the kernel, if we defined the swap device number SWAP_DEV, then the compiled kernel has the memory swap functionality. For Linux 0.12, the switching device uses a designated independent partition on the hard disk that does not contain a file system. When the swapping program is initialized, it first reads page 0 on the swap device. This page is the swap area management page, which contains the bitmap used by the swapping page management. The 10 characters starting from the 4068th byte are the swap device feature string "SWAP-SPACE". If the feature string is not on the partition, the given partition is not a valid swap device.

The swap.c program mainly includes swap mapping bitmap management functions and swap device access functions. The get_swap_page() and swap_free() functions are respectively used to apply a swap page and release the specified page in the swap device based on the swapping bitmap; the swap_out() and swap_in() functions are respectively used to output/input the memory page information to/from the swap device. The latter two functions use the read_swap_page() and write_swap_page() functions to access the specified swap device. These two functions are defined in the include/linux/mm.h header file in the form of macros:

```
#define read_swap_page(nr,buffer)    ll_rw_page(READ,SWAP_DEV,(nr),(buffer));
#define write_swap_page(nr,buffer)   ll_rw_page(WRITE,SWAP_DEV,(nr),(buffer));
```

The ll_rw_page() function is a low-level page read and write function of the block device. The code is implemented in the kernel/blk_drv/ll_rw_blk.c file. It can be seen that the swap device access functions are essentially the device page access functions that specifies the device number.

During the kernel initialization process, if the system defines the swap device number SWA_DEV, the kernel executes the swap processing initialization function init_swapping(). This function first checks if the system does have a swap device based on the array of blocks in the system, and if the swap partition of the device is valid. It then requests a memory page and reads the first swap management page (page 0) on the swap partition into the memory page. Page 0 stores the swap page bitmap mapping information, where each bit represents an swap page. If a bit is 0, it indicates that the swap page on the corresponding device has been used (occupied) or unavailable; if the bit is 1, indicating that the corresponding swap page is available. Since a page has a total of SWAP_BITS (4096 X 8 = 32768) bits, the swap partition can manage up to 32,768 pages.

In a running Linux, if a block device partitioner (such as fdisk) initializes the swap partition to have 'swap_size' swap pages, the first page (page 0) of the swap partition on the device will be used for swap management (occupied), so the first bit in the swap bitmap should also be zero. Therefore, the number of swap pages actually available on the device is 'swap_size - 1', and their page number ranges are [1 -- swap_size-1], so their corresponding bits in the bitmap are all 1s (idle). Bits in the range of [swap_size -- SWAP_BITS] in the bitmap are initialized to an unavailable state (all 0s) because there is no corresponding swap page on the device. Therefore, in the initial normal case, the bitmap bit status should be as shown below (the shaded portion is not available for swapping):

```
bits in the bitmap: 0, 1, 1, 1, ... , 1,           0,         ..., 0.
swap page number  : 0, 1, 2, 3, ... , swap_size-1, swap_size, ..., SWAP_BITS-1.
```

13.4.2 Code annotation

Program 13-3 linux/mm/swap.c

```
1  /*
2   *  linux/mm/swap.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   * This file should contain most things doing the swapping from/to disk.
9   * Started 18.12.91
10  */
11
  // <string.h> String header file. Defines some embedded functions about string operations.
  // <linux/mm.h> Memory management header file. Contains page size definitions and some page
  //     release function prototypes.
  // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
  //     of the initial task 0, and some embedded assembly function macro statements about the
   //    descriptor parameter settings and acquisition.
  // <linux/head.h> Head header file. A simple structure for the segment descriptor is defined,
  //     along with several selector constants.
  // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
  //     used functions of the kernel.
12 #include <string.h>
13
14 #include <linux/mm.h>
15 #include <linux/sched.h>
16 #include <linux/head.h>
17 #include <linux/kernel.h>
18
   // Each byte has 8 bits, so a page (4096 bytes) has a total of 32768 bits. If one bit corresponds
   // to one page of memory, the bitmap of one page can manage up to 32,768 pages, corresponding
   // to 128 MB of memory capacity. If the bit is set, it indicates that the corresponding swap
   // page is idle.
19 #define SWAP_BITS (4096<<3)              // define swap total bits in a page (32768).
20
   //// bitop() is a bit manipulation macro. By giving a different "op", we can define three
   // operations for testing, setting, or clearing specified bit.
   // The parameter 'addr' of the inline assembly function specifies the linear address; 'nr' is
   // the bit offset at the specified address. The macro places the value of the 'nr'th bit at
   // the given address 'addr' into the carry flag, sets or resets the bit and returns the carry
   // flag value (ie, returns the original bit value).
   // The first instruction on line 25 combines with the "op" to form different instructions:
   // "op" = "", the instruction bt - Bit test, and sets the carry with the original value.
   // "op" = "s", the instuction bts - Bit test and set, and the original bit is sets to the carry.
   // "op" = "r", instruction btr - Bit test and reset, and the original bit is set to the carry.
   // Input: %0 - (return value), %1 - bit offset (nr); %2 - base address (addr); %3 - plus
   // operation register initial value (0).
   // The inline assembly code saves the bit specified by the base address (%2) and the bit offset
   // (%1) to the carry flag CF, and then sets (resets) the bit. The instruction ADCL is loaded
   // with a carry bit to set the operand (%0) according to the carry bit CF. If CF = 1, the return
   // register value = 1, otherwise the return register value = 0.
   #define bitop(name,op) \
   static inline int name(char * addr,unsigned int nr) \
   { \
   int __res; \
   __asm__ __volatile__("bt" op " %1,%2; adcl $0,%0" \
   :"=g" (__res) \
   :"r" (nr),"m" (*(addr)),"0" (0)); \
   return __res; \
   }

   // Here we define 3 inline functions according to different op characters.
   bitop(bit,"")                  // define  bit(char * addr, unsigned int nr)
   bitop(setbit,"s")              // define  setbit(char * addr, unsigned int nr)
   bitop(clrbit,"r")              // define  clrbit(char * addr, unsigned int nr)

   static char * swap_bitmap = NULL;
   int SWAP_DEV = 0;              // The swap device number set when the kernel is initialized.

   /*
    * We never page the pages in task[0] - kernel memory.
    * We page all other pages.
    */
   // The first virtual memory page, the virtual memory page starting at the end of task 0 (64MB).
   #define FIRST_VM_PAGE (TASK_SIZE>>12)              // = 64MB/4KB = 16384
   #define LAST_VM_PAGE (1024*1024)                   // = 4GB/4KB  = 1048576
   #define VM_PAGES (LAST_VM_PAGE - FIRST_VM_PAGE)    // = 1032192 (count from 0).

   //// Apply and get a swap page number.
   // Scans the entire swapping bitmap (except for bit 0 of the bitmap page itself), resets the
   // first found bit, and returns its position, which is the current free swap page number. Returns
   // the swap page number if the operation is successful, otherwise returns 0.
   static int get_swap_page(void)
   {
           int nr;

           if (!swap_bitmap)
                   return 0;
           for (nr = 1; nr < 32768 ; nr++)
                   if (clrbit(swap_bitmap,nr))
                           return nr;             // return the current free swap page number.
           return 0;
   }

   // Release the sawp page specified in the swap device.
   // Set the bit in the swap bitmap corresponding to the specified page number by the parameter.
   // In a swap bitmap, if a bit is 1, it means that the corresponding swap page is idle. Therefore,
   // if the original bit is equal to 1, it means that the original page of the swap device is
   // not occupied, or the bitmap is in error. Then an error message is displayed and returned.
   void swap_free(int swap_nr)
   {
           if (!swap_nr)
                   return;
   62          if (swap_bitmap && swap_nr < SWAP_BITS)
   63                  if (!setbit(swap_bitmap,swap_nr))
   64                          return;
   65          printk("Swap-space bad (swap_free())\n\r");
   66          return;
   67  }
   68
       // Swap the specified page into memory.
       // The page of the specified page table entry is read from the swap device into the newly
       // requested memory page. At the same time, modify the corresponding bit in the swap bitmap
       // (set), and modify the contents of the page table entry, let it point to the memory page,
       // and set the corresponding flag.
   69  void swap_in(unsigned long *table_ptr)
   70  {
   71          int swap_nr;
   72          unsigned long page;
   73
       // The function first checks the validity of the swap bitmap and parameters. If the swap bitmap
       // does not exist, or the page corresponding to the specified page table entry already exists
       // in the memory, or the swapping page number is 0, a warning message is displayed and exits.
       // For the memory page that has been placed in the swap device, the corresponding page table
       // entry should be the swap page number*2, ie (swap_nr << 1). See the description of line 111
       // in the function try_to_swap_out() below.
   74          if (!swap_bitmap) {
   75                  printk("Trying to swap in without swap bit-map");
   76                  return;
   77          }
   78          if (1 & *table_ptr) {
   79                  printk("trying to swap in present page\n\r");
   80                  return;
   81          }
   82          swap_nr = *table_ptr >> 1;
   83          if (!swap_nr) {
   84                  printk("No swap page in swap_in\n\r");
   85                  return;
   86          }
       // Then apply for a memory page and read the page with the page number swap_nr from the swap
       // device. After the page is swapped in using the read_swap_page(), the corresponding bit in
       // the swap bitmap is set. Ifit is originally set, it means that the same page is read again
       // from the switching device, so the warning message is displayed. Finally, let the page table
       // entry point to the physical page, and set the page modified, user readable and writable and
       // presence flags (Dirty, U/S, R/W, P).
   87          if (!(page = get_free_page()))
   88                  oom();
   89          read_swap_page(swap_nr, (char *) page);       // defined in file nclude/linux/mm.h
   90          if (setbit(swap_bitmap,swap_nr))
   91                  printk("swapping in multiply from same page\n\r");
   92          *table_ptr = page | (PAGE_DIRTY | 7);
   93  }
   94
       // Try to swap out the page.
       // If the memory page has not been modified, it does not need to be saved in the swap device,
       // because the corresponding page can also be read directly from the corresponding image file.
       // So you can directly release the corresponding physical page. Otherwise, apply for an swap
       // page number and then swap out the page. At this time, the swap page number is to be saved
       // in the corresponding page table entry, and it is still necessary to keep the page table entry
       // bit P = 0.  The parameter 'table_ptr' is a pointer to a page table entry. Returns 1 if the
       // page is swapped or released successfully, otherwise returns 0.
       int try_to_swap_out(unsigned long * table_ptr)
       {
               unsigned long page;
               unsigned long swap_nr;

       // The function first determines the validity of the parameter. If the memory page that needs
       // to be swapped out does not exist (or is invalid), the code exits; If the physical page address
       // specified by the page table entry is greater than the memory-managed high-end PAGING_MEMORY
       // (15MB), it also exits.
               page = *table_ptr;
               if (!(PAGE_PRESENT & page))
                       return 0;
               if (page - LOW_MEM > PAGING_MEMORY)
                       return 0;
       // If the memory page has been modified, but the page is shared, then in order to improve the
       // efficiency of the operation, such pages should not be swapped out, so the function returns
       // 0 and exists. Otherwise we get a swap page number and save it in the page table entry, then
       // swap the page out and release the corresponding physical memory page.
               if (PAGE_DIRTY & page) {
                       page &= 0xfffff000;                     // get physical page address.
                       if (mem_map[MAP_NR(page)] != 1)
                               return 0;
                       if (!(swap_nr = get_swap_page()))       // get a swap page number.
                               return 0;
       // For pages to be swapped, the swap page number (swap_nr << 1) is stored in the corresponding
       // page table entry. The reason of multiplying 2 is to free the presence bit (P, bit 0) of the
       // page table entry. Only pages with a bit P=0 and a page table entry content other than 0 will
       // be in the swap device. The Intel manual clearly states that when a table entry has a
       // P = 0 (invalid page), all other bits (bits 31 - 1) are free to use. The write page function
       // write_swap_page(nr, buffer) is defined as ll_rw_page(WRITE, SWAP_DEV, (nr), (buffer)), see
       // line 12 of the linux/mm.h file.
                       *table_ptr = swap_nr<<1;
                       invalidate();                          // refresh the CPU transform cache.
                       write_swap_page(swap_nr, (char *) page);
                       free_page(page);
                       return 1;
               }
       // Otherwise, it means that the page has not been modified, so you don't have to swap it out,
       // but you can release it directly.
               *table_ptr = 0;
               invalidate();
               free_page(page);
               return 1;
       }

       /*
        * Ok, this has a rather intricate logic - the idea is to make good
        * and fast machine code. If we didn't worry about that, things would
```

```
126  * be easier.
127  */
    // Put the memory page in the swap device.
    // Starting from the page directory entry (FIRST_VM_PAGE>>10) corresponding to the linear
    // address 64MB, the entire 4GB linear space is searched. During this time we tried to swap
    // the corresponding memory page to the swap device. Returns 1 if the page is successfully swapped
    // out, otherwise returns 0. The two static local variables are used to temporily stored the
    // current search point for the beginning of the next searching. This function will be called
    // in get_free_page() (defined at line 172).
128 int swap_out(void)
129 {
130         static int dir_entry = FIRST_VM_PAGE>>10;  // 16, first directory entry of task 1.
131         static int page_entry = -1;
132         int counter = VM_PAGES;                    // 1032192, see line 44.
133         int pg_table;
134 
    // First we loop through the page directory table to find the page directory entry pg_table
    // containing the valid secondary page table, and exit the loop if found., Otherwise, we adjust
    // the number of remaining secondary page tables 'counter' of the directory entry, and then
    // continue to check the next directory entry. If the appropriate (existing) page directory
    // entry has not been found after all searches, the code returns with 0.
135         while (counter>0) {
136                 pg_table = pg_dir[dir_entry];         // content of the directory entry.
137                 if (pg_table & 1)                     // exit the loop if the table exists.
138                         break;
139                 counter -= 1024;                      // One page table has 1024 items.
140                 dir_entry++;                          // next directory entry.
141                 if (dir_entry >= 1024)
142                         dir_entry = FIRST_VM_PAGE>>10;
143         }
    // After getting the page table pointer in the current directory entry, the swap function
    // try_to_swap_out() is called one by one for all 1024 pages in the page table to try to swap
    // out it. Returns 1 if a page is successfully swapped out to the swap device. If all page tables
   // for all directory entries have failed, the warning message is displayed and returns 0.
144         pg_table &= 0xfffff000;                        // page table pointer (address).
145         while (counter-- > 0) {
146                 page_entry++;                          // Page table entry (initially -1).
    // If we have tried to process all the items in the current page table, but still can not
    // successfully find a page that can be swapped out, that is, the page table item index number
    // is greater than or equal to 1024, then use the same way as the previous 135 - 143 lines To
    // select the secondary page table in the next page directory entry.
147                 if (page_entry >= 1024) {
148                         page_entry = 0;
149                 repeat:
150                         dir_entry++;
151                         if (dir_entry >= 1024)
152                                 dir_entry = FIRST_VM_PAGE>>10;
153                         pg_table = pg_dir[dir_entry];  // content of page directory entry.
154                         if (!(pg_table&1))
155                                 if ((counter -= 1024) > 0)
156                                         goto repeat;
157                                 else
158                                         break;
159                         pg_table &= 0xfffff000;        // get page table ponter.
160                 }
161                 if (try_to_swap_out(page_entry + (unsigned long *) pg_table))
162                         return 1;
163         }
164         printk("Out of swap-memory\n\r");
165         return 0;
166 }
167
168 /*
169  * Get physical address of first (actually last :-) free page, and mark it
170  * used. If no free pages left, return 0.
171  */
    //// Get a free physical page in the main memory area.
    // If there is no physical memory page available, then we perform page swap processing and then
    // apply for and try to get the page again.
    // Input: %1(ax = 0) ; %2(LOW_MEM) Memory start position managed by byte bitmap;
    //  %3(cx = PAGING_PAGES); %4(edi = mem_map + PAGING_PAGES - 1).
    // The function returns the address of the new page in %0(ax = physical page start address).
    // The above %4 register actually points to the last byte of the memory byte bitmap mem_map[]. This
    // function scans all page flags backwards from the end of the bitmap (the total number of pages
    // is PAGING_PAGES), and returns the page address if the page is free (bitmap byte is 0).
    // NOTE! This function simply points out a free physical page in the main memory area, but it
    // is not mapped to the address space of any process. The put_page() function in the memory.c
    // program is used to map a specified page into the address space of a process. Of course, using
    // this function for the kernel does not require the use of put_page() for mapping because the
    // kernel code and data space (16MB) are mapped to the physical address space peer-to-peer.
    // Line 174 defines a local register variable. This variable will be saved in the eax register
    // for efficient access and operation. This method of defining variables is mainly used in inline
    // assembly files.
172 unsigned long get_free_page(void)
173 {
174 register unsigned long __res asm("ax");
175
    // First we look up the byte with a value 0 in the memory byte bitmap and then clear the
    // corresponding physical memory page. If the resulting page address is larger than the actual
    // physical memory capacity, look for it again. If no free page is found, perform the swap
    // operation and then look it up again. Finally, the free physical page address is returned.
176 repeat:
    // Find a free page uing the memory byte bitmap mem_map[].
177 __asm__("std ; repne ; scasb\n\t"      // al(0) compared with content of each page (di).
178         "jne 1f\n\t"                   // jump to label 1 if none of it equals 0.
179         "movb $1,1(%%edi)\n\t"         // set byte [1+edi] to 1 of the page.
180         "sall $12,%%ecx\n\t"           // pages * 4K = relative address of the page.
181         "addl %2,%%ecx\n\t"            // plus LOW_MEM to get the absolute page address.
    // Zero the memory page.
182         "movl %%ecx,%%edx\n\t"         // load edx register with page start address.
183         "movl $1024,%%ecx\n\t"         // load ecx with counting number 1024.
184         "leal 4092(%%edx),%%edi\n\t"   // load edi with page end address (4092 + edx).
185         "rep ; stosl\n\t"              // reset (clear) each byte in the page.
186         "movl %%edx,%%eax\n"           // load eax with the page start address.
187         "1:"
188         :"=a" (__res)
```

```
        :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
        "D" (mem_map+PAGING_PAGES-1)
        :"di","cx","dx");
        if (__res >= HIGH_MEMORY)   // search again if the page is out of main page area.
                goto repeat;
        if (!__res && swap_out())   // do swapping if no free page found and search again.
                goto repeat;
        return __res;               // return the address of the free page.
}

// Memory page swapping initialization.
// The function first checks if the device has a swap partition based on the device's partition
// array (array of blocks) and checks if the swap partition is valid. Then apply to get a page
// of memory to store swap page bitmap array swap_bitmap[], and read the swap management page
// (the first page) from the device's swap partition into the swap bitmap array. Then carefully
// check whether each bit in the swap bitmap array and finally return.
void init_swapping(void)
{
// Blk_size[] is a pointer array to the number of blocks of the block device specified by the
// major device number. Each of its items corresponds to the total number of data blocks owned
// by one sub-device, and each sub-device corresponds to one partition of the block device.
// If no swap device number is defined in the system, the code returns; if the swap device does
// not set an array of blocks, a warning message is displayed and returned.
        extern int *blk_size[];              // defined in blk_drv/ll_rw_blk.c, line 49.
        int swap_size,i,j;

        if (!SWAP_DEV)
                return;
        if (!blk_size[MAJOR(SWAP_DEV)]) {
                printk("Unable to get size of swap device\n\r");
                return;
        }
// Then get and check the total number of blocks 'swap_size' in the swap partition of the swap
// device. If it is 0, it will return; if the total number of blocks is less than 100, a warning
// message will be displayed and then exit.
        swap_size = blk_size[MAJOR(SWAP_DEV)][MINOR(SWAP_DEV)];
        if (!swap_size)
                return;
        if (swap_size < 100) {
                printk("Swap device too small (%d blocks)\n\r",swap_size);
                return;
        }
// Then we convert the total number of swap blocks into the corresponding total number of swap
// pages (blocks/4). This value cannot be greater than the number of pages that SWAP_BITS can
// represent (32768) . Then get a free memory page to store the swap bitmap array 'swap_bitmap',
// where each bit represents one swap page.
        swap_size >>= 2;
        if (swap_size > SWAP_BITS)
                swap_size = SWAP_BITS;
        swap_bitmap = (char *) get_free_page();
        if (!swap_bitmap) {
                printk("Unable to start swapping: out of memory :-)\n\r");
                return;
        }
// Then we read page 0 on the device swap partition into the swap_bitmap page, which is the
// swap area management page. Among them, the beginning of the 4086th byte contains the swap
// device feature string "SWAP-SPACE". If the feature string is not found, it is not a valid
// swap device. So we display a warning message, release the memory page we just got and exit
// the function. Otherwise, the feature string (10 bytes) in the memory page is cleared.
// The macro read_swap_page(nr, buffer) is defined in file include/linux/mm.h, line 11.
        read_swap_page(0,swap_bitmap);
        if (strncmp("SWAP-SPACE",swap_bitmap+4086,10)) {
                printk("Unable to find swap-space signature\n\r");
                free_page((long) swap_bitmap);
                swap_bitmap = NULL;
                return;
        }
        memset(swap_bitmap+4086,0,10);
// Then we check the read swap bitmap, which contains a total of 32,768 bits. If the bit of
// the bitmap is 0, it means that the corresponding swap page on the device is used (occupied).
// If the bit is 1, it indicates that the corresponding swap page is available (idle). So for
// the swap partition on the device, the first page (page 0) is used for swap management and
// is already occupied (bit 0 is 0). The swap page [1 -- swap_size-1] is available, so their
// corresponding bits in the bitmap should be 1 (idle). Bits in the [swap_size -- SWAP_BITS]
// range in the bitmap should also be initialized to 0 (occupied) because there is no
// corresponding swap pages. Below, when checking the bitmap, the bitmap is checked in two steps
// based on the unavailable and available parts.
// First check the bitmap bits of the unavailable swap page, they should all be 0 (occupied).
// If any of these bits is 1 (idle), there is a problem with the bitmap. The error message is
// then displayed, the page occupied by the bitmap is released, and the function is exited.
        for (i = 0 ; i < SWAP_BITS ; i++) {
                if (i == 1)
                        i = swap_size;
                if (bit(swap_bitmap,i)) {
                        printk("Bad swap-space bit-map\n\r");
                        free_page((long) swap_bitmap);
                        swap_bitmap = NULL;
                        return;
                }
        }
// Then check and count whether all bits between [1 to swap_size-1] are 1s (idle). If the
// statistics show that there is no free swap page, it means that there is a problem with the
// swap function, so the page occupied by the bitmap is released and the function is exited.
// Otherwise, the swap device works ok and the number of swap pages and the total bytes of swap
// space are displayed.
        j = 0;
        for (i = 1 ; i < swap_size ; i++)
                if (bit(swap_bitmap,i))
                        j++;
        if (!j) {
                free_page((long) swap_bitmap);
                swap_bitmap = NULL;
                return;
        }
        printk("Swap device ok: %d pages (%d bytes) swap-space\n\r",j,j*4096);
}
```

254

13.5 Summary

This chapter describes how the kernel manages and accesses physical and virtual memory in the system. It focuses on the memory allocation management mechanism of Intel CPU, and how the Linux kernel divides the memory space. At the same time, it gives the principle of copy-on-write mechanism and demand loading mechanism. Finally, we illustrate and describe the bitmap-swap page management and processing mechanism of the swap partition on the device.

In the next chapter, we fully describe all the header files included in the Linux kernel source code, including important data structures and macro definitions in almost all kernel code.

14 Header Files (include)

The program should declare the function first before using it. For ease of use, it is common practice to put the same type of functions or data structures and declarations of constants in a header file. Any related type definitions and macro definitions can also be included in the header file. The preprocessor directive "#include" is used in the program source file to reference the relevant header file.

A control line statement as shown below in the program will cause the line to be replaced by the contents of the file 'filename':

```
# include <filename>
```

Of course, the filename 'filename' cannot contain > and newline characters, as well as the ", ', \, or /* characters. The compiler will search for this file in a set of pre-set places. Similarly, the following form of control line will cause the compiler to first search for the 'filename' file in the directory where the source program is located:

```
# include "filename"
```

If the file is not found, the compiler will perform the same search process as above. In this form, the file name 'filename' also cannot contain newline characters and ", ', \, or /* characters, but the > character is allowed.

In general application source code, the header files are inextricably linked to the library files in the development environment. Each function in the library needs to be declared in the relevant header file. The header files in the application development environment (usually placed in the /usr/include/ directory) can be thought of as an integral part of the functions in the libraries they provide (eg libc.a), and are instructions or interface statements for library functions. After the compiler converts the source code program into an object module, the linker combines all the object modules of the program, including the modules in any library files used, to form an executable program.

For the standard C library, there are about 15 basic header files. Each header file represents a functional description or structure definition of a particular class of functions, such as I/O manipulation functions, character handling functions, and so on. A detailed description of the standard library and its implementation can be found in the book "The Standard C Library" by Mr. Plauger.

For the kernel source code described in this book, the header files involved can be seen as a summary of the services provided by the kernel and its libraries, and are the header files specific to the kernel and its related programs. In these header files, all the data structures, initialization data, constants, and macro definitions used by the kernel are mainly described, as well as a small amount of program code. In addition to several specialized header files (such as the block device header file blk.h), the header files used in the Linux 0.12 kernel are placed in the include/ directory of the kernel source tree. Therefore, compiling this Linux kernel does not require the use of any header files located in the /usr/include/ directory provided by the development environment. Of course, except for the tools/build.c program. Because although this program is included in the kernel source tree, it is just a utility or application for creating a kernel image file that will not be linked into the kernel code.

Starting with kernel version 0.95, the header files in the kernel source tree need to be copied to the /usr/include/linux directory to compile the kernel smoothly. That is, starting with this version of the kernel, the header files have been merged with the header files used by the Linux development environment.

14.1 Files in the include/ directory

The header files used by the kernel are stored in the include/ directory of the kernel source tree. The files in this directory are shown in Listing 14-1. One point to note here is that for ease of use and compatibility, Linus uses a similar naming convention for standard kernel C header files when compiling kernel program header files. The names of many header files, and even some of the contents of the files, are basically the same as those of the standard C library. But these header files are still kernel-specific or program-specific that is closely tied to the kernel. On a Linux system, they coexist with the header files of the standard library. The usual practice is to place these header files in a subdirectory of the standard library header file directory for use by programs that require kernel data structures or constants.

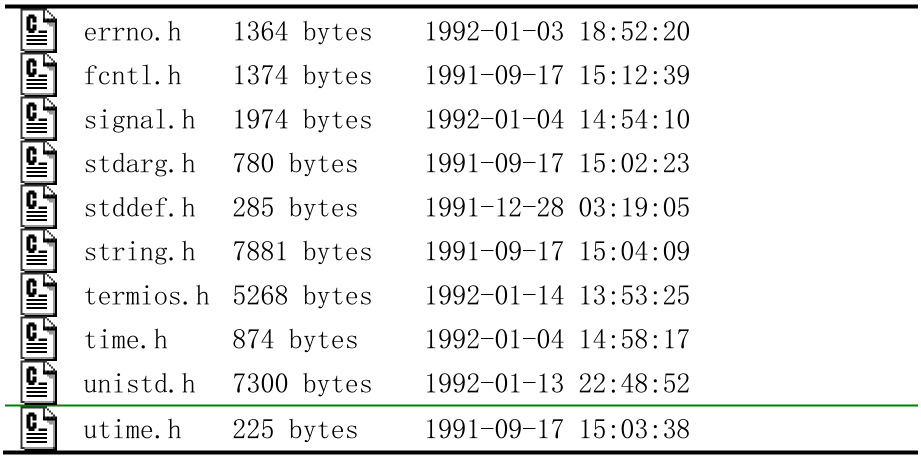
In addition, due to copyright issues, Linus also attempted to rewrite some header files to replace the header files of the standard C library with copyright restrictions. So the header files in these kernel sources have some overlap with the header files in the development environment. In Linux systems, for application development, the kernel header files in the asm/, linux/, and sys/ subdirectories in Listing 14-1 usually need to be copied to the directory where the standard C library header files are located (/usr/include), while other files do not conflict with the standard library header files, you can directly put them in the standard library header file directory, or change them to the three subdirectories here.

The asm/ directory is primarily used to store header files for function declarations or data structures that are closely related to the computer architecture being used. For example, the Intel CPU port IO assembly macro file io.h, the interrupt descriptor set assembly macro header file system.h, and so on. The linux/ directory contains some header files used by the Linux kernel program, including the header file sched.h used by the scheduler, the memory management header file mm.h, and the terminal management data structure file tty.h. The sys/ directory contains several header files related to kernel resources. The sys/ directory stores several header files related to kernel resources. However, starting from version 0.98, the header files in the sys/ directory under the kernel directory tree are all moved to the linux/ directory.

There are 36 header files (*.h) in the Linux 0.12 kernel, including 4 in the asm/ subdirectory, 11 in the linux/subdirectory, and 8 in the sys/ subdirectory. Starting from the next section, we first describe the 13 header files in the include/ directory, and then describe the files in each subdirectory in turn. The order of description is sorted by file name.

List 14-1 Files in the linux/include/ directory

	Filename	Size	Last Modified Time	Description
📁	asm/		1992-01-09 16:46:04	
📁	linux/		1992-01-12 19:43:55	
📁	sys/		1992-01-09 16:46:03	
📄	a.out.h	6047 bytes	1991-09-17 15:10:49	
📄	const.h	321 bytes	1991-09-17 15:12:39	
📄	ctype.h	1049 bytes	1991-11-07 17:30:47	



errno.h	1364 bytes	1992-01-03 18:52:20
fcntl.h	1374 bytes	1991-09-17 15:12:39
signal.h	1974 bytes	1992-01-04 14:54:10
stdarg.h	780 bytes	1991-09-17 15:02:23
stddef.h	285 bytes	1991-12-28 03:19:05
string.h	7881 bytes	1991-09-17 15:04:09
termios.h	5268 bytes	1992-01-14 13:53:25
time.h	874 bytes	1992-01-04 14:58:17
unistd.h	7300 bytes	1992-01-13 22:48:52
utime.h	225 bytes	1991-09-17 15:03:38

14.2 a.out.h

14.2.1 Function

In the Linux kernel, the a.out.h file is used to define the executable file structure loaded in a.out format, mainly used in executable file loader program fs/exec.c. This file is not part of the standard C library, but a kernel-specific header file. However, since there is no conflict with the header file name of the standard library, it can generally be placed in the /usr/include/ directory on Linux systems for use by programs that involve related content. This header file defines an a.out (Assembley out) format for the object file. This object file format is used for .o files and executables used in Linux 0.12 systems.

The a.out.h file contains three data structure definitions and some related macro definitions, so the file can be divided into three parts accordingly:

- Lines 1-108 give and describe the object header structure and associated macro definitions;
- Lines 109-185 are definitions and descriptions of the structure of symbol entries;
- Lines 186-217 define and describe the structure of the relocation table entry.

Due to the large amount of content in the file, a detailed description of three data structures and associated macro definitions is placed after the program list. Here is a brief introduction to the exec structure in the a.out format file.

An a.out executable file consists of an exec header section at the beginning and subsequent code, data, and other parts. The header part of a executable file mainly contains an exec structure, which contains information such as the magic number field, code and data length, symbol table length, and code execution start position. The kernel uses this information to load the executable into memory and execute it, and the linker (ld) uses these parameters to combine some of the module files into one executable. This is the only necessary component of a object file.

Starting with the 0.96 kernel, the Linux system directly uses GNU's header file a.out.h. As a result, programs compiled under Linux 0.9x cannot run on Linux 0.1x systems. Below we analyze the differences between the two a.out header files, and explain how to make some executable files compiled under 0.9x without using dynamic link library can also run under 0.1x.

The main difference between the a.out.h file used by Linux 0.12 and the GNU file of the same name is the first field a_magic of the exec structure. The file field name of GNU is a_info, and the field is further divided into three sub-domains: Flags, Machine Type, and Magic Number. At the same time, the corresponding macros N_MACHTYPE and N_FLAGS are defined for the machine type field, as shown in Figure 14-1.

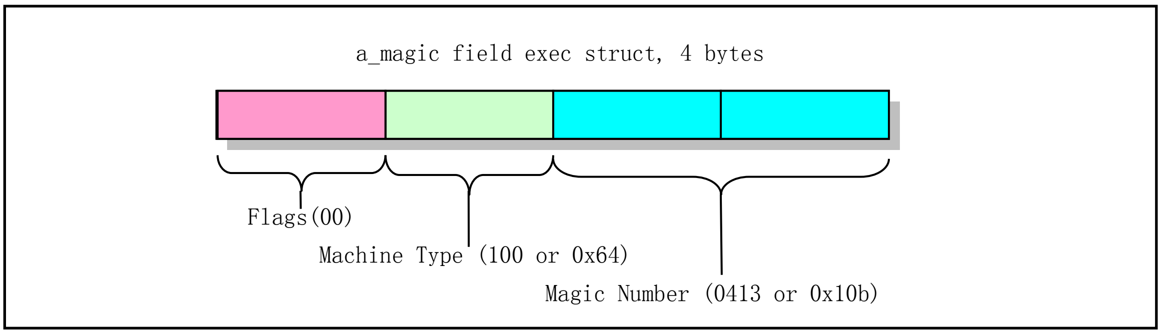

a_magic field exec struct, 4 bytes
Flags(00)
Machine Type (100 or 0x64)
Magic Number (0413 or 0x10b)

Figure 14-1 The first field in the exec structure a_magic(a_info)

In the Linux 0.9x system, for executable files linked with static libraries, the values in parentheses in the figure are the default values for each field. The 4 bytes at the beginning of this binary executable file are:

```
0x0b, 0x01, 0x64, 0x00
```

The a.out header file in this kernel only defines the magic number field. Therefore, the first 4 bytes of a binary executable file in a.out format on a Linux 0.1x system are:

```
0x0b, 0x01, 0x00, 0x00
```

It can be seen that the difference between the executable file in the a.out format of GNU and the executable file compiled on the Linux 0.1x system is only in the machine type field. So we can clear the machine type field (3rd byte) of the a.out format executable file on Linux 0.9x and run it on the 0.1x system. As long as the system-call invoked by the ported executable file is already implemented in the 0.1x system. The author used this approach when starting to rebuild many of the commands in the Linux 0.1x root file system. In other respects, GNU's a.out.h header file is no different from a.out.h here.

14.2.2 Code annotation

Program 14-1 linux/include/a.out.h

```
1 #ifndef _A_OUT_H
2 #define _A_OUT_H
3
4 #define __GNU_EXEC_MACROS__
5
  // Lines 6-108 are the first part of the document, which mainly defines the execution structure
  // of the object file and the macros of related operations.
  // Below is the object file header structure, see the detailed description after the program.
6 struct exec {
7   unsigned long a_magic;        /* Use macros N_MAGIC, etc for access */
8   unsigned a_text;              /* length of text, in bytes */
```

```
  unsigned a_data;            /* length of data, in bytes */
  unsigned a_bss;             /* length of uninitialized data area for file, in bytes */
  unsigned a_syms;            /* length of symbol table data in file, in bytes */
  unsigned a_entry;           /* start address */
  unsigned a_trsize;          /* length of relocation info for text, in bytes */
  unsigned a_drsize;          /* length of relocation info for data, in bytes */
};

// Macro definition, used to take the magic number in the above exec structure.
#ifndef N_MAGIC
#define N_MAGIC(exec) ((exec).a_magic)
#endif

#ifndef OMAGIC
/* Code indicating object file or impure executable.  */
// Historically, on the PDP-11 computer, the magic number (magic number) was octal number 0407
// (0x107). Historically, on the PDP-11 computer, the magic number (magic number) was octal
// number 0407 (0x107), which was at the beginning of the executable file header structure.
// It was originally a jump instruction of the PDP-11, indicating that it jumped to the beginning
// of the code after the next 7 words. In this way, the loader can jump directly to the beginning
// of the instruction after putting the executable file into memory. There is no program to
// use this method, but this octal number is retained as a flag (magic number) for identifying
// the file type. 'OMAGIC' can be considered as 'Old Magic'.
#define OMAGIC 0407
/* Code indicating pure executable.  */
// NMAGIC (New Magic), used after 1975. It involves the virtual storage mechanism.
#define NMAGIC 0410                // 0410 == 0x108
/* Code indicating demand-paged executable.  */
// This type of header structure occupies 1KB of space at the beginning of the file.
#define ZMAGIC 0413                // 0413 == 0x10b
#endif /* not OMAGIC */
// There is also a QMAGIC, which is used to save disk capacity and store the header structure
// and code of the file on the disk in a compact manner.
// The following macro (Bad Magic) is used to determine the correctness of the magic number
// field. Returns true if the magic number cannot be recognized.
#ifndef N_BADMAG
#define N_BADMAG(x)                                         \
 (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC              \
  && N_MAGIC(x) != ZMAGIC)
#endif

#define _N_BADMAG(x)                                        \
 (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC              \
  && N_MAGIC(x) != ZMAGIC)

// A macro definition that gives the remaining length between the end of the header structure
// and the 1KB position.
#define _N_HDROFF(x) (SEGMENT_SIZE - sizeof (struct exec))

// The following macros are used to manipulate the contents of the object file, including .o
// module files and executable files.

// The starting offset of the code portion.
// If the file is of type ZMAGIC, ie an executable file, the code portion begins at a 1024-byte
// offset from the execution file; otherwise the code portion begins with the end of the execution
// header structure (32 bytes), ie the file is a module File (OMAGIC type).
#ifndef N_TXTOFF
#define N_TXTOFF(x) \
 (N_MAGIC(x) == ZMAGIC ? _N_HDROFF((x)) + sizeof (struct exec) : sizeof (struct exec))
#endif

// The data part start offset, starting at the end of the code section.
#ifndef N_DATOFF
#define N_DATOFF(x) (N_TXTOFF(x) + (x).a_text)
#endif

// The code relocation info offset, starting at the end of the data section.
#ifndef N_TRELOFF
#define N_TRELOFF(x) (N_DATOFF(x) + (x).a_data)
#endif

// The data relocation info offset, starting at the end of the code relocation info.
#ifndef N_DRELOFF
#define N_DRELOFF(x) (N_TRELOFF(x) + (x).a_trsize)
#endif

// The symbol table offset, start at the end of the data segment relocation table.
#ifndef N_SYMOFF
#define N_SYMOFF(x) (N_DRELOFF(x) + (x).a_drsize)
#endif

// The string information offset, after the symbol table.
#ifndef N_STROFF
#define N_STROFF(x) (N_SYMOFF(x) + (x).a_syms)
#endif

// The following is location operation where a executable is loaded into logical space.
/* Address of text segment in memory after it is loaded.  */
#ifndef N_TXTADDR
#define N_TXTADDR(x) 0                        // The code segment begings at address 0.
#endif

/* Address of data segment in memory after it is loaded.
   Note that it is up to you to define SEGMENT_SIZE
   on machines not listed here.  */
#if defined(vax) || defined(hp300) || defined(pyr)
#define SEGMENT_SIZE PAGE_SIZE
#endif
#ifdef  hp300
#define PAGE_SIZE       4096
#endif
#ifdef  sony
#define SEGMENT_SIZE    0x2000
#endif  /* Sony.  */
#ifdef is68k
#define SEGMENT_SIZE 0x20000
```

```
 86 #endif
 87 #if defined(m68k) && defined(PORTAR)
 88 #define PAGE_SIZE 0x400
 89 #define SEGMENT_SIZE PAGE_SIZE
 90 #endif
 91
    // Here, the kernel defines the memory page as 4KB, and the segment size is defined as 1KB,
    // so the above definition is not used.
 92 #define PAGE_SIZE 4096
 93 #define SEGMENT_SIZE 1024
 94
    // The size defined by the segment (considering the carry).
 95 #define _N_SEGMENT_ROUND(x) (((x) + SEGMENT_SIZE - 1) & ~(SEGMENT_SIZE - 1))
 96
    // Code segment end address.
 97 #define _N_TXTENDADDR(x) (N_TXTADDR(x)+(x).a_text)
 98
    // Data segment start address.
    // If the file is of the OMAGIC type, the data segment immediately follows the code segment.
    // Otherwise the data segment address starts from the segment boundary after the code segment
    // (1KB boundary alignment), for example for a ZMAGIC type file.
 99 #ifndef N_DATADDR
100 #define N_DATADDR(x) \
101     (N_MAGIC(x)==OMAGIC? (_N_TXTENDADDR(x)) \
102      : (_N_SEGMENT_ROUND (_N_TXTENDADDR(x))))
103 #endif
104
105 /* Address of bss segment in memory after it is loaded.  */
    // The uninitialized data segment bbs is located and follows the data segment.
106 #ifndef N_BSSADDR
107 #define N_BSSADDR(x) (N_DATADDR(x) + (x).a_data)
108 #endif
109
    // Lines 110--185 are part 2. It describes the symbol table entries in the object file and defines
    // related operation macros. See the detailed instructions after the program listing.
    // The symbol table entry (record) structure in the a.out object file.
110 #ifndef N_NLIST_DECLARED
111 struct nlist {
112   union {
113     char *n_name;
114     struct nlist *n_next;
115     long n_strx;
116   } n_un;
117   unsigned char n_type;               // The byte is divided into 3 fields, and
118   char n_other;                       // lines 146-154 are the mask code for each field.
119   short n_desc;
120   unsigned long n_value;
121 };
122 #endif
123
    // The constants for the n_type field in the nlist structure are defined below.
124 #ifndef N_UNDF
125 #define N_UNDF 0
```

```
#endif
#ifndef N_ABS
#define N_ABS 2
#endif
#ifndef N_TEXT
#define N_TEXT 4
#endif
#ifndef N_DATA
#define N_DATA 6
#endif
#ifndef N_BSS
#define N_BSS 8
#endif
#ifndef N_COMM
#define N_COMM 18
#endif
#ifndef N_FN
#define N_FN 15
#endif

// The following 3 constants are the mask (octal) of the n_type field in the nlist structure.
#ifndef N_EXT
#define N_EXT 1              // 0x01 (0b0000,0001) symbol is external (global) ?.
#endif
#ifndef N_TYPE
#define N_TYPE 036           // 0x1e (0b0001,1110) The type bits of the symbol.
#endif
#ifndef N_STAB               // STAB -- Symbol table types.
#define N_STAB 0340          // 0xe0 (0b1110,0000) These bits are used for symbol debugger.
#endif

/* The following type indicates the definition of a symbol as being
   an indirect reference to another symbol.  The other symbol
   appears as an undefined reference, immediately following this symbol.

   Indirection is asymmetrical.  The other symbol's value will be used
   to satisfy requests for the indirect symbol, but not vice versa.
   If the other symbol does not have a definition, libraries will
   be searched to find a definition.  */
#define N_INDR 0xa

/* The following symbols refer to set elements.
   All the N_SET[ATDB] symbols with the same name form one set.
   Space is allocated for the set in the text section, and each set
   element's value is stored into one word of the space.
   The first word of the space is the length of the set (number of elements).

   The address of the set is made into an N_SETV symbol
   whose name is the same as the name of the set.
   This symbol acts like a N_DATA global symbol
   in that it can satisfy undefined external references.  */

/* These appear as input to LD, in a .o file.  */
```

```
#define N_SETA  0x14        /* Absolute set element symbol */
#define N_SETT  0x16        /* Text set element symbol */
#define N_SETD  0x18        /* Data set element symbol */
#define N_SETB  0x1A        /* Bss set element symbol */

/* This is output from LD.  */
#define N_SETV  0x1C        /* Pointer to set vector in data area.  */

#ifndef N_RELOCATION_INFO_DECLARED

/* This structure describes a single relocation to be performed.
   The text-relocation section of the file is a vector of these structures,
   all of which apply to the text section.
   Likewise, the data-relocation section applies to the data section.  */

// The code and data relocation info structure in a object file of a.out.
struct relocation_info
{
  /* Address (within segment) to be relocated.  */
  int r_address;
  /* The meaning of r_symbolnum depends on r_extern.  */
  unsigned int r_symbolnum:24;
  /* Nonzero means value is a pc-relative offset
     and it should be relocated for changes in its own address
     as well as for changes in the symbol or section specified.  */
  unsigned int r_pcrel:1;
  /* Length (as exponent of 2) of the field to be relocated.
     Thus, a value of 2 indicates 1<<2 bytes.  */
  unsigned int r_length:2;
  /* 1 => relocate with value of symbol.
          r_symbolnum is the index of the symbol
          in file's the symbol table.
     0 => relocate with the address of a segment.
          r_symbolnum is N_TEXT, N_DATA, N_BSS or N_ABS
          (the N_EXT bit may be set also, but signifies nothing).  */
  unsigned int r_extern:1;
  /* Four bits that aren't used, but when writing an object file
     it is desirable to clear them.  */
  unsigned int r_pad:4;
};
#endif /* no N_RELOCATION_INFO_DECLARED.  */


#endif /* __A_OUT_GNU_H__ */

```

14.2.3 Information

14.2.3.1 a.out executable file format

The Linux kernel version 0.12 only supports the format of a.out (Assembler output) executable files and object files. Although this format has been gradually not used, but the ELF (Executable and Link Format) format is more fully used, due to its simplicity, it is suitable as a material for learning. Below we give a comprehensive introduction to the a.out format.

Three data structures and some macros are declared in the header file a.out.h. These data structures describe the structure of the object files on the system. In the Linux 0.12 system, the compiled object module file (referred to as the module file) and the binary executable file generated by the linker are in the a.out format. Here we collectively refer to them as object files. An object file can consist of up to seven parts (sections). They are in order:

a) **exec header --** This section contains some parameters (exec structure) that the kernel uses to load the executable file into memory and execute it, and the linker (ld) uses these parameters to combine some of the module files into one executable file. This is the only necessary component of the object file.
b) **text segment --** Contains the instruction code (text) and related data that is loaded into memory when the program is executed. It can be loaded in read-only form.
c) **data segment --** This section contains data that has already been initialized and is always loaded into readable and writable memory.
d) **text relocations --** This section contains records data for use by the linker. Used to locate and update a pointer or address in a code segment when combining object module files.
e) **data relocation --** Similar to the role of the code relocation section, but for the relocation of pointers in the data segment.
f) **symbol table --** This section also contains records data for use by the linker to cross-reference named variables and functions (symbols) between binary object module files.
g) **string table --** This part contains character strings corresponding to the symbol names.

Each object or binary file begins with an execution data structure (exec structure) in the form of:

```
struct exec {
  unsigned long a_magic        // Use macros for access, such as N_MAGIC.
  unsigned a_text              // length of code, in bytes.
  unsigned a_data              // length of data, in bytes.
  unsigned a_bss               // length of the uninitialized data area for file, in bytes.
  unsigned a_syms              // length of the symbol table data in the file, in bytes.
  unsigned a_entry             // Execution start address.
  unsigned a_trsize            // length of the relocation info for text, in bytes.
  unsigned a_drsize            // length of the relocation info for data, in bytes.
};
```

The functionality of each field is as follows:

- a_magic -- This field contains three subfields, the flag field, the machine type id field, and the magic number field, as shown in Figure 14-1. However, for the Linux 0.12 system, its object file only uses the magic number subfield and is accessed using the macro N_MAGIC(), which uniquely determines the difference between the binary executable file and other loaded files. This subfield must contain one of the following values:
  - OMAGIC -- Indicates that the text and data segments are immediately following the execution header and are stored consecutively. The kernel loads both text and data segments into readable and writable memory. The magic number of the object file compiled by the compiler is OMAGIC (octal 0407).
  - NMAGIC -- Like OMAGIC, text and data segments follow the execution header and are stored continuously. However, the kernel loads the text into read-only memory and loads the data

  segment into writable memory at the next page boundary after the text.

  - ZMAGIC -- The kernel loads separate pages from the binary executable when necessary. The execution header, text segment, and data segment are all processed by the linker into blocks of multiple page sizes. The text page loaded by the kernel is read-only, and the page of the data segment is writable. The magic number of the executable file generated by the linker is ZMAGIC (0413, ie 0x10b).

- a_text -- This field contains the size of the text segment, the number of bytes.
- a_data -- This field contains the size of the data segment, the number of bytes.
- a_bss -- Contains the length of the 'bss segment' that the kernel uses to set the initial break(brk) after the data segment. When the kernel is loading the program, this writable memory appears to be behind the data segment and is initially all zeros.
- a_syms -- Contains the size in bytes of the symbol table section.
- a_entry -- The memory address of the program execution start point after the kernel has loaded the executable file into memory.
- a_trsize -- This field contains the size of the text relocation table, in bytes.
- a_drsize -- This field contains the size of the data relocation table, in bytes.

Several macros are defined in the a.out.h header file. These macros use the exec structure to test for consistency or to locate various section offsets in the executable file. These macros are:

N_BADMAG(exec) Returns a non-zero value if the a_magic field cannot be recognized.
N_TXTOFF(exec) The starting byte offset of the code segment.
N_DATOFF(exec) The starting byte offset of the data segment.
N_DRELOFF(exec) The starting byte offset of the data relocation table.
N_TRELOFF(exec) The starting byte offset of the text relocation table.
N_SYMOFF(exec) The starting byte offset of the symbol table.
N_STROFF(exec) The starting byte offset of the string table.

The relocation record has a standard format, which is described using a relocation information (relocation_info) structure, as shown below.

```
struct relocation_info
{
  int r_address;               // The address that needs to be relocated within the segment.
  unsigned int r_symbolnum:24;  // The meaning is related to r_extern.
                               // Specifies a symbol or a segment in the symbol table.
  unsigned int r_pcrel:1;       // A pc-related flag.
  unsigned int r_length:2;      // Length (as exponent of 2) of the field to be relocated.
  unsigned int r_extern:1;      // 1:relocate with value of symbol, 0:with address of segment.
  unsigned int r_pad:4;         // 4 bits are not used, but it is best to clear them.
};
```

The meaning of each field in the structure is as follows:

- r_address -- This field contains the byte offset of the pointer that the linker needs to process (edit). The offset of the text relocation is counted from the beginning of the text segment, and the offset of the data relocation is calculated from the beginning of the data segment. The linker adds the value already stored at the offset to the new value calculated using the relocation record.

- r_symbolnum -- This field contains the ordinal number (not the byte offset) of a symbol structure in the symbol table. After the linker calculates the absolute address of the symbol, it adds the address to the pointer being relocated. (If the r_extern bit is 0, then the situation is different, see below.)
- r_pcrel -- If this bit is set, the linker considers that a pointer is being updated, which uses the pc-related addressing mode and is part of the machine code instruction. When the running program uses this relocated pointer, the address of the pointer is implicitly added to the pointer.
- r_length -- This field contains the power of 2 of the length of the pointer: 0 means 1 byte long, 1 means 2 bytes long, 2 means 4 bytes long.
- r_extern -- If set, it indicates that the relocation requires an external reference; the linker must use a symbol address to update the pointer. When the bit is 0, the relocation is "local"; the linker updates the pointer to reflect the changes in the load addresses of the various segments, rather than reflecting changes in the value of a symbol. In this case, the contents of the r_symbolnum field are an n_type value; such field tells the linker what segment the relocated pointer points into.
- r_pad -- These 4 bits are not used in Linux systems. It is best to set all 0 when writing a object file.

Symbols map names to addresses (or more generally, strings are mapped to values). Due to the linker's adjustment of the address, the name of a symbol must be used to indicate its address until it has been assigned an absolute address value. A symbol consists of a fixed-length record in the symbol table and a variable-length name in the string table. The symbol table is an array of nlist structures, as shown below.

```
struct nlist {
  union {
    char          *n_name;
    struct nlist *n_next;
    long           n_strx;
  } n_un;
  unsigned char n_type;            // divided into 3 fields, and lines 146-154 are their masks.
  char          n_other;
  short         n_desc;
  unsigned long n_value;
};
```

The meaning of each field is:

- n_un.n_strx -- Contains the byte offset of the symbol name in the string table. When a program accesses a symbol table using the nlist() function, the field is replaced with the n_un.n_name field, which is a pointer to a string in memory.
- n_type -- Used by the linker to determine how to update the value of the symbol. The 8-bit wide n_type field can be divided into three subfields using the bitmasks code starting at line 146--154, as shown in Figure 14-2. For symbols of N_EXT type location bits, the linker treats them as "external" symbols and allows other binary object files to reference them. The N_TYPE mask is used to select the bits of interest to the linker:
  - N_UNDF -- An undefined symbol. The linker must locate an external symbol with the same name in another binary object file to determine the absolute data value of the symbol. In special cases, if the n_type field is non-zero and no binary file defines this symbol, the linker resolves the symbol into an address in the BSS segment, reserving bytes of length equal to n_value. If the

symbol is not defined in more than one binary object file and the binary object files do not match their length values, the linker will select the longest value found across all binary object files.

- N_ABS -- An absolute symbol. The linker does not update an absolute symbol.
- N_TEXT -- A text (code) symbol. The value of this symbol is the text address, and the linker updates its value when it merges the binary object files.
- N_DATA -- A data symbol; similar to N_TEXT, but for data addresses. The value of the corresponding text and data symbol is not the offset of the file but the address; In order to find the offset of the file, it is necessary to determine the address at which the relevant section starts loading and subtract it, and then add the offset of the section.
- N_BSS -- A BSS symbol; similar to a text or data symbol, but without a corresponding offset in the binary object file.
- N_FN -- A file name symbol. When merging a binary object file, the linker inserts the symbol before the symbol in the binary file. The name of the symbol is the file name given to the linker, and its value is the address of the first text segment in the binary file. File name symbols are not required for linking and loading, but are very useful for debugging programs.
- N_STAB -- The mask code is used to select the bits of interest to the symbolic debugger (eg gdb); its value is specified in stab().

- n_other -- This field provides symbol independence information about the symbol relocation operation according to the segment determined by n_type. Currently, the lowest 4 bits of the n_other field contain one of two values: AUX_FUNC and AUX_OBJECT. AUX_FUNC associates symbols with callable functions, and AUX_OBJECT associates symbols with data, regardless of whether they are in text segments or data segments. This field is mainly used by the linker ld for the creation of dynamic executable programs.
- n_desc -- Reserved for use by the debugger; the linker does not process it. Different debuggers use this field for different purposes.
- n_value -- Contains the value of the symbol. For text, data, and BSS symbols, this is an address; for other symbols (such as debugger symbols), the value can be arbitrary.

A string table consists of symbol strings of length unsigned long followed by a null. The length represents the byte size of the entire table, so the minimum value (that is, the offset of the first string) on a 32-bit machine is always 4.

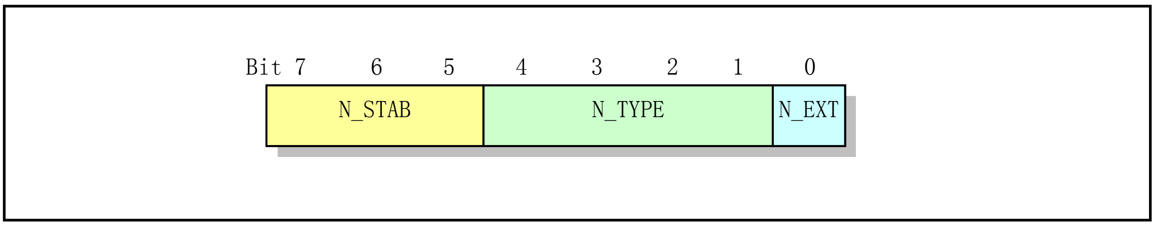


Figure 14-2 Symbol type field n_type

14.3 const.h

14.3.1 Function

The const.h file defines some of the flag constants used by the file modes and type field i_mode in the file i-node.

14.3.2 Code annotation

Program 14-2 linux/include/const.h

```
#ifndef _CONST_H
#define _CONST_H

#define BUFFER_END 0x200000          // the memory end used by the buffer (not used).

// Each flag bit of the i_mode field in the i-node structure.
#define I_TYPE          0170000      // Indicate the i-node type (type mask).
#define I_DIRECTORY     0040000      // Is a directory file.
#define I_REGULAR       0100000      // Is a regular file, not a dir or a special file.
#define I_BLOCK_SPECIAL 0060000      // Is a block device special file.
#define I_CHAR_SPECIAL  0020000      // Is a character device special file.
#define I_NAMED_PIPE    0010000      // Is a named pipe node.
#define I_SET_UID_BIT   0004000      // Set efficient user ID type at execution time.
#define I_SET_GID_BIT   0002000      // Set efficient group ID type at execution time.

#endif

```

14.4 ctype.h

14.4.1 Function

The ctype.h file is a header file for character testing and processing, and is one of the header files for the standard C library. It defines some macros for character type checking and conversion. It defines some macros for character type checking and conversion. For example, determine if a character c is a numeric character (isdigit(c)) or a space character (isspace(c)). During the check process, an array or table (defined in lib/ctype.c) is used, which defines the properties and types of all the characters in the ASCII table. When a macro is used, the character code is taken as an index value in the table _ctype[], and a byte is obtained from the table, so that the relevant bit is obtained.

In addition, macro names that begin with two underscores or begin with an underscore followed by an uppercase letter are usually reserved for the header writer, such as the names __abc and _SP.

14.4.2 Code annotation

Program 14-3 linux/include/ctype.h

```
#ifndef _CTYPE_H
#define _CTYPE_H

#define _U      0x01    /* upper */          // used for uppercase characters [A-Z].
#define _L      0x02    /* lower */          // used for lowercase characters [a-z].
#define _D      0x04    /* digit */          // used for digitals [0-9].
#define _C      0x08    /* cntrl */          // used for control characters.
#define _P      0x10    /* punct */          // used for punctuation characters.
#define _S      0x20    /* white space (space/lf/tab) */
#define _X      0x40    /* hex digit */           // used for hexadecimal digits.
#define _SP     0x80    /* hard space (0x20) */   // used for the space character (0x20).

// An array (table) of character attributes that defines the attributes of each character.
// The other is a temporary character variable. They are all defined in lib/ctype.c file.
extern unsigned char _ctype[];
extern char _ctmp;

// Below are some macros that determine the character type.
#define isalnum(c) ((_ctype+1)[c]&(_U|_L|_D))      // is a character or digital.
#define isalpha(c) ((_ctype+1)[c]&(_U|_L))         // is a character.
#define iscntrl(c) ((_ctype+1)[c]&(_C))            // is a control character.
#define isdigit(c) ((_ctype+1)[c]&(_D))            // is a digital.
#define isgraph(c) ((_ctype+1)[c]&(_P|_U|_L|_D))   // is a graphic character.
#define islower(c) ((_ctype+1)[c]&(_L))            // is a lowercase character.
#define isprint(c) ((_ctype+1)[c]&(_P|_U|_L|_D|_SP))   // is a printable character.
#define ispunct(c) ((_ctype+1)[c]&(_P))            // is a punctuation mark.
#define isspace(c) ((_ctype+1)[c]&(_S))            // is a space, \f, \n, \r, \t, \v.
#define isupper(c) ((_ctype+1)[c]&(_U))            // is an uppercase character.
#define isxdigit(c) ((_ctype+1)[c]&(_D|_X))        // is a hexadecimal number.

// In the following two macro definitions, the macro parameter is prefixed (unsigned), so ’c’
// should be bracketed, which means it is (c). Because ’c’ may be a complex expression in the
// program. For example, if the argument is a + b, without parentheses, it becomes: (unsigned)
// a + b in the macro definition, which is obviously wrong. After bracketing, it can be correctly
// represented as (unsigned) (a + b).
#define isascii(c) (((unsigned) c)<=0x7f)          // is an ASCII character.
#define toascii(c) (((unsigned) c)&0x7f)           // convert to ASCII character.

// The reason for using a temporary variable ’_ctmp’ in the following two macro definitions
// is that the macro’s parameters can only be used once in the macro definition. But this is
// not safe for multithreading because two or more threads may use this public temporary variable
// at the same time. So from kernel 2.2.x, these two macro definitions are changed to use two
// functions.
#define tolower(c) (_ctmp=c,isupper(_ctmp)?_ctmp-('A'-'a'):_ctmp)     // to lowercase char.
#define toupper(c) (_ctmp=c,islower(_ctmp)?_ctmp-('a'-'A'):_ctmp)     // to uppercase char.

#endif

```

14.5 errno.h

14.5.1 Function

There is a variable called 'errno' in the UNIX type system or the standard C language. Whether this variable is needed in the C standard has caused a lot of controversy in the C standardization organization (X3J11). But the result of the debate was that the 'errno' was not removed, instead a header file named "errno.h" was created. Because the standardization organization wants each library function or data object to be declared in a corresponding standard header file.

The main reason for the debate is that for each system-call in the kernel, if the return value is the result of the system-call, it is difficult to report the error. If we let each function return a true/false indication value and the resulting value returns separately, we can't easily get the result of the system-call. One solution is to combine these two methods: For a particular system-call, you can specify an error return value that is different from the range of valid result values. For example, a pointer can take a null value, and for a pid it can return a value of -1. In many other cases, '-1' can be used to indicate an error value as long as it does not conflict with the resulting value. However, the standard C library function return value only tells whether an error has occurred, and the type of error must be known from other places, so the variable 'errno' is used.

In order to be compatible with the design mechanism of the standard C library, the library file in the Linux kernel also uses this processing method. Therefore, this header file of the standard C library is also borrowed. See the lib/open.c program and the system-call macro definition in unistd.h for examples. In some cases, although the program knows the error from the returned '-1' value, but wants to know the specific error number, you can determine the error number of the last error by reading the value of 'errno'.

14.5.2 Code annotation

Program 14-4 linux/include/errno.h

```
#ifndef _ERRNO_H
#define _ERRNO_H

/*
 * ok, as I hadn't got any other source of information about
 * possible error numbers, I was forced to use the same numbers
 * as minix.
 * Hopefully these are posix or something. I wouldn't know (and posix
 * isn't telling me - they want $$$ for their f***ing standard).
 *
 * We don't use the _SIGN cludge of minix, so kernel returns must
 * see to the sign by themselves.
 *
 * NOTE! Remember to change strerror() if you change this file!
 */

// System-calls and many library functions return a special value to indicate an operation
// failure or an error. This value is usually chosen to be '-1', or some other specific value.
// But this return value only indicates that an error has occurred. If we need to know the type
// of error, we need to look at the variable 'errno' which represents the system error number.
// This variable is declared in the errno.h file and is initialized to 0 when the program begins
// execution.
extern int errno;

// In the event of an error, the system-call puts the error number in the variable 'errno'
// (negative value) and returns -1. Therefore, if the program needs to know the specific error
// number, you need to check the value of 'errno'.
#define ERROR           99                 // General error.
#define EPERM            1                 // The operation is not permited.
#define ENOENT           2                 // The file or directory does not exist.
#define ESRCH            3                 // The specified process does not exist.
#define EINTR            4                 // Interrupted system-call.
#define EIO              5                 // Input/output error.
#define ENXIO            6                 // Specified device or address doesn't exist.
#define E2BIG            7                 // The parameter list is too long.
#define ENOEXEC          8                 // The format of executable file is incorrect.
#define EBADF            9                 // File handle (descriptor) is incorrect.
#define ECHILD          10                 // The child process does not exist.
#define EAGAIN          11                 // The resource is temporarily unavailable.
#define ENOMEM          12                 // No enough memory.
#define EACCES          13                 // No access permissions.
#define EFAULT          14                 // The address is wrong.
#define ENOTBLK         15                 // Not a block device file.
#define EBUSY           16                 // The resource is busy.
#define EEXIST          17                 // File already exists.
#define EXDEV           18                 // Illegal connection to device.
#define ENODEV          19                 // The device does not exist.
#define ENOTDIR         20                 // Not a directory file.
#define EISDIR          21                 // Is a directory file.
#define EINVAL          22                 // Invalid argument.
#define ENFILE          23                 // The system has too many open files.
#define EMFILE          24                 // Too many open files.
#define ENOTTY          25                 // Inappropriate IO (no tty terminal).
#define ETXTBSY         26                 //  (No longer use).
#define EFBIG           27                 // File size too big.
#define ENOSPC          28                 // The device is full (the device has no space).
#define ESPIPE          29                 // Invalid file pointer relocation.
#define EROFS           30                 // The file system is read only.
#define EMLINK          31                 // Too many links.
#define EPIPE           32                 // The pipe is wrong.
#define EDOM            33                 // Domain error.
#define ERANGE          34                 // The result range error.
#define EDEADLK         35                 // Resource deadlocks.
#define ENAMETOOLONG    36                 // The filename is too long.
#define ENOLCK          37                 // No locks are available.
#define ENOSYS          38                 // Not yet implemented.
#define ENOTEMPTY       39                 // The directory is not empty.

/* Should never be seen by user programs */
#define ERESTARTSYS     512                // Re-execute the system-call.
#define ERESTARTNOINTR  513                // Re-execute the system-call, no interrupt.

#endif

```

14.6 fcntl.h

14.6.1 Function

The fcntl.h file is the file control option header file. It mainly defines the file control function fcntl () and some of the options used in the file creation or open function. The fcntl() function is implemented in the linux/fs/fcntl.c file, and is used to perform various specified operations on the file descriptor (handle). The specific operation is specified by the function parameter cmd (command). .

14.6.2 Code annotation

Program 14-5 linux/include/fcntl.h

```
#ifndef _FCNTL_H
#define _FCNTL_H

#include <sys/types.h>        // Type header file. The basic system data types are defined.

/* open/fcntl - NOCTTY, NDELAY isn't implemented yet */
#define O_ACCMODE        00003                    // File access mode mask.
// File access modes used by the open() and fcntl(), and only one of the three can be used.
#define O_RDONLY           00                     // Open file in read-only mode.
#define O_WRONLY           01                     // Open file in write-only mode.
#define O_RDWR             02                     // Open file in read-write mode.
// Below are the file creation and operation flags for open(). Can be used with the above
// access mode in a 'bit or' mode.
#define O_CREAT          00100    /* not fcntl */ // Create if file does not exist.
#define O_EXCL           00200    /* not fcntl */ // Exclusive use of file.
#define O_NOCTTY         00400    /* not fcntl */ // No control terminal.
#define O_TRUNC          01000    /* not fcntl */ // truncated to zero if write operation.
#define O_APPEND         02000                    // Open file in append mode.
#define O_NONBLOCK       04000    /* not fcntl */ // Open file in non-blocking manner.
#define O_NDELAY         O_NONBLOCK

/* Defines for fcntl-commands. Note that currently
 * locking isn't supported, and other things aren't really
 * tested.
 */
// The command (cmd) used by the file descriptor's operation function fcntl().
#define F_DUPFD          0        /* dup */          // duplicate a file handle.
#define F_GETFD          1        /* get f_flags */  // get file handle flags (FD_CLOEXEC).
#define F_SETFD          2        /* set f_flags */  // set file handle flags.
#define F_GETFL          3        /* more flags (cloexec) */
#define F_SETFL          4        // get/set file state flag & access mode.
// Below are file lock commands. The parameter 'lock' of fcntl() points to flock structure.
#define F_GETLK          5        /* not implemented */ // get flock that blocks the lock.
#define F_SETLK          6        // Set (F_RDLCK or F_WRLCK) or clear lock (F_UNLCK).
#define F_SETLKW         7        // Set or clear the lock in wait mode.
```

```
31
32 /* for F_[GET|SET]FL */
   // The file handle needs to be closed when executing the exec() function.
33 #define FD_CLOEXEC      1       /* actually anything with low bit set goes */
34
35 /* Ok, these are locking features, and aren't implemented at any
36  * level. POSIX wants them.
37  */
38 #define F_RDLCK         0       // Share or read file lock.
39 #define F_WRLCK         1       // Exclusive or write file lock.
40 #define F_UNLCK         2       // File unlock.
41
42 /* Once again - not implemented, but ... */
   // The following is a file lock data structure that describes the type (l_type) of the affected
   // file segment, the start offset (l_whence), the relative offset (l_start), the lock length
   // (l_len), and the pid that implements the lock.
43 struct flock {
44         short l_type;           // Lock type (F_RDLCK, F_WRLCK, F_UNLCK).
45         short l_whence;         // Start offset (SEEK_SET, SEEK_CUR or SEEK_END).
46         off_t l_start;          // The beginning of the lock. Relative offset (in bytes).
47         off_t l_len;            // The size of the lock; if 0, it is the end of the file.
48         pid_t l_pid;            // The process id of the lock.
49 };
50
   // The following are function prototypes using the above flags or commands.
   // Create a new file or rewrite an existing file. The parameter 'filename' is the file name
   // of the file to be created, and 'mode' is the property (see include/sys/stat.h).
51 extern int creat(const char * filename,mode_t mode);
   // File handle operation function. They can affect file open operations. The parameter 'fildes'
   // is the file handle (descriptor), and 'cmd' is the operation command, see lines 23-30 above.
   // The function can be in the following forms:
   // int fcntl(int fildes, int cmd);
   // int fcntl(int fildes, int cmd, long arg);
   // int fcntl(int fildes, int cmd, struct flock *lock);
52 extern int fcntl(int fildes,int cmd, ...);
   // open a file. Used to establish a connection between a file and a file handle.
   // The parameter 'flags' is a combination of the flags on lines 7-17 above.
53 extern int open(const char * filename, int flags, ...);
54
55 #endif
56
```

14.7 signal.h

14.7.1 Functionality

Signals provide a way to handle asynchronous events, and signals are also known as soft interrupts. By sending a signal to a process, we can control the execution state of the process (pause, resume, or terminated). The signal.h file defines the names and basic operational functions of all the signals used in the kernel. The most important function are the functions signal() and sigaction() that change the way the signal is processed.

As you can see from the header file, the Linux kernel implements all 20 signals required by POSIX.1. So we can say that Linux was completely designed with compatibility with the standard at the outset. The specific implementation of the function can be found in the program kernel/signal.c.

14.7.2 Code annotation

Program 14-6 linux/include/signal.h

```
#ifndef _SIGNAL_H
#define _SIGNAL_H

#include <sys/types.h>       // Type header file. The basic system data types are defined.

typedef int sig_atomic_t;              // define signal atomic operation type.
typedef unsigned int sigset_t;         /* 32 bits */      // define signal set type.

#define _NSIG             32           // number of signals.
#define NSIG            _NSIG          // NSIG = _NSIG

// The following are the signals defined in the Linux 0.12 kernel. This includes all 20 signals
// required by POSIX.1.
#define SIGHUP           1      // Hang Up     -- Hang up the control terminal or process.
#define SIGINT           2      // Interrupt   -- Interrupt from the keyboard.
#define SIGQUIT          3      // Quit        -- Exit command from the keyboard.
#define SIGILL           4      // Illeagle    -- Illegal instruction.
#define SIGTRAP          5      // Trap        -- Track breakpoints.
#define SIGABRT          6      // Abort
#define SIGIOT           6      // IO Trap
#define SIGUNUSED        7      // Unused
#define SIGFPE           8      // FPE         -- Coprocessor error.
#define SIGKILL          9      // Kill        -- Force the process to terminate.
#define SIGUSR1         10      // User1       -- User defined signal 1.
#define SIGSEGV         11      // Segment Violation -- Invalid memory reference.
#define SIGUSR2         12      // User2       -- User defined signal 2.
#define SIGPIPE         13      // Pipe        -- Pipe write error, no reader.
#define SIGALRM         14      // Alarm       -- Real-time timer alarm.
#define SIGTERM         15      // Terminate   -- Process terminated.
#define SIGSTKFLT       16      // Stack Fault -- Stack error (coprocessor).
#define SIGCHLD         17      // Child       -- Child process is stopped or terminated.
#define SIGCONT         18      // Continue    -- Resume the execution of process.
#define SIGSTOP         19      // Stop        -- Stop the execution of process.
#define SIGTSTP         20      // TTY Stop    -- Stop sig sent by process, can be ignored.
#define SIGTTIN         21      // TTY In      -- Background process requests input.
#define SIGTTOU         22      // TTY Out     -- Background process requests output.

/* Ok, I haven't implemented sigactions, but trying to keep headers POSIX */
// The above comment is obsolete because sigaction() has been implemented in the 0.12 kernel.
// The following is the symbolic constant that can be taken from the 'sa_flags' flag field in
// the sigaction structure.
// SA_NOCLDSTOP - When child is in stopped state, the SIGCHLD signal is not processed.
// SA_INTERRUPT - The system call is not restarted after it is interrupted by the signal.
// SA_NOMASK    - This signal is not blocked from being received in its signal handler.
```

```
   // SA_ONESHOT - The signal handler is restored to its default one once it has been called.
37 #define SA_NOCLDSTOP    1
38 #define SA_INTERRUPT    0x20000000
39 #define SA_NOMASK       0x40000000
40 #define SA_ONESHOT      0x80000000
41
   // The following constants are used for sigprocmask(how, ) -- to add/remove a given signal
   // to/from the blocking signal set, and change the blocking signal set (mask code).
   // Used to change the behavior of this function.
42 #define SIG_BLOCK       0  /* for blocking signals */
43 #define SIG_UNBLOCK     1  /* for unblocking signals */
44 #define SIG_SETMASK     2  /* for setting the signal mask */
45
   // The following three constant symbols all represent function pointers that have no return
   // value and have an INT integer parameter. These three pointers are logically the addresses
   // of functions that are practically impossible. They can be used as the second parameter of
   // the signal() function below to tell the kernel to let the kernel process the signal, ignore
   // the processing of the signal, or signal processing returns an error. For how to use them,
   // see kernel/signal.c, lines 156--158.
46 #define SIG_DFL         ((void (*)(int))0)      /* default signal handling */
47 #define SIG_IGN         ((void (*)(int))1)      /* ignore signal */
48 #define SIG_ERR         ((void (*)(int))-1)     /* error return from signal */
49
   // The following defines a macro for initial setting the sigaction structure signal mask.
50 #ifdef notdef
51 #define sigemptyset(mask) ((*(mask) = 0), 1)    // Clear mask.
52 #define sigfillset(mask) ((*(mask) = ~0), 1)    // All bits of the mask are set.
53 #endif
54
   // The following is the sigaction data structure, where the meaning of each field is:
   // 'sa_handler' is the action that is specified for a signal. This signal can be ignored with
   // SIG_DFL, or SIG_IGN above, or it can be a pointer to a function that handles the signal.
   // 'sa_mask' gives the masking code for the signal, which will block the processing of these
   // signals when the signal program is executed.
   // 'sa_flags' specifies the set of signals that change the signal processing, which is defined
   // by the bit flags of lines 37-40.
   // 'sa_restorer' is a recovery function pointer, provided by the function library Libc, used
   // to clean up the user stack.  See signal.c.
   // In addition, the signal that causes the trigger signal processing will also be blocked unless
   // the SA_NOMASK flag is used.
55 struct sigaction {
56         void (*sa_handler)(int);
57         sigset_t sa_mask;
58         int sa_flags;
59         void (*sa_restorer)(void);
60 };
61
   // The signal() function below is used to install a new signal handler for signal _sig, similar
   // to sigaction().This function takes two arguments: one is to specify the signal _sig to be
   // captured, and the other is a function pointer _func with one argument and no return value.
   // The return value of this function is also a function pointer with an int argument (the last
   //  (int)) and no return value, which is the original handle of the signal.
62 void (*signal(int _sig, void (*_func)(int)))(int);
```

```
    // The following two functions are used to send signals. kill() is used to send a signal to
    // any process or process group; raise() is used to send a signal to the current process itself,
    // which is equivalent to kill(getpid(), sig). See kernel/exit.c, line 205.
63 int raise(int sig);
64 int kill(pid_t pid, int sig);
    // In the task structure of the process, in addition to a 32-bit signal field 'signal' indicating
    // the signal to be processed of the current process, there is a 32-bit blocking signal set
    // field 'blocked' for masking the signal, each bit representing a corresponding blocked signal.
    // Modifying the mask signal set can block or unblock the specified signal. The following five
    // functions are used to manipulate the process to mask the signal set. Although it is very
    // simple to implement, it is not implemented in this version of the kernel.
    // The functions sigaddset() and sigdelset() are used to add, delete, and modify signals in
    // the signal set. Sigaddset() is used to add the given signal signo to the signal set pointed
    // to by the mask, sigdelset() is the opposite. The functions sigemptyset() and sigfillset()
    // are used to initialize the process masking signal set. Before using the signal set, each
    // program needs to initialize the mask signal set using one of these two functions. Sigemptyset()
    // is used to clear all masked signals, that is, to respond to all signals; sigfillset() puts
    // all signals into the signal set, ie masks all signals. Of course SIGINT and SIGSTOP cannot
    // be blocked. The sigismember() function is used to test if a specified signal is in the signal
    // set (1 - yes, 0 - no, -1 - error).
65 int sigaddset(sigset_t *mask, int signo);
66 int sigdelset(sigset_t *mask, int signo);
67 int sigemptyset(sigset_t *mask);
68 int sigfillset(sigset_t *mask);
69 int sigismember(sigset_t *mask, int signo); /* 1 - is, 0 - not, -1 error */
    // Check the signal in 'set' to see if there is a pending signal. And in 'set' returns the currently
    // blocked signal set in the process.
70 int sigpending(sigset_t *set);
    // The following function is used to change the signal set that the process is currently blocking.
    // If 'oldset' is not NULL, then it returns the current masked signal set. If the 'set' pointer
    // is not NULL, modify the process mask signal set according to the 'how' (line 42-44).
71 int sigprocmask(int how, sigset_t *set, sigset_t *oldset);
    // The following function temporarily replaces the signal mask of the process with 'sigmask'
    // and then pauses the process until it receives a signal. If a signal is captured and returned
    // from the signal handler, the function returns and the signal mask is restored to the value
    // before the call.
72 int sigsuspend(sigset_t *sigmask);
    // The sigaction() function is used to change the action taken by the process when it receives
    // signal, that is, to change the processing handler of the signal. See the description in the
    // kernel/signal.c program.
73 int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
74
75 #endif /* _SIGNAL_H */
76
```

14.8 stdarg.h

14.8.1 Function

One of the biggest features of the C language is that it allows programmers to customize functions with variable number of parameters. In order to access the parameters in these variable parameter lists, you need to use the macros in the stdarg.h file. The stdarg.h header file is modified by the C standardization organization according to the varargs.h file of the BSD system.

Stdarg.h is a standard parameter header file that defines a list of variable parameters in the form of macros. It mainly describes a type (va_list) and three macros (va_start, va_arg and va_end) for the vsprintf, vprintf, vfprintf functions. When reading this file, we need to first understand how to use the variable parameter function. See the description after the kernel/vsprintf.c list.

14.8.2 Code annotation

Program 14-7 linux/include/stdarg.h

```
 1 #ifndef _STDARG_H
 2 #define _STDARG_H
 3
 4 typedef char *va_list;  // Defining va_list is a character pointer type.
 5
 6 /* Amount of space required in an argument list for an arg of type TYPE.
 7    TYPE may alternatively be an expression whose type is used.  */
 8
   // The following sentence defines the byte length of the rounded TYPE type, which is a multiple
   // of the int length (4).
 9 #define __va_rounded_size(TYPE)  \
10   (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
11
   // The following macro initializes the pointer AP to point to the first argument passed to the
   // function's variable argument list. Before calling va_arg or va_end for the first time, you
   // must first call the va_start macro. The parameter LASTARG is the identifier of the rightmost
   // parameter in the function definition, ie an identifier to the left of '...'. The AP is a
   // variable parameter table parameter pointer, and LASTARG is the last specified parameter.
   // &(LASTARG) is used to get its address (ie its pointer) and the pointer is a character type.
   // After adding the width value of LASTARG, the AP is a pointer to the first parameter in the
   // variable parameter list. This macro has no return value.
   // The function __builtin_saveregs() on line 17 is defined in gcc's library program libgcc2.c
   // and is used to hold registers. See the "Implementing the Varargs Macros" section of the gcc
   // manual "Target Description Macros" for instructions.
12 #ifndef __sparc__
13 #define va_start(AP, LASTARG)                                    \
14  (AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
15 #else
16 #define va_start(AP, LASTARG)                                    \
17  (__builtin_saveregs (),                                         \
18   AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
19 #endif
20
   // The macro below is used by the called function to complete a normal return. va_end can modify
   // the AP to not be used until va_start is called again. va_end must be called after va_arg
   // has read all the parameters.
21 void va_end (va_list);          /* Defined in gnulib */
22 #define va_end(AP)
23
   // The following macro is used to extend the expression to have the same type and value as the
   // next passed parameter. For default values, va_arg can be used with characters, unsigned
   // characters, and floating point types. The first time you use va_arg, it returns the first
   // argument in the table, and each subsequent call returns the next argument in the table. This
   // is done by first accessing the AP and then increasing its value to point to the next item.
   // va_arg uses TYPE to complete access and locate the next item. Each time va_arg is called,
   // it modifies the AP to indicate the next parameter in the table.
   #define va_arg(AP, TYPE)                                \
    (AP += __va_rounded_size (TYPE),                       \
     *((TYPE *) (AP - __va_rounded_size (TYPE))))

   #endif /* _STDARG_H */

```

14.9 stddef.h

14.9.1 Functionality

The stddef.h header file is created by the C standardization organization (X3J11), and the meaning of the file name is the standard (std) definition (def). It is mainly used to store "standard definitions" such as some commonly used constant symbols and function prototypes in UNIX-like systems. Another confusing header file is stdlib.h, which is also created by the standardization organization, and is mainly used to declare various function prototype declarations that are not related to other header file types. But the contents of these two header files often make it impossible to figure out which declarations are in which header file.

Some members of the standardization organization believe that C should also be a useful programming language in a stand-alone environment that does not fully support the standard C library. For a standalone environment, the C standard requires that it provide all the attributes of the C language. For the standard C library, such an implementation only needs to provide support for the functions in the four header files: float.h, limits.h, stdarg.h, and stddef.h. This requirement clarifies what the stddef.h file should contain, and the other three header files are basically used for more specific aspects:

- float.h Describe the floating point representation feature;
- limits.h Describe the integer representation characteristics;
- stdarg.h Provides macro definitions for accessing variable parameter list.

Any other type or macro definition used in a standalone environment should be placed in the stddef.h file, but later organizational members have relaxed these restrictions, causing some definitions to appear in multiple header files. For example, the NULL macro definition also appears in the other four header files. Therefore, in order to prevent conflicts, the stddef.h file first uses the undef command to cancel the original definition when defining NULL (line 14). In addition, the types and macros defined in this file have one thing in common: they have been tried to be included in the features of the C language, but since various compilers have defined this information in their own way, it’s very hard to write code that replaces all of these definitions. This file is rarely used in the Linux 0.12 kernel.

14.9.2 Code annotation

Program 14-8 linux/include/stddef.h

```
 1 #ifndef _STDDEF_H
```

```
#define _STDDEF_H

#ifndef _PTRDIFF_T
#define _PTRDIFF_T
typedef long ptrdiff_t;              // The type of result of subtracting two pointers.
#endif

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned long size_t;        // The type of result returned by sizeof().
#endif

#undef NULL
#define NULL ((void *)0)             // null pointer.

// The following defines a macro that calculates the offset position of a member in a type.
// By using this macro, you can determine the byte offset of a member (field) from the beginning
// of the structure in the structure type that contains it. The result of the macro is an integer
// constant expression of type size_t. Here is a trick usage: ((TYPE *)0) means that an integer
// type 0 is cast into a data object pointer type, and then the operation is performed on the
// result.
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)

#endif

```

14.10 string.h

14.10.1 Functionality

The string.h header file defines all string manipulation functions as inline functions, and uses inline assembly language to improve execution speed. In addition, a NULL macro and a SIZE_T type are defined at the beginning of the file.

A header file of the same name is also provided in the standard C library, but the function implementation is in the standard C library, and its corresponding header file contains only the declaration of the relevant function. For the string.h file listed here, Linus gives the implementation of each function, but each function has the 'extern' and 'inline' keyword prefixes, that is, some inline functions are defined. Therefore, for a program containing this header file, if the inline function used for some reason cannot be embedded in the calling code, the same name function defined in the kernel function library lib/ directory will be used, see the lib/string.c program. In that string.c, the program first defines 'extern' and 'inline' to be empty, and then contains the string.h header file. Therefore, the string.c program actually contains the function implementation code declared in the string.h header file.

To make it easy to write comments on one line, some abbreviations are used here in the source without causing confusion, for example: string - str; character - char; pointer - ptr; address - addr. source - src; destination - dest; length - len.

14.10.2 Code annotation

Program 14-9 linux/include/string.h

```
#ifndef _STRING_H_
#define _STRING_H_

#ifndef NULL
#define NULL ((void *) 0)
#endif

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif

extern char * strerror(int errno);

/*
 * This string-include defines all string functions as inline
 * functions. Use gcc. It also assumes ds=es=data space, this should be
 * normal. Most of the string-functions are rather heavily hand-optimized,
 * see especially strtok,strstr,str[c]spn. They should work, but are not
 * very easy to understand. Everything is done entirely within the register
 * set, making the functions fast and clean. String instructions have been
 * used through-out, making for "slightly" unclear code :-)
 *
 *              (C) 1991 Linus Torvalds
 */

//// Copy a source string to destination string until it encounters a NULL character.
// Params: dest - the destination str pointer, src - the source str pointer.
// In the embedded assembly code, %0 - register ESI(src), %1 - EDI(dest).
extern inline char * strcpy(char * dest,const char *src)
{
__asm__("cld\n"                         // Clear direction flag.
        "1:\tlodsb\n\t"                 // Load 1 byte from DS:[ESI] into AL and update ESI.
        "stosb\n\t"                     // Store the byte in AL to ES:[EDI] and update EDI.
        "testb %%al,%%al\n\t"           // The byte just stored is 0?
        "jne 1b"                        // If not, jump backward to label 1, else end.
        ::"S" (src),"D" (dest):"si","di","ax");
return dest;                            // Returns the destination str pointer.
}

//// Copy count bytes of the source string to the destination.
// If the source str length is less than count bytes, null chars are appended to the dest string.
// Parameters: dest - the destination str pointer, src - the source str pointer, count - number
// of bytes copied. %0 - esi(src), %1 - edi(dest), %2 - ecx(count).
extern inline char * strncpy(char * dest,const char *src,int count)
{
__asm__("cld\n"                         // Clear direction flag.
        "1:\tdecl %2\n\t"               // Register ECX-- (count--).
        "js 2f\n\t"                     // If count<0 then jump forward to label 2 and end.
        "lodsb\n\t"                  // Get 1 byte from DS:[ESI] to AL, and ESI++.
        "stosb\n\t"                  // Store the byte to ES:[EDI], and EDI++.
        "testb %%al,%%al\n\t"        // Is this byte 0?
        "jne 1b\n\t"                 // If not, jump forward to label 1 to continue copy.
        "rep\n\t"                    // Else, store remain number of NULLs into dest str.
        "stosb\n"
        "2:"
        ::"S" (src),"D" (dest),"c" (count):"si","di","ax","cx");
return dest;                         // Returns the dest string pointer.
}

//// Copy the source string to the end of the destination string.
// Parameters: dest - the destination str pointer, src - the source str pointer.
// In the assembly code, %0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1).
extern inline char * strcat(char * dest,const char * src)
{
__asm__("cld\n\t"
        "repne\n\t"                  // Compare AL and ES:[EDI] byte and update EDI++,
        "scasb\n\t"                  // Until byte in dest is 0, the EDI points to last byte.
        "decl %1\n"                  // Let ES:[EDI] points to the location of 0.
        "1:\tlodsb\n\t"              // Take source str byte DS:[ESI] to AL, and ESI++.
        "stosb\n\t"                  // Store this byte to ES:[EDI] and EDI++.
        "testb %%al,%%al\n\t"        // Is this byte 0?
        "jne 1b"                     // If not,jump back to label 1 to continue, else end.
        ::"S" (src),"D" (dest),"a" (0),"c" (0xffffffff):"si","di","ax","cx");
return dest;                         // Returns the dest str pointer.
}

//// Copy count bytes of source str to the end of the dest str, and finally add a null char.
// dest - the destination str, src - the source str, count - number of bytes to copy.
// In the assembly code, %0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1), %4 - (count).
extern inline char * strncat(char * dest,const char * src,int count)
{
__asm__("cld\n\t"
        "repne\n\t"                  // Compare AL and ES:[EDI], and update EDI++,
        "scasb\n\t"                  // Until 0 in dest, the EDI points to last byte.
        "decl %1\n\t"                // Let ES:[EDI] points to the location of 0.
        "movl %4,%3\n"               // Place the bytes to be copied into ECX.
        "1:\tdecl %3\n\t"            // ECX-- (counts from 0).
        "js 2f\n\t"                  // Ecx < 0 ?, yes, jump forward to the label 2.
        "lodsb\n\t"                  // Otherwise take the bytes at DS:[ESI] to AL, ESI++.
        "stosb\n\t"                  // Store to ES:[EDI], and EDI++.
        "testb %%al,%%al\n\t"        // The byte value is 0?
        "jne 1b\n"                   // If not, jump back to label 1 and continue copying.
        "2:\txorl %2,%2\n\t"         // Clear AL to zero.
        "stosb"                      // Save to ES:[EDI].
        ::"S" (src),"D" (dest),"a" (0),"c" (0xffffffff),"g" (count)
        :"si","di","ax","cx");
return dest;                         // Returns the dest string pointer.
}

//// Compare two strings.
// Parameters: cs - string 1, ct - string 2.
// %0 - eax(__res) return value, %1 - edi(cs) str1 pointer, %2 - esi(ct) str2 pointer.
// If str1 > str2, returns 1; str1 = str2 returns 0; str1 < str2 returns -1.
extern inline int strcmp(const char * cs,const char * ct)
{
register int __res __asm__("ax");    // __res is a register variable (eax).
__asm__("cld\n"
        "1:\tlodsb\n\t"              // Get str2 byte DS:[ESI] to AL, and ESI++.
        "scasb\n\t"                  // Compare AL with str1 byte ES:[EDI], and EDI++.
        "jne 2f\n\t"                 // If they are not equal, jump forward to label 2.
        "testb %%al,%%al\n\t"        // Is this byte 0 (end of string)?
        "jne 1b\n\t"                 // No, jump back to label 1, and continue comparison.
        "xorl %%eax,%%eax\n\t"       // Yes, EAX is cleared, jumps forward to label 3,end.
        "jmp 3f\n"
        "2:\tmovl $1,%%eax\n\t"      // EAX is set to 1.
        "jl 3f\n\t"                  // If str2 < str1, returns a positive value, end.
        "negl %%eax\n"               // Else EAX = -EAX, return a negative value, and end.
        "3:"
        :"=a" (__res):"D" (cs),"S" (ct):"si","di");
return __res;                        // Returns the comparison result.
}

//// String 1 is compared to the first count characters of string 2.
// Parameters: cs - str1, ct - str2, count - the number of characters to compare.
// %0 - eax(__res), %1 - edi(cs) str1 pointer, %2 - esi(ct) str2 pointer, %3 - ecx(count).
// If str1 > str2, returns 1; str1 = str2 returns 0; str1 < str2 returns -1.
extern inline int strncmp(const char * cs,const char * ct,int count)
{
register int __res __asm__("ax");    // __res is a register variable (eax).
__asm__("cld\n"
        "1:\tdecl %3\n\t"            // Register ECX-- (count--).
        "js 2f\n\t"                  // If count<0, then jump forward to label 2.
        "lodsb\n\t"                  // Load str2 char DS:[ESI] to AL, and ESI++.
        "scasb\n\t"                  // Compare AL with str1 char ES:[EDI], and EDI++.
        "jne 3f\n\t"                 // If they are not equal, jump forward to label 3.
        "testb %%al,%%al\n\t"        // Is it a NULL char?
        "jne 1b\n"                   // No, jump back to label 1,continue the comparison.
        "2:\txorl %%eax,%%eax\n\t"   // Yes, then EAX is cleared (return value).
        "jmp 4f\n"                   // Jump forward to label 4 and end.
        "3:\tmovl $1,%%eax\n\t"      // Set EAX to 1.
        "jl 4f\n\t"                  // If the result is str2 < str1, 1 is returned.
        "negl %%eax\n"               // Else EAX = -EAX, return a negative value and end.
        "4:"
        :"=a" (__res):"D" (cs),"S" (ct),"c" (count):"si","di","cx");
return __res;                        // Returns the comparison result.
}

//// Look for the first matching character in the string.
// Parameters: s - the string, c - the character to be found.
// In the assembly code, %0 - eax(__res), %1 - esi (str pointer s), %2 - eax (char c).
// Returns a pointer to the first occurrence of a matching character in the string. If no
// matching characters are found, a null pointer is returned.
extern inline char * strchr(const char * s,char c)
{
```

```
register char * __res __asm__("ax");
__asm__("cld\n\t"
        "movb %%al,%%ah\n"          // Move the char to be compared to AH.
        "1:\tlodsb\n\t"             // Get a string char DS:[ESI] to AL, and ESI++.
        "cmpb %%ah,%%al\n\t"        // The char AL is compared with the specified char AH.
        "je 2f\n\t"                 // If they are the same, jump forward to the label 2.
        "testb %%al,%%al\n\t"       // Is the char in AL a NULL char? (End of string?)
        "jne 1b\n\t"                // If not, back to label 1 and continue comparison.
        "movl $1,%1\n"              // If yes, no match char is found, and set ESI to 1.
        "2:\tmovl %1,%0\n\t"        // Put pointer at the next byte of match char into EAX.
        "decl %0"                   // Adjust the pointer to point to the matching char.
        :"=a" (__res):"S" (s),"0" (c):"si");
return __res;                       // Returns the pointer.
}

//// Look for the last occurrence of the given character in the string. (reverse search string)
// Parameters: s - the string, c - the character to be found.
// In the assembly code, %0 - eax(__res), %1 - esi (string pointer s), %2 - eax (char c).
// Returns the pointer to the last occurrence of a matching character in the string. If no
// matching characters are found, a null pointer is returned.
extern inline char * strrchr(const char * s,char c)
{
register char * __res __asm__("dx");
__asm__("cld\n\t"
        "movb %%al,%%ah\n"          // Move the char to be compared to AH.
        "1:\tlodsb\n\t"             // Get a string char DS:[ESI] to AL, and ESI++.
        "cmpb %%ah,%%al\n\t"        // The char AL is compared with the specified char AH.
        "jne 2f\n\t"                // If not the same, jump forward to the label 2.
        "movl %%esi,%0\n\t"         // Store the char pointer to EDX.
        "decl %0\n"                 // Adjust the pointer to point to the matching char.
        "2:\ttestb %%al,%%al\n\t"   // Is the char in AL a NULL char? (End of string?)
        "jne 1b"                    // If not, back to label 1 and continue comparison.
        :"=d" (__res):"0" (0),"S" (s),"a" (c):"ax","si");
return __res;                       // Returns the pointer.
}

// Find the first sequence of chars in string1, any char in the sequence is contained in string2.
// Parameters: cs - string1 pointer, ct - string2 pointer.
// %0 - esi(__res), %1 - eax(0), %2 - ecx(-1), %3 - esi(str1 ptr cs), %4 - (str2 ptr ct).
// Returns the length of the first sequence of chars in str1 contained in str2.
extern inline int strspn(const char * cs, const char * ct)
{
register char * __res __asm__("si");
__asm__("cld\n\t"
        "movl %4,%%edi\n\t"         // Calc the len of str2, store its pointer to EDI.
        "repne\n\t"                 // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
        "scasb\n\t"                 // If they are not equal, continue to compare (ECX--).
        "notl %%ecx\n\t"            // Each bit in ECX is inversed.
        "decl %%ecx\n\t"            // ECX--, The length of str2 is obtained.
        "movl %%ecx,%%edx\n"        // Put the length of str2 into EDX temporarily.
        "1:\tlodsb\n\t"             // Put the str1 char DS:[ESI] to AL, and ESI++.
        "testb %%al,%%al\n\t"       // Is the char equal to 0 (end of str1)?
        "je 2f\n\t"                 // If yes, jump forward to label 2 and end.
        "movl %4,%%edi\n\t"          // Get the str2 pointer into EDI.
        "movl %%edx,%%ecx\n\t"       // Put the str2 length into ECX.
        "repne\n\t"                  // Compare AL and str2 char ES:[EDI], and EDI++.
        "scasb\n\t"                  // If they are equal, continue to compare.
        "je 1b\n"                    // If they are equal, jump backward to label 1.
        "2:\tdecl %0"                // ESI--, points to the char in str1 contained in str2.
        :"=S" (__res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
        :"ax","cx","dx","di");
return __res-cs;                     // Returns the length of the sequence of chars.
}

//// Search the first char sequence in string1 that does not contain any of the chars in string2.
// Parameters: cs - string1 pointer, ct - string2 pointer.
// %0 - esi(__res), %1 - eax(0), %2 - ecx(-1), %3 - esi (str1 ptr cs), %4 - (str2 ptr ct).
// Returns the length of first char sequence in str1 that doesn't contain any of chars in str2.
extern inline int strcspn(const char * cs, const char * ct)
{
register char * __res __asm__("si");
__asm__("cld\n\t"
        "movl %4,%%edi\n\t"          // Calc the len of str2, store its pointer to EDI.
        "repne\n\t"                  // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
        "scasb\n\t"                  // If they are not equal, continue to compare (ECX--).
        "notl %%ecx\n\t"             // Calc the length of str2 by inversing each bits
        "decl %%ecx\n\t"             // in ECX and decreasing ECX by one.
        "movl %%ecx,%%edx\n"         // Put str2 length into EDX temporarily.
        "1:\tlodsb\n\t"              // Take the str1 char DS:[ESI] to AL, and ESI++.
        "testb %%al,%%al\n\t"        // Is the char equal to 0 (end of str1)?
        "je 2f\n\t"                  // If yes, jump forward to label 2 and end.
        "movl %4,%%edi\n\t"          // Get the str2 pointer into EDI.
        "movl %%edx,%%ecx\n\t"       // Put the str2 length into ECX.
        "repne\n\t"                  // Compare AL and str2 char ES:[EDI], and EDI++.
        "scasb\n\t"                  // If they not are equal, continue to compare.
        "jne 1b\n"                   // If they not are equal, jump backward to label 1.
        "2:\tdecl %0"                // ESI--, points to the char in str1 contained in str2.
        :"=S" (__res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
        :"ax","cx","dx","di");
return __res-cs;                     // Returns the length of the sequence of chars.
}

//// In string1, look for any first character contained in string2.
// Parameters: cs - string1 pointer, ct - string2 pointer.
// %0 -esi(__res), %1 -eax(0), %2 -ecx(0xffffffff), %3 -esi(str1 ptr cs), %4 -(str2 ptr ct).
// Returns the pointer in string1 containing the first character in string2.
extern inline char * strpbrk(const char * cs,const char * ct)
{
register char * __res __asm__("si");
__asm__("cld\n\t"
        "movl %4,%%edi\n\t"          // Calc the len of str2, store its pointer to EDI.
        "repne\n\t"                  // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
        "scasb\n\t"                  // If they are not equal, continue to compare (ECX--).
        "notl %%ecx\n\t"             // Calc the length of str2 by inversing each bits
        "decl %%ecx\n\t"             // in ECX and decreasing ECX by one.
        "movl %%ecx,%%edx\n"         // Store str2 length into EDX temporarily.
        "1:\tlodsb\n\t"             // Get the str1 char DS:[ESI] to AL, and ESI++.
        "testb %%al,%%al\n\t"       // Is the char equal to 0 (end of str1)?
        "je 2f\n\t"                 // If yes, jump forward to label 2.
        "movl %4,%%edi\n\t"         // Get the str2 pointer into EDI.
        "movl %%edx,%%ecx\n\t"      // Put the str2 length into ECX.
        "repne\n\t"                 // Compare AL and str2 char ES:[EDI], and EDI++.
        "scasb\n\t"                 // If they not are equal, continue to compare.
        "jne 1b\n\t"                // If they not are equal, jump backward to label 1.
        "decl %0\n\t"               // ESI--, points to a char in str1 contained in str2.
        "jmp 3f\n"                  // Jump forward to label 3 and end.
        "2:\txorl %0,%0\n"          // If no match is found, it will return NULL.
        "3:"
        :"=S" (__res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
        :"ax","cx","dx","di");
return __res;                       // Returns the pointer in str1.
}

//// In string1, look for the first substring that matches the entire string2.
// Parameters: cs - string1 pointer, ct - string2 pointer.
// %0 -eax(__res), %1 -eax(0), %2 -ecx(0xffffffff), %3 -esi(str1 ptr cs), %4 -(str2 ptr ct).
// Returns the first substring pointer in string1 that matches entire string2.
extern inline char * strstr(const char * cs,const char * ct)
{
register char * __res __asm__("ax");
__asm__("cld\n\t" \
        "movl %4,%%edi\n\t"         // Calc the len of str2, store its pointer to EDI.
        "repne\n\t"                 // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
        "scasb\n\t"                 // If they are not equal, continue to compare (ECX--).
        "notl %%ecx\n\t"            // Calc str2 len by inversing all bits and dec 1 in ECX.
        "decl %%ecx\n\t"    /* NOTE! This also sets Z if searchstring='' */
        "movl %%ecx,%%edx\n"        // Store str2 length into EDX temporarily.
        "1:\tmovl %4,%%edi\n\t"     // Get the str2 pointer into EDI.
        "movl %%esi,%%eax\n\t"      // Copy the str1 pointer to EAX.
        "movl %%edx,%%ecx\n\t"      // Put the str2 length into ECX.
        "repe\n\t"                  // Comp str1,str2 char DS:[ESI],ES:[EDI], ESI++,EDI++.
        "cmpsb\n\t"                 // If the chars are equal, they continue to compare.
                                    // If they are all equal, jump to label 2, end.
        "je 2f\n\t"         /* also works for empty string, see above */
        "xchgl %%eax,%%esi\n\t"     // str1 ptr => ESI, comparison result str1 ptr => EAX.
        "incl %%esi\n\t"            // str1 pointer points to the next char.
        "cmpb $0,-1(%%eax)\n\t"     // Is the byte pointed to by ptr in str1 (EAX-1) 0?
        "jne 1b\n\t"               // If not, go label1, continue comp from 2nd char of str1.
        "xorl %%eax,%%eax\n\t"      // Clear EAX, indicating that no match was found.
        "2:"
        :"=a" (__res):"0" (0),"c" (0xffffffff),"S" (cs),"g" (ct)
        :"cx","dx","di","si");
return __res;                       // Returns the result.
}

//// Calculate the length of a string.
// Parameter: s - a string.
// %0 - ecx(__res), %1 - edi (string pointer s), %2 - eax(0), %3 - ecx(0xffffffff).
// Returns the length of the string.
```

```
extern inline int strlen(const char * s)
{
register int __res __asm__("cx");     // __res is a register variable (ECX).
__asm__("cld\n\t"                     // Clear the direction flag.
        "repne\n\t"                   // AL(0) is compared with char es:[edi] in the string,
        "scasb\n\t"                   // If they are not equal, continue to compare.
        "notl %0\n\t"                 // Inverse each bit in ECX.
        "decl %0"                     // ECX--, the length of the string is obtained.
        :"=c" (__res):"D" (s),"a" (0),"0" (0xffffffff):"di");
return __res;                         // Returns the string length.
}

// a temporary string pointer, used to store the pointer to the parsed string (s) below.
extern char * ___strtok;              // string token.

//// Using the characters in string2, string1 is segmented into a token sequence.
// String1 is considered to be a sequence containing zero or more tokens and separated by one
// or more characters in the separator string2. When strtok() is called for the first time,
// a pointer to the first char of the first token in string1 is returned, and a null character
// is written to the separator when the token is returned. Subsequent calls to strtok() with
// null as the first argument will continue to scan string1 in this way until there is no token.
// The split string2 can be different during different invocations.
// Params: s - the string1 to be processed, ct - the string2 containing each separator.
// %0-ebx(__res), %1-esi(__strtok); %2-ebx(__strtok), %3-esi(str1 ptr s), %4-(str2 ptr ct).
// Returns a token in the string s, or a null pointer if no token is found. Subsequent calls
// to strtok() with a null string s will search for the next token in the original string s.
extern inline char * strtok(char * s,const char * ct)
{
register char * __res __asm__("si");
__asm__("testl %1,%1\n\t"             // First test if ESI (str1 pointer s) is NULL.
        "jne 1f\n\t"                  // If not, means the first call, and jump label 1.
        "testl %0,%0\n\t"             // If NULL, means subsequent call, test EBX(__strtok).
        "je 8f\n\t"                   // If EBX is NULL, it cannot be processed,jump to end.
        "movl %0,%1\n"                // Copy the EBX pointer(current str1 ptr) to ESI.
        "1:\txorl %0,%0\n\t"          // Clear EBX.
        "movl $-1,%%ecx\n\t"          // Set ECX = 0xffffffff.
        "xorl %%eax,%%eax\n\t"        // Clear EAX.
        "cld\n\t"                     // Clear direction flag.
        "movl %4,%%edi\n\t"           // Let's find the str2 length. EDI points to str2.
        "repne\n\t"                   // Compare AL(0) with char at ES:[EDI], and EDI++.
        "scasb\n\t"                   // Until end of str2 (null char), or count ECX = 0.
        "notl %%ecx\n\t"              // Inverse ECX and subtract 1,to get str2 length.
        "decl %%ecx\n\t"              // If str2 len is 0, go forward to label 7.
        "je 7f\n\t"                   /* empty delimeter-string */
        "movl %%ecx,%%edx\n"          // Store str2 length to EDX temporarily.
        "2:\tlodsb\n\t"               // Get str1 char DS:[ESI] to AL, and ESI++.
        "testb %%al,%%al\n\t"         // Is the char 0 (end of str1)?
        "je 7f\n\t"                   // If yes, jump forward to label 7.
        "movl %4,%%edi\n\t"           // EDI again points to first char of the str2.
        "movl %%edx,%%ecx\n\t"        // Copy str2 length into the counter ECX.
        "repne\n\t"                   // Compare str1 char AL with all chars in str2,
        "scasb\n\t"                   // to check if the char is a separator.
        "je 2b\n\t"                   // If found it in str2, jump backward to label 2.
        "decl %1\n\t"                 // Else the str1 pointer ESI points to the char now.
        "cmpb $0,(%1)\n\t"            // Is this a NULL character?
        "je 7f\n\t"                   // If yes, jump forward to label 7.
        "movl %1,%0\n"                // The char pointer ESI is stored in EBX.
        "3:\tlodsb\n\t"               // Get next char of str1 to DS:[ESI], and ESI++.
        "testb %%al,%%al\n\t"         // Is the char 0 (end of str1)?
        "je 5f\n\t"                   // If yes, means str1 ends and jump forward to label 5.
        "movl %4,%%edi\n\t"           // EDI again points to first char of the str2.
        "movl %%edx,%%ecx\n\t"        // Copy str2 length into the counter ECX.
        "repne\n\t"                   // Compare str1 char AL with all chars in str2,
        "scasb\n\t"                   // to check if the char is a separator.
        "jne 3b\n\t"                  // If not, jump to label3 to detect the next char in str1.
        "decl %1\n\t"                 // If it is a separator, ESI--, points to the separator.
        "cmpb $0,(%1)\n\t"            // Is this a NULL character?
        "je 5f\n\t"                   // If yes, jump forward to label 5.
        "movb $0,(%1)\n\t"            // If not, replaced the separator with a NULL char.
        "incl %1\n\t"                 // ESI points to next char in str1 (remaining str).
        "jmp 6f\n"                    // Jump forward to label 6.
        "5:\txorl %1,%1\n"            // Clear ESI.
        "6:\tcmpb $0,(%0)\n\t"        // Does the EBX pointer point to a NULL char?
        "jne 7f\n\t"                  // If not, jump forward to label 7.
        "xorl %0,%0\n"                // If yes, let the EBX = NULL. (end of str1)
        "7:\ttestl %0,%0\n\t"         // EBX is NULL?
        "jne 8f\n\t"                  // If not, jump forward to label 8, end return.
        "movl %0,%1\n"                // Set ESI to NULL, means no more token in str1.
        "8:"
        :"=b" (__res),"=S" (___strtok)
        :"0" (___strtok),"1" (s),"g" (ct)
        :"ax","cx","dx","di");
return __res;                         // Returns pointer to the new token.
}

//// Memory copy. Copy n bytes from source address src to the destination address dest.
// Params: dest - the destination address, src - the source address, n - number of bytes.
// %0 - ecx(n), %1 - esi(src), %2 - edi(dest).
extern inline void * memcpy(void * dest,const void * src, int n)
{
__asm__("cld\n\t"                     // Clear direction flag.
        "rep\n\t"                     // Repeatedly copying ECX bytes,
        "movsb"                       // From DS:[ESI++] to ES:[EDI++].
        ::"c" (n),"S" (src),"D" (dest)
        :"cx","si","di");
return dest;                          // Returns the dest address.
}

//// The memory moves. Same as memory copy operation, but consider the direction of the move.
// Params: dest - the destination address, src - the source address, n - number of bytes.
// If (dest < src) then: %0 - ecx(n), %1 - esi(src), %2 - edi(dest);
// otherwise: %0 - ecx(n), %1 - esi(src + n - 1 ), %2 - edi(dest + n - 1).
// This directional operation is to prevent data overlapping during copying.
extern inline void * memmove(void * dest,const void * src, int n)
{
if (dest<src)
```

```
__asm__("cld\n\t"                    // Clear direction flag.
        "rep\n\t"                    // From DS:[ESI++] to ES:[EDI++],
        "movsb"                      // Repeatedly copying ECX bytes.
        ::"c" (n),"S" (src),"D" (dest)
        :"cx","si","di");
else
__asm__("std\n\t"                    // Set the direction and start copying from the end.
        "rep\n\t"                    // From DS:[ESI--] to ES:[EDI--],
        "movsb"                      // Repeatedly copying ECX bytes.
        ::"c" (n),"S" (src+n-1),"D" (dest+n-1)
        :"cx","si","di");
return dest;                         // Returns the dest address.
}

//// Compare n bytes of two memory blocks, and don't stop comparing even if encounter NULL char.
// Params: cs - mem block1 address, ct - mem block2 address, count - number of bytes compared.
// %0 - eax(__res), %1 - eax(0), %2 - edi (mem block1), %3 - esi (mem block2), %4 - ecx(count).
// If block1 > block2 returns 1; block1 < block2, returns -1; block1 == block2, returns 0.
extern inline int memcmp(const void * cs,const void * ct,int count)
{
register int __res __asm__("ax");
__asm__("cld\n\t"
        "repe\n\t"                   // Compare DS:[ESI++] with ES:[EDI++].
        "cmpsb\n\t"                  // If equal, continue comparing. repeat ECX times.
        "je 1f\n\t"                  // If all the same, jump to label 1, return 0 (EAX).
        "movl $1,%%eax\n\t"          // else set EAX to 1.
        "jl 1f\n\t"                  // If value of mem block2 <block1, jump to label 1.
        "negl %%eax\n"               // else invert EAX value.
        "1:"
        :"=a" (__res):"0" (0),"D" (cs),"S" (ct),"c" (count)
        :"si","di","cx");
return __res;                        // Returns the compare result (in EAX).
}

//// Look for the specified character in a memory block of n bytes.
// Params: cs - the memory block address, c - the specified char, count - the compare size.
//%0 - edi(__res), %1 - eax (char c), %2 - edi (mem block address cs), %3 - ecx (num of bytes).
// Returns the pointer to the first matching location, or NULL if not found.
extern inline void * memchr(const void * cs,char c,int count)
{
register void * __res __asm__("di");
if (!count)                          // If memory block size is 0, then NULL is returned.
        return NULL;
__asm__("cld\n\t"
        "repne\n\t"                  // Compare the char in AL with the ES:[EDI++],
        "scasb\n\t"                  // Repeat if it is not equal (up to ECX times).
        "je 1f\n\t"                  // If equal, jump forward to label 1.
        "movl $1,%0\n"               // else set EDI to 1.
        "1:\tdecl %0"                // Let EDI point to the char searched.(or NULL).
        :"=D" (__res):"a" (c),"D" (cs),"c" (count)
        :"cx");
return __res;                        // Return the char pointer.
}
```

```
//// Fill in the memory block with the specified character.
// Fill in the memory area pointed to by 's' with the char 'c' and fill in 'count' bytes.
//%0 - eax (char c), %1 - edi (memory address s), %2 - ecx (number of bytes count).
extern inline void * memset(void * s,char c,int count)
{
__asm__("cld\n\t"
        "rep\n\t"                       // Repeat ECX times,
        "stosb"                         // Fill the char in AL into ES:[EDI++].
        ::"a" (c),"D" (s),"c" (count)
        :"cx","di");
return s;                               // Returns the memory block pointer.
}

#endif

```

14.11 termios.h

14.11.1 Functionality

The termios.h file contains terminal I/O interface definitions, including termios data structures and some function prototypes for common terminal interface settings. These functions are used to read or set the properties of the terminal, line control, read or set the baud rate, and read or set the group ID of the terminal front end process. Although this is an early Linux header file, it is fully compliant with the current POSIX standard and has been appropriately extended.

The two terminal data structures termio and termios defined in this file belong to two types of UNIX series (or clones), termio is defined in AT&T system V, and termios is specified by POSIX standard. The two structures are basically the same, except that termio uses a short integer type to define the mode flag set, while termios uses a long integer to define the mode flag set. Since both structures are currently in use, most systems support them for compatibility. In addition, a similar sgtty structure has been used before, and it has not been used at present.

14.11.2 Code annotation

Program 14-10 linux/include/termios.h

```
#ifndef _TERMIOS_H
#define _TERMIOS_H

#include <sys/types.h>

#define TTY_BUF_SIZE 1024               // The buffer size in the tty (in bytes).

/* 0x54 is just a magic number to make these relatively uniqe ('T') */

// The following is the command set used by the tty's ioctl, which encodes it in the low byte.
// Get the information in the terminal termios structure (see tcgetattr()).
#define TCGETS          0x5401        // Terminal Command GET Settings.
// Set the information in the terminal termios structure (see tcsetattr(), TCSANOW).
#define TCSETS          0x5402
// Before setting the information in termios, you need to wait for all the data in the output
// queue to be processed (used up). This command is required for modifying the parameters that
// affect the output (see tcsetattr(), TCSADRAIN option).(TCSETSW - TCSETS Wait)
#define TCSETSW         0x5403
// Before setting the termios information, you need to wait for all the data in the output queue
// to be processed, and flush (clear) the input queue (See tcsetattr(), TCSAFLUSH option).
#define TCSETSF         0x5404
// Take the attribute information in the termio structure (see tcgetattr()).
#define TCGETA          0x5405
// Set the attribute information in the termio structure (see tcsetattr(), TCSANOW option).
#define TCSETA          0x5406
// Before setting the attribute information of termio, you need to wait for all the data in
// the output queue to be processed (run out). This type of command is required for modifying
// the parameters that affect the output (see tcsetattr(), TCSADRAIN option).
#define TCSETAW         0x5407
// Before setting the termio attributes, you need to wait for all data in the output queue to
// be processed, and flush (used up) the input queue (see tcsetattr(), TCSAFLUSH option).
#define TCSETAF         0x5408
// Wait for the output queue to be processed (empty). If the parameter value is 0, send a break
// (see tcsendbreak(), tcdrain()).
#define TCSBRK          0x5409
// Start/stop control. If the parameter value is 0, the output is suspended; if it is 1, the
// pending output is re-opened; if it is 2, the input is suspended; if it is 3, the pending
// input is re-opened (see tcflow()).
#define TCXONC          0x540A
// Flushes data that has been written but has not been sent, or has been received but not yet
// read. If the argument is 0, the input queue is flushed (cleared); if it is 1, the output
// queue is flushed; if it is 2, the input and output queues are flushed (see tcflush()).
#define TCFLSH          0x540B
// Set the terminal serial line exclusive mode.
#define TIOCEXCL        0x540C       // Terminal I/O Control Exclude.
// Reset the terminal serial line exclusive mode.
#define TIOCNXCL        0x540D
// Set tty as the control terminal. (TIOCNOTTY - prohibits the tty as a control terminal).
#define TIOCSCTTY       0x540E
// Get the group ID of the specified terminal device process (see tcgetpgrp()).
#define TIOCGPGRP       0x540F      // Terminal IO Control Get PGRP.
// Set the group ID of the specified terminal device process (see tcsetpgrp()).
#define TIOCSPGRP       0x5410
// Returns the number of characters in the output queue that have not been sent.
#define TIOCOUTQ        0x5411
// Simulate the terminal input. The command takes a pointer to a character as a parameter and
// pretends that the character is typed on the terminal. The user must have superuser privileges
// or read permission on the control terminal.
#define TIOCSTI         0x5412
// Get the terminal device window size information (see the winsize structure).
#define TIOCGWINSZ      0x5413
// Set the terminal device window size information (see the winsize structure).
#define TIOCSWINSZ      0x5414
```

```
   // Returns current status bit flag set for the MODEM status control pin (see lines 185-196).
30 #define TIOCMGET        0x5415
   // Set the status of the MODEM state individual control line (true or false).
31 #define TIOCMBIS        0x5416
   // Clear (reset) the status of the MODEM state individual control line.
32 #define TIOCMBIC        0x5417
   // Set the status of the MODEM state control line. If a bit is set, the status line corresponding
   // to the MODEM will be set to be valid.
33 #define TIOCMSET        0x5418
   // Get the software carrier detect flag (1 - on; 0 - off). For locally connected terminals or
   // other devices, the software carrier flag is turned on, and it is turned off for terminals
   // or devices that use MODEM lines. In order to be able to use these two ioctl calls, the tty
   // line should be opened in O_NDELAY mode, so open() will not wait for the carrier.
34 #define TIOCGSOFTCAR    0x5419
   // Set the software carrier detect flag (1 - on; 0 - off).
35 #define TIOCSSOFTCAR    0x541A
   // Returns the number of characters in the input queue that have not been fetched.
36 #define FIONREAD        0x541B
37 #define TIOCINQ         FIONREAD
38
   // Window size attribute structure. It can be used for screen-based applications. The TIOCGWINSZ
   // and TIOCSWINSZ commands in ioctls can be used to read or set this information.
39 struct winsize {
40         unsigned short ws_row;       // window character rows.
41         unsigned short ws_col;       // window character colums.
42         unsigned short ws_xpixel;    // window width in pixels.
43         unsigned short ws_ypixel;    // window hight in pixels.
44 };
45
   // The termio structure of the AT&T system V.
46 #define NCC 8                        // the size of the control character array in termio.
47 struct termio {
48         unsigned short c_iflag;      /* input mode flags */
49         unsigned short c_oflag;      /* output mode flags */
50         unsigned short c_cflag;      /* control mode flags */
51         unsigned short c_lflag;      /* local mode flags */
52         unsigned char c_line;        /* line discipline */
53         unsigned char c_cc[NCC];     /* control characters */
54 };
55
   // The termios structure defined by POSIX.
56 #define NCCS 17                      // the size of the control character array in termios.
57 struct termios {
58         tcflag_t c_iflag;            /* input mode flags */
59         tcflag_t c_oflag;            /* output mode flags */
60         tcflag_t c_cflag;            /* control mode flags */
61         tcflag_t c_lflag;            /* local mode flags */
62         cc_t c_line;                 /* line discipline */
63         cc_t c_cc[NCCS];             /* control characters */
64 };
65
   // The following is the index of the item in the control character array c_cc[]. The initial
   // value of this array is defined in include/linux/tty.h. The program can change the value in
// this array. If _POSIX_VDISABLE(\0) is defined, then when the value of an array is equal to
// _POSIX_VDISABLE, it means that the special characters in the array are prohibited.
/* c_cc characters */
#define VINTR 0         // c_cc[VINTR]    = INTR    (^C), \003.
#define VQUIT 1         // c_cc[VQUIT]    = QUIT    (^\), \034.
#define VERASE 2        // c_cc[VERASE]   = ERASE   (^H), \177.
#define VKILL 3         // c_cc[VKILL]    = KILL    (^U), \025.
#define VEOF 4          // c_cc[VEOF]     = EOF     (^D), \004.
#define VTIME 5         // c_cc[VTIME]    = TIME    (\0), \0.   timer value (see below).
#define VMIN 6          // c_cc[VMIN]     = MIN     (\1), \1.   timer value.
#define VSWTC 7         // c_cc[VSWTC]    = SWTC    (\0), \0.   switch char.
#define VSTART 8        // c_cc[VSTART]   = START   (^Q), \021.
#define VSTOP 9         // c_cc[VSTOP]    = STOP    (^S), \023.
#define VSUSP 10        // c_cc[VSUSP]    = SUSP    (^Z), \032. suspend char.
#define VEOL 11         // c_cc[VEOL]     = EOL     (\0), \0.
#define VREPRINT 12     // c_cc[VREPRINT] = REPRINT (^R), \022.
#define VDISCARD 13     // c_cc[VDISCARD] = DISCARD (^O), \017.
#define VWERASE 14      // c_cc[VWERASE]  = WERASE  (^W), \027. word erase char.
#define VLNEXT 15       // c_cc[VLNEXT]   = LNEXT   (^V), \026.
#define VEOL2 16        // c_cc[VEOL2]    = EOL2    (\0), \0.

// The constants of various flags used in the input mode field c_iflag in the termios.
/* c_iflag bits */
#define IGNBRK  0000001           // Ignore the BREAK condition when input.
#define BRKINT  0000002           // Generates a SIGINT signal on BREAK.
#define IGNPAR  0000004           // Ignore the character of the parity error.
#define PARMRK  0000010           // Mark parity error.
#define INPCK   0000020           // Allow input to check parity.
#define ISTRIP  0000040           // Mask the 8th bit of the character.
#define INLCR   0000100           // Map line feed NL to a carriage return CR.
#define IGNCR   0000200           // Ignore the CR.
#define ICRNL   0000400           // Map CR to NL.
#define IUCLC   0001000           // Convert uppercase chars to lowercase chars.
#define IXON    0002000           // Allow start/stop (XON/XOFF) output control.
#define IXANY   0004000           // Allow any character to restart the output.
#define IXOFF   0010000           // Allow start/stop (XON/XOFF) input control.
#define IMAXBEL 0020000           // Rings when the input queue is full.

// The constants of various flags used in the output mode field c_oflag in the termios.
/* c_oflag bits */
#define OPOST   0000001           // Perform output processing.
#define OLCUC   0000002           // Convert lowercase chars to uppercase chars.
#define ONLCR   0000004           // Map NL to CR-NL.
#define OCRNL   0000010           // Map CR to NL.
#define ONOCR   0000020           // The CR is not output in the column 0.
#define ONLRET  0000040           // The NL performs the function of carriage return.
#define OFILL   0000100           // Use fill chars when deferred instead of time delays.
#define OFDEL   0000200           // The fill char is DEL. The default is NULL if not set.
#define NLDLY   0000400           // Choose a line feed delay.
#define   NL0   0000000           // NL delay type 0.
#define   NL1   0000400           // NL delay type 1.
#define CRDLY   0003000           // Choose a carriage return delay.
#define   CR0   0000000           // CR delay type 0.
```

```
#define   CR1   0001000            // CR delay type 1.
#define   CR2   0002000            // CR delay type 2.
#define   CR3   0003000            // CR delay type 2.
#define TABDLY  0014000            // Select horizontal TAB delay.
#define   TAB0  0000000            // TAB delay type 0.
#define   TAB1  0004000            // TAB delay type 1.
#define   TAB2  0010000            // TAB delay type 2.
#define   TAB3  0014000            // TAB delay type 3.
#define   XTABS 0014000            // Replace TAB with spaces, it is the number of spaces.
#define BSDLY   0020000            // Select the backspace (BS) delay.
#define   BS0   0000000            // BS delay type 0.
#define   BS1   0020000            // BS delay type 1.
#define VTDLY   0040000            // Select vertical tabulation delay.
#define   VT0   0000000            // VT delay type 0.
#define   VT1   0040000            // VT delay type 1.
#define FFDLY   0040000            // Select Form feed delay.
#define   FF0   0000000            // FF delay type 0.
#define   FF1   0040000            // FF delay type 1.

// The flags used in the control mode field c_cflag in the termios (octals).
/* c_cflag bit meaning */
#define CBAUD   0000017         // Transmit baud rate bit mask.
#define  B0     0000000         /* hang up */
#define  B50    0000001         // baud rate 50.
#define  B75    0000002
#define  B110   0000003
#define  B134   0000004
#define  B150   0000005
#define  B200   0000006
#define  B300   0000007
#define  B600   0000010
#define  B1200  0000011
#define  B1800  0000012
#define  B2400  0000013
#define  B4800  0000014
#define  B9600  0000015
#define  B19200 0000016
#define  B38400 0000017         // baud rate 38400.
#define EXTA B19200             // Extended baud rate A.
#define EXTB B38400             // Extended baud rate B.

#define CSIZE   0000060         // Character bit width mask.
#define   CS5   0000000         // 5 bits per character.
#define   CS6   0000020         // 6 bits.
#define   CS7   0000040         // 7 bits.
#define   CS8   0000060         // 8 bits.
#define CSTOPB  0000100         // Set two stop bits instead of one.
#define CREAD   0000200         // Allow to receive.
#define PARENB  0000400         // Enable parity operation.
#define PARODD  0001000         // The input check is odd check.
#define HUPCL   0002000         // Hang up after the last process is closed.
#define CLOCAL  0004000         // Ignore the MODEM control lines.
#define CIBAUD  03600000        /* input baud rate (not used) */
```

```
#define CRTSCTS 020000000000     /* flow control */

// The flags used in the local mode field c_lflag in the termios.
/* c_lflag bits */
#define ISIG    0000001         // signal generated when revice INTR, QUIT, SUSP or DSUSP .
#define ICANON  0000002         // Enable canonical mode (cooked mode).
#define XCASE   0000004         // If ICANON is set, terminal displays uppercase chars.
#define ECHO    0000010         // Echo the input characters.
#define ECHOE   0000020         // If ICANON, ERASE/WERASE erase the previous char/word.
#define ECHOK   0000040         // If ICANON, KILL char will erase the current line.
#define ECHONL  0000100         // If ICANON, NL is echoed even if ECHO is not enabled.
// The input/output queues are not flushed when the SIGINT and SIGQUIT signals are generated,
// and the input queue is flushed when the SIGSUSP signal is generated.
#define NOFLSH  0000200
// Send the SIGTTOU signal to the process group of the background process, which tries to write
// its own control terminal.
#define TOSTOP  0000400
// If ECHO is set, ASCII control characters other than TAB, NL, START, and STOP will be echoed
// back to the '^X' pattern, and the X value is the control value +0x40.
#define ECHOCTL 0001000
#define ECHOPRT 0002000        // If ICANON & IECHO, chars will be displayed when erasing.
#define ECHOKE  0004000        // If ICANON, KILL is echoed by erasing each char on he line.
#define FLUSHO  0010000        // Output is flushed. it can be toggled by DISCARD char.
#define PENDIN  0040000        // All chars in inqueue are reprinted when next char is read.
#define IEXTEN  0100000        // Enable implementation-defined input processing.

/* modem lines */
#define TIOCM_LE        0x001       // Line Enable.
#define TIOCM_DTR       0x002       // Data Terminal Ready.
#define TIOCM_RTS       0x004       // Request to Send.
#define TIOCM_ST        0x008       // Serial Transfer.
#define TIOCM_SR        0x010       // Serial Receive.
#define TIOCM_CTS       0x020       // Clear To Send.
#define TIOCM_CAR       0x040       // Carrier Detect.
#define TIOCM_RNG       0x080       // Ring indicate.
#define TIOCM_DSR       0x100       // Data Set Ready.
#define TIOCM_CD        TIOCM_CAR
#define TIOCM_RI        TIOCM_RNG

/* tcflow() and TCXONC use these */
#define TCOOFF          0          // Suspend output ("Terminal Control Output OFF").
#define TCOON           1          // Restart suspended output.
#define TCIOFF          2          // Send a STOP to stop device transmiting data to system.
#define TCION           3          // Send a START to start device transmiting to system.

/* tcflush() and TCFLSH use these */
#define TCIFLUSH        0          // Clear the received data but do not read it.
#define TCOFLUSH        1          // Clear the output data but not transmit it.
#define TCIOFLUSH       2          // Clear in/out data but not read/transmit it.

/* tcsetattr uses these */
#define TCSANOW         0          // Enable settings now.
#define TCSADRAIN       1          // Change occurs after all output has been transmitted.
```

```
#define TCSAFLUSH       2          // Change occurs after input/output are all flushed.

    // The following functions are implemented in the function library libc.a in the build
    // environment, not in the kernel. In the library implementation, these functions are implemented
    // by calling the system-call ioctl(). For ioctl() see the fs/ioctl.c program.

    // Returns the receive baud rate in the termios structure referred to by termios_p.
extern speed_t cfgetispeed(struct termios *termios_p);
    // Returns the send baud rate in the termios structure.
extern speed_t cfgetospeed(struct termios *termios_p);
    // Set the receive baud rate in the termios structure to 'speed'.
extern int cfsetispeed(struct termios *termios_p, speed_t speed);
    // Set the send baud rate in the termios structure.
extern int cfsetospeed(struct termios *termios_p, speed_t speed);
    // Wait for the written data of the object pointed to by fildes to be transmitted.
extern int tcdrain(int fildes);
    // Suspend/restart the reception/transmission data of the object pointed to by fildes.
extern int tcflow(int fildes, int action);
    // Discards all written but not yet transmitted data for the fildes specified object, as well
    // as all data that has been received but not yet read.
extern int tcflush(int fildes, int queue_selector);
    // Get the parameters of the object corresponding to the handle fildes and save it in termios.
extern int tcgetattr(int fildes, struct termios *termios_p);
    // If the terminal uses asynchronous serial transmission, a series of zero value bits are
    // continuously transmitted for a certain period of time.
extern int tcsendbreak(int fildes, int duration);
    // Use the data of the termios structure to set the parameters related to the terminal.
extern int tcsetattr(int fildes, int optional_actions,
        struct termios *termios_p);

#endif
```

14.11.3 Information

14.11.3.1 Control Character TIME、MIN

When the canonical mode flag ICANON is disabled, the input is in the non-canonical mode (raw mode). In non-canonical mode, input characters are not processed into lines, and input characters are immediately readable, without the need to enter (user type) carriage-return, line feed, etc. line-defining characters. Therefore, erasing and terminating processing will not occur. The settings of MIN (c_cc[VMIN]) and TIME (c_cc[VTIME]) in termios structure are used to determine how to handle the received characters, how many characters are read by the associated read() system-call, and when to return to the user program.

MIN represents the minimum amount of characters that need to be read when a read operation is satisfied (ie, when the character is returned to the user). TIME is a timing value counted in 1/10 second for timeout values for burst and short-term data transfers. The four combinations of these two control characters and their interactions are described as follows:

◆ MIN > 0, TIME > 0:

In this case, TIME acts as a timer between characters and starts to function after receiving the first character. It will be reset and restarted every time it receives a character. The interaction between MIN and

TIME is as follows: Once a character is received, the inter-character timer begins to work. If MIN characters are received before the timer expires (note that the timer restarts every time a character is received), the read operation is satisfied. If the timer expires before the MIN characters are received, the characters that have been received at this time are returned to the user. Note that if TIME times out, at least one character will be returned because the timer will only start functioning after receiving a character. So in this case (MIN > 0, TIME > 0), the read operation will sleep until the first character is received to activate the MIN and TIME mechanisms. If the number of characters read is less than the number of existing characters, the timer will not be reactivated and subsequent read operations will be satisfied immediately.

◆ MIN > 0, TIME = 0:

In this case, since the value of TIME is 0, the timer does not work and only MIN makes sense. Only when the MIN characters are received, the waiting read operation will be satisfied (the waiting operation will sleep until MIN characters are received). Programs that use this to read the record-based terminal IO will be blocked (arbitrarily) indefinitely in the read operation.

◆ MIN = 0, TIME > 0:

In this case, since MIN=0, TIME no longer functions as a timer between characters, but a read operation timer and functions at the beginning of the read operation. The read operation is satisfied as soon as a character is received or the timer expires. Note that in this case, if the timer expires, no characters will be read. If the timer does not time out, the read operation will only be satisfied after reading one character. Therefore, in this case, the read operation is not blocked indefinitely (indeterminately) to wait for characters. After the start of the read operation, if no characters are received within TIME*0.10 seconds, the read operation will return with 0 characters.

◆ MIN = 0, TIME = 0:

In this case, the read operation will return immediately. The minimum number of characters requested to be read or the number of existing characters in the buffer queue will be returned without waiting for more characters to be input into the buffer.

In general, in non-canonical mode, these two values are timeout timing values and character count values. MIN indicates the minimum number of characters that need to be read in order to satisfy the read operation. TIME is a timing value counted by one tenth of a second. When both are set, the read operation will wait until at least one character is read, then return after reading the MIN characters or return the read character due to the TIME timeout. If only MIN is set, the read operation will not return until the MIN characters are read. If only TIME is set, the read will return immediately after reading at least one character or timing out. If neither is set, the read will return immediately, giving only the number of bytes currently read.

14.12 time.h

14.12.1 Functionality

The time.h header file refers to functions that process time and date. There is a very interesting description of time in MINIX: “Time processing is more complicated, such as what is GMT (Greenwich Mean Time, now UTC time), local time or other time. Even though Bishop Ussher (1581-1656) once calculated, according to the Bible, the world begins at 9 am on October 12, 4004 BC. But in the UNIX world, time began at midnight on January 1, 1970, GMT, before it was empty and chaotic". Where UTC means universal time code.

This file is one of the header files in the standard C library. Since some of the UNIX operating system developers were amateur astronomy enthusiasts at the time, they were particularly strict with the time representation in UNIX systems, so that time and date representations and calculations are particularly complicated in UNIX-like or in standard C-compatible systems. This file defines one constant symbol (macro), four types, and some time and date operation conversion functions. In addition, some of the function declarations given in this file are functions provided by the standard C library, which are not included in the kernel.

In the Linux 0.12 kernel, this file mainly provides the tm structure type for the init/main.c and kernel/mktime.c files, which is used by the kernel to obtain real-time clock information (calendar time) from the system CMOS chip, so that the system boot time can be set. The boot time is the time (in seconds) elapsed since 0:00 on January 1, 1970, and will be stored in the global variable startup_time for all code reading by the kernel.

14.12.2 Code annotation

Program 14-11 linux/include/time.h

```
#ifndef _TIME_H
#define _TIME_H

#ifndef _TIME_T
#define _TIME_T
typedef long time_t;               // The time from GMT on January 1, 1970 at midnight.
#endif

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif

#ifndef NULL
#define NULL ((void *) 0)
#endif

#define CLOCKS_PER_SEC 100         // System clock tick frequency, 100HZ.

typedef long clock_t;              // Ticks passed by system from the start of a process.

struct tm {
        int tm_sec;                // Seconds [0, 59].
        int tm_min;                // Minutes [0, 59].
        int tm_hour;               // Hours [0, 59].
        int tm_mday;               // The number of days in a month [0, 31].
        int tm_mon;                // The number of months in a year [0, 11].
        int tm_year;               // The number of years since 1900.
        int tm_wday;               // One day of the week [0, 6] (Sunday =0).
        int tm_yday;               // One day in a year [0, 365].
        int tm_isdst;              // Summer time sign. > 0 -in use; = 0 -not used; < 0 -invalid.
};

// A macro that checks to see if it is a leap year.
#define __isleap(year)  \
  ((year) % 4 == 0 && ((year) % 100 != 0 || (year) % 1000 == 0))

// Here are some function prototypes for time operations.
// Get processor usage time. Returns an appr.processor time (ticks) used by the program.
clock_t clock(void);
// Get time. Returns the number of seconds since 1970.1.1:0:0:0 (the calendar time).
time_t time(time_t * tp);
// Calculate the time difference. Returns the number of seconds elapsed between time2 and time1.
double difftime(time_t time2, time_t time1);
// Convert the time represented by the tm structure to calendar time.
time_t mktime(struct tm * tp);

// Converts the time in tm structure into a string and returns a pointer to the string.
char * asctime(const struct tm * tp);
// Convert the calendar time to a string, such as "Wed Jun 30 21:49:08:1993\n".
char * ctime(const time_t * tp);
// Convert the calendar time to UTC time represented by the tm structure.
struct tm * gmtime(const time_t *tp);
// Converts the calendar time to the time in specified time zone represented by tm structure.
struct tm *localtime(const time_t * tp);
// The time represented by the tm structure is converted to a string of maximum length 'smax'
// using the format string 'fmt' and the result is stored in 's'.
size_t strftime(char * s, size_t smax, const char * fmt, const struct tm * tp);
// Initialize the time conversion information and initialize the zname variable using the
// environment variable TZ. This function is called automatically in the time conversion function
// associated with the time zone.
void tzset(void);

#endif

```

14.13 unistd.h

14.13.1 Functionality

Standard symbol constants and types are defined in the unistd.h header file. There are many different symbol constants and types defined in this file, as well as some function declarations.如 If the symbol __LIBRARY__ is defined in the program, it will also include the kernel system-call number and the inline assembly macro _syscall0(), _syscall1(), and so on.

14.13.2 Code annotation

Program 14-12 linux/include/unistd.h

```
#ifndef _UNISTD_H
#define _UNISTD_H

/* ok, this may be a joke, but I'm working on it */
// The symbol constant below indicates the version of the IEEE Standard 1003.1 implementation
// that the kernel follows, which is an integer value.
```

```
#define _POSIX_VERSION 198808L

#define _POSIX_CHOWN_RESTRICTED /* only root can do a chown (I think..) */
// Path names longer than NAME_MAX will generate an error and will not be automatically truncated.
#define _POSIX_NO_TRUNC         /* no pathname truncation (but see in kernel) */
// _POSIX_VDISABLE is used to control the function of some special characters of the terminal.
// When a character code value of the c_cc[] array in a terminal termios structure is equal
// to the value of _POSIX_VDISABLE, it means that the corresponding special character function
// is prohibited.
#define _POSIX_VDISABLE '\0'    /* character to disable things like ^C */
// Indicates that the system implementation supports job control.
#define _POSIX_JOB_CONTROL
// Each process has a saved set-user-ID and a saved set-group-ID.
#define _POSIX_SAVED_IDS        /* Implemented, for whatever good it is */

#define STDIN_FILENO    0           // Standard input file handle (descriptor) number.
#define STDOUT_FILENO   1           // Standard output file handle number.
#define STDERR_FILENO   2           // Standard error output file handle number.

#ifndef NULL
#define NULL    ((void *)0)         // Define the value of a null pointer.
#endif

// The symbol constants defined below are used for the access() function.
/* access */
#define F_OK    0                   // Check if the file exists.
#define X_OK    1                   // Check if it is executable (searchable).
#define W_OK    2                   // Check if it is writable.
#define R_OK    4                   // Check if it is readable.

// The following symbol constants are used for the lseek() and fcntl() functions.
/* lseek */
#define SEEK_SET        0           // Set file read/write pointer to the offset.
#define SEEK_CUR        1           // Set to the current value plus the offset.
#define SEEK_END        2           // Set to the file length plus the offset.

// The following symbol constants are used in the sysconf() function.
/* _SC stands for System Configuration. We don't use them much */
#define _SC_ARG_MAX             1   // The maximum number of arguments.
#define _SC_CHILD_MAX           2   // The maximum number of child processes.
#define _SC_CLOCKS_PER_SEC      3   // Ticks per second.
#define _SC_NGROUPS_MAX         4   // The maximum number of groups.
#define _SC_OPEN_MAX            5   // The maximum number of opened files.
#define _SC_JOB_CONTROL         6   // Job control.
#define _SC_SAVED_IDS           7   // The saved identifier.
#define _SC_VERSION             8   // Version.

// The following symbol constants are used for the pathconf() function.
/* more (possibly) configurable things - now pathnames */
#define _PC_LINK_MAX            1   // The maximum number of links.
#define _PC_MAX_CANON           2   // The maximum number of regular files.
#define _PC_MAX_INPUT           3   // Maximum input length.
#define _PC_NAME_MAX            4   // The maximum length of the name.
```

```
#define _PC_PATH_MAX          5    // The maximum length of a path.
#define _PC_PIPE_BUF          6    // Pipe buffer size.
#define _PC_NO_TRUNC          7    // The file name is not truncated.
#define _PC_VDISABLE          8    // The specified control char is disabled.
#define _PC_CHOWN_RESTRICTED  9    // Change the owner is restricted.

  // <sys/stat.h> File status header file. Contains file or file system state structures stat{}
  //     and constants.
  // <sys/time.h> The timeval structure and the itimerval structure are defined.
  // <sys/times.h> Defines the running time structure tms and the times() function prototype in
  //     the process.
  // <sys/utsname.h> System name structure header file.
  // <sys/resource.h> Resource file. Contains information on the limits and utilization of system
  //     resources used by processes.
  // <utime.h> User time header file. The access and modification time structures and the utime()
  //     prototype are defined.
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/times.h>
#include <sys/utsname.h>
#include <sys/resource.h>
#include <utime.h>

#ifdef __LIBRARY__

   // The following is the system-call symbol constants implemented by the kernel, used as an index
   // in the system-call function table (see include/linux/sys.h).
#define __NR_setup      0       /* used only by init, to get system going */
#define __NR_exit       1
#define __NR_fork       2
#define __NR_read       3
#define __NR_write      4
#define __NR_open       5
#define __NR_close      6
#define __NR_waitpid    7
#define __NR_creat      8
#define __NR_link       9
#define __NR_unlink     10
#define __NR_execve     11
#define __NR_chdir      12
#define __NR_time       13
#define __NR_mknod      14
#define __NR_chmod      15
#define __NR_chown      16
#define __NR_break      17
#define __NR_stat       18
#define __NR_lseek      19
#define __NR_getpid     20
#define __NR_mount      21
#define __NR_umount     22
#define __NR_setuid     23
#define __NR_getuid     24
#define __NR_stime      25
```

```
#define __NR_ptrace     26
#define __NR_alarm      27
#define __NR_fstat      28
#define __NR_pause      29
#define __NR_utime      30
#define __NR_stty       31
#define __NR_gtty       32
#define __NR_access     33
#define __NR_nice       34
#define __NR_ftime      35
#define __NR_sync       36
#define __NR_kill       37
#define __NR_rename     38
#define __NR_mkdir      39
#define __NR_rmdir      40
#define __NR_dup        41
#define __NR_pipe       42
#define __NR_times      43
#define __NR_prof       44
#define __NR_brk        45
#define __NR_setgid     46
#define __NR_getgid     47
#define __NR_signal     48
#define __NR_geteuid    49
#define __NR_getegid    50
#define __NR_acct       51
#define __NR_phys       52
#define __NR_lock       53
#define __NR_ioctl      54
#define __NR_fcntl      55
#define __NR_mpx        56
#define __NR_setpgid    57
#define __NR_ulimit     58
#define __NR_uname      59
#define __NR_umask      60
#define __NR_chroot     61
#define __NR_ustat      62
#define __NR_dup2       63
#define __NR_getppid    64
#define __NR_getpgrp    65
#define __NR_setsid     66
#define __NR_sigaction  67
#define __NR_sgetmask   68
#define __NR_ssetmask   69
#define __NR_setreuid   70
#define __NR_setregid   71
#define __NR_sigsuspend 72
#define __NR_sigpending 73
#define __NR_sethostname 74
#define __NR_setrlimit  75
#define __NR_getrlimit  76
#define __NR_getusage  77
#define __NR_gettimeofday 78
```

```
141 #define __NR_settimeofday 79
142 #define __NR_getgroups  80
143 #define __NR_setgroups  81
144 #define __NR_select     82
145 #define __NR_symlink    83
146 #define __NR_lstat      84
147 #define __NR_readlink   85
148 #define __NR_uselib     86
149
    // The following defines the system-call embedded assembly macro function.
    // The system-call macro function with no arguments: type name(void).
    // %0 - eax(__res), %1 - eax(__NR_##name). Where name is the name of a system-call, combined
    // with __NR_ to form the above system-call symbol constant, which is used to address the
    // function pointer in the system call table. In the macro definition, if there are two
    // consecutive well signs '##' between the two symbols, it means that the two symbols are
    // connected together to form a new symbol when the macro is replaced. For example, '__NR_##name'
    // on line 156 below, after replacing the parameter 'name' (for example, 'fork'), the last
    // occurrence in the program will be the symbol '__NR_fork'.
    // Returns: if the return value is greater than or equal to 0, the value is returned, otherwise
    // the error number errno is set and -1 is returned.
150 #define _syscall0(type,name) \
151 type name(void) \
152 { \
153 long __res; \
154 __asm__ volatile ("int $0x80" \          // system interrupt 0x80.
155         : "=a" (__res) \                 // The return value is placed in eax(__res).
156         : "0" (__NR_##name)); \          // The input is system interrupt number __NR_name.
157 if (__res >= 0) \                        // If value >=0, the value is returned directly.
158         return (type) __res; \
159 errno = -__res; \                        // Otherwise set error number and return -1.
160 return -1; \
161 }
162
    // A system-call macro function with 1 parameter: type name(atype a)
    // %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a).
163 #define _syscall1(type,name,atype,a) \
164 type name(atype a) \
165 { \
166 long __res; \
167 __asm__ volatile ("int $0x80" \
168         : "=a" (__res) \
169         : "0" (__NR_##name),"b" ((long)(a))); \
170 if (__res >= 0) \
171         return (type) __res; \
172 errno = -__res; \
173 return -1; \
174 }
175
    // A system call macro function with 2 parameters: type name(atype a, btype b)
    // %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b).
176 #define _syscall2(type,name,atype,a,btype,b) \
177 type name(atype a,btype b) \
178 { \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name), "b" ((long)(a)), "c" ((long)(b))); \
if (__res >= 0) \
        return (type) __res; \
errno = -__res; \
return -1; \
}

// A system call macro function with 3 parameters: type name(atype a, btype b, ctype c)
// %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b), %4 - edx(c).
// The system call of this kernel has up to three parameters. If there is more data, you can
// put the data in a buffer and pass the buffer pointer as a parameter to the kernel.
#define _syscall3(type,name,atype,a,btype,b,ctype,c) \
type name(atype a,btype b,ctype c) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(a)),"c" ((long)(b)),"d" ((long)(c))); \
if (__res>=0) \
        return (type) __res; \
errno=-__res; \
return -1; \
}

#endif /* __LIBRARY__ */

extern int errno;                       // Error number, a global variable.

// The following are the function prototype definitions for each system-call.
// see include/linux/sys.h for details.
int access(const char * filename, mode_t mode);
int acct(const char * filename);
int alarm(int sec);
int brk(void * end_data_segment);
void * sbrk(ptrdiff_t increment);
int chdir(const char * filename);
int chmod(const char * filename, mode_t mode);
int chown(const char * filename, uid_t owner, gid_t group);
int chroot(const char * filename);
int close(int fildes);
int creat(const char * filename, mode_t mode);
int dup(int fildes);
int execve(const char * filename, char ** argv, char ** envp);
int execv(const char * pathname, char ** argv);
int execvp(const char * file, char ** argv);
int execl(const char * pathname, char * arg0, ...);
int execlp(const char * file, char * arg0, ...);
int execle(const char * pathname, char * arg0, ...);
// The keyword 'volatile' before the function name is used to tell the compiler gcc that the
// function will not return. This will allow gcc to produce better code. More importantly, using
// this keyword avoids generating false warnings for certain (uninitialized variables). This
// is equivalent to gcc's function attribute description:
// void do_exit(int error_code) __attribute__ ((noreturn));
volatile void exit(int status);
volatile void _exit(int status);
int fcntl(int fildes, int cmd, ...);
int fork(void);
int getpid(void);
int getuid(void);
int geteuid(void);
int getgid(void);
int getegid(void);
int ioctl(int fildes, int cmd, ...);
int kill(pid_t pid, int signal);
int link(const char * filename1, const char * filename2);
int lseek(int fildes, off_t offset, int origin);
int mknod(const char * filename, mode_t mode, dev_t dev);
int mount(const char * specialfile, const char * dir, int rwflag);
int nice(int val);
int open(const char * filename, int flag, ...);
int pause(void);
int pipe(int * fildes);
int read(int fildes, char * buf, off_t count);
int setpgrp(void);
int setpgid(pid_t pid,pid_t pgid);
int setuid(uid_t uid);
int setgid(gid_t gid);
void (*signal(int sig, void (*fn)(int)))(int);
int stat(const char * filename, struct stat * stat_buf);
int fstat(int fildes, struct stat * stat_buf);
int stime(time_t * tptr);
int sync(void);
time_t time(time_t * tloc);
time_t times(struct tms * tbuf);
int ulimit(int cmd, long limit);
mode_t umask(mode_t mask);
int umount(const char * specialfile);
int uname(struct utsname * name);
int unlink(const char * filename);
int ustat(dev_t dev, struct ustat * ubuf);
int utime(const char * filename, struct utimbuf * times);
pid_t waitpid(pid_t pid,int * wait_stat,int options);
pid_t wait(int * wait_stat);
int write(int fildes, const char * buf, off_t count);
int dup2(int oldfd, int newfd);
int getppid(void);
pid_t getpgrp(void);
pid_t setsid(void);
int sethostname(char *name, int len);
int setrlimit(int resource, struct rlimit *rlp);
int getrlimit(int resource, struct rlimit *rlp);
int getrusage(int who, struct rusage *rusage);
int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(struct timeval *tv, struct timezone *tz);
int getgroups(int gidsetlen, gid_t *gidset);
int setgroups(int gidsetlen, gid_t *gidset);
int select(int width, fd_set * readfds, fd_set * writefds,
        fd_set * exceptfds, struct timeval * timeout);

#endif

```

14.14 utime.h

14.14.1 Functionality

The utime.h file defines the file access and modification time structure utimbuf{}, and the utime() function prototype, where time is in seconds.

14.14.2 Code annotation

Program 14-13 linux/include/utime.h

```
#ifndef _UTIME_H
#define _UTIME_H

// <sys/types.h> Type header file. The basic system data types and file system parameter
//       structure are defined.
#include <sys/types.h>  /* I know - shouldn't do this, but .. */

struct utimbuf {
        time_t actime;          // File access time. seconds from 1970.1.1:0:0:0.
        time_t modtime;         // File modified time. seconds from 1970.1.1:0:0:0.
};

// Function to set file access and modify time.
extern int utime(const char *filename, struct utimbuf *times);

#endif

```

14.15 Files in the include/asm/ directory

List 14-2 linux/include/asm/目录下的文件

Filename	Size	Last Modified Time (GMT)	Description
io.h	477 bytes	1991-08-07 10:17:51	
memory.h	507 bytes	1991-06-15 20:54:44	
segment.h	1366 bytes	1991-11-25 18:48:24	
system.h	1707 bytes	1992-01-13 13:02:10	

14.16 io.h

14.16.1 Functionality

The embedded assembly macro functions that access the hardware IO port are defined in the io.h file: outb(), inb(), and outb_p() and inb_p().The main difference between the first two functions and the latter two is that the jmp instruction is used in the latter code for time delay.

14.16.2 Code annotation

Program 14-14 linux/include/asm/io.h

```
//// Hardware port byte output function.
// Parameters: value - the byte of the output; port - the port.
#define outb(value,port) \
__asm__ ("outb %%al,%%dx"::"a" (value),"d" (port))


//// Hardware port byte input function.
// Returns the input byte.
#define inb(port) ({ \
unsigned char _v; \
__asm__ volatile ("inb %%dx,%%al":"=a" (_v):"d" (port)); \
_v; \
})

//// Hardware port byte output function with delay. Use two jump statements to delay for a while.
// Parameters: value - the byte to be exported; port - the port.
#define outb_p(value,port) \
__asm__ ("outb %%al,%%dx\n" \
                "\tjmp 1f\n" \            // Jump forward to label 1 (the next statement).
                "1:\tjmp 1f\n" \          // Jump forward to label 1.
                "1:"::"a" (value),"d" (port))

```

```
//// Hardware port byte input function with delay. Use two jump statements to delay for a while.
// Returns the input byte.
#define inb_p(port) ({ \
unsigned char _v; \
__asm__ volatile ("inb %%dx,%%al\n" \
        "\tjmp 1f\n" \                      // Jump forward to label 1 (the next statement).
        "1:\tjmp 1f\n" \                    // Jump forward to label 1.
        "1:":"=a" (_v):"d" (port)); \
_v; \
})

```

14.17 memory.h

14.17.1 Functionality

The memory.h file contains a memory copy embedded assembly macro function memcpy().It is identical to memcpy() defined in string.h, except that the latter is defined in the form of an embedded assembly C function.

14.17.2 Code annotation

Program 14-15 linux/include/asm/memory.h

```
/*
 *  NOTE!!! memcpy(dest,src,n) assumes ds=es=normal data segment. This
 *  goes for all kernel functions (ds=es=kernel space, fs=local data,
 *  gs=null), as well as for all well-behaving user programs (ds=es=
 *  user data space). This is NOT a bug, as any user program that changes
 *  es deserves to die if it isn't careful.
 */
//// Memory block copy. Copy n bytes from source address src to the destination dest.
// %0 - edi (address dest), %1 - esi (address src), %2 - ecx (number of bytes n).
#define memcpy(dest,src,n) ({ \
void * _res = dest; \
__asm__ ("cld;rep;movsb" \          // Copy from DS:[ESI++] to ES:[EDI++],total ECX(n) bytes.
        ::"D" ((long)(_res)),"S" ((long)(src)),"c" ((long) (n)) \
        :"di","si","cx"); \
_res; \
})

```

14.18 segment.h

14.18.1 Functionality

The segment.h file defines some functions that access the Intel CPU segment registers, as well as some memory manipulation functions associated with the segment registers. In a Linux system, when the user program starts executing kernel code through a system-call, the kernel code first loads the kernel data segment descriptor (segment value 0x10) in the GDT into the registers DS and ES, that is, DS and ES are used for access the kernel data segment; and the task data segment descriptor (segment value 0x17) in the LDT is loaded into the FS, that is, the FS is used to access the user data segment. See lines 92--96 in kernel/sys_call.s. Therefore, when executing kernel code, you need to use a special method to access the data in the user program (task). Functions such as get_fs_byte() and put_fs_byte() in this file are specifically used to access data in the user program.

14.18.2 Code annotation

Program 14-16 linux/include/asm/segment.h

```
//// Get a byte at the specified address in the FS segment.
// Parameters: addr - the specified memory address.
// %0 - (returned byte _v); %1 - (memory address addr).
// Returns a byte at memory FS:[addr].
extern inline unsigned char get_fs_byte(const char * addr)
{
        unsigned register char _v;     // a register variable for efficient access.

        __asm__ ("movb %%fs:%1,%0":"=r" (_v):"m" (*addr));
        return _v;
}

//// Get a word at the specified address in the FS segment.
// %0 - (returned word _v); %1 - (memory address addr).
// Returns a word at memory FS:[addr].
extern inline unsigned short get_fs_word(const unsigned short *addr)
{
        unsigned short _v;

        __asm__ ("movw %%fs:%1,%0":"=r" (_v):"m" (*addr));
        return _v;
}

//// Get a long word at the specified address in the FS segment.
// %0 - (returned long word _v); %1 - (memory address addr).
// Returns a long word at memory FS:[addr].
extern inline unsigned long get_fs_long(const unsigned long *addr)
{
        unsigned long _v;

        __asm__ ("movl %%fs:%1,%0":"=r" (_v):"m" (*addr)); \
        return _v;
}

//// Put a byte at the specified memory address in the FS segment.
// Parameters: val - the byte value; addr - the memory address.
// %0 - register (byte value val); %1 - (memory address addr).
extern inline void put_fs_byte(char val,char *addr)
{
__asm__ ("movb %0,%%fs:%1"::"r" (val),"m" (*addr));
}

//// Put a word at th specified memory address in the FS segment.
// Parameters: val - the word value; addr - the memory address.
// %0 - register (word value val); %1 - (memory address addr).
extern inline void put_fs_word(short val,short * addr)
{
__asm__ ("movw %0,%%fs:%1"::"r" (val),"m" (*addr));
}

//// Put a long word at th specified memory address in the FS segment.
// Parameters: val - the long word value; addr - the memory address.
// %0 - register (long word value val); %1 - (memory address addr).
extern inline void put_fs_long(unsigned long val,unsigned long * addr)
{
__asm__ ("movl %0,%%fs:%1"::"r" (val),"m" (*addr));
}

/*
 * Someone who knows GNU asm better than I should double check the followig.
 * It seems to work, but I don't know if I'm doing something subtly wrong.
 * --- TYT, 11/24/91
 * [ nothing wrong here, Linus ]
 */

//// Get FS segment register value (selector).
extern inline unsigned long get_fs()
{
        unsigned short _v;
        __asm__("mov %%fs,%%ax":"=a" (_v):);
        return _v;
}

//// Get DS segment register value.
extern inline unsigned long get_ds()
{
        unsigned short _v;
        __asm__("mov %%ds,%%ax":"=a" (_v):);
        return _v;
}

//// Set FS segment register.
extern inline void set_fs(unsigned long val)
{
        __asm__("mov %0,%%fs"::"a" ((unsigned short) val));
}


```

14.19 system.h

14.19.1 Functionality

The system.h file contains embedded assembly macros that set or modify segment descriptors/interrupt gate descriptors. Among them, the function move_to_user_mode() is used for the kernel to manually switch (move) to the initial process (task 0) when the kernel initialization ends, that is, to run from the code of the privilege level 0 code to the privilege level 3. The method used is to simulate the interrupt call return process, that is, use the IRET instruction to implement the privilege level change and the stack switch, thereby moving the CPU execution control flow to the environment of the initial task 0, as shown in Figure 14-3.

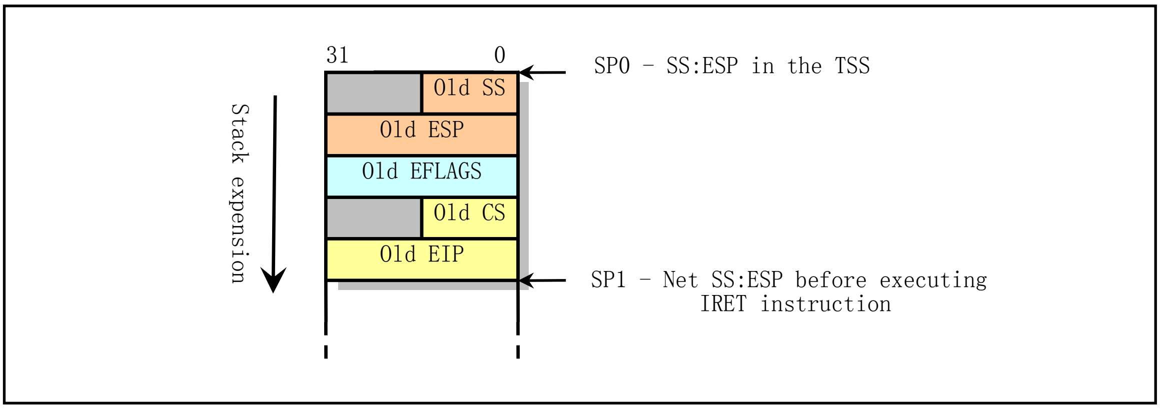


Figure 14-3 Using IRET instruction to implement privilege level change

The use of this method for privilege level changes and control transfer is caused by the CPU protection mechanism. By call gate, interrupt or trap gate, the CPU allows low-level (such as privilege level 3) code to call or transfer control to high-level code, but not vice versa. So the kernel uses the method of simulating IRET instruction returns to low-level code from a high level.

Before executing the task 0 code, the kernel first sets up the stack to simulate the content placement in the stack when the privilege level switch has just entered the interrupt call procedure. The IRET instruction is then executed, causing the system to move to task 0 for execution. When the IRET instruction is executed, the contents of the stack are as shown in Figure 14-3, at which point the stack pointer is ESP1. The stack of task 0 is the stack of the kernel. After the IRET is executed, it is moved to task 0 and executed. Since the task 0 descriptor privilege level is 3, the SS:ESP on the stack will also be popped up. So after IRET, ESP is equal to ESP0. Note that the interrupt return instruction IRET here does not cause the CPU to perform the task switching operation because the flag NT has been reset in sched_init() before executing this function. Executing the IRET instruction while NT is in the reset state does not cause the CPU to perform a task switching operation. It can be seen that the start of task 0 is purely manual.

Task 0 is a special process whose data segment and code segment map directly to the kernel code and data space, that is, 640K memory space starting from physical address 0, and its stack address is the stack used by the kernel code. Therefore, the original SS and the original ESP in the stack in the figure are formed by directly pushing the existing kernel stack pointer onto the stack.

Another part of the system.h file gives macros that set different types of descriptor entries in the interrupt descriptor table IDT. _set_gate() is a multi-parameter macro that is a generic macro that is called by the macro set_intr_gate() that sets the interrupt gate descriptor, the macro set_trap_gate() that sets the trap gate descriptor, and the macro set_system_gate() that sets the system gate descriptor. The format of the Interrupt Gate and Trap Gate descriptor entries in the IDT table is shown in Figure 14-4.

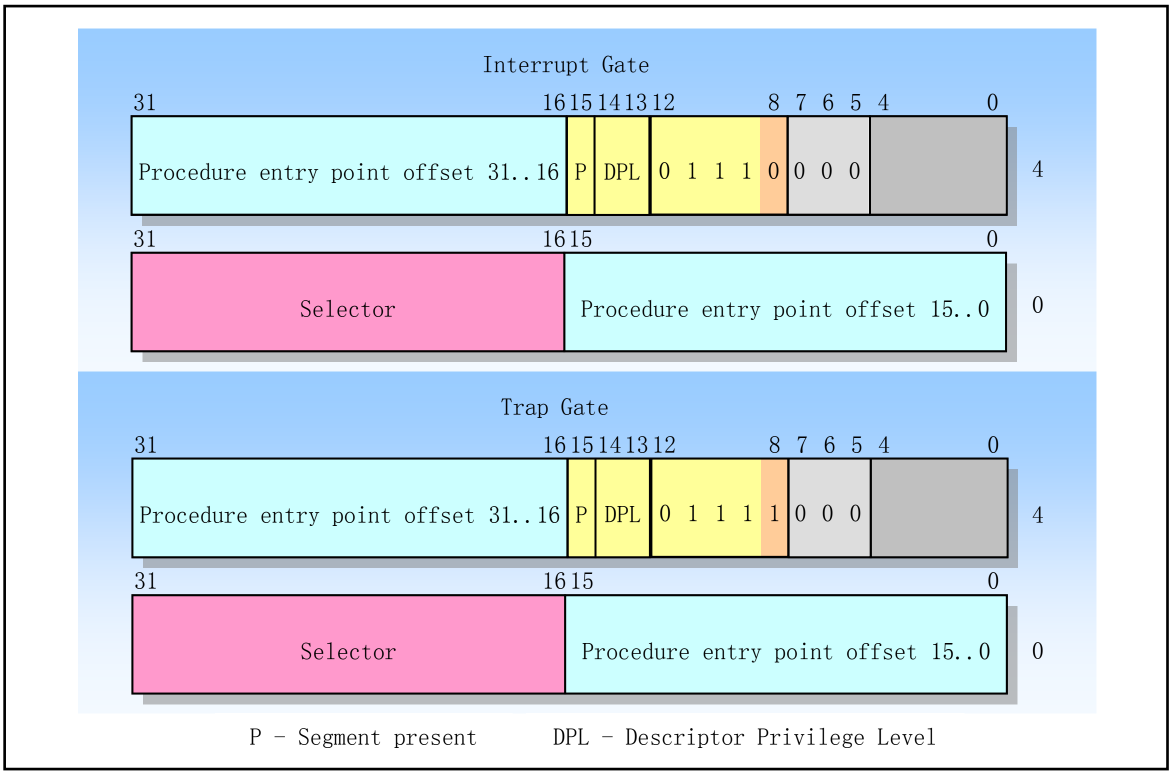

Interrupt Gate
31
16 15 14 13 12
8 7 6 5 4
0
Procedure entry point offset 31..16
P
DPL
0 1 1 1
0
0 0 0
4
31
16 15
0
Selector
Procedure entry point offset 15..0
0
Trap Gate
31
16 15 14 13 12
8 7 6 5 4
0
Procedure entry point offset 31..16
P
DPL
0 1 1 1
1
0 0 0
4
31
16 15
0
Selector
Procedure entry point offset 15..0
0
P - Segment present
DPL - Descriptor Privilege Level

Figure 14-4 Interrupt Gate and Trap Gate Descriptor Format in Table IDT

Where P is the segment presence flag; DPL is the privilege level of the descriptor. The difference between the interrupt gate and the trap gate is the effect of the EFLAGS interrupt enable flag IF. The interrupt executed by the interrupt gate descriptor resets the IF flag, so this way prevents other interrupts from interfering with the current interrupt processing, and the subsequent interrupt end instruction IRET will restore the original value of the IF flag from the stack; The interrupt executed by the trap gate will not affect the IF flag.

In the generic macro _set_gate (gate_addr, type, dpl, addr) used to set the descriptor, the parameter gate_addr specifies the physical memory address at which the descriptor is located. 'type' indicates the type of descriptor to be set, which corresponds to the lower 4 bits of the 6th byte in the descriptor format in Figure 14-4, so type=14(0x0E) indicates the interrupt gate descriptor, type=15 (0x0F) indicates the trap gate descriptor. The parameter 'dpl' is the DPL in the corresponding descriptor format, and 'addr' is the 32-bit offset address of the interrupt processing process corresponding to the descriptor. Because the interrupt processing is part of the kernel segment code, their segment selector values are all 0x0008 (specified in the EAX register high word).

The last part of the system.h file is used to set the general segment descriptor content and set the task state segment descriptor and the local table segment descriptor in the global descriptor table GDT. The meanings of the parameters of these macros is similar to the above.

14.19.2 Code annotation

Program 14-17 linux/include/asm/system.h

```
//// Move to user mode to run.
// This function uses the IRET instruction to move from kernel mode to initial task 0.
#define move_to_user_mode() \
__asm__ ("movl %%esp,%%eax\n\t" \      // Save stack pointer ESP to EAX register.
        "pushl $0x17\n\t" \            // First push the user stack segment SS,
        "pushl %%eax\n\t" \            // then push the stack pointer ESP on to stack,
        "pushfl\n\t" \                 // and push the EFLAGS register too.
        "pushl $0x0f\n\t" \            // Then push the code segment CS of task0,
        "pushl $1f\n\t" \              // and push the offset (EIP) at label 1.
        "iret\n" \                     // Execute the IRET, causes control ret to label 1.
        "1:\tmovl $0x17,%%eax\n\t" \   // At this point, kernel begins to execute task 0.
        "movw %%ax,%%ds\n\t" \         // Initialize segment register points to the data
        "movw %%ax,%%es\n\t" \         // segment of this local descriptor table.
        "movw %%ax,%%fs\n\t" \
        "movw %%ax,%%gs" \
        :::"ax")

#define sti() __asm__ ("sti"::)        // enable interrupt.
#define cli() __asm__ ("cli"::)        // disable interrupt.
#define nop() __asm__ ("nop"::)        // no op.

#define iret() __asm__ ("iret"::)      // interrupt return.

//// Macro for setting the gate descriptor.
// The gate descriptor located at the address gate_addr is set according to the interrupt or
// exception handling procedure address addr, the gate descriptor type type, and the privilege
// level information dpl in the parameter. (Note: The "offset" below is relative to the kernel
// code or data segment).
// Parameters: gate_addr - gate descriptor address; type - descriptor type field value;
// dpl - descriptor privilege level;  addr - offset address.
// %0 - (type flag word combined by 'dpl', 'type'); %1 - (descriptor low 4 byte address);
// %2 - (descriptor high 4 byte address);  %3 - EDX (program offset address addr);
// %4 - EAX (high word contains segment selector 0x0008).
#define _set_gate(gate_addr,type,dpl,addr) \
// The offset address low word and selector are combined into a descriptor lower 4 bytes (EAX).
// Combine the type flag word with the offset high word into descriptor high 4 bytes (EDX).
// Finally, the lower 4 bytes and the upper 4 bytes of the gate descriptor are set separately.
__asm__ ("movw %%dx,%%ax\n\t" \
        "movw %0,%%dx\n\t" \
        "movl %%eax,%1\n\t" \
        "movl %%edx,%2" \
        : \
        : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \     // %0
        "o" (*((char *) (gate_addr))), \                    // %1
        "o" (*(4+(char *) (gate_addr))), \                  // %2
        "d" ((char *) (addr)),"a" (0x00080000))             // %3, %4

```

```
   //// Set the interrupt gate (automatically mask subsequent interrupts).
   // Parameters: n - the interrupt number; addr - the interrupt program offset address.
   // Where '&idt[n]' is the offset value of the corresponding entry of the interrupt number n
   // in the interrupt descriptor table IDT; the type of the interrupt descriptor is 14, and the
   // privilege level is 0.
33 #define set_intr_gate(n,addr) \
34         _set_gate(&idt[n],14,0,addr)
35
   //// Set trap gate.
   // Parameters: n - the interrupt number; addr - the interrupt program offset address.
   // '&idt[n]' is the offset of the corresponding entry of the interrupt number 'n' in the IDT;
   // the type of the interrupt descriptor is 15, and the privilege level is 0.
36 #define set_trap_gate(n,addr) \
37         _set_gate(&idt[n],15,0,addr)
38
   //// Set system trap gate.
   // The descriptor set by set_trap_gate() above has a privilege level of 0, and here is 3.
   // Therefore, the interrupt processing set by set_system_gate() can be executed by all programs,
   // such as single-step debugging, overflow error, and boundary out of error processing.
   // Parameters: n - the interrupt number; addr - the interrupt program offset address.
   // '&idt[n]' is the offset of the corresponding entry of the interrupt number n in the IDT;
   // the type of the interrupt descriptor is 15, and the privilege level is 3.
39 #define set_system_gate(n,addr) \
40         _set_gate(&idt[n],15,3,addr)
41
   //// Set the segment descriptor (not used in the kernel).
   // Parameters: gate_addr - descriptor address; type - type field value in the descriptor;
   // dpl - descriptor privilege level; base - segment base address; limit - segment limit.
   // See the format of the segment descriptor. Note that the assignment object here is incorrect
   // (reversed). Line 43 should be '*((gate_addr)+1)', and line 49 is '*(gate_addr)'. However,
   // this macro is not used in the kernel code, so Linus is not aware :-)
42 #define _set_seg_desc(gate_addr,type,dpl,base,limit) {\
43         *(gate_addr) = ((base) & 0xff000000) | \             // Descriptor lower 4 bytes
44                 (((base) & 0x00ff0000)>>16) | \
45                 ((limit) & 0xf0000) | \
46                 ((dpl)<<13) | \
47                 (0x00408000) | \
48                 ((type)<<8); \
49         *((gate_addr)+1) = (((base) & 0x0000ffff)<<16) | \   // Descriptor high 4 bytes.
50                 ((limit) & 0x0ffff); }
51
   //// Set the task status segment/local table descriptor in the global table GDT.
   // The length of the status segment and the local table segment are both set to 104 bytes.
   // Parameters: n - the address corresponding to the descriptor item n in the global table GDT;
   // addr - the base address of the memory where the state segment/local table is located;
   // type - the flag type byte in the descriptor.
   // %0 - eax (address addr); %1 - (address of descriptor item n); %2 - (the offset 2 of the
   // descriptor item n); %3 - (the offset 4 of descriptor item n); %4 - (the offset 5 of the
   // descriptor item n); %5 - (the offset 6 of descriptor item n); %6 - (the offset 7 of the
   // descriptor item n);
52 #define _set_tssldt_desc(n,addr,type) \
53 __asm__ ("movw $104,%1\n\t" \      // The TSS length is stored in length field (0-th byte).
54         "movw %%ax,%2\n\t" \       // Put tne low word of base into the 2-3rd byte.
```

```
        "rorl $16,%%eax\n\t" \     // Rotate base high word into AX (low word to high).
        "movb %%al,%3\n\t" \       // Move low byte of the base high word to the 4th byte.
        "movb $" type ",%4\n\t" \  // Move flag type byte into the 5th byte.
        "movb $0x00,%5\n\t" \      // The sixth byte of the descriptor is set to zero.
        "movb %%ah,%6\n\t" \       // Move high byte of the base high word to the 7th byte.
        "rorl $16,%%eax" \         // Loop 16 bits to the right, EAX restores the original.
        ::"a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
         "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
        )

//// Set the task status segment (TSS) descriptor in the global table GDT.
// n - is a pointer to the descriptor; addr - is the base address of the segment in the descriptor
// entry. The type of TSS descriptor is of type 0x89.
#define set_tss_desc(n,addr) _set_tssldt_desc(((char *) (n)),addr,"0x89")
//// Set the local table (LDT) descriptor in the global table.
// n - is a pointer to the descriptor; addr - is the base address of the segment in the descriptor
// entry. The type of the local table segment descriptor is 0x82.
#define set_ldt_desc(n,addr) _set_tssldt_desc(((char *) (n)),addr,"0x82")

```

14.20 Files in the directory include/linux/

List 14-3 Files in the directory linux/include/linux/

Filename	Size	Last Modified Time(GMT)	Description
config.h	1545 bytes	1992-01-11 00:13:18	
fdreg.h	2466 bytes	1991-11-02 10:48:44	
fs.h	5754 bytes	1992-01-12 07:00:20	
hdreg.h	1968 bytes	1991-10-13 15:32:15	
head.h	304 bytes	1991-06-19 19:24:13	
kernel.h	1036 bytes	1992-01-12 02:17:34	
math_emu.h	4924 bytes	1992-01-01 17:33:04	
mm.h	1101 bytes	1992-01-13 15:46:41	
sched.h	7351 bytes	1992-01-13 22:24:42	
sys.h	3402 bytes	1992-01-13 21:42:37	
tty.h	2801 bytes	1992-01-08 22:51:56	

14.21 config.h

14.21.1 Functionality

Config.h is the kernel configuration header file that defines the machine configuration information used by the uname command, as well as the keyboard language type and hard disk type (HD_TYPE) options used.

14.21.2 Code annotation

Program 14-18 linux/include/linux/config.h

```
#ifndef _CONFIG_H
#define _CONFIG_H

/*
 * Defines for what uname() should return
 */
#define UTS_SYSNAME "Linux"
#define UTS_NODENAME "(none)"   /* set by sethostname() */
#define UTS_RELEASE ""          /* patchlevel */
#define UTS_VERSION "0.12"
#define UTS_MACHINE "i386"      /* hardware type */

/* Don't touch these, unless you really know what your doing. */
```

```
// The following symbolic constants are used to indicate the memory location during system boot
// and load of the kernel and the default maximum kernel system module size.
#define DEF_INITSEG     0x9000        // The segment to which the boot sector be moved.
#define DEF_SYSSEG      0x1000        // The segment to which the the system module loaded.
#define DEF_SETUPSEG    0x9020        // The segment where the setup program is located.
#define DEF_SYSSIZE     0x3000        // The maximum system module size (in units of 16).

/*
 * The root-device is no longer hard-coded. You can change the default
 * root-device by changing the line ROOT_DEV = XXX in boot/bootsect.s
 */

/*
 * The keyboard is now defined in kernel/chr_dev/keyboard.S
 */

/*
 * Normally, Linux can get the drive parameters from the BIOS at
 * startup, but if this for some unfathomable reason fails, you'd
 * be left stranded. For this case, you can define HD_TYPE, which
 * contains all necessary info on your harddisk.
 *
 * The HD_TYPE macro should look like this:
 *
 * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
 *
 * In case of two harddisks, the info should be sepatated by
 * commas:
 *
 * #define HD_TYPE { h,s,c,wpcom,lz,ctl },{ h,s,c,wpcom,lz,ctl }
 */
/*
 This is an example, two drives, first is type 2, second is type 3:

#define HD_TYPE { 4,17,615,300,615,8 }, { 6,17,615,300,615,0 }

 NOTE: ctl is 0 for all drives with heads<=8, and ctl=8 for drives
 with more than 8 heads.

 If you want the BIOS to tell what kind of drive you have, just
 leave HD_TYPE undefined. This is the normal thing to do.
*/

#endif

```

14.22 fdreg.h

14.22.1 Functionality

The fdreg.h header file is used to describe some of the parameters commonly used in floppy devices and the I/O ports used. Because the control of the floppy disk drive is cumbersome and there are many commands, it is best to refer to the book about the principle of the microcomputer interface before reading the code to understand how the floppy disk controller (FDC) works. Then you will think that the definition here is still reasonable and orderly.

When programming a floppy disk device, you need to access 4 ports, one for each register or multiple registers. For a 1.2M floppy disk controller there will be some of the ports given in Table 14-1.

Table 14-1 Floppy disk controller ports

I/O port	Read/Write	Register name
0x3f2	Write only	Digital output (control) Register
0x3f4	Read only	FDC main status register
0x3f5	Read/Write	FDC data register
0x3f7	Read only	Digital input register
0x3f7	Write only	Floppy disk control register (rate control)

The digital output port (digital control port) is an 8-bit register that controls the drive motor on, drive selection, start/reset FDC, and enable/disable DMA and interrupt requests.

The FDC main status register is also an 8-bit register that reflects the basic state of the floppy disk controller and the floppy disk drive (FDD). Typically, the status bits of the main status register are read before the CPU sends a command to the FDC or before the FDC obtains the result of the operation to determine if the current FDC data register is ready and to determine the direction of data transfer.

The data port of the FDC corresponds to multiple registers (write-only command register and parameter register, read-only result register), but only one register can appear on data port 0x3f5 at any one time. When accessing a write-only register, the DIO direction bit of the main state must be 0 (CPU FDC), and vice versa when accessing the read-only register. When reading the result, the result is only read after the FDC is not busy. Usually, the result data has a maximum of 7 bytes.

The floppy disk controller can accept a total of 15 commands. Each command goes through three phases: the command phase, the execution phase, and the results phase.

The command phase is that the CPU sends command bytes and parameter bytes to the FDC. The first byte of each command is always the command byte (command code) followed by a parameter of 0-8 bytes. The execution phase is the operation specified by the FDC execution command. In the execution phase, the CPU does not intervene. Generally, the FDC issues an interrupt request to let the CPU know the end of the command execution. If the FDC command sent by the CPU is to transfer data, the FDC can be performed in an interrupt mode or in a DMA manner. The interrupt mode transfers 1 byte at a time. The DMA mode is under the management of the DMA controller, and the FDC and the memory transfer data until all the data is transmitted. At this time, the DMA controller notifies the FDC of the transmission byte count termination signal, and finally the FDC issues an interrupt request signal to inform the CPU that the execution phase is over. The result phase is that the CPU reads the FDC data register return value to obtain the result of the FDC command execution. The result data returned is 0--7 bytes in length. For commands that do not return result data, the FDC should be sent a status to detect the interrupt status command acquisition operation.

14.22.2 Code annotation

Program 14-19 linux/include/linux/fdreg.h

```
/*
 * This file contains some defines for the floppy disk controller.
 * Various sources. Mostly "IBM Microcomputers: A Programmers
 * Handbook", Sanches and Canton.
 */
#ifndef _FDREG_H    // This definition is used to exclude duplicate header files in code.
#define _FDREG_H

// Prototype declaration of some floppy disk type functions.
extern int ticks_to_floppy_on(unsigned int nr);
extern void floppy_on(unsigned int nr);
extern void floppy_off(unsigned int nr);
extern void floppy_select(unsigned int nr);
extern void floppy_deselect(unsigned int nr);

// The definition of some ports and symbols for the floppy disk controller.
/* Fd controller regs. S&C, about page 340 */
#define FD_STATUS       0x3f4           // Main status register port.
#define FD_DATA         0x3f5           // Data port.
#define FD_DOR          0x3f2           /* Digital Output Register */
#define FD_DIR          0x3f7           /* Digital Input Register (read) */
#define FD_DCR          0x3f7           /* Diskette Control Register (write)*/

/* Bits of main status register */
#define STATUS_BUSYMASK 0x0F            /* drive busy mask */  // (one bit per driver).
#define STATUS_BUSY     0x10            /* FDC busy */
#define STATUS_DMA      0x20            /* 0- DMA mode */
#define STATUS_DIR      0x40            /* 0- cpu->fdc */
#define STATUS_READY    0x80            /* Data reg ready */

/* Bits of FD_ST0 */
#define ST0_DS          0x03            /* drive select mask */
#define ST0_HA          0x04            /* Head (Address) */
#define ST0_NR          0x08            /* Not Ready */
#define ST0_ECE         0x10            /* Equipment chech error */
#define ST0_SE          0x20            /* Seek end */    // or recalitrate end.
// Interrupt code bit (interrupt reason), 00 - command ends normally; 01 - command ends
// abnormally; 10 - command is invalid; 11 - FDD ready state changes.
#define ST0_INTR        0xC0            /* Interrupt code mask */

/* Bits of FD_ST1 */
#define ST1_MAM         0x01            /* Missing Address Mark */
#define ST1_WP          0x02            /* Write Protect */
#define ST1_ND          0x04            /* No Data - unreadable */  // sector not found.
```

```
#define ST1_OR          0x10            /* OverRun */      // Data transfer timeout.
#define ST1_CRC         0x20            /* CRC error in data or addr */
#define ST1_EOC         0x80            /* End Of Cylinder */

/* Bits of FD_ST2 */
#define ST2_MAM         0x01            /* Missing Addess Mark (again) */
#define ST2_BC          0x02            /* Bad Cylinder */
#define ST2_SNS         0x04            /* Scan Not Satisfied */
#define ST2_SEH         0x08            /* Scan Equal Hit */
#define ST2_WC          0x10            /* Wrong Cylinder */
#define ST2_CRC         0x20            /* CRC error in data field */
#define ST2_CM          0x40            /* Control Mark = deleted */

/* Bits of FD_ST3 */
#define ST3_HA          0x04            /* Head (Address) */
#define ST3_TZ          0x10            /* Track Zero signal (1=track 0) */
#define ST3_WP          0x40            /* Write Protect */

/* Values for FD_COMMAND */
#define FD_RECALIBRATE  0x07            /* move to track 0 */      // recalibrate.
#define FD_SEEK         0x0F            /* seek track */
#define FD_READ         0xE6            /* read with MT, MFM, SKip deleted */
#define FD_WRITE        0xC5            /* write with MT, MFM */
#define FD_SENSEI       0x08            /* Sense Interrupt Status */
// Set the drive parameters (step rate, head unload time, etc.).
#define FD_SPECIFY      0x03            /* specify HUT etc */

/* DMA commands */
#define DMA_READ        0x46           // The mode word of DMA read disk (to port 12, 11).
#define DMA_WRITE       0x4A           // The mode word of DMA write disk.

#endif

```

14.23 fs.h

14.23.1 Functionality

The fs.h header file mainly defines some constants and structures about the file system, including the data structure of the buffer block in the buffer cache, the super block and i-node structure in the MINIX 1.0 file system, and the file table structure and some pipeline operation macros. .

14.23.2 Code annotation

Program 14-20 linux/include/linux/fs.h

```
/*
 * This file has definitions for some important file table
 * structures etc.
 */
```

```

#ifndef _FS_H
#define _FS_H

#include <sys/types.h>       // type header file. The basic system data types are defined.

/* devices are as follows: (same as minix, so we can use the minix
 * file system. These are major numbers.)
 *
 * 0 - unused (nodev)
 * 1 - /dev/mem              // memory device.
 * 2 - /dev/fd
 * 3 - /dev/hd
 * 4 - /dev/ttyx             // tty serial terminal device.
 * 5 - /dev/tty
 * 6 - /dev/lp               // printer device.
 * 7 - unnamed pipes
 */

#define IS_SEEKABLE(x) ((x)>=1 && (x)<=3)    // Determine if a device can find a location.

#define READ 0
#define WRITE 1
#define READA 2         /* read-ahead - don't pause */
#define WRITEA 3        /* "write-ahead" - silly, but somewhat useful */

void buffer_init(long buffer_end);       // buffer cache init function.

#define MAJOR(a) (((unsigned)(a))>>8)    // get device major number.
#define MINOR(a) ((a)&0xff)              // get device minor number.

#define NAME_LEN 14                      // name length is 14.
#define ROOT_INO 1                       // root i-node number.

#define I_MAP_SLOTS 8                    // the number of i-node bitmap slots (blocks).
#define Z_MAP_SLOTS 8                    // the number of logical block bitmap slots.
#define SUPER_MAGIC 0x137F               // File system magic number.

#define NR_OPEN 20                       // The maximum number of files opened by process.
#define NR_INODE 32                      // The maximum number of I-nodes used by system.
#define NR_FILE 64                       // The maximum number of files in the system.
#define NR_SUPER 8                       // The maximum number of superblocks in system.
#define NR_HASH 307                      // Buffer hash-table array items.
#define NR_BUFFERS nr_buffers            // The number of buffer blocks in the system.
#define BLOCK_SIZE 1024                  // Data block size (in bytes).
#define BLOCK_SIZE_BITS 10               // The number of bits used by the block size.
#ifndef NULL
#define NULL ((void *) 0)
#endif

// The number of i-nodes that each block can store (1024/32 = 32).
#define INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct d_inode)))
// The number of directory entries that can be stored in each block (1024/16 = 64).
```

```
#define DIR_ENTRIES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct dir_entry)))

// Get pipe head, tail, size, pipe empty? full?
#define PIPE_READ_WAIT(inode) ((inode).i_wait)
#define PIPE_WRITE_WAIT(inode) ((inode).i_wait2)
#define PIPE_HEAD(inode) ((inode).i_zone[0])
#define PIPE_TAIL(inode) ((inode).i_zone[1])
#define PIPE_SIZE(inode) ((PIPE_HEAD(inode)-PIPE_TAIL(inode))&(PAGE_SIZE-1))
#define PIPE_EMPTY(inode) (PIPE_HEAD(inode)==PIPE_TAIL(inode))
#define PIPE_FULL(inode) (PIPE_SIZE(inode)==(PAGE_SIZE-1))

#define NIL_FILP        ((struct file *)0)   // null file structure pointer.
#define SEL_IN          1
#define SEL_OUT         2
#define SEL_EX          4

typedef char buffer_block[BLOCK_SIZE];       // buffer block array (1024 items).

// Buffer block header data structure. (very important!!!)
// bh is often used in the code to represent the abbreviation of buffer_head.
struct buffer_head {
        char * b_data;                  /* pointer to data block (1024 bytes) */
        unsigned long b_blocknr;        /* block number */
        unsigned short b_dev;           /* device (0 = free) */
        unsigned char b_uptodate;
        unsigned char b_dirt;           /* 0-clean,1-dirty */   // Modified flag.
        unsigned char b_count;          /* users using this block */
        unsigned char b_lock;           /* 0 - ok, 1 -locked */
        struct task_struct * b_wait;        // task wait queue.
        struct buffer_head * b_prev;        // the previous block on the hash queue.
        struct buffer_head * b_next;        // the next block on the hash queue.
        struct buffer_head * b_prev_free;   // the previous block on the free list.
        struct buffer_head * b_next_free;   // the next block on the free list.
};

// I-node data structure (32 bytes) on disk.
struct d_inode {
        unsigned short i_mode;        // File type and attribute (rwx bit).
        unsigned short i_uid;         // User id (file owner identifier).
        unsigned long i_size;         // File size (in bytes).
        unsigned long i_time;         // Modified time (from 1970.1.1:0, in seconds).
        unsigned char i_gid;          // Group id (the group in which file owner belong).
        unsigned char i_nlinks;       // Number of links (entries pointed to the i-node).
        unsigned short i_zone[9];     // logical block number array.
};                                    // direct(0-6), indirect(7) or secondary indirect(8).

// This is the i-node structure in memory. The first seven items are exactly the same as d_inode.
struct m_inode {
        unsigned short i_mode;
        unsigned short i_uid;
        unsigned long i_size;
        unsigned long i_mtime;
        unsigned char i_gid;
```

```
104         unsigned char i_nlinks;
105         unsigned short i_zone[9];
106 /* these are in memory also */
107         struct task_struct * i_wait;  // task waiting queue for waiting for the i-node.
108         struct task_struct * i_wait2;   /* for pipes */
109         unsigned long i_atime;        // i-node access time.
110         unsigned long i_ctime;        // i-node change time.
111         unsigned short i_dev;         // the device where the i-node is located.
112         unsigned short i_num;         // i-node number.
113         unsigned short i_count;       // i-node used count, 0 indicates it's idle (free).
114         unsigned char i_lock;         // lock flag.
115         unsigned char i_dirt;         // modified flag
116         unsigned char i_pipe;         // the i-node is used for pipe.
117         unsigned char i_mount;        // mount flag.
118         unsigned char i_seek;         // used for lseek method of the file.
119         unsigned char i_update;       // updated flag.
120 };
121
    // File structure (used to establish a relationship between a file handle and the i-node)
122 struct file {
123         unsigned short f_mode;        // file mode (RW bits).
124         unsigned short f_flags;       // file open and control flags.
125         unsigned short f_count;       // file reference count.
126         struct m_inode * f_inode;     // file's i-node.
127         off_t f_pos;                  // read and write position in the file.
128 };
129
    // In-memory disk super block structure.
130 struct super_block {
131         unsigned short s_ninodes;       // number of i-nodes in the file system.
132         unsigned short s_nzones;        // number of zones (logica blocks).
133         unsigned short s_imap_blocks;   // number of data blocks occupied by i-node map.
134         unsigned short s_zmap_blocks;   // number of blocks occupied by logical block map.
135         unsigned short s_firstdatazone; // the first block number in the data zone.
136         unsigned short s_log_zone_size; // Log2(number of data blocks / logical block).
137         unsigned long s_max_size;       // the maximum file size.
138         unsigned short s_magic;         // file system magic number.
139 /* These are only in memory */
140         struct buffer_head * s_imap[8]; // an array of i-node bitmap buffer blocks.
141         struct buffer_head * s_zmap[8]; // an array of logical block bitmap buffer blocks.
142         unsigned short s_dev;           // the device number of the super block.
143         struct m_inode * s_isup;        // The root i-node of the mounted file system.
144         struct m_inode * s_imount;     // The i-node to which the file system is installed.
145         unsigned long s_time;           // modified time.
146         struct task_struct * s_wait;    // the wait queue for processes waiting for it.
147         unsigned char s_lock;           // the lock flag.
148         unsigned char s_rd_only;        // read only flag.
149         unsigned char s_dirt;           // tirty flag.
150 };
151
    // Super block structure on disk. It is exactly the same as lines 131-138 above.
152 struct d_super_block {
153         unsigned short s_ninodes;
```

```
        unsigned short s_nzones;
        unsigned short s_imap_blocks;
        unsigned short s_zmap_blocks;
        unsigned short s_firstdatazone;
        unsigned short s_log_zone_size;
        unsigned long s_max_size;
        unsigned short s_magic;
};

// File directory entry structure (16 bytes).
struct dir_entry {
        unsigned short inode;           // i-node number.
        char name[NAME_LEN];            // File name, NAME_LEN = 14.
};

extern struct m_inode inode_table[NR_INODE];     // i-node table (32 entries).
extern struct file file_table[NR_FILE];          // file table (64 items).
extern struct super_block super_block[NR_SUPER]; // super block array (8 items).
extern struct buffer_head * start_buffer;        // starting location of the buffer cache.
extern int nr_buffers;                           // the number of buffers.

//// The following are prototypes of disk manipulation function.
// Check if the floppy disk in the drive has changed.
extern void check_disk_change(int dev);
// Check the floppy disk replacement. Returns 1 if floppy is replaced, otherwise returns 0.
extern int floppy_change(unsigned int nr);
// Set the amount of time to wait to start the drive (set the wait timer).
extern int ticks_to_floppy_on(unsigned int dev);
// Start the specified drive.
extern void floppy_on(unsigned int dev);
// Turn off the specified floppy drive.
extern void floppy_off(unsigned int dev);

//// The following are function prototypes for file system operation management.
// The size of the file specified by the i-node is truncated to 0.
extern void truncate(struct m_inode * inode);
// Refresh (synchronize) the i-node information.
extern void sync_inodes(void);
// Waiting for the specified i-node.
extern void wait_on(struct m_inode * inode);
// block bitmap operation. Get the block number of the data block 'block' on the device.
extern int bmap(struct m_inode * inode,int block);
// Create a logical block on the device corresponding to the block 'block' and return the logical
// block number on the device.
extern int create_block(struct m_inode * inode,int block);
// Get the i-node number of the specified path name.
extern struct m_inode * namei(const char * pathname);
// Get the i-node of the specified path name without following the symbolic link.
extern struct m_inode * lnamei(const char * pathname);
// Prepare to open the file according to the path name.
extern int open_namei(const char * pathname, int flag, int mode,
        struct m_inode ** res_inode);
// Release (put back) an i-node (to write device).
```

```
extern void iput(struct m_inode * inode);
// Reads an i-node of from device.
extern struct m_inode * iget(int dev,int nr);
// Obtain an idle i-node entry from the inode table.
extern struct m_inode * get_empty_inode(void);
// Get (Apply a) pipe i-node. Returns the pointer to the i-node (fail if NULL).
extern struct m_inode * get_pipe_inode(void);
// Find the specified data block in the hash table. Returns the buffer head pointer.
extern struct buffer_head * get_hash_table(int dev, int block);
// Read the specified block from the device (first look in the hash table).
extern struct buffer_head * getblk(int dev, int block);
// Low-level read/write block function.
extern void ll_rw_block(int rw, struct buffer_head * bh);
// Low-level read/write data pages, that is, 4 data blocks at a time.
extern void ll_rw_page(int rw, int dev, int nr, char * buffer);
// Release the specified buffer block.
extern void brelse(struct buffer_head * buf);
// Read the specified data block.
extern struct buffer_head * bread(int dev,int block);
// Read a page (4 buffer blocks) to the specified memory address.
extern void bread_page(unsigned long addr,int dev,int b[4]);
// Reads a specified block of data and marks the block that will be read later.
extern struct buffer_head * breada(int dev,int block,...);
// Request a disk block from device dev and return the logical block number
extern int new_block(int dev);
// Frees the logic blocks in the device data area. Reset the logic block bitmap bits.
extern void free_block(int dev, int block);
// Create a new i-node for device and return the i-node number.
extern struct m_inode * new_inode(int dev);
// Release (free) an i-node (when deleting a file).
extern void free_inode(struct m_inode * inode);
// Refresh the specified device buffer.
extern int sync_dev(int dev);
// Get a super block of the specified device.
extern struct super_block * get_super(int dev);
extern int ROOT_DEV;       // root device number.

// Mount the root file system.
extern void mount_root(void);

#endif

```

14.24 hdreg.h

14.24.1 Functionality

The hdreg.h file mainly defines some command constant symbols for programming the hard disk controller. This includes the controller port, the status of each bit of the hard disk status register, controller commands, and error status constant symbols. The data structure of the hard disk partition table is also given.

14.24.2 Code annotation

Program 14-21 linux/include/linux/hdreg.h

```
/*
 * This file contains some defines for the AT-hd-controller.
 * Various sources. Check out some definitions (see comments with
 * a ques).
 */
#ifndef _HDREG_H
#define _HDREG_H

/* Hd controller regs. Ref: IBM AT Bios-listing */
#define HD_DATA         0x1f0   /* _CTL when writing */
#define HD_ERROR        0x1f1   /* see err-bits */
#define HD_NSECTOR      0x1f2   /* nr of sectors to read/write */
#define HD_SECTOR       0x1f3   /* starting sector */
#define HD_LCYL         0x1f4   /* starting cylinder */
#define HD_HCYL         0x1f5   /* high byte of starting cyl */
#define HD_CURRENT      0x1f6   /* 101dhhhh , d=drive, hhhh=head */
#define HD_STATUS       0x1f7   /* see status-bits */
#define HD_PRECOMP HD_ERROR     /* same io address, read=error, write=precomp */
#define HD_COMMAND HD_STATUS    /* same io address, read=status, write=cmd */

#define HD_CMD          0x3f6   // Control register port.

/* Bits of HD_STATUS */
#define ERR_STAT        0x01    // Command execution error.
#define INDEX_STAT      0x02    // Received the index.
#define ECC_STAT        0x04    /* Corrected error */  // ECC checksum error.
#define DRQ_STAT        0x08    // Request service.
#define SEEK_STAT       0x10    // End of seek.
#define WRERR_STAT      0x20    // Drive error.
#define READY_STAT      0x40    // Drive ready.
#define BUSY_STAT       0x80    // Controller busy.

/* Values for HD_COMMAND */
#define WIN_RESTORE             0x10   // Drive reset (recalibration).
#define WIN_READ                0x20   // Read sector.
#define WIN_WRITE               0x30   // Write sector.
#define WIN_VERIFY              0x40   // Sector verify.
#define WIN_FORMAT              0x50   // Format track.
#define WIN_INIT                0x60   // Controller initialize.
#define WIN_SEEK                0x70   // Seek track.
#define WIN_DIAGNOSE            0x90   // Controller diagnose.
#define WIN_SPECIFY             0x91   // Establish drive parameters.

/* Bits for HD_ERROR */
// When execute a diagnostic command, its meaning is different from other commands, as follows:
// ==============================================================================
//           Diagnostic command              Other command
// ------------------------------------------------------------------------------
// 0x01      No error                  Data mark lost
// 0x02      Controller error          Track 0 error.
// 0x03      Sector buffer error
// 0x04      ECC part error            Command abort
// 0x05      Control process error
// 0x10                                ID not found.
// 0x40                                ECC error.
// 0x80                                Bad sector
//--------------------------------------------------------------------------------
#define MARK_ERR        0x01    /* Bad address mark ? */
#define TRK0_ERR        0x02    /* couldn't find track 0 */
#define ABRT_ERR        0x04    /* ? */
#define ID_ERR          0x10    /* ? */
#define ECC_ERR         0x40    /* ? */
#define BBD_ERR         0x80    /* ? */

// Hard disk partition table structure, see the information after the list below.
struct partition {
        unsigned char boot_ind;         /* 0x80 - active (unused) */
        unsigned char head;             /* ? */
        unsigned char sector;           /* ? */
        unsigned char cyl;              /* ? */
        unsigned char sys_ind;          /* ? */
        unsigned char end_head;         /* ? */
        unsigned char end_sector;       /* ? */
        unsigned char end_cyl;          /* ? */
        unsigned int start_sect;        /* starting sector counting from 0 */
        unsigned int nr_sects;          /* nr of sectors in partition */
};

#endif

```

14.24.3 Information

14.24.3.1 Hard disk partition table

In order to facilitate management of data or to achieve shared hard disk resources by multiple operating systems, the hard disk can be logically divided into 1--4 partitions. The sector numbers between each partition are contiguous. The partition table consists of four entries, each of which consists of 16 bytes and corresponds to the information of one partition. Each entry has a partition size, a cylinder number, a track number, and a sector number, as shown in Table 14-2. The partition table is stored at the 0x1BE--0x1FD position of the first sector of the 0 cylinder 0 head of the hard disk.

Table 14-2 Hard disk partition table entry structure

Offset	Name	Size	Description
0x00	boot_ind	1 byte	Boot index. Only one partition of the 4 partitions can be booted at a time. 0x00 - Do not boot from this partition; 0x80 - Boot from this partition.
0x01	head	1 byte	Partition start head number. The head number ranges from 0 to 255.
0x02	sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the beginning of the partition.

0x03	cyl	1 byte	The lower 8 bits of the cylinder number at the starting of the partition.
0x04	sys_ind	1 byte	Partition type. 0x0b - DOS; 0x80 - Old Minix; 0x83 - Linux . . .
0x05	end_head	1 byte	The head number at the end of the partition. It ranges from 0 to 255.
0x06	end_sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the end of the partition .
0x07	end_cyl	1 byte	The lower 8 bits of the cylinder number at the end of the partition.
0x08-0x0b	start_sect	4 byte	The physical sector number at the beginning of the partition. It counts from 0 in the order of the sector number of the entire hard disk.
0x0c-0x0f	nr_sects	4 byte	The number of sectors occupied by the partition.

14.25 head.h

14.25.1 Functionality

The head.h header file defines the simple structure of the descriptor in the Intel CPU and specifies the item number of the descriptor.

14.25.2 Code annotation

Program 14-22 linux/include/linux/head.h

```
#ifndef _HEAD_H
#define _HEAD_H

// The data structure of the segment descriptor is defined below. This structure only states
// that each descriptor is composed of 8 bytes, and each descriptor table has 256 entries.
typedef struct desc_struct {
        unsigned long a,b;
} desc_table[256];

// Declare the memory page directory table used by the paging management mechanism. Each
// directory entry is 4 bytes. For this kernel, the table starts at physical address 0.
extern unsigned long pg_dir[1024];
extern desc_table idt,gdt;       // Interrupt descriptor table, global descriptor table.

#define GDT_NUL 0                // The 0th item of the GDT, not used.
#define GDT_CODE 1               // The first item is the kernel code segment descriptor.
#define GDT_DATA 2               // The second item is the kernel data segment descriptor.
#define GDT_TMP 3                // The third item is system segment descriptor, not used.

#define LDT_NUL 0                // The 0th item of the LDT, not used.
#define LDT_CODE 1               // The first item is the user code segment descriptor .
#define LDT_DATA 2               // The second item is the user data segment descriptor.

#endif

```

14.26 kernel.h

14.26.1 Functionality

The kernel.h file defines some function prototypes commonly used by the kernel.

14.26.2 Code annotation

Program 14-23 linux/include/linux/kernel.h

```
/*
 * 'kernel.h' contains some often-used function prototypes etc
 */
// Verify that the memory block is overrun. (kernel/fork.c, 24).
void verify_area(void * addr,int count);
// Display kernel error messages and then enter an infinite loop. (kernel/panic.c, 16).
volatile void panic(const char * str);
// The function for the process exit. (kernel/exit.c, 262).
volatile void do_exit(long error_code);
// Standard print (display) function. (init/main.c, 179).
int printf(const char * fmt, ...);
// Kernel-specific print function with the same functionality as printf()(kernel/printk.c).
int printk(const char * fmt, ...);
// The display function of the console. (kernel/chr_drv/console.c, 995).
void console_print(const char * str);
// Write a string of the specified length to the tty. (kernel/chr_drv/tty_io.c, 339).
int tty_write(unsigned ch,char * buf,int count);
// Generic kernel memory allocation function. ( lib/malloc.c, 117).
void * malloc(unsigned int size);
// Frees the memory occupied by the specified object. ( lib/malloc.c, 182).
void free_s(void * obj, int size);
// Hard disk processing timed out. (kernel/blk_drv/hd.c, 318).
extern void hd_times_out(void);
// Stop beeping. (kernel/chr_drv/console.c, 944).
extern void sysbeepstop(void);
// Black screen processing. (kernel/chr_drv/console.c, 981).
extern void blank_screen(void);
// Restore the screen that is being blacked out. (kernel/chr_drv/console.c, 988).
extern void unblank_screen(void);

extern int beepcount;          // The beep time tick counts (chr_drv/console.c, 950).
extern int hd_timeout;         // Hard disk timeout ticks (kernel/blk_drv/blk.h).
extern int blankinterval;      // Screen black screen interval(chr_drv/console.c,138).
extern int blankcount;         // Black screen time count (chr_drv/console.c, 139).

#define free(x) free_s((x), 0)

/*
 * This is defined as a macro, but at some point this might become a
 * real subroutine that sets a flag if it returns true (to do
 * BSD-style accounting where the process is flagged if it uses root
 * privs).  The implication of this is that you should do normal
 * permissions checks first, and check suser() last.
 */
#define suser() (current->euid == 0)            // Check if it is a super user.

```

14.27 math_emu.h

14.27.1 Functionality

The math_emu.h file contains the constant definitions and structures involved in Chapter 11 (Mathematic Coprocessor), including some of the data structures used by the kernel code to simulate various types of data when simulating mathematical coprocessors.

14.27.2 Code annotation

Program 14-24 linux/include/linux/math_emu.h

```
 1 /*
 2  * linux/include/linux/math_emu.h
 3  *
 4  * (C) 1991 Linus Torvalds
 5  */
 6 #ifndef _LINUX_MATH_EMU_H
 7 #define _LINUX_MATH_EMU_H
 8
   // The scheduler header file defines the task structure, the initial task 0, and some embedded
   // assembly function macro statements about the descriptor parameter settings and acquisition.
 9 #include <linux/sched.h>
10
   // The structure of the data on the stack when CPU generates exception INT 7 (device not exist)
   // is similar to the data distribution in the kernel stack when a system-call is invoked.
11 struct info {
12         long ___math_ret;      // The return address of the math_emulate() caller (INT 7).
13         long ___orig_eip;      // A place to temporarily save the original EIP.
14         long ___edi;           // The registers that the exception handler pushed.
15         long ___esi;
16         long ___ebp;
   // When interrupt 7 returns, it will execute the return processing code of the system call.
   // Below (lines 18--30) are identical to the structure in stack when system-call is invoked.
17         long ___sys_call_ret;
18         long ___eax;
19         long ___ebx;
20         long ___ecx;
21         long ___edx;
22         long ___orig_eax;      // If it's not a sys-call but other interrupts, it is -1.
23         long ___fs;
24         long ___es;
25         long ___ds;
26          long ___eip;          // Lines 26 -- 30 are pushed by the CPU automatically.
27          long ___cs;
28          long ___eflags;
29          long ___esp;
30          long ___ss;
31 };
32
   // Constants defined to facilitate the reference to fields in info structure (data in the stack).
33 #define EAX (info->___eax)
34 #define EBX (info->___ebx)
35 #define ECX (info->___ecx)
36 #define EDX (info->___edx)
37 #define ESI (info->___esi)
38 #define EDI (info->___edi)
39 #define EBP (info->___ebp)
40 #define ESP (info->___esp)
41 #define EIP (info->___eip)
42 #define ORIG_EIP (info->___orig_eip)
43 #define EFLAGS (info->___eflags)
44 #define DS (*(unsigned short *) &(info->___ds))
45 #define ES (*(unsigned short *) &(info->___es))
46 #define FS (*(unsigned short *) &(info->___fs))
47 #define CS (*(unsigned short *) &(info->___cs))
48 #define SS (*(unsigned short *) &(info->___ss))
49
   // Terminate the math coprocessor emulation operation (in file math_emulation.c, line 488).
   // The actual effect of the macro definition on lines 52-53 below is to redefine __math_abort
   // as a function that does not return (that is, add volatile before). The first part of the macro:
   // '(volatile void (*)(struct info *, unsigned int))' is a function type definition that is used
   // to re-specify the definition of the __math_abort function. This is followed by its corresponding
   // parameters. Putting the keyword volatile in front of the function name to decorate the function
   // is used to tell the gcc compiler that the function will not return, so that gcc can produces
   // better code.
50 void __math_abort(struct info *, unsigned int);
51
52 #define math_abort(x,y) \
53 (((volatile void (*)(struct info *,unsigned int)) __math_abort)((x),(y)))
54
55 /*
56  * Gcc forces this stupid alignment problem: I want to use only two longs
57  * for the temporary real 64-bit mantissa, but then gcc aligns out the
58  * structure to 12 bytes which breaks things in math_emulate.c. Shit. I
59  * want some kind of "no-alignt" pragma or something.
60  */
61
   // Temporary real structure.
   // It has a total of 64 bit mantissas. Where 'a' is the lower 32 bits, 'b' is the upper 32 bits
   // (including 1 fixed bit), and 'exponent' is the exponent value.
62 typedef struct {
63       long a,b;
64       short exponent;
65 } temp_real;
66
```

```
// The structure designed to solve the alignment problem mentioned in the original note above
// and works like the temp_real structure above.
typedef struct {
     short m0,m1,m2,m3;
     short exponent;
} temp_real_unaligned;

// Assign the temp_real type value 'a' to the 80387 stack register 'b' (ST(i)).
#define real_to_real(a,b) \
((*(long long *) (b) = *(long long *) (a)),((b)->exponent = (a)->exponent))

// Long real (double precision) structure.
typedef struct {
     long a,b;              // 'a' is the lower 32 bits; 'b' is the upper 32 bits.
} long_real;

typedef long short_real;     //Define a short real type.

// Temporary integer structure.
typedef struct {
     long a,b;              // 'a' is the lower 32 bits; 'b' is the upper 32 bits.
     short sign;
} temp_int;

// The structure corresponding to the contents of the status word register inside the 80387
// coprocessor (see Figure 11-6).
struct swd {
     int ie:1;         // Invalid operation exception.
     int de:1;         // Denormalized exception.
     int ze:1;         // Divide by zero exception.
     int oe:1;         // Overflow exception.
     int ue:1;         // Underflow exception.
     int pe:1;         // Precision exception.
     int sf:1;         // Stack error flag, caused by overflow of the accumulator.
     int ir:1;         // Ir, b: Set if any of the above 6 unmasked exceptions occur.
     int c0:1;         // c0--c3: Condition code bits.
     int c1:1;
     int c2:1;
     int top:3;        // Indicates the 80-bit register currently at the top of the stack.
     int c3:1;
     int b:1;
};

// 80387 internal register control mode constants.
#define I387 (current->tss.i387)          // 80387 status information of the process.
#define SWD (*(struct swd *) &I387.swd)   // Status control word in 80387.
#define ROUNDING ((I387.cwd >> 10) & 3)   // Get the rounding mode in the control word.
#define PRECISION ((I387.cwd >> 8) & 3)   // Get the precision mode in the control word.

// Constant that define the significant digits of a precision.
#define BITS24         0         // Precision Effective bits: 24 bits.
#define BITS53         2         // 53 bits.
#define BITS64         3         // 64 bits.
```

```

// Define rounding mode constants.
#define ROUND_NEAREST  0         // Round to the nearest or even.
#define ROUND_DOWN     1         // Trend to negative infinite.
#define ROUND_UP       2         // Trend to positive infinite.
#define ROUND_0        3         // Trend to cut to zero.

// Constant definitions.
#define CONSTZ   (temp_real_unaligned) {0x0000,0x0000,0x0000,0x0000,0x0000}  // 0
#define CONST1   (temp_real_unaligned) {0x0000,0x0000,0x0000,0x8000,0x3FFF}  // 1.0
#define CONSTPI  (temp_real_unaligned) {0xC235,0x2168,0xDAA2,0xC90F,0x4000}  // Pi
#define CONSTLN2 (temp_real_unaligned) {0x79AC,0xD1CF,0x17F7,0xB172,0x3FFE}  // Loge(2)
#define CONSTLG2 (temp_real_unaligned) {0xF799,0xFBCF,0x9A84,0x9A20,0x3FFD}  // Log10(2)
#define CONSTL2E (temp_real_unaligned) {0xF0BC,0x5C17,0x3B29,0xB8AA,0x3FFF}  // Log2(e)
#define CONSTL2T (temp_real_unaligned) {0x8AFE,0xCD1B,0x784B,0xD49A,0x4000}  // Log2(10)

// Set 80387 states.
#define set_IE() (I387.swd |= 1)
#define set_DE() (I387.swd |= 2)
#define set_ZE() (I387.swd |= 4)
#define set_OE() (I387.swd |= 8)
#define set_UE() (I387.swd |= 16)
#define set_PE() (I387.swd |= 32)

// Set 80387 control conditions
#define set_C0() (I387.swd |= 0x0100)
#define set_C1() (I387.swd |= 0x0200)
#define set_C2() (I387.swd |= 0x0400)
#define set_C3() (I387.swd |= 0x4000)

/* ea.c */
// Calculates the effective address used by the operand in the emulation instruction, that is,
// calculates the effective address according to the addressing mode byte in the instruction.
// Params: __info - the contents of the stack at time of interrupt; __code - instruction code.
// Returns effective address.
char * ea(struct info * __info, unsigned short __code);

/* convert.c */
// Various data type conversion functions implemented in the convert.c file.
void short_to_temp(const short_real * __a, temp_real * __b);
void long_to_temp(const long_real * __a, temp_real * __b);
void temp_to_short(const temp_real * __a, short_real * __b);
void temp_to_long(const temp_real * __a, long_real * __b);
void real_to_int(const temp_real * __a, temp_int * __b);
void int_to_real(const temp_int * __a, temp_real * __b);

/* get_put.c */
// Access functions of various types.
void get_short_real(temp_real *, struct info *, unsigned short);
void get_long_real(temp_real *, struct info *, unsigned short);
void get_temp_real(temp_real *, struct info *, unsigned short);
void get_short_int(temp_real *, struct info *, unsigned short);
void get_long_int(temp_real *, struct info *, unsigned short);
void get_longlong_int(temp_real *, struct info *, unsigned short);
void get_BCD(temp_real *, struct info *, unsigned short);
void put_short_real(const temp_real *, struct info *, unsigned short);
void put_long_real(const temp_real *, struct info *, unsigned short);
void put_temp_real(const temp_real *, struct info *, unsigned short);
void put_short_int(const temp_real *, struct info *, unsigned short);
void put_long_int(const temp_real *, struct info *, unsigned short);
void put_longlong_int(const temp_real *, struct info *, unsigned short);
void put_BCD(const temp_real *, struct info *, unsigned short);

/* add.c */
// A function that simulates a floating-point addition instruction.
void fadd(const temp_real *, const temp_real *, temp_real *);

/* mul.c */
// Simulate floating point multiply instructions.
void fmul(const temp_real *, const temp_real *, temp_real *);

/* div.c */
// Simulate floating point division instructions.
void fdiv(const temp_real *, const temp_real *, temp_real *);

/* compare.c */
// Simulate floating point comparison instructions.
void fcom(const temp_real *, const temp_real *);    // FCOM, compare two numbers.
void fucom(const temp_real *, const temp_real *);   // FUCOM, no order comparison.
void ftst(const temp_real *);                // FTST, top stack accumulator compared to 0.

#endif

```

14.28 mm.h

14.28.1 Functionality

The mm.h file is the memory management header file. It mainly defines the size of the memory page and several page release function prototypes.

14.28.2 Code annotation

Program 14-25 linux/include/linux/mm.h

```
#ifndef _MM_H
#define _MM_H

// Define the memory page size (in bytes). Note that the cache block size is 1024 bytes.
#define PAGE_SIZE 4096

// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
//     manipulation function prototypes.
#include <linux/kernel.h>
#include <signal.h>

extern int SWAP_DEV;          // Memory page swap device number (mm/memory.c, line 36).

// The swapped memory page is read in or written to the swap device. The ll_rw_page() function
// is defined in the file blk_drv/ll_rw_block.c. The parameter 'nr' is the page number in the
// main memory area; 'buffer' is the read/write buffer.
#define read_swap_page(nr,buffer) ll_rw_page(READ,SWAP_DEV,(nr),(buffer));
#define write_swap_page(nr,buffer) ll_rw_page(WRITE,SWAP_DEV,(nr),(buffer));

// Get free physical page in the main memory area. Returns 0 if there is no more memory available.
extern unsigned long get_free_page(void);
// Map a physical memory page whose content has been modified to a specified location in the
// linear address space. Almost exactly the same as put_page().
extern unsigned long put_dirty_page(unsigned long page,unsigned long address);
// Release a page of physical memory starting with address 'addr'.
extern void free_page(unsigned long addr);
// Free the swapping page specified in the swap device (mm/swap.c, line 58).
void swap_free(int page_nr);
// Swap the specified page from the device into memory (mm/swap.c, line 69).
void swap_in(unsigned long *table_ptr);

//// Out of memory (oom) processing function.
extern inline volatile void oom(void)
{
// do_exit() should use the exit code. The error code with the same value of the signal value
// SIGSEGV(11) is "Resource is temporarily unavailable", which is synonymous.
        printk("out of memory\n\r");
        do_exit(SIGSEGV);
}

// Invalidate the translation look-aside buffer (TLB) on-chip.
// In order to improve the efficiency of address translation, the CPU stores the most recently
// used page table data in the internal cache of the chip called translation lookaside buffer
// (TLB). After modifying the page table information, you need to refresh the buffer. This is
// done by reloading the page directory base register (PDBR) CR3. Below, EAX = 0, which is the
// base address of the page directory.
#define invalidate() \
__asm__("movl %%eax,%%cr3"::"a" (0))

/* these are not to be changed without changing head.s etc */
// The maximum memory capacity supported by the Linux 0.12 kernel by default is 16MB, and these
// definitions can be modified to accommodate more memory.
#define LOW_MEM 0x100000                    // Low end of the physical memory (1MB).
extern unsigned long HIGH_MEMORY;           // The highest address of physical memory.
#define PAGING_MEMORY (15*1024*1024)        // The size of the paged memory (15MB).
#define PAGING_PAGES (PAGING_MEMORY>>12)    // The number of pages after paging (3840).
#define MAP_NR(addr) (((addr)-LOW_MEM)>>12) // Map the memory address to the page number.
#define USED 100                            // Page used flag, see memory.c, line 449.

```

```
    // Memory mapping byte map, 1 byte represents 1 page. The corresponding byte of each page is
    // used to mark the number of times the page is currently referenced (used). It can map up to
    // 15Mb of memory space. In the function mem_init() in the memory.c program, the position that
    // cannot be used as the main memory area page is set to USED (100) in advance.
extern unsigned char mem_map [ PAGING_PAGES ];

    // The symbol constants defined below correspond to some of the flag bits in the page directory
    // and page table (secondary page table) entries.
#define PAGE_DIRTY      0x40          // Bit 6, the page is dirty (modified).
#define PAGE_ACCESSED   0x20          // Bit 5, the page was accessed.
#define PAGE_USER       0x04          // Bit 2, the page belongs to: 1-user; 0-superuser.
#define PAGE_RW         0x02          // Bit 1, read/write rights: 1 - write; 0 - read.
#define PAGE_PRESENT    0x01          // Bit 0, page exists: 1 - present; 0 - not exist.

#endif

```

14.29 sched.h

14.29.1 Functionality

Sched.h is the scheduler header file, which defines the task structure task_struct, initial task 0 data, and some embedded assembly function macros for memory descriptor parameter settings and acquisition and task context switch macro switch_to().Below we describe in detail the execution process of the task switch macro.

The task switch macro switch_to(n) (starting at line 222) first declares a structure 'struct {long a,b;} __tmp', which is used to reserve 8 bytes of space on the kernel state stack. This space is used to store the selector for the task status segment TSS of the new task that will be switched to. Then test if we are performing the operation to switch to the current task, and if so, do not need to do anything, just exit. Otherwise we save the selector of the new task TSS to the offset position 4 in the temporary structure __tmp, at which point the data in __tmp is set to:

```
__tmp+0: Undefined (long)
__tmp+4: New task TSS selector (word)
__tmp+6: Undefined (word)
```

Next, we exchange the new task pointer in the ECX register with the current task pointer in the global variable 'current', let 'current' contain the pointer value of the new task we are going to switch to, and ECX saves the current task. Then execute the instruction LJMP that indirectly jumps to __tmp. The instruction that jumps to the new task TSS selector will ignore the undefined value part of __tmp, and the CPU will automatically jump to the new task specified by the TSS segment to execute, and the task (current task) will be suspended. This is why we don't need to set other undefined parts of the structure variable __tmp. See Figure 4-37 in Section 4.7 for a schematic diagram of the task switching operation.

After a period of time, the LJMP instruction of a task will jump to the TSS segment selector of the task, causing the CPU to switch back to the task and start execution from the next instruction of LJMP. At this point ECX contains a pointer to the current task, so we can use this pointer to check if it is the last (most recently) task that used the math coprocessor. If the task has not used the coprocessor, it will exit immediately. Otherwise, the CLTS instruction is executed to reset the task switching flag TS in the control register CR0. The CPU sets this flag whenever the task is switched and tests the flag before executing the coprocessor instruction. This method of processing the TS flag in the Linux system allows the kernel to avoid unnecessary saving and recovery operations on the coprocessing state, thereby improving the execution performance of the coprocessor.

14.29.2 Code annotation

Program 14-26 linux/include/linux/sched.h

```
#ifndef _SCHED_H
#define _SCHED_H

#define HZ 100                   // Define system clock tick frequency (10ms per tick)

#define NR_TASKS        64               // The max number of tasks in the system.
#define TASK_SIZE       0x04000000       // The size of each task (64MB).
#define LIBRARY_SIZE    0x00400000       // The size of the loaded library (4MB).

#if (TASK_SIZE & 0x3fffff)
#error "TASK_SIZE must be multiple of 4M"
#endif

#if (LIBRARY_SIZE & 0x3fffff)
#error "LIBRARY_SIZE must be a multiple of 4M"
#endif

#if (LIBRARY_SIZE >= (TASK_SIZE/2))
#error "LIBRARY_SIZE too damn big!"
#endif

#if (((TASK_SIZE>>16)*NR_TASKS) != 0x10000)
#error "TASK_SIZE*NR_TASKS must be 4GB"
#endif

// The location where the library is loaded in the process logical address space (at 60MB).
#define LIBRARY_OFFSET (TASK_SIZE - LIBRARY_SIZE)

// The following macros CT_TO_SECS and CT_TO_USECS are used to convert the current system ticks
// into seconds and microseconds.
#define CT_TO_SECS(x)    ((x) / HZ)
#define CT_TO_USECS(x)   (((x) % HZ) * 1000000/HZ)

#define FIRST_TASK task[0]            // Task 0 is special, so we define a symbol for it.
#define LAST_TASK task[NR_TASKS-1]    // The last task in the task array.

// <linux/head.h> Head header file. A simple structure for the segment descriptor is defined,
//      along with several selector constants.
// <linux/fs.h> File system header file. Define the file table structure (file,buffer_head,
//      m_inode, etc.).
// <linux/mm.h> Memory management header file. Contains page size definitions and some page
//      release function prototypes.
// <sys/param.h> Parameter file. Some hardware-related parameter values are given.
// <sys/time.h> The timeval structure and the itimerval structure are defined.
// <sys/resource.h> Resource file. Contains information on the limits and utilization of system
//     resources used by processes.
// <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
//     manipulation function prototypes.
#include <linux/head.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <signal.h>

#if (NR_OPEN > 32)
#error "Currently the close-on-exec-flags and select masks are in one long, max 32 files/proc"
#endif

// This defines the operating state in which the process may be.
#define TASK_RUNNING            0   // process is running or is ready to run.
#define TASK_INTERRUPTIBLE      1   // in an interruptible wait state.
#define TASK_UNINTERRUPTIBLE    2   // in an uninterruptible wait state (mainly I/O op).
#define TASK_ZOMBIE             3   // in a dead state and the father has not yet signaled.
#define TASK_STOPPED            4   // process has stopped.

#ifndef NULL
#define NULL ((void *) 0)
#endif

// Copy the page directory table of a process. ( mm/memory.c, 118 )
// Linus considers this to be one of the most complex functions in the kernel.
extern int copy_page_tables(unsigned long from, unsigned long to, long size);
// Frees the memory specified by the page table and the page table itself. ( mm/memory.c, 69 )
extern int free_page_tables(unsigned long from, unsigned long size);

// The initialization function of the scheduler. (kernel/sched.c, 417 )
extern void sched_init(void);
// Process scheduling function. (kernel/sched.c, 119)
extern void schedule(void);
// The initialization function for exception (trap) processing. (kernel/traps.c, 185)
extern void trap_init(void);
// Display kernel error messages and then enter an infinite loop. (kernel/panic.c, 16)
extern void panic(const char * str);
// Write a string of the specified length to the tty. (kernel/chr_drv/tty_io.c, 339)
extern int tty_write(unsigned minor,char * buf,int count);

typedef int (*fn_ptr)();        // Define a function pointer type.

// The following is the structure used by the math coprocessor, which is mainly used to save
// the execution status information of i387 when the process is switched out.
struct i387_struct {
        long    cwd;            // Control word.
        long    swd;            // Status word.
        long    twd;            // Tag word.
        long    fip;            // Coprocessor code ip pointer.
        long    fcs;            // Coprocessor code segment register.
        long    foo;            // The offset of the memory operand.
        long    fos;            // The segment of the memory operand.
        long    st_space[20];   /* 8*10 bytes for each FP-reg = 80 bytes */
};

// Task status segment (TSS) data structure.
struct tss_struct {
        long    back_link;      /* 16 high bits zero */
        long    esp0;
        long    ss0;            /* 16 high bits zero */
        long    esp1;
        long    ss1;            /* 16 high bits zero */
        long    esp2;
        long    ss2;            /* 16 high bits zero */
        long    cr3;
        long    eip;
        long    eflags;
        long    eax,ecx,edx,ebx;
        long    esp;
        long    ebp;
        long    esi;
        long    edi;
        long    es;             /* 16 high bits zero */
        long    cs;             /* 16 high bits zero */
        long    ss;             /* 16 high bits zero */
        long    ds;             /* 16 high bits zero */
        long    fs;             /* 16 high bits zero */
        long    gs;             /* 16 high bits zero */
        long    ldt;            /* 16 high bits zero */
        long    trace_bitmap;   /* bits: trace 0, bitmap 16-31 */
        struct i387_struct i387;
};

// Below is the task (process) data structure, or process control block, or process descriptor.
// See section 5.7 for detailed descriptions.
//struct task_struct {
//   long state;                    // -1 unrunnable, 0 runnable (ready), > 0 stopped.
//   long counter;                  // Task run time tick (decrement), run time slice.
//   long priority;                 // Priority. When task starts running, counter=priority.
//   long signal;                   // Signal bitmap, each bit is a signal( = bit offset + 1).
//   struct sigaction sigaction[32]; // Signal attribute struct. Signal operation and flags.
//   long blocked;                  // Process signal mask (Bitmap of masked signal).
//   int exit_code;                 // Exit code after task stops, its parent will get it.
//   unsigned long start_code;      // Code start location in linear address space.
//   unsigned long end_code;        // Code length or size (bytes).
//   unsigned long end_data;        // Code size + data siez (bytes).
//   unsigned long brk;             // Total size (number of bytes).
//   unsigned long start_stack;     // Stack bottom location.
//   long pid;                      // Process identifier.
//   long pgrp;                     // Process group number.
//   long session;                  // Process session number.
//   long leader;                // Leader session number.
//   int groups[NGROUPS];        // Group numbers. A process can belong to more groups.
//   task_struct *p_pptr;        // Pointer to parent process.
//   task_struct *p_cptr;        // Pointer to youngest child process.
//   task_struct *p_ysptr;       // Pointer to younger sibling process created afterwards.
//   task_struct *p_osptr;       // Pointer to older sibling process created earlier.
//   unsigned short uid;         // User id.
//   unsigned short euid;        // Effective user id.
//   unsigned short suid;        // Saved user id.
//   unsigned short gid;         // Group id.
//   unsigned short egid;        // Effective group id.
//   unsigned short sgid;        // Saved group id.
//   long timeout;               // Kernel timing timeout value.
//   long alarm;                 // Alarm timing value (ticks).
//   long utime;                 // User state running time (ticks).
//   long stime;                 // System state runtime (ticks).
//   long cutime;                // Child process user state runtime.
//   long cstime;                // Child process system state runtime.
//   long start_time;            // Time the process started running.
//   struct rlimit rlim[RLIM_NLIMITS];  // Resource usage statistics array.
//   unsigned int flags;         // per process flags.
//   unsigned short used_math;   // Flag: Whether a coprocessor is used.
//   int tty;                    // The tty subdevice number used. -1 means no use.
//   unsigned short umask;       // The mask bit of the file creation attribute.
//   struct m_inode * pwd;       // Current working directory i node structure pointer.
//   struct m_inode * root;      // Root i-node structure pointer.
//   struct m_inode * executable; // The pointer to i-node structure of the executable file.
//   struct m_inode * library;   // The loaded library i-node structure pointer.
//   unsigned long close_on_exec; // A bitmap flags that close file handles on execution.
//   struct file * filp[NR_OPEN]; // File structure pointer table, up to 32 items.
//                               // The index is the value of file descriptor.
//   struct desc_struct ldt[3];  // LDT. 0-empty, 1-code seg cs, 2-data & stack seg ds & ss.
//   struct tss_struct tss;      // The task status segment structure TSS of the process.
//};
struct task_struct {
/* these are hardcoded - don't touch */
        long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter;
        long priority;
        long signal;
        struct sigaction sigaction[32];
        long blocked;   /* bitmap of masked signals */
/* various fields */
        int exit_code;
        unsigned long start_code,end_code,end_data,brk,start_stack;
        long pid,pgrp,session,leader;
        int     groups[NGROUPS];
        /*
         * pointers to parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->p_pptr->pid)
         */
        struct task_struct      *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
        unsigned short uid,euid,suid;
        unsigned short gid,egid,sgid;
        unsigned long timeout,alarm;
        long utime,stime,cutime,cstime,start_time;
        struct rlimit rlim[RLIM_NLIMITS];
        unsigned int flags;     /* per process flags, defined below */
        unsigned short used_math;
/* file system info */
        int tty;                /* -1 if no tty, so it must be signed */
        unsigned short umask;
        struct m_inode * pwd;
        struct m_inode * root;
        struct m_inode * executable;
        struct m_inode * library;
        unsigned long close_on_exec;
        struct file * filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
        struct desc_struct ldt[3];
/* tss for this task */
        struct tss_struct tss;
};

/*
 * Per process flags
 */
#define PF_ALIGNWARN    0x00000001      /* Print alignment warning msgs */
                                        /* Not implemented yet, only for 486*/

/*
 *  INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x9ffff (=640kB)
 */
// Hard-coded information corresponding to the first task of the above task structure.
#define INIT_TASK \
/* state etc */ { 0,15,15, \          // state, counter, priority
/* signals */   0,{{},},0, \          // signal, sigaction[32], blocked
/* ec,brk... */ 0,0,0,0,0,0, \  // exit_code,start_code,end_code,end_data,brk,start_stack
/* pid etc.. */ 0,0,0,0, \            // pid, pgrp, session, leader
/* suppl grps*/ {NOGROUP,}, \         // groups[]
/* proc links*/ &init_task.task,0,0,0, \     // p_pptr, p_cptr, p_ysptr, p_osptr
/* uid etc */   0,0,0,0,0,0, \        // uid, euid, suid, gid, egid, sgid
/* timeout */   0,0,0,0,0,0,0, \  // alarm,utime,stime,cutime,cstime,start_time,used_math
/* rlimits */   { {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff},  \
                  {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}, \
                  {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}}, \
/* flags */     0, \                  // flags
/* math */      0, \              // used_math, tty,umask,pwd,root,executable,close_on_exec
/* fs info */   -1,0022,NULL,NULL,NULL,NULL,0, \
/* filp */      {NULL,}, \            // filp[20]
        { \                           // ldt[3]
                {0,0}, \
/* ldt */       {0x9f,0xc0fa00}, \  // code size 640K,base 0x0,G=1,D=1,DPL=3,P=1 TYPE=0xa
                {0x9f,0xc0f200}, \  // data size 640K,base 0x0,G=1,D=1,DPL=3,P=1 TYPE=0x2
        }, \
/*tss*/ {0,PAGE_SIZE+(long)&init_task,0x10,0,0,0,0,(long)&pg_dir,\       // tss
         0,0,0,0,0,0,0,0, \
         0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
         _LDT(0),0x80000000, \
                {} \
        }, \
}

extern struct task_struct *task[NR_TASKS];      // An array of task pointers.
extern struct task_struct *last_task_used_math;
extern struct task_struct *current;             // The current process pointer.
extern unsigned long volatile jiffies;          // Ticks (10ms/tick) from start of boot.
extern unsigned long startup_time;              // Boot time. seconds since 1970:0:0:0.
extern int jiffies_offset;               // The number of ticks that need to be adjusted.

#define CURRENT_TIME (startup_time+(jiffies+jiffies_offset)/HZ)  // Current time(seconds).

// Add a timer (ticks, call the function *fn() when timing is up). (kernel/sched.c )
extern void add_timer(long jiffies, void (*fn)(void));
// Uninterruptible waiting for sleep. (kernel/sched.c)
extern void sleep_on(struct task_struct ** p);
// Interrupted waiting for sleep. (kernel/sched.c )
extern void interruptible_sleep_on(struct task_struct ** p);
// Clearly wake up the process of sleep. (kernel/sched.c )
extern void wake_up(struct task_struct ** p);
// Check if the current process is in the specified user group grp.
extern int in_group_p(gid_t grp);

/*
 * Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall
 * 4-TSS0, 5-LDT0, 6-TSS1 etc ...
 */
// Look for the entry of the first TSS in the global table. 0-nul (not used), 1-code segment
// cs, 2-data segment ds, 3-system segment syscall, 4-task state segment TSS0, 5-local table
// LTD0, 6-task state segment TSS1, and so on.
// As can be guessed from the original comment, Linus had wanted to put the code of the system
// call in the fourth independent segment of the GDT table. But then did not do that, so he
// kept the fourth descriptor item (the syscall item) in the GDT table idle.
// The following define selector indexes of the first TSS and LDT descriptors in GDT table.
#define FIRST_TSS_ENTRY 4
#define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1)

// This macro is used to calculate the selector (offset) of the TSS descriptor of the n-th task
// in the GDT. Since each descriptor occupies 8 bytes, FIRST_TSS_ENTRY<<3 indicates the starting
// offset position of the descriptor in the GDT table. Since each task uses 1 TSS and 1 LDT
// descriptor, which occupies a total of 16 bytes, n<<4 is required to indicate the corresponding
// TSS start position. The value obtained by this macro is also the index value of the selector
// of the TSS. The latter macro defines the selector (offset) of the LDT descriptor in the GDT.
#define _TSS(n) ((((unsigned long) n)<<4)+(FIRST_TSS_ENTRY<<3))
#define _LDT(n) ((((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))

// The embedded assembly macro is used to load the TSS segment selector of the n-th task into
```

```
    // the task register TR. The latter macro is used to load the LDT segment selector of the n-th
    // task into the local descriptor table register LDTR.
208 #define ltr(n) __asm__("ltr %%ax"::"a" (_TSS(n)))
209 #define lldt(n) __asm__("lldt %%ax"::"a" (_LDT(n)))

    // Get the task number of the current running task (it is the index in the task array, which
    // is different from the process number pid), used in program kernel/traps.c, line 78.
    // Returns: n - the current task number.
210 #define str(n) \
211 __asm__("str %%ax\n\t" \                 // Save the TSS selector in the task register to EAX.
212         "subl %2,%%eax\n\t" \            // (EAX - FIRST_TSS_ENTRY * 8) => EAX
213         "shrl $4,%%eax" \                // (EAX / 16) => EAX = current task number.
214         :"=a" (n) \
215         :"a" (0),"i" (FIRST_TSS_ENTRY<<3))
216 /*
217  *      switch_to(n) should switch tasks to task nr n, first
218  * checking that n isn't the current task, in which case it does nothing.
219  * This also clears the TS-flag if the task we switched to has used
220  * tha math co-processor latest.
221  */
    // Jumping to a address composed by TSS selector will cause the CPU to switch task.
    // Input: %0 - points to __tmp; %1 - points to __tmp.b for the new TSS selector; DX - TSS segment
    // selector for new task n; ECX - task structure pointer for new task n task[ n].
    // The temporary data structure __tmp is used to construct the operand of the far jump instruction
    // on line 228. The operand consists of a 4-byte offset address and a 2-byte segment selector.
    // Therefore, the value of 'a' in __tmp is a 32-bit offset, and the lower 2 bytes of 'b' are
    // selectors for the new TSS segment (high 2 bytes are not used). Jumping to the TSS segment
    // selector causes the task to switch to the process corresponding to the TSS. The 'a' value
    // is useless for long jumps that cause task switching. The indirect jump instruction on line
    // 228 uses a 6-byte operand as the long pointer to the jump destination. The format is:
    //     JMP 16-bit segment selector: 32-bit offset.
    // After the task is switched back, it is determined by comparing the original task pointer
    // with the last used coprocessor task pointer saved in the last_task_used_math variable when
    // determining whether the original task was executed last time. See description of the
    // math_state_restore() function in kernel/sched.c.
222 #define switch_to(n) {\
223 struct {long a,b;} __tmp; \
224 __asm__("cmpl %%ecx,_current\n\t" \      // Is task n the current task? (current ==task[n]?)
225         "je 1f\n\t" \                    // If so, do nothing and return.
226         "movw %%dx,%1\n\t" \             // The new task TSS selector is stored in __tmp.b
227         "xchgl %%ecx,_current\n\t" \     // Current = task[n]; ECX =the task switched out.
228         "ljmp %0\n\t" \                  // Long jump to *&__tmp, causing task switching.
    // The following statement will not continue until the task is switched back. First check if
    //the original task used the coprocessor last time. If yes, clear the TS flag in CR0.
229         "cmpl %%ecx,_last_task_used_math\n\t" \  // original task used math last time?
230         "jne 1f\n\t" \                   // If not, jump forward and exit.
231         "clts\n" \                       // If yes, then clear the TS flag in CR0.
232         "1:" \                           // exit.
233         ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
234         "d" (_TSS(n)),"c" ((long) task[n])); \
235 }
236
    // Page alignment (nowhere in the kernel to reference this!)
```

```
#define PAGE_ALIGN(n) (((n)+0xfff)&0xfffff000)

// Set each base address fields in the descriptor at address addr.
// %0 - addr offset 2; %1 - addr offset 4; %2 - addr offset 7; EDX - base address.
#define _set_base(addr,base) \
__asm__("movw %%dx,%0\n\t" \          // Lower 16 bits(15-0) of the base=>[addr+2].
        "rorl $16,%%edx\n\t" \        // Upper 16 bits(31-16) of the base in EDX => DX.
        "movb %%dl,%1\n\t" \          // Lower 8 bits(23-16) of the upper 16 bits =>[addr+4]
        "movb %%dh,%2" \              // Upper 8 bits(31-24) of the upper 16 bits =>[addr+7]
        ::"m" (*((addr)+2)), \
          "m" (*((addr)+4)), \
          "m" (*((addr)+7)), \
          "d" (base) \
        :"dx")                        // Tell gcc that the value in EDX has been changed.

// Set the segment limit field in the descriptor at address addr.
// %0 - address addr; %1 - addr offset 6; EDX - segment length limit.
#define _set_limit(addr,limit) \
__asm__("movw %%dx,%0\n\t" \          // Lower 16 bits(15-0) of the segment limit =>[addr].
        "rorl $16,%%edx\n\t" \        // Upper 4 bits(19-16) of the limit in EDX => DL.
        "movb %1,%%dh\n\t" \          // Original [addr+6] => DH, the upper 4 bits are flags.
        "andb $0xf0,%%dh\n\t" \       // Clear lower 4 bits of DH (will be stored at 19-16).
        "orb %%dh,%%dl\n\t" \         // Combine original high 4-bit flags and upper 4 bits
        "movb %%dl,%1" \              // (19-16)of the limit into 1 byte,stored at [addr+6].
        ::"m" (*(addr)), \
          "m" (*((addr)+6)), \
          "d" (limit) \
        :"dx")

// Set the base address field of the descriptor in the local descriptor table LDT.
// Set the segment limit field of the descriptor in the LDT.
#define set_base(ldt,base) _set_base( ((char *)&(ldt)) , base )
#define set_limit(ldt,limit) _set_limit( ((char *)&(ldt)) , (limit-1)>>12 )

// Get base from the descriptor at the address addr. It is the opposite of _set_base().
// EDX - store base (__base); %1 - addr offset 2; %2 - addr offset 4; %3 - addr offset 7.
#define _get_base(addr) ({\
unsigned long __base; \
__asm__("movb %3,%%dh\n\t" \          // Upper 8 bits(31-24) of upper 16 bits [addr+7] =>DH.
        "movb %2,%%dl\n\t" \          // Lower 8 bits(23-16) of upper 16 bits [addr+4] =>DL.
        "shll $16,%%edx\n\t" \        // Upper 16 bits are moved to the upper 16 bits of EDX.
        "movw %1,%%dx" \              // Lower 16 (15-0) bits of the base at [addr+2] => DX.
        :"=d" (__base) \              // Thus EDX contains a 32-bit segment base address.
        :"m" (*((addr)+2)), \
         "m" (*((addr)+4)), \
         "m" (*((addr)+7))); \
__base;})

// Take the base address in the segment descriptor pointed to by 'ldt' in the LDT.
#define get_base(ldt) _get_base( ((char *)&(ldt)) )

// Take the segment limit in the descriptor specified by the segment selector 'segment'.
// The instruction LSL is an abbreviation of Load Segment Limit. It takes the scattered limit
// length bits from the descriptor of the specified segment and puts the complete segment limit
// value into the specified register. The resulting segment limit value is the actual number
// of bytes minus 1, so you need to add 1 to return.
// %0 - the length of the segment (bytes); %1 - the segment selector 'segment'.
#define get_limit(segment) ({ \
unsigned long __limit; \
__asm__("lsll %1,%0\n\tincl %0":"=r" (__limit):"r" (segment)); \
__limit;})

#endif

```

14.30 sys.h

14.30.1 Functionality

The sys.h header file lists the prototypes of all system-call functions in the kernel, as well as the system-call function pointer table.

14.30.2 Code annotation

Program 14-27 linux/include/linux/sys.h

```
/*
 * Why isn't this a .c file?  Enquiring minds....
 */

extern int sys_setup();         // 0 - System initializations.    (kernel/blk_drv/hd.c,74)
extern int sys_exit();          // 1 - Program exit.              (kernel/exit.c, 365)
extern int sys_fork();          // 2 - Create a new process.      (kernel/sys_call.s,222)
extern int sys_read();          // 3 - Read file.                 (fs/read_write.c,55)
extern int sys_write();         // 4 - Write file.                (fs/read_write.c,83)
extern int sys_open();          // 5 - Open file.                 (fs/open.c,171)
extern int sys_close();         // 6 - Close file.                (fs/open.c,219)
extern int sys_waitpid();       // 7 - Wait process to terminate. (kernel/exit.c,370)
extern int sys_creat();         // 8 - Create file.               (fs/open.c,214)
extern int sys_link();          // 9 - Ceate hard linke to a file. (fs/namei.c,837)
extern int sys_unlink();        // 10 - Delete a filename(or file) (fs/namei.c,709)
extern int sys_execve();        // 11 - Execute a program.        (kernel/sys_call.s,214)
extern int sys_chdir();         // 12 - Change current directory. (fs/open.c,76)
extern int sys_time();          // 13 - Get current time.         (kernel/sys.c,134)
extern int sys_mknod();         // 14 - Create block/char special file.  (fs/namei.c,464)
extern int sys_chmod();         // 15 - Change file mode.         (fs/open.c,106)
extern int sys_chown();         // 16 - Chane file owner or group. (fs/open.c,122)
extern int sys_break();         // 17 -                           (kernel/sys.c,33)*
extern int sys_stat();          // 18 - Get file status using path name. (fs/stat.c,36)
extern int sys_lseek();         // 19 - Reposite r/w file offset. (fs/read_write.c,25)
extern int sys_getpid();        // 20 - Get process id.           (kernel/sched.c,380)
extern int sys_mount();         // 21 - Mount a file-system.      (fs/super.c,199)
extern int sys_umount();        // 22 - Unmount a file-system.    (fs/super.c,166)
extern int sys_setuid();       // 23 - Set process user id.      (kernel/sys.c,196)
extern int sys_getuid();       // 24 - Get process user id.      (kernel/sched.c,390)
extern int sys_stime();        // 25 - Set system time.          (kernel/sys.c,207)
extern int sys_ptrace();       // 26 - Process tracing.          (kernel/sys.c,38)*
extern int sys_alarm();        // 27 - Set alarm time.           (kernel/sched.c,370)
extern int sys_fstat();        // 28 - Get file status by using handle. (fs/stat.c,58)
extern int sys_pause();        // 29 - Pause the process running. (kernel/sched.c,164)
extern int sys_utime();        // 30 - Set file access and modified time. (fs/open.c,25)
extern int sys_stty();         // 31 - Modify terminal settings. (kernel/sys.c,43)*
extern int sys_gtty();         // 32 - Get terminal settings.    (kernel/sys.c,48)*
extern int sys_access();       // 33 - Check file access permission. (fs/open.c,48)
extern int sys_nice();         // 34 - Set execution priority.   (kernel/sched.c,410)
extern int sys_ftime();        // 35 - Get date and time.        (kernel/sys.c,28)*
extern int sys_sync();         // 36 - Synchronous data with dev. (fs/buffer.c,44)
extern int sys_kill();         // 37 - Terminate a process.      (kernel/exit.c,205)
extern int sys_rename();       // 38 - Change filename.          (kernel/sys.c,53)*
extern int sys_mkdir();        // 39 - Make a directory.         (fs/namei.c,515)
extern int sys_rmdir();        // 40 - Remove a directory.       (fs/namei.c,635)
extern int sys_dup();          // 41 - Duplicate a file handle.  (fs/fcntl.c,42)
extern int sys_pipe();         // 42 - Create a pipe.            (fs/pipe.c,76)
extern int sys_times();        // 43 - Get running time.         (kernel/sys.c,216)
extern int sys_prof();         // 44 - Execution time zone.      (kernel/sys.c,58)*
extern int sys_brk();          // 45 - Change data segment len.  (kernel/sys.c,228)
extern int sys_setgid();       // 46 - Set process group id.     (kernel/sys.c,98)
extern int sys_getgid();       // 47 - Set process group id.     (kernel/sched.c,400)
extern int sys_signal();       // 48 - Signal processing.        (kernel/signal.c,85)
extern int sys_geteuid();      // 49 - Get efficient user id.    (kenrl/sched.c,395)
extern int sys_getegid();      // 50 - Get efficient group id.   (kenrl/sched.c,405)
extern int sys_acct();         // 51 - Process accounting.       (kernel/sys.c,109)*
extern int sys_phys();         // 52 - Map phy mem to process space.  (kernel/sys.c,114)*
extern int sys_lock();         // 53 -                           (kernel/sys.c,119)*
extern int sys_ioctl();        // 54 - Device i/o control.       (fs/ioctl.c,31)
extern int sys_fcntl();        // 55 - File operation control.   (fs/fcntl.c,47)
extern int sys_mpx();          // 56 -                           (kernel/sys.c,124)*
extern int sys_setpgid();      // 57 - Set process group id.     (kernel/sys.c,245)
extern int sys_ulimit();       // 58 - Statistical resource usage. (kernel/sys.c,129)
extern int sys_uname();        // 59 - Show system info.         (kernel/sys.c,343)
extern int sys_umask();        // 60 - Get default file creation mode.  (kernel/sys.c,515)
extern int sys_chroot();       // 61 - Change root directory.    (fs/open.c,91)
extern int sys_ustat();        // 62 - Get file system states.   (fs/open.c,20)
extern int sys_dup2();         // 63 - Duplicate file handle.    (fs/fcntl.c,36)
extern int sys_getppid();      // 64 - Get parenet process id.   (kernel/sched.c,385)
extern int sys_getpgrp();      // 65 - Get pid (getpgid(0))      (kernel/sys.c,271)
extern int sys_setsid();       // 66 - Set new session id.       (kernel/sys.c,276)
extern int sys_sigaction();    // 67 - Set signal operation.     (kernel/signal.c,100)
extern int sys_sgetmask();     // 68 - Get signal mask code.     (kernel/signal.c,14)
extern int sys_ssetmask();     // 69 - Set signal mask code.     (kernel/signal.c,19)
extern int sys_setreuid();     // 70 - Set real/efficient uid.   (kernel/sys.c,159)
extern int sys_setregid();     // 71 - Set real/efficient pid.   (kernel/sys.c,74)
extern int sys_sigpending();   // 73 - Check pending signals.    (kernel/signal.c,27)
extern int sys_sigsuspend();   // 72 - Suspending a process.     (kernel/signal.c,48)
extern int sys_sethostname();  // 74 - Set host name.            (kernel/sys.c,357)
extern int sys_setrlimit();    // 75 - Set resource using limit. (kernel/sys.c,387)
extern int sys_getrlimit();    // 76 - Get resource using limit.  (kernel/sys.c,375)
extern int sys_getrusage();    // 77 - Get resource usage.        (kernel/sys.c,412)
extern int sys_gettimeofday(); // 78 - Get time of the day.       (kernel/sys.c,440)
extern int sys_settimeofday(); // 79 - Set time of the day.       (kernel/sys.c,466)
extern int sys_getgroups();    // 80 - Get process all group ids. (kernel/sys.c,289)
extern int sys_setgroups();    // 81 - Set process group array.   (kernel/sys.c,307)
extern int sys_select();       // 82 - wait for file change state. (fs/select.c,216)
extern int sys_symlink();      // 83 - Create file sysmbol link.  (fs/namei.c, 767)
extern int sys_lstat();        // 84 - Get link file state.       (fs/stat.c, 47)
extern int sys_readlink();     // 85 - Read link file contents.   (fs/stat.c, 69)
extern int sys_uselib();       // 86 - Select shared lib.         (fs/exec.c, 42)

// The function pointer table of the system-call, which is used by the system-call interrupt
// handler (int 0x80) as a jump table.
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
sys_setreuid,sys_setregid, sys_sigsuspend, sys_sigpending, sys_sethostname,
sys_setrlimit, sys_getrlimit, sys_getrusage, sys_gettimeofday,
sys_settimeofday, sys_getgroups, sys_setgroups, sys_select, sys_symlink,
sys_lstat, sys_readlink, sys_uselib };

/* So we don't have to do any more manual updating.... */
int NR_syscalls = sizeof(sys_call_table)/sizeof(fn_ptr);

```

14.31 tty.h

14.31.1 Functionality

The tty.h file defines the terminal data structure and some constants, as well as the macros used by the tty queue buffer operations.

14.31.2 Code annotation

Program 14-28 linux/include/linux/tty.h

```
/*
 * 'tty.h' defines some structures used by tty_io.c and some defines.
 *
 * NOTE! Don't touch this without checking that nothing in rs_io.s or
 * con_io.s breaks. Some constants are hardwired into the system (mainly
 * offsets into 'tty_queue'
 */

#ifndef _TTY_H
#define _TTY_H

#define MAX_CONSOLES    8         // The maximum number of virtual consoles.
#define NR_SERIALS      2         // The number of serial terminals.
#define NR_PTYS         4         // The number of psesudo terminals.

extern int NR_CONSOLES;           // The number of virtual consoles.

// <termios.h> Terminal input and output function header file. It mainly defines the terminal
// interface that controls the asynchronous communication port.
#include <termios.h>

#define TTY_BUF_SIZE 1024         // The size of the tty queue buffer.

 // Tty character buffer queue data structure. Used for read, write, and auxiliary (canonical)
 // buffer queues in the tty_struc structure.
 // The first field 'data' contains the character line value (not the number of characters) in
 // the queue buffer. For a serial terminal, the serial port address is stored therein.
struct tty_queue {
        unsigned long data;               // Char lines in the buffer or the serial port.
        unsigned long head;               // The data header in the buffer.
        unsigned long tail;               // The data tail in the buffer.
        struct task_struct * proc_list;   // process list waiting for this queue buffer.
        char buf[TTY_BUF_SIZE];           // The buffer.
};

 // These macros are used to check the terminal type.
#define IS_A_CONSOLE(min)       (((min) & 0xC0) == 0x00)      // console.
#define IS_A_SERIAL(min)        (((min) & 0xC0) == 0x40)      // serial terminal.
#define IS_A_PTY(min)           ((min) & 0x80)                // psesudo terminal.
#define IS_A_PTY_MASTER(min)    (((min) & 0xC0) == 0x80)      // master pty.
#define IS_A_PTY_SLAVE(min)     (((min) & 0xC0) == 0xC0)      // slate pty.
#define PTY_OTHER(min)          ((min) ^ 0x40)                // other type terminal.

 // The buffer operation macros in the tty queue are defined below. (tail is in front, head is
 // in the back, see figure in tty_io.c).
 // The buffer pointer 'a' is shifted forward/backward by 1 byte, and if it has exceeded the
 // right/left side of the buffer, the pointer moves cyclically.
#define INC(a) ((a) = ((a)+1) & (TTY_BUF_SIZE-1))
#define DEC(a) ((a) = ((a)-1) & (TTY_BUF_SIZE-1))
 // Empty the buffer.  // The size of the remaining free area of the buffer.
 // The last position in the buffer. // Buffer is full.// The number of chars in the buffer.
#define EMPTY(a) ((a)->head == (a)->tail)
#define LEFT(a) (((a)->tail-(a)->head-1)&(TTY_BUF_SIZE-1))
#define LAST(a) ((a)->buf[(TTY_BUF_SIZE-1)&((a)->head-1)])
#define FULL(a) (!LEFT(a))
#define CHARS(a) (((a)->head-(a)->tail)&(TTY_BUF_SIZE-1))
 // Get a character from the 'tail' in the 'queue' buffer, and tail++.
 // Place a character at the 'head' in the 'queue' queue buffer, and head++.
```

```
44 #define GETCH(queue,c) \
45 (void)({c=(queue)->buf[(queue)->tail];INC((queue)->tail);})
46 #define PUTCH(c,queue) \
47 (void)({(queue)->buf[(queue)->head]=(c);INC((queue)->head);})
48
   // Check the type of characters typed on the terminal keyboard.
49 #define INTR_CHAR(tty) ((tty)->termios.c_cc[VINTR])        // Send signal SIGINT.
50 #define QUIT_CHAR(tty) ((tty)->termios.c_cc[VQUIT])        // Send signal SIGQUIT.
51 #define ERASE_CHAR(tty) ((tty)->termios.c_cc[VERASE])      // Erase a char.
52 #define KILL_CHAR(tty) ((tty)->termios.c_cc[VKILL])        // Kill a line of chars.
53 #define EOF_CHAR(tty) ((tty)->termios.c_cc[VEOF])          // End of file char.
54 #define START_CHAR(tty) ((tty)->termios.c_cc[VSTART])      // Start output.
55 #define STOP_CHAR(tty) ((tty)->termios.c_cc[VSTOP])        // Stop output.
56 #define SUSPEND_CHAR(tty) ((tty)->termios.c_cc[VSUSP])     // Send signal SIGTSTP.
57
   // Terminal data structure.
58 struct tty_struct {
59         struct termios termios;                     // Terminal io mode & control structure.
60         int pgrp;                                   // The pgroup the terminal belongs to.
61         int session;                                // The session.
62         int stopped;                                // Terminal stopped flag.
63         void (*write)(struct tty_struct * tty);     // tty write function pointer.
64         struct tty_queue *read_q;                   // tty read queue.
65         struct tty_queue *write_q;                  // tty write queue.
66         struct tty_queue *secondary;                // tty aux or canonical queue.
67         };
68
69 extern struct tty_struct tty_table[];
70 extern int fg_console;                              // Front console number.
71
   // The following macro obtains a pointer to the tty structure corresponding to the terminal
   // number 'nr' in tty_table[] according to the terminal type.
   // The second half of line 73 is used to select the corresponding tty structure in the tty_table[]
   // table based on the sub-device number 'dev'. If dev = 0, it means that the foreground terminal
   // is being used, so you can use the terminal number 'fg_console' as the tty_table[] entry index
   // to get the tty structure. If dev is greater than 0, then it should be considered in two cases:
   // (1) dev is the virtual terminal number; (2) dev is the serial terminal number or pseudo terminal
   // number. For virtual terminals, the tty structure in tty_table[] is indexed by dev-1(0 --
   // 63). For other types of terminals, their tty structure index entry is dev.
   // For example, if dev = 64, which means it is a serial terminal 1, its tty structure is
   // ttb_table[dev]. If dev = 1, the tty structure of the corresponding terminal is tty_table[0].
   // See lines 70--73 of the tty_io.c program.
72 #define TTY_TABLE(nr) \
73 (tty_table + ((nr) ? (((nr) < 64)? (nr)-1:(nr)) : fg_console))
74
   // Here is the initial value of the special character array c_cc[] that can be changed in the
   // terminal termios structure. POSIX.1 defines 11 special characters, but the Linux system
   // additionally defines the six special characters used by SVR4. If _POSIX_VDISABLE(\0) is
   // defined, then when an item value is equal to _POSIX_VDISABLE, the corresponding special
   // character is prohibited. They are represented by octals in the initial values on line 81.
75 /*      intr=^C         quit=^|         erase=del       kill=^U
76         eof=^D          vtime=\0        vmin=\1         sxtc=\0
77         start=^Q        stop=^S         susp=^Z         eol=\0
```

```
        reprint=^R      discard=^U      werase=^W       lnext=^V
        eol2=\0
*/
#define INIT_C_CC "\003\034\177\025\004\0\1\0\021\023\032\0\022\017\027\026\0"

void rs_init(void);                  // serial terminal init.(kernel/chr_drv/serial.c)
void con_init(void);                 // Control terminal init.(kernel/chr_drv/console.c)
void tty_init(void);                 // (kernel/chr_drv/tty_io.c)

int tty_read(unsigned c, char * buf, int n);     // (kernel/chr_drv/tty_io.c)
int tty_write(unsigned c, char * buf, int n);    // (kernel/chr_drv/tty_io.c)

void con_write(struct tty_struct * tty);         // (kernel/chr_drv/console.c)
void rs_write(struct tty_struct * tty);          // (kernel/chr_drv/serial.c)
void mpty_write(struct tty_struct * tty);        // (kernel/chr_drv/pty.c)
void spty_write(struct tty_struct * tty);        // (kernel/chr_drv/pty.c)

void copy_to_cooked(struct tty_struct * tty);    // (kernel/chr_drv/tty_io.c)

void update_screen(void);                        // (kernel/chr_drv/console.c)

#endif

```

14.32 Header files in the include/sys/ directory

The include/sys/ directory contains eight header files that are closely related to the system hardware resources and their settings, as shown in Listing 14-4.

List 14-4 Files in the linux/include/sys/ directory

Filename	Size	Last Modified Time (GMT)	Description
param.h	196 bytes	1992-01-06 21:10:22	
resource.h	1809 bytes	1992-01-03 18:52:56	
stat.h	1376 bytes	1992-01-11 18:42:48	
time.h	1799 bytes	1992-01-09 03:51:28	
times.h	200 bytes	1991-09-17 15:03:06	
types.h	928 bytes	1992-01-14 13:50:35	
utsname.h	272 bytes	1992-01-04 15:05:42	
wait.h	593 bytes	1991-12-22 15:08:01	

14.33 param.h

14.33.1 Functionality

The param.h file contains and defines some parameter values related to the system hardware.

14.33.2 Code annotation

Program 14-29 linux/include/sys/param.h

```
#ifndef _SYS_PARAM_H
#define _SYS_PARAM_H

#define HZ 100                  // The system clock frequency, 100 times per second.
#define EXEC_PAGESIZE 4096      // Executable page size.

#define NGROUPS         32      /* Max number of groups per user */
#define NOGROUP         -1

#define MAXHOSTNAMELEN 8        // The maximum length of the host name, 8 bytes.

#endif

```

14.34 resource.h

14.34.1 Functionality

The resource.h header file contains information about the limits and utilization of the system resources used by the process. It defines the rusage structure and symbolic constants RUSAGE_SELF, RUSAGE_CHILDREN used by the system-call (or library function) getrusage(). It also defines the rlimit structure used by system-calls or functions getrlimit() and setrlimit() and the symbol constants used in the arguments.

The information accessed by getrlimit() and setrlimit() is in the rlim[] array of the process task structure. The array has a total of RLIM_NLIMITS items, each of which is an rlimit structure that defines the restrictions on the use of a resource, as shown in Figure 14-5. As shown in the figure, six resource limits are defined for a process in the Linux 0.12 kernel, and they are defined on lines 41-46 in this header file.

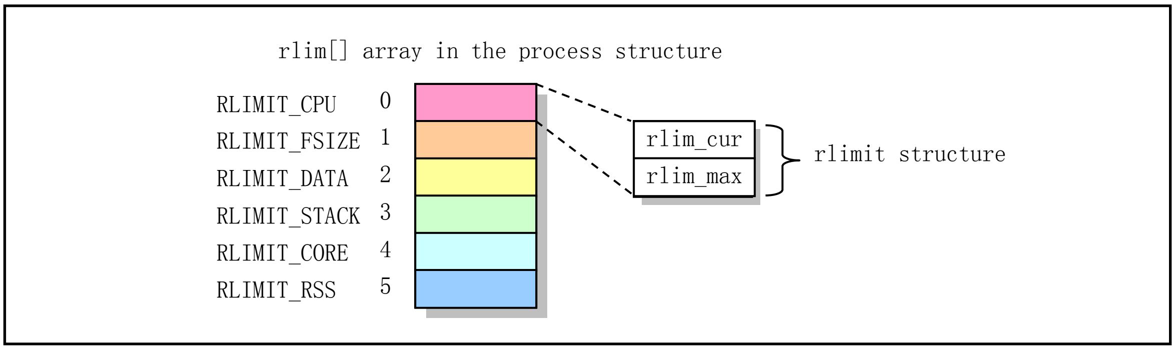


Figure 14-5 The uses of the rlim[] array in the process descriptor

14.34.2 Code annotation

Program 14-30 linux/include/sys/resource.h

```
1 /*
2  * Resource control/accounting header file for linux
3  */
4
5 #ifndef _SYS_RESOURCE_H
6 #define _SYS_RESOURCE_H
7
  // The following symbol constants and structures are used for getrusage(). See line 412 of the
  // kernel/sys.c file.
8 /*
9  * Definition of struct rusage taken from BSD 4.3 Reno
10 *
11 * We don't support all of these yet, but we might as well have them....
12 * Otherwise, each time we add new items, programs which depend on this
13 * structure will lose.  This reduces the chances of that happening.
14 */
  // Below is the symbol constants used by the parameter 'who' of the getrusage().
  // Returns the resource utilization information of the current process.
```

```
   // Returns the resource utilization of child processes who has terminated and is waiting.
15 #define RUSAGE_SELF     0          // Returns resource utilization of the current process.
16 #define RUSAGE_CHILDREN -1         // Returns resource utilization of the child processes.
17
   // Rusage is the resource utilization statistical structure of the process, and is used by
   // getrusage() to return the statistical value of the resource utilization of the specified
   // process. The Linux 0.12 kernel uses only the first two fields, which are all timeval structures
   // (include/sys/time.h). The ru_utime field is used to store the running time statistics of
   // the user in the user state; the ru_stime field is used to store the running time statistics
   // of the process in the kernel state.
18 struct  rusage {
19         struct timeval ru_utime;        /* user time used */
20         struct timeval ru_stime;        /* system time used */
21         long    ru_maxrss;              /* maximum resident set size */
22         long    ru_ixrss;               /* integral shared memory size */
23         long    ru_idrss;               /* integral unshared data size */
24         long    ru_isrss;               /* integral unshared stack size */
25         long    ru_minflt;              /* page reclaims */
26         long    ru_majflt;              /* page faults */
27         long    ru_nswap;               /* swaps */
28         long    ru_inblock;             /* block input operations */
29         long    ru_oublock;             /* block output operations */
30         long    ru_msgsnd;              /* messages sent */
31         long    ru_msgrcv;              /* messages received */
32         long    ru_nsignals;            /* signals received */
33         long    ru_nvcsw;               /* voluntary context switches */
34         long    ru_nivcsw;              /* involuntary " */
35 };
36
   // The following are the symbol constants and structures used by getrlimit() and setrlimit().
37 /*
38  * Resource limits
39  */
   // The following are the types of resources defined in the Linux 0.12 kernel. They are the range
   // of values for the first parameter resource in getrlimit() and setrlimit(). In fact, these
   // symbolic constants are the indexes of the items of the rlim[] array in the process task
   // structure. Each item in the rlim[] array is an rlimit structure, see line 58 below.
40
41 #define RLIMIT_CPU      0          /* CPU time in ms */
42 #define RLIMIT_FSIZE    1          /* Maximum filesize */
43 #define RLIMIT_DATA     2          /* max data size */
44 #define RLIMIT_STACK    3          /* max stack size */
45 #define RLIMIT_CORE     4          /* max core file size */
46 #define RLIMIT_RSS      5          /* max resident set size */
47
48 #ifdef notdef
49 #define RLIMIT_MEMLOCK  6          /* max locked-in-memory address space*/
50 #define RLIMIT_NPROC    7          /* max number of processes */
51 #define RLIMIT_OFILE    8          /* max number of open files */
52 #endif
53
   // This symbolic constant defines the types of resources that are restricted in Linux. The number
   // of resource types defined here is 6, so only the first 6 items are valid.
```

```
#define RLIM_NLIMITS    6

#define RLIM_INFINITY   0x7fffffff     // The resource is unlimited or cannot be modified.

// Resource limit structure.
struct rlimit {
        int     rlim_cur;              // Current resource limit, or soft limit.
        int     rlim_max;              // Hard limit.
};

#endif /* _SYS_RESOURCE_H */

```

14.35 stat.h

14.35.1 Functionality

The stat.h header file shows the data returned by the file function stat() and its structure type, as well as some property operation test macros and function prototypes.

14.35.2 Code annotation

Program 14-31 linux/include/sys/stat.h

```
#ifndef _SYS_STAT_H
#define _SYS_STAT_H

#include <sys/types.h>

// File status data structure. All field values are available from the file's inode structure.
struct stat {
        dev_t   st_dev;         // The device number that contains the file.
        ino_t   st_ino;         // File i-node number.
        umode_t st_mode;        // File type and modes (see below).
        nlink_t st_nlink;       // The number of links to the file.
        uid_t   st_uid;         // The user ID of the file.
        gid_t   st_gid;         // The group ID of the file.
        dev_t   st_rdev;        // Device number (if it is a special char or block file).
        off_t   st_size;        // File size (in bytes).
        time_t  st_atime;       // Last access time.
        time_t  st_mtime;       // Last modify time.
        time_t  st_ctime;       // The time the inode was last changed.
};

//
// The following are some of the symbol constants defined for the values used by the st_mode
// field. These values are all expressed in octal. For the sake of remembering, these symbolic
// names are a combination of the initials or abbreviations of some English words. For example,
// each uppercase letter of the name S_IFMT represents the words State, Inode, File, Mask, and
// Type; and the name S_IFREG is a combination of the initial letters of State, Inode, File,
// and REGular; the name S_IRWXU is State, Inode, Read, Write, eXecute and User. Other names
// can be deduced by analogy.
// File types:
#define S_IFMT  00170000      // File type bit mask (in octal).
#define S_IFLNK  0120000      // Symbolic link.
#define S_IFREG  0100000      // Regular file.
#define S_IFBLK  0060000      // Block special device files, such as harddisk dev/hd0.
#define S_IFDIR  0040000      // Directory.
#define S_IFCHR  0020000      // Char device file.
#define S_IFIFO  0010000      // FIFO special file.
// File mode bits:
// S_ISUID is used to test whether the set-user-ID flag of the file is set. If the flag is set,
// the efficient user ID of the process will be set to the user ID of the file owner when the
// file is executed. S_ISGID performs the same processing for the group ID.
#define S_ISUID  0004000      // Set the user ID (set-user-ID) at execution time.
#define S_ISGID  0002000      // Set the group ID (set-group-ID) at execution time.
#define S_ISVTX  0001000      // For directories, the restricted delete flag.

#define S_ISLNK(m)     (((m) & S_IFMT) == S_IFLNK)    // Test if it's a symbolic link file.
#define S_ISREG(m)     (((m) & S_IFMT) == S_IFREG)    // a regular file.
#define S_ISDIR(m)     (((m) & S_IFMT) == S_IFDIR)    // a directory.
#define S_ISCHR(m)     (((m) & S_IFMT) == S_IFCHR)    // a char device file.
#define S_ISBLK(m)     (((m) & S_IFMT) == S_IFBLK)    // a block device file.
#define S_ISFIFO(m)    (((m) & S_IFMT) == S_IFIFO)    // a FIFO special file.

// File access permission:
#define S_IRWXU 00700      // The owner can read, write, execute/search (U for User).
#define S_IRUSR 00400      // The owner can read.
#define S_IWUSR 00200      // The owner can write.
#define S_IXUSR 00100      // The owner can execute/search.

#define S_IRWXG 00070      // Group members can read, write, execute/search (G for Group).
#define S_IRGRP 00040      // Group members can read.
#define S_IWGRP 00020      // Group members can write.
#define S_IXGRP 00010      // Group members can execute/search.

#define S_IRWXO 00007      // Others can read, write, execute/search (O stands for Other).
#define S_IROTH 00004      // Others can read (the last 3 letters represent Other).
#define S_IWOTH 00002      // Others can write.
#define S_IXOTH 00001      // Others can execute/search.

extern int chmod(const char *_path, mode_t mode);        // Change file modes.
extern int fstat(int fildes, struct stat *stat_buf);     // Get file state info by fhandle.
extern int mkdir(const char *_path, mode_t mode);        // Make a directory.
extern int mkfifo(const char *_path, mode_t mode);       // Make a pipe file.
extern int stat(const char *filename, struct stat *stat_buf); // Get file state info.
extern mode_t umask(mode_t mask);                        // Set mode mask.

#endif

```

14.36 time.h

14.36.1 Functionality

The time.h header file defines the timeval structure and the internally used itimerval structure, as well as the time zone constants.

14.36.2 Code annotation

Program 14-32 linux/include/sys/time.h

```
#ifndef _SYS_TIME_H
#define _SYS_TIME_H

/* gettimofday returns this */
struct timeval {
        long    tv_sec;         /* seconds */
        long    tv_usec;        /* microseconds */
};

// Time zone structure.
// TZ is the abbreviation of Time Zone, and DST is the Daylight Saving Time.
struct timezone {
        int     tz_minuteswest; /* minutes west of Greenwich */
        int     tz_dsttime;     /* type of dst correction */
};

#define DST_NONE        0       /* not on dst */
#define DST_USA         1       /* USA style dst */
#define DST_AUST        2       /* Australian style dst */
#define DST_WET         3       /* Western European dst */
#define DST_MET         4       /* Middle European dst */
#define DST_EET         5       /* Eastern European dst */
#define DST_CAN         6       /* Canada */
#define DST_GB          7       /* Great Britain and Eire */
#define DST_RUM         8       /* Rumania */
#define DST_TUR         9       /* Turkey */
#define DST_AUSTALT     10      /* Australian style with shift in 1986 */

// A setting macros for the file descriptor set, used for the select() function.
#define FD_SET(fd,fdsetp)     (*(fdsetp) |= (1 << (fd)))     // Set the fd in the fd set.
#define FD_CLR(fd,fdsetp)     (*(fdsetp) &= ~(1 << (fd)))    // Clear the fd in the set.
#define FD_ISSET(fd,fdsetp)   ((*(fdsetp) >> fd) & 1)        // Is the fd in the set ?
#define FD_ZERO(fdsetp)       (*(fdsetp) = 0)                // Clear all fds in the set.

/*
 * Operations on timevals.
 *
 * NB: timercmp does not work for >= or <=.
 */
// The operation macro of the timeval time structure.
#define timerisset(tvp)         ((tvp)->tv_sec || (tvp)->tv_usec)
```

```
#define timercmp(tvp, uvp, cmp) \
        ((tvp)->tv_sec cmp (uvp)->tv_sec || \
         (tvp)->tv_sec == (uvp)->tv_sec && (tvp)->tv_usec cmp (uvp)->tv_usec)
#define timerclear(tvp)         ((tvp)->tv_sec = (tvp)->tv_usec = 0)

/*
 * Names of the interval timers, and structure
 * defining a timer setting.
 */
#define ITIMER_REAL     0               // Decrease in real time.
#define ITIMER_VIRTUAL  1               // Decrease in the virtual time of the process.
#define ITIMER_PROF     2               // Decrease in process virtual time or runtime.

// Internal time structure.
struct  itimerval {
        struct  timeval it_interval;    /* timer interval */
        struct  timeval it_value;       /* current value */
};

#include <time.h>
#include <sys/types.h>

int gettimeofday(struct timeval * tp, struct timezone * tz);
int select(int width, fd_set * readfds, fd_set * writefds,
        fd_set * exceptfds, struct timeval * timeout);

#endif /*_SYS_TIME_H*/

```

14.37 times.h

14.37.1 Functionality

The times.h header file mainly defines the file access and modification time structure tms. It will be returned by the times() function. Where time_t is defined in sys/types.h. A function prototype times() is also defined.

14.37.2 Code annotation

Program 14-33 linux/include/sys/times.h

```
#ifndef _TIMES_H
#define _TIMES_H

#include <sys/types.h>     // Type header file. The basic system data types are defined.

struct tms {
        time_t tms_utime;  // CPU time used by the user.
        time_t tms_stime;  // CPU time used by the system (kernel).
        time_t tms_cutime; // User CPU time used by the terminated child process.
        time_t tms_cstime; // The system CPU time used by the terminated child.
};

extern time_t times(struct tms * tp);

#endif

```

14.38 types.h

14.38.1 Functionality

The types.h header file defines the basic data types. All types are defined as the appropriate mathematical type length. In addition, size_t is an unsigned integer type; off_t is an extended signed integer type; pid_t is a signed integer type.

14.38.2 Code annotation

Program 14-34 linux/include/sys/types.h

```
#ifndef _SYS_TYPES_H
#define _SYS_TYPES_H

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;        // Used for the size (length) of the object.
#endif

#ifndef _TIME_T
#define _TIME_T
typedef long time_t;                // Used for time (in seconds).
#endif

#ifndef _PTRDIFF_T
#define _PTRDIFF_T
typedef long ptrdiff_t;
#endif

#ifndef NULL
#define NULL ((void *) 0)
#endif

typedef int pid_t;                 // Used for process id and process group id.
typedef unsigned short uid_t;      // Used for the user id.
typedef unsigned char gid_t;       // Used for the group id.
typedef unsigned short dev_t;      // Used for the device number.
typedef unsigned short ino_t;      // Used for the inode number.
typedef unsigned short mode_t;     // Used for some file modes.
typedef unsigned short umode_t;    //
typedef unsigned char nlink_t;     // Used for the file links counting.
typedef int daddr_t;
typedef long off_t;                // Used for the offset in a file.
typedef unsigned char u_char;      // unsigned char.
typedef unsigned short ushort;     // unsigned short.

typedef unsigned char cc_t;
typedef unsigned int speed_t;
typedef unsigned long tcflag_t;

typedef unsigned long fd_set;     // File descriptor set. Each bit represents 1 descriptor.

typedef struct { int quot,rem; } div_t;     // Used for DIV operation.
typedef struct { long quot,rem; } ldiv_t;   // Used for long DIV operation.

// File system parameter structure for the ustat() function. The last two fields are unused
// and always return NULLs.
struct ustat {
        daddr_t f_tfree;                    // Total free blocks in the system.
        ino_t f_tinode;                     // Total free inodes.
        char f_fname[6];                    // File system name.
        char f_fpack[6];                    // The packed file system name.
};

#endif

```

14.39 utsname.h

14.39.1 Functionality

utsname.h is the system name structure header file. It defines the utsname structure and the function prototype uname().This function uses the information in the utsname structure to give information such as the system identifier, version number, and hardware type. In POSIX, the size of the character array should be unspecified, but the data stored in it must be NULL terminated. Therefore, the kernel's utsname structure definition does not meet POSIX requirements (the string array size is defined as 9). In addition, the name utsname is an abbreviation for Unix Timesharing System name.

14.39.2 Code annotation

Program 14-35 linux/include/sys/utsname.h

```
#ifndef _SYS_UTSNAME_H
#define _SYS_UTSNAME_H

#include <sys/types.h>      // The basic system data types are defined.
#include <sys/param.h>      // Some hardware-related parameter values are given.

```

```
struct utsname {
        char sysname[9];                // system name.
        char nodename[MAXHOSTNAMELEN+1];    // The node name in the network.
        char release[9];                // release level.
        char version[9];                // version.
        char machine[9];                // hardware type.
};

extern int uname(struct utsname * utsbuf);

#endif

```

14.40 wait.h

14.40.1 Functionality

This header file describes the information when the process is waiting, including some symbol constants and wait(), waitpid() function prototype declarations.

14.40.2 Code annotation

Program 14-36 linux/include/sys/wait.h

```
#ifndef _SYS_WAIT_H
#define _SYS_WAIT_H

#include <sys/types.h>

#define _LOW(v)         ( (v) & 0377)         // Get the low byte (in octal).
#define _HIGH(v)        ( ((v) >> 8) & 0377)  // Get the high byte.

/* options for waitpid, WUNTRACED not supported */
// [ Note: In fact, the 0.12 kernel already supports the WUNTRACED option.]
// The following constant symbols are options used in the function waitpid().
#define WNOHANG         1                   // Don't hang and return immediately.
#define WUNTRACED       2                   // Reports the child status that was stopped.

// The macros are used to check the meaning of the status word returned by the waitpid().
#define WIFEXITED(s)    (!((s)&0xFF)        // True if the child exits normally.
#define WIFSTOPPED(s)   (((s)&0xFF)==0x7F)  // True if the child is stopping.
#define WEXITSTATUS(s)  (((s)>>8)&0xFF)     // The exit status.
#define WTERMSIG(s)     ((s)&0x7F)          // Signal that caused process to terminate.
#define WCOREDUMP(s)    ((s)&0x80)          // Check if a core dump has been performed.
#define WSTOPSIG(s)     (((s)>>8)&0xFF)     // Signal that caused process to stop.
// True if the child process exited due to an uncaptured signal.
#define WIFSIGNALED(s)  (((unsigned int)(s)-1 & 0xFFFF) < 0xFF)

// The wait() and waitpit() functions allow a process to get state information for one of its
```

```
   // child processes. The various options of the function allow you to get the status information
   // of the child process that has been terminated or stopped. If there are status information
   // for two or more child processes, the order of the reports is not specified.
   // wait() will suspend the current process until one of its children exits (terminates), or
   // receives a signal requesting termination of the process, or needs to call a signal handler.
   // Waitpid() suspends the current process until the child specified by pid exits or receives
   // a signal requesting termination of the process, or a signal handler needs to be called.
   // If pid= -1, options=0, then waitpid() acts the same as the wait() function, otherwise its
   // behavior will vary with the pid and options parameters (see kernel/exit.c, 142).
   // The parameter 'pid' is the process id; '*stat_loc' is a pointer to the location of the status
   // information; 'options' is the wait option, see lines 10, 11 above.
21 pid_t wait(int *stat_loc);
22 pid_t waitpid(pid_t pid, int *stat_loc, int options);
23
24 #endif
25
```

14.41 Summary

This chapter describes all the header files used by the kernel. From the next chapter we will introduce the library file code used by the kernel. The code for these library files will be linked into the kernel code when the kernel is compiled.

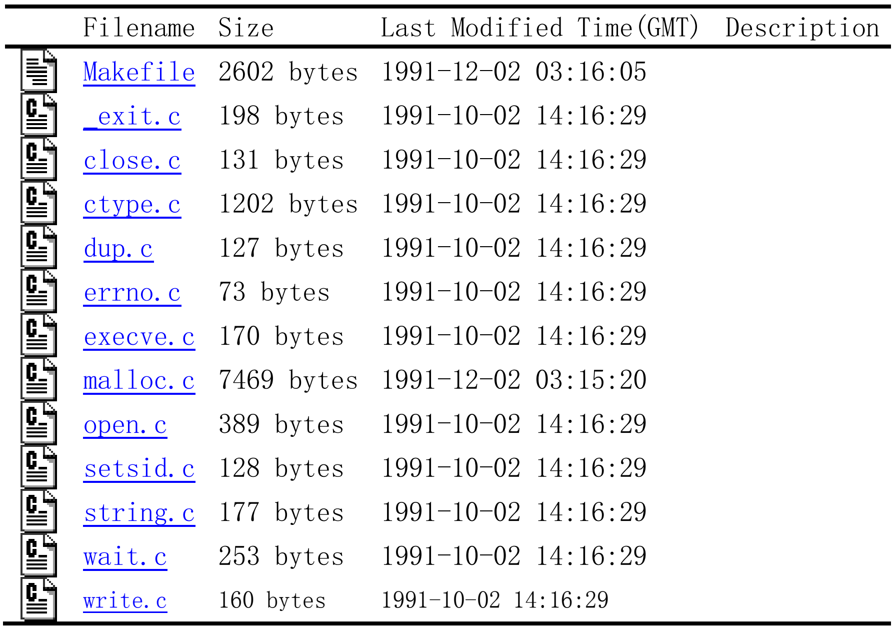
15 Library files (lib)

The C language library is a collection of reusable program modules, while the Linux kernel library files are a combination of some commonly used functions that are compiled for use by the kernel. The C files in Listing 15-1 are the programs that makes up the modules in the kernel library file. It mainly includes process execution and exit functions, file access operation functions, memory allocation functions, and string manipulation functions.

Specifically, the functions implemented are: exit function _exit(), close file function close(), copy file descriptor function dup(), file open function open(), write file function write(), execute program function execve( ), the memory allocation function malloc(), wait for the child process state function wait(), create the session system call setsid(), and all string manipulation functions implemented in include/string.h.

Except for a malloc.c program written by Mr. Tytso, the size of the program is very short, and some are only one or two lines of code. They basically invoke the system-calls directly to implement their functions.

List 15-1 Files in the /linux/lib/ directory



Filename	Size	Last Modified Time(GMT)	Description
Makefile	2602 bytes	1991-12-02 03:16:05	
_exit.c	198 bytes	1991-10-02 14:16:29	
close.c	131 bytes	1991-10-02 14:16:29	
ctype.c	1202 bytes	1991-10-02 14:16:29	
dup.c	127 bytes	1991-10-02 14:16:29	
errno.c	73 bytes	1991-10-02 14:16:29	
execve.c	170 bytes	1991-10-02 14:16:29	
malloc.c	7469 bytes	1991-12-02 03:15:20	
open.c	389 bytes	1991-10-02 14:16:29	
setsid.c	128 bytes	1991-10-02 14:16:29	
string.c	177 bytes	1991-10-02 14:16:29	
wait.c	253 bytes	1991-10-02 14:16:29	
write.c	160 bytes	1991-10-02 14:16:29	

In the kernel compilation phase, the relevant instructions in the kernel Makefile will compile these programs into .o modules, and then build them into a lib.a library and linked to the kernel module. Different from the various library files provided by the usual compilation environment (such as libc.a, libufc.a provided by gcc, etc.), the functions in this library are mainly used in the init/main.c program of the kernel initialization stage, for its execution of the init() function in the user mode. So the included functions are few and very simple. But it is implemented in exactly the same way as a general library.

Creating a function library usually uses the command ar (archive abbreviation). For example, to create a function library libmine.a with 3 modules a.o, b.o, and c.o, you need to execute the following command:

```
ar -rc libmine.a a.o b.o c.o d.o
```

To add function module dup.o to this library file, execute the following command:

```
ar -rs dup.o
```

15.1 _exit.c

15.1.1 Functionality

The _exit.c file is used by the program to call the kernel's exit system-call function.

15.1.2 Code annotation

Program 15-1 linux/lib/_exit.c

```
/*
 *  linux/lib/_exit.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <unistd.h> Linux standard header file. Various symbol constants and types are defined and
//     various functions are declared. If '__LIBRARY__' is defined, it also includes the
//     system-call number and the inline assembly _syscall0().
#define __LIBRARY__
#include <unistd.h>

//// The program exits (terminates) function.
// The library function directly calls the system interrupt int 0x80, function number __NR_exit.
// Parameters: exit_code - exit code.
// The keyword 'volatile' before the function name is used to tell the compiler gcc that the
// function will not return. This will allow gcc to produce better code and, more importantly,
// use this keyword to avoid some false warnings.
volatile void _exit(int exit_code)
{
// %0 - eax(__NR_exit); %1 - ebx(exit_code)。
        __asm__("int $0x80"::"a" (__NR_exit),"b" (exit_code));
}

```

15.1.3 Information

For a description of the system-call interrupt number, see the description in the include/unistd.h file.

15.2 close.c

15.2.1 Functionality

The file close function close() is defined in the close.c file.

15.2.2 Code annotation

Program 15-2 linux/lib/close.c

```
/*
 *  linux/lib/close.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <unistd.h> Linux standard header file. Various symbol constants and types are defined and
//     various functions are declared. If ’__LIBRARY__’ is defined, it also includes the
//     system-call number and the inline assembly _syscall0().
#define __LIBRARY__
#include <unistd.h>

// Close the file function.
// The following macro corresponds to function prototype: int close(int fd). It directly calls
// the system int 0x80, with the parameter __NR_close. Where fd is the file descriptor.
_syscall1(int,close,int,fd)

```

15.3 ctype.c

15.3.1 Functionality

The ctype.c program is used to provide auxiliary array structure data for ctype.h for type determination of characters.

15.3.2 Code annotation

Program 15-3 linux/lib/ctype.c

```
/*
 *  linux/lib/ctype.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <ctype.h> The character type file. Defines some macros for character type conversion.
#include <ctype.h>

char _ctmp;      // a tem variable for macros that convert characters in the ctype.h file.
```

```
// The following is an array of character attributes that define the attributes corresponding
// to each character. These attribute types (such as _C, etc.) are defined in ctype.h. It is
// used to check whether the character is a control character (_C), uppercase character (_U),
// lowercase character (_L), etc.
unsigned char _ctype[] = {0x00,                         /* EOF */
_C, _C, _C, _C, _C, _C, _C, _C,                         /* 0-7 */
_C, _C|_S, _C|_S, _C|_S, _C|_S, _C|_S, _C, _C,          /* 8-15 */
_C, _C, _C, _C, _C, _C, _C, _C,                         /* 16-23 */
_C, _C, _C, _C, _C, _C, _C, _C,                         /* 24-31 */
_S|_SP, _P, _P, _P, _P, _P, _P, _P,                     /* 32-39 */
_P, _P, _P, _P, _P, _P, _P, _P,                         /* 40-47 */
_D, _D, _D, _D, _D, _D, _D, _D,                         /* 48-55 */
_D, _D, _P, _P, _P, _P, _P, _P,                         /* 56-63 */
_P, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, _U,       /* 64-71 */
_U, _U, _U, _U, _U, _U, _U, _U,                         /* 72-79 */
_U, _U, _U, _U, _U, _U, _U, _U,                         /* 80-87 */
_U, _U, _U, _P, _P, _P, _P, _P,                         /* 88-95 */
_P, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, _L,       /* 96-103 */
_L, _L, _L, _L, _L, _L, _L, _L,                         /* 104-111 */
_L, _L, _L, _L, _L, _L, _L, _L,                         /* 112-119 */
_L, _L, _L, _P, _P, _P, _P, _C,                         /* 120-127 */
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         /* 128-143 */
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         /* 144-159 */
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         /* 160-175 */
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         /* 176-191 */
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         /* 192-207 */
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         /* 208-223 */
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,         /* 224-239 */
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};        /* 240-255 */


```

15.4 dup.c

15.4.1 Functionality

The dup.c program includes a function dup() that creates a copy of the file descriptor. After a successful return, the new and original descriptors can be used interchangeably. They share locks, file read and write pointers, and file flags. For example, if the file read/write position pointer is modified by one of the descriptors using lseek(), the file read/write pointer is also changed for the other descriptor. This function uses the smallest unused descriptor to create a new descriptor, but the two descriptors do not share the close-on-exec flag.

15.4.2 Code annotation

Program 15-4 linux/lib/dup.c

```
/*
 *  linux/lib/dup.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <unistd.h> Linux standard header file. Various symbol constants and types are defined and
//     various functions are declared. If '__LIBRARY__' is defined, it also includes the
//     system-call number and the inline assembly _syscall0().
#define __LIBRARY__
#include <unistd.h>

//// Duplicate file descriptor (handle) function.
// The macro below corresponds to the function prototype: int dup(int fd). It directly calls
// the system int 0x80, the parameter is __NR_dup. Where fd is the file descriptor.
_syscall1(int,dup,int,fd)

```

15.5 errno.c

15.5.1 Functionality

The program only defines an variable errno to store the error number when the function call fails. Please refer to the description in the include/errno.h file.

15.5.2 Code annotation

Program 15-5 linux/lib/errno.c

```
/*
 *  linux/lib/errno.c
 *
 *  (C) 1991  Linus Torvalds
 */

int errno;

```

15.6 execve.c

15.6.1 Functionality

The execve.c program contains a system-call function that runs the executables.

15.6.2 Code annotation

Program 15-6 linux/lib/execve.c

```
/*
```

```
 *  linux/lib/execve.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <unistd.h> Linux standard header file. Various symbol constants and types are defined and
//     various functions are declared. If '__LIBRARY__' is defined, it also includes the
//     system-call number and the inline assembly _syscall0().
#define __LIBRARY__
#include <unistd.h>

//// Load and execute the child process (other programs) function.
// The macro corresponds to function: int execve(const char * file, char ** argv, char ** envp).
// Parameters: file - the executable file name; argv - an array of command line argument pointers;
// envp - an array of environment variable pointers. It directly calls the system int 0x80,
// the parameter is __NR_execve. See include/unistd.h and fs/exec.c.
_syscall3(int,execve,const char *,file,char **,argv,char **,envp)

```

15.7 malloc.c

15.7.1 Functionality

The malloc.c program mainly includes the memory allocation function malloc().In order not to be confused with the malloc() function used by the user program, it is called kmalloc() from the kernel version 0.98, and the free_s() function is renamed to kfree_s().

Note that the memory allocation functions of the same name used by the application are generally implemented in the library file of the development environment, such as the libc.a library in the GCC environment. Since the library functions in the development environment are themselves linked to the user program, they cannot directly use the functions such as get_free_page() in the kernel to implement the memory allocation function. Of course, they also do not need to directly manage the memory page, because the memory allocation function in the library libc.a only needs to dynamically adjust the set value at the end of the process data segment according to the program request, and does not cover the end stack and the environment parameter area. The rest of the specific memory mapping and other operations are done by the kernel. This operation of adjusting the end position of the process data segment is the main purpose of the memory allocation function in the library, and the kernel system-call brk() is called, see line 228 of the kernel/sys.c program. So if you can view the source code of the library function implementation in the development environment, you will find that the memory allocation functions such as malloc() and calloc() only call the kernel system-call brk() in addition to managing the dynamic application memory area. The memory allocation functions in the development environment library are identical to the functions here only in the aspects that they all require dynamic management of allocated memory. The management methods they use are basically the same.

The malloc() function uses the bucket principle to manage the allocated memory. The basic idea is to use the bucket directory (hereinafter referred to as the directory) for the different memory block sizes requested. For example, if the size of the request memory block is 32 bytes or less but more than 16 bytes, the memory block is allocated using the bucket descriptor list corresponding to the second item in the bucket directory. The basic structure is shown in Figure 15-1. The maximum memory size that the function can allocate at a time is one memory page, which is 4096 bytes.

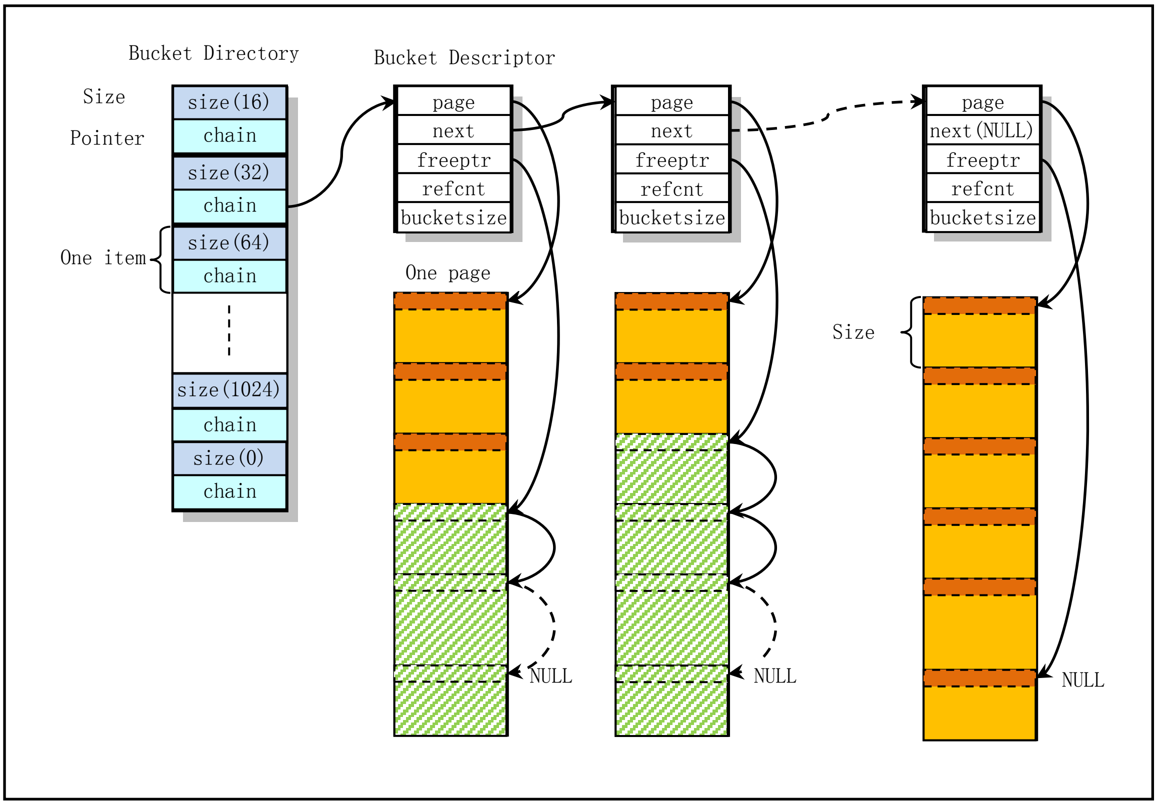


Figure 15-1 Use the bucket principle for memory allocation management

The first time you call the malloc() function, you first need to create a free bucket descriptor list for the page, which holds the descriptors that have not been used or have been reclaimed. The structure of the linked list is shown in Figure 15-2, where free_bucket_desc is the pointer to the linked list header. Extracting/putting a descriptor from/into the linked list starts from the beginning of the linked list. When a descriptor is fetched, the first descriptor pointed to by the header pointer is fetched; when an idle descriptor is released, it is also placed at the head of the list.

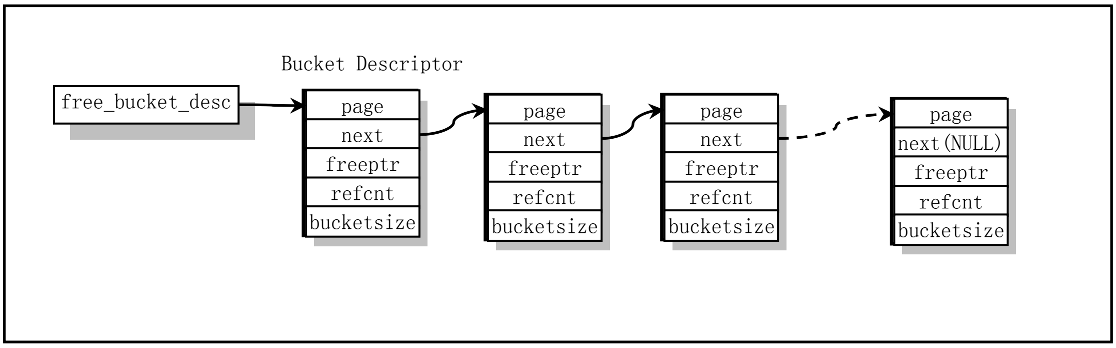


Figure 15-2 Free bucket descriptor chain table structure

During the running of the system, if all bucket descriptors are occupied at a certain time, then free_bucket_desc will be NULL. Therefore, without the bucket descriptor being released, the next time you need to use the free bucket descriptor, the program will apply for a page again and create a new free bucket descriptor list on it as shown in the figure.

The basic steps of the malloc() function are as follows:

1. First search the directory and look for the descriptor list corresponding to the directory entry that matches the size of the requested memory block. When the memory size of the directory entry is larger than the requested byte size, the corresponding directory entry is found. If the search for the entire directory does not find a suitable directory entry, then the memory block requested by the user is too large.
2. Find a descriptor with free space in the descriptor list corresponding to the directory entry. If the free memory pointer freeptr of a descriptor is not NULL, it means that the corresponding descriptor is found. If we don't find a descriptor with free space, then we need to create a new descriptor. The steps to create a new descriptor are as follows:
   a. If the idle descriptor list header pointer is still NULL, it means that the malloc() function is called for the first time, or all empty bucket descriptors are used up. In this case, you need to use the function init_bucket_desc() to create a list of idle descriptors.
   b. Then get a descriptor from the header of the idle descriptor chain, initialize the descriptor, make its object reference count 0, the object size is equal that of the directory entry, and apply for a memory page, let the descriptor page pointer points to the memory page, and the free memory pointer of the descriptor also points to the beginning of the page.
   c. c. Initialize the page for the memory page according to the size of the object used in this directory entry, and establish a linked list of all objects. That is, each object's head stores a pointer to the next object, and the last object stores a NULL at the beginning.
   d. Then insert the descriptor into the beginning of the descriptor list for the corresponding directory entry.
3. Copy the free memory pointer freeptr of the descriptor to the memory pointer returned to the user, and then adjust the freeptr to point to the next free object location in the memory page corresponding to the descriptor, and increment the descriptor reference count by one.

The free_s() function is used to reclaim the memory blocks released by the user. The basic method is to first convert the address of the corresponding page according to the address of the memory block, and then search all the descriptors in the directory to find the descriptor corresponding to the page. The released memory block is chained into the free object list pointed to by the freeptr, and the descriptor's object reference count value is decremented by one. If the reference count value is equal to zero at this time, it means that the page corresponding to the descriptor is completely free, and the memory page can be released and the descriptor is retracted into the idle descriptor list.

15.7.2 **Code annotation**

Program 15-7 linux/lib/malloc.c

```
/*
 * malloc.c --- a general purpose kernel memory allocator for Linux.
 *
 * Written by Theodore Ts'o (tytso@mit.edu), 11/29/91
 *
 * This routine is written to be as fast as possible, so that it
 * can be called from the interrupt level.
 *
 * Limitations: maximum size of memory we can allocate using this routine
 *      is 4k, the size of a page in Linux.
 *
 * The general game plan is that each page (called a bucket) will only hold
 * objects of a given size.  When all of the object on a page are released,
 * the page can be returned to the general free pool.  When malloc() is
 * called, it looks for the smallest bucket size which will fulfill its
 * request, and allocate a piece of memory from that bucket pool.
 *
 * Each bucket has as its control block a bucket descriptor which keeps
 * track of how many objects are in use on that page, and the free list
 * for that page.  Like the buckets themselves, bucket descriptors are
 * stored on pages requested from get_free_page().  However, unlike buckets,
 * pages devoted to bucket descriptor pages are never released back to the
 * system.  Fortunately, a system should probably only need 1 or 2 bucket
 * descriptor pages, since a page can hold 256 bucket descriptors (which
 * corresponds to 1 megabyte worth of bucket pages.)  If the kernel is using
 * that much allocated memory, it's probably doing something wrong.  :-)
 *
 * Note: malloc() and free() both call get_free_page() and free_page()
 *      in sections of code where interrupts are turned off, to allow
 *      malloc() and free() to be safely called from an interrupt routine.
 *      (We will probably need this functionality when networking code,
 *      particularily things like NFS, is added to Linux.)  However, this
 *      presumes that get_free_page() and free_page() are interrupt-level
 *      safe, which they may not be once paging is added.  If this is the
 *      case, we will need to modify malloc() to keep a few unused pages
 *      "pre-allocated" so that it can safely draw upon those pages if
 *      it is called from an interrupt routine.
 *
 *      Another concern is that get_free_page() should not sleep; if it
 *      does, the code is carefully ordered so as to avoid any race
 *      conditions.  The catch is that if malloc() is called re-entrantly,
 *      there is a chance that unecessary pages will be grabbed from the
 *      system.  Except for the pages for the bucket descriptor page, the
 *      extra pages will eventually get released back to the system, though,
 *      so it isn't all that bad.
 */

// <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
//     used functions of the kernel.
// <linux/mm.h> Memory management header file. Contains page size definitions and some page
//     release function prototypes.
// <asm/system.h> System header file. An embedded assembly macro that defines or modifies
//     descriptors/interrupt gates, etc. is defined.
#include <linux/kernel.h>
#include <linux/mm.h>
#include <asm/system.h>

 // Bucket descriptor structure.
struct bucket_desc {    /* 16 bytes */
```

```
        void                    *page;          // Memory page pointer.
        struct bucket_desc      *next;          // The next descriptor pointer.
        void                    *freeptr;       // A pointer to the free memory.
        unsigned short          refcnt;         // Refernce count.
        unsigned short          bucket_size;    // The size of the bucket.
};

// Bucket descriptor directory structure.
struct _bucket_dir {     /* 8 bytes */
        int                     size;           // The size (in bytes) of this bucket.
        struct bucket_desc      *chain;         // Bucket descriptor list pointer.
};

/*
 * The following is the where we store a pointer to the first bucket
 * descriptor for a given size.
 *
 * If it turns out that the Linux kernel allocates a lot of objects of a
 * specific size, then we may want to add that specific size to this list,
 * since that will allow the memory to be allocated more efficiently.
 * However, since an entire page must be dedicated to each specific size
 * on this list, some amount of temperance must be exercised here.
 *
 * Note that this list *must* be kept in order.
 */
// Bucket directory list.
struct _bucket_dir bucket_dir[] = {
        { 16,   (struct bucket_desc *) 0},      // A 16-byte memory block.
        { 32,   (struct bucket_desc *) 0},      // A 32-byte memory block.
        { 64,   (struct bucket_desc *) 0},
        { 128,  (struct bucket_desc *) 0},
        { 256,  (struct bucket_desc *) 0},
        { 512,  (struct bucket_desc *) 0},
        { 1024, (struct bucket_desc *) 0},
        { 2048, (struct bucket_desc *) 0},
        { 4096, (struct bucket_desc *) 0},      // A 4096-byte memory block.
        { 0,    (struct bucket_desc *) 0}};     /* End of list marker */

/*
 * This contains a linked list of free bucket descriptor blocks
 */
struct bucket_desc *free_bucket_desc = (struct bucket_desc *) 0;

/*
 * This routine initializes a bucket description page.
 */
//// Initialize the bucket descriptor.
// Create a free bucket descriptor list and let 'free_bucket_desc' point to the first free bucket
// descriptor.
static inline void init_bucket_desc()
{
        struct bucket_desc *bdesc, *first;
        int     i;
```

```

// First apply for a page of memory for storing the bucket descriptor. Then calculate the number
// of bucket descriptors that can be stored in a page of memory, and then establish a one-way
// link pointer to it.
        first = bdesc = (struct bucket_desc *) get_free_page();
        if (!bdesc)
                panic("Out of memory in init_bucket_desc()");
        for (i = PAGE_SIZE/sizeof(struct bucket_desc); i > 1; i--) {
                bdesc->next = bdesc+1;
                bdesc++;
        }
        /*
         * This is done last, to avoid race conditions in case
         * get_free_page() sleeps and this routine gets called again....
         */
// Add the free bucket descriptor pointer 'first' to the beginning of the list.
        bdesc->next = free_bucket_desc;
        free_bucket_desc = first;
}

//// Memory allocation function.
// Parameters: len - the size of the requested memory block.
// Returns: a pointer to the allocated memory. Returns NULL if it fails.
void *malloc(unsigned int len)
{
        struct _bucket_dir      *bdir;
        struct bucket_desc      *bdesc;
        void                    *retval;

        /*
         * First we search the bucket_dir to find the right bucket change
         * for this request.
         */
// Search the bucket directory for a bucket descriptor list that is suitable for applying the
// size of the memory block. If the bucket size of the directory entry is greater than the
// requested number of bytes, the corresponding bucket directory entry is found.
        for (bdir = bucket_dir; bdir->size; bdir++)
                if (bdir->size >= len)
                        break;
// If the search for the entire directory does not find a directory entry of the appropriate
// size, it indicates that the requested memory block size is too large, exceeding the allocation
// limit of the program (up to 1 page). Then the error message is displayed and the crash occurs.
        if (!bdir->size) {
                printk("malloc called with impossibly large argument (%d)\n",
                        len);
                panic("malloc: bad arg");
        }
        /*
         * Now we search for a bucket descriptor which has free space
         */
        cli();       /* Avoid race conditions */
// Search the descriptor list in the corresponding bucket directory entry to find the bucket
// descriptor with free space. If the free memory pointer 'freeptr' of the bucket descriptor
// is not empty, it indicates that the corresponding bucket descriptor was found.
        for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next)
                if (bdesc->freeptr)
                        break;
        /*
         * If we didn't find a bucket with free space, then we'll
         * allocate a new one.
         */
        if (!bdesc) {
                char            *cp;
                int             i;

// If free_bucket_desc is still empty, it means that the function is called for the first time,
// or all empty bucket descriptors in the linked list are used up. At this point you need to
// apply for a page and build and initialize an idle descriptor list on it. free_bucket_desc
// will point to the first free bucket descriptor.
                if (!free_bucket_desc)
                        init_bucket_desc();

// Take the free bucket descriptor pointed to by free_bucket_desc and let free_bucket_desc point
// to the next free bucket descriptor. Then initialize the new bucket descriptor: make the
// references count equal to 0; the bucket size is equal to the size of the corresponding bucket
// directory; apply a memory page, let the page pointer point to the page; the free memory pointer
// also points to the beginning of the page, because it is all idle at this time.
                bdesc = free_bucket_desc;
                free_bucket_desc = bdesc->next;
                bdesc->refcnt = 0;
                bdesc->bucket_size = bdir->size;
                bdesc->page = bdesc->freeptr = (void *) cp = get_free_page();
// If the application for memory page operation fails, an error occurs and the machine crashes.
// Otherwise, the page size is divided by the bucket size specified by the bucket directory
// entry, and the first 4 bytes of each object are set to pointers to the next object. The pointer
// of the last object is set to 0 (NULL).
                if (!cp)
                        panic("Out of memory in kernel malloc()");
                /* Set up the chain of free objects */
                for (i=PAGE_SIZE/bdir->size; i > 1; i--) {
                        *((char **) cp) = cp + bdir->size;
                        cp += bdir->size;
                }
                *((char **) cp) = 0;
// Then, the next descriptor pointer field of the bucket descriptor is pointed to the descriptor
// pointed to by the bucket directory entry pointer originally, and the bucket directory's chain
// points to the bucket descriptor, that is, the descriptor is inserted into the descriptor
// chain header.
                bdesc->next = bdir->chain;     /* OK, link it in! */
                bdir->chain = bdesc;
        }
// The return pointer is equal to the current free pointer of the page. The free space pointer
// is then adjusted to point to the next free object, and the object reference count in the
// corresponding page in the descriptor is incremented by one. Finally enable the interrupt
// and return the pointer to the free memory object.
        retval = (void *) bdesc->freeptr;
        bdesc->freeptr = *((void **) retval);
        bdesc->refcnt++;
        sti();      /* OK, we're safe again */
        return(retval);
}

/*
 * Here is the free routine.  If you know the size of the object that you
 * are freeing, then free_s() will use that information to speed up the
 * search for the bucket descriptor.
 *
 * We will #define a macro so that "free(x)" is becomes "free_s(x, 0)"
 */
//// Release the bucket object.
// Parameters: obj - the corresponding object pointer; size - the object size.
void free_s(void *obj, int size)
{
        void            *page;
        struct _bucket_dir      *bdir;
        struct bucket_desc      *bdesc, *prev;

        /* Calculate what page this object lives in */
        page = (void *)  ((unsigned long) obj & 0xfffff000);
        /* Now search the buckets looking for that page */
        for (bdir = bucket_dir; bdir->size; bdir++) {
                prev = 0;
                /* If size is zero then this conditional is always false */
                if (bdir->size < size)
                        continue;
// Search all the descriptors in the directory entry to find the corresponding page. If a
// descriptor page pointer is equal to 'page', it means that the corresponding descriptor is
// found, so it jumps to label 'found'. If the descriptor does not contain a corresponding page,
// then the descriptor pointer 'prev' is pointed to the descriptor.
// If all the descriptors searching for the corresponding directory entry do not find the
// specified page, an error message is displayed and the computer crashes.
                for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next) {
                        if (bdesc->page == page)
                                goto found;
                        prev = bdesc;
                }
        }
        panic("Bad address passed to kernel free_s()");
found:
// After finding the corresponding bucket descriptor, first turn off the interrupt. The object
// memory block is then chained into the free block object list and the object reference count
// for that descriptor is decremented by one.
        cli();   /* To avoid race conditions */
        *((void **)obj) = bdesc->freeptr;
        bdesc->freeptr = obj;
        bdesc->refcnt--;

// If the reference count is equal to 0, then we can release the corresponding memory page and
// the bucket descriptor.
```

```
        if (bdesc->refcnt == 0) {
                /*
                 * We need to make sure that prev is still accurate.  It
                 * may not be, if someone rudely interrupted us....
                 */
// If 'prev' is not the previous descriptor of the searched descriptor, then the previous
// descriptor of the current descriptor is re-searched.
                if ((prev && (prev->next != bdesc)) ||
                    (!prev && (bdir->chain != bdesc)))
                        for (prev = bdir->chain; prev; prev = prev->next)
                                if (prev->next == bdesc)
                                        break;
// If the previous descriptor is found, the current descriptor is removed from the descriptor
// chain. If prev==NULL, it means that the current descriptor is the first descriptor of the
// directory entry, that is, the chain in the directory entry should directly point to the current
// descriptor 'bdesc', otherwise it indicates that there is a problem with the linked list.
// Therefore, in order to remove the current descriptor from the linked list, you should have
// 'chain' pointing to the next descriptor.
                if (prev)
                        prev->next = bdesc->next;
                else {
                        if (bdir->chain != bdesc)
                                panic("malloc bucket chains corrupted");
                        bdir->chain = bdesc->next;
                }
// Finally, the memory page operated by the current descriptor is released, and the descriptor
// is inserted at the beginning of the idle descriptor list.
                free_page((unsigned long) bdesc->page);
                bdesc->next = free_bucket_desc;
                free_bucket_desc = bdesc;
        }
        sti();          // Enable interrupt and return.
        return;
}
```

15.8 open.c

15.8.1 Functionality

The open() library function in the open.c file is used to open a file with the specified file name. When open() is called successfully, the file descriptor of the file is returned. This call creates a new open file and is not shared with any other process. When the exec() function is executed, the new file descriptor will remain open at all times. The file's read and write pointer is set at the beginning of the file.

The function parameter 'flag' can be one of O_RDONLY, O_WRONLY, and O_RDWR, which means that the file is read-only open, write-only open, and read-write open, respectively, and can be used with some other flags. See also the implementation of the sys_open() function in the fs/open.c program (line 171).

15.8.2 **Code annotation**

Program 15-8 linux/lib/open.c

```
/*
 *  linux/lib/open.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <unistd.h> Linux standard header file. Various symbol constants and types are defined and
//     various functions are declared. If '__LIBRARY__' is defined, it also includes the
//     system-call number and the inline assembly _syscall0().
// <stdarg.h> Standard parameter header file. Define a list of variable parameters in the form
//     of macros. It mainly describes one type (va_list) and three macros (va_start, va_arg and
//     va_end) for the vsprintf, vprintf, and vfprintf functions.
#define __LIBRARY__
#include <unistd.h>
#include <stdarg.h>

//// Open file library function.
// Open a file or create a file when it does not exist.
// Parameters: filename - file name; flag - file open flag; ...
// Returns the file descriptor. If an error occurs, the error code is set and -1 is returned.
int open(const char * filename, int flag, ...)
{
        register int res;
        va_list arg;

// Use the va_start() macro function to get the pointer of the parameter after the flag. Then
// call the system interrupt int 0x80 with the function number __NR_open to open the file.
// %0 - eax (descriptor returned or error code); %1 - eax (system-call function __NR_open);
// %2 - ebx (filename); %3 - ecx (open file flag); % 4 - edx (following the file mode).
        va_start(arg,flag);
        __asm__("int $0x80"
                :"=a" (res)
                :"" (__NR_open),"b" (filename),"c" (flag),
                "d" (va_arg(arg,int)));
// If the system interrupt call returns a value greater than or equal to 0, indicating that
// it is a file descriptor, it will return directly. Otherwise, the return value is less than
// 0, then it is an error code. So set the error code and return -1.
        if (res>=0)
                return res;
        errno = -res;
        return -1;
}

```

15.9 setsid.c

15.9.1 Functionality

The setsid.c program includes a setsid() system call function. This function is used to create a new session if the calling process is not a leader of a group. The calling process will become the leader of the new session, the group leader of the new process group, and there is no controlling terminal. The group ID and session ID of the calling process are set to the PID of the process. The calling process will become the only process in the new process group and the new session.

15.9.2 Code annotation

Program 15-9 linux/lib/setsid.c

```
/*
 *  linux/lib/setsid.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <unistd.h> Linux standard header file. Various symbol constants and types are defined and
//     various functions are declared. If '__LIBRARY__' is defined, it also includes the
//     system-call number and the inline assembly _syscall0().
#define __LIBRARY__
#include <unistd.h>

//// Create a session and set the process group number.
// The following system-call macro corresponds to the function: pid_t setsid().
// Returns the session identifier (session ID) of the calling process.
_syscall0(pid_t,setsid)

```

15.10 string.c

15.10.1 Functionality

All string manipulation functions already exist in the string.h header file, but appear as inline code. Here, we give the implementation code that contains the string function in string.c by first declaring the 'extern' and 'inline' prefixes to be empty, and then including the string.h header file. See the instructions before the include/string.h header file.

15.10.2 Code annotation

Program 15-10 linux/lib/string.c

```
/*
 *  linux/lib/string.c
 *
 *  (C) 1991  Linus Torvalds
 */

#ifndef __GNUC__
#error I want gcc!
#endif

#define extern
#define inline
#define __LIBRARY__
#include <string.h>

```

15.11 wait.c

15.11.1 Functionality

The wait.c program includes the functions waitpid() and wait(). These two functions allow the process to get state information for one of its child processes. Various options allow you to get child process status information that has been terminated or stopped. If there are status information for two or more child processes, the order of the reports is not specified.

wait() will suspend the current process until one of its child processes exits (terminates), or receives a signal requesting termination of the process, or a signal handler needs to be called.

waitpid() suspends the current process until the child process specified by pid exits (terminates) or receives a signal requesting termination of the process, or a signal handler needs to be called.

If pid= -1, options=0, waitpid() acts the same as the wait() function, otherwise its behavior will vary depending on the pid and options parameters (see kernel/exit.c, 370).

15.11.2 Code annotation

Program 15-11 linux/lib/wait.c

```
/*
 *  linux/lib/wait.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <unistd.h> Linux standard header file. Various symbol constants and types are defined and
//     various functions are declared. If '__LIBRARY__' is defined, it also includes the
//     system-call number and the inline assembly _syscall0().
// <sys/wait.h> wait header file. Define the system call wait() core waitpid() and related
//     constant symbols.
#define __LIBRARY__
#include <unistd.h>
#include <sys/wait.h>
```

```

//// Wait for the process to terminate.
// The macro corresponds to the function: pid_t waitpid(pid_t pid, int * wait_stat, int options)
// Parameters: pid - the process id of the process waiting to be terminated, or other specific
// value used to specify a special case; wait_stat - used to store state information;
// options - WNOHANG or WUNTRACED or 0.
_syscall3(pid_t,waitpid,pid_t,pid,int *,wait_stat,int,options)

//// wait() system-call. It directly calls the waitpid() function.
pid_t wait(int * wait_stat)
{
        return waitpid(-1,wait_stat,0);
}

```

15.12 write.c

15.12.1 Functionality

The write.c program includes a write function write() to the file descriptor. This function writes the data of the 'count' byte to the file 'buf' of the file specified by the file descriptor.

15.12.2 Code annotation

Program 15-12 linux/lib/write.c

```
/*
 *  linux/lib/write.c
 *
 *  (C) 1991  Linus Torvalds
 */

// <unistd.h> Linux standard header file. Various symbol constants and types are defined and
//     various functions are declared. If '__LIBRARY__' is defined, it also includes the
//     system-call number and the inline assembly _syscall0().
#define __LIBRARY__
#include <unistd.h>

//// Write the file.
// The macro corresponds to the function: int write(int fd, const char * buf, off_t count)
// Parameters: fd - file descriptor; buf - write buffer pointer; count - number of write bytes.
// Returns: the number of bytes written back on success (0 means write 0 bytes); -1 will be
// returned on error and the error code is set.
_syscall3(int,write,int,fd,const char *,buf,off_t,count)

```

15.13 Summary

This chapter describes several library function files used by several tasks that the kernel runs in user mode at initialization time. These library functions are implemented in exactly the same way as the generic library functions used in the development environment.

In the next chapter, we explain a tool, tools/build.c, contained in the kernel code tree that is used to combine all kernel modules to generate a kernel image file.

16 Building Kernel (tools)

The 'tools' directory in the Linux kernel source code contains a utility program build.c that generates kernel disk image files. This program is compiled separately into an executable file and will be called in the Makefile to be used to connect and merge all kernel compiled modules into a working image file Image. According to the contents of the Makefile, the Make program will first compile boot/bootsect.s and boot/setup.s using the 8086 assembler to generate two object module files 'bootsect' and 'setup' in MINIX format; then compile with GNU. Gcc/gas, compiles and links all other programs in the source code to generate the object module 'system' in a.out format. Finally, use the build tool to remove the extra header data from the three modules and stitch them together into a kernel image file 'Image'. The basic compilation linking/combination structure is shown in Figure 16-1.

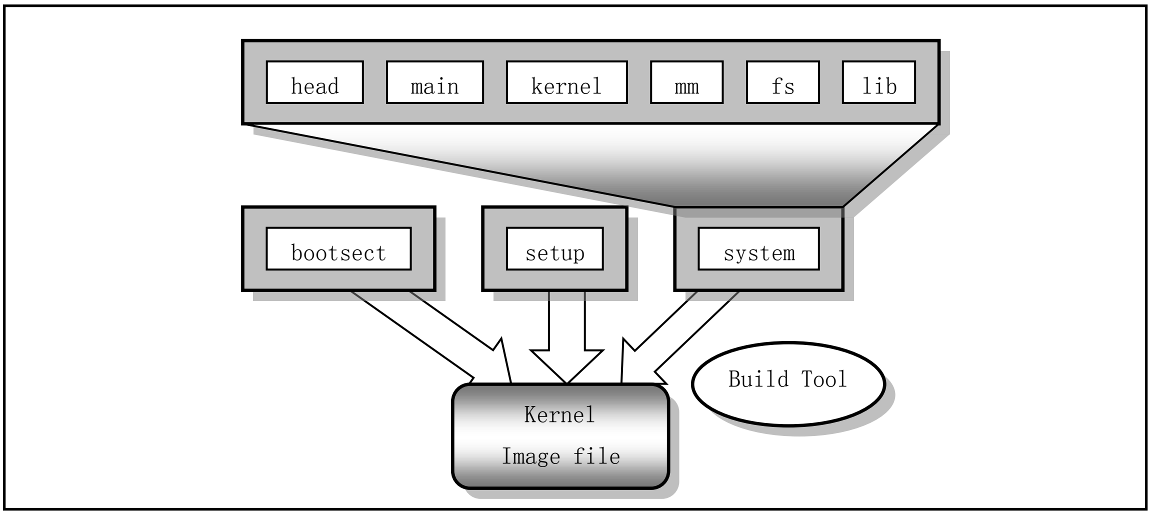


Figure 16-1 Kernel compilation linking/combination structure

16.1 build.c

16.1.1 Functionality

On the linux/Makefile line 42--44, the command line form for executing the build program is as follows:

```
    tools/build  boot/bootsect boot/setup tools/system $(ROOT_DEV) $(SWAP_DEV) > Image
```

The build program uses five parameters, namely the bootsect, setup, system, the optional root fs device name ROOT_DEV, and the optional swap device SWAP_DEV. The bootsect and setup modules are compiled by as86, they have the MINIX executable file format (see the description of the program list), and the system module is linked by modules compiled from other source code, with GNU a.out executable file format. The main job of the build program is to remove the MINIX execution file header information of bootsect and setup, remove the a.out header information in the system module, retain only their code and data parts, and then combine them in order, and write them sequentially into a file named ‘Image’.

The program first checks the last optional parameter on the command line: root device file name. If it exists, the status information structure (stat) of the device file is read, and the device number is extracted. If this parameter is not present on the command line, the default value is used. Then the bootsect file is processed, the minix execution header information of the file is read, the validity is checked, and then the subsequent 512-byte boot code is read to determine whether it has the bootable flag 0xAA55, and then the root device number acquired earlier is written to the 508, 509 offset, and finally the 512-byte code data is written to the stdout standard output, which is redirected to the Image file by the Make file. The next step is to process the setup file in a similar way. If the file is less than 4 sectors in length, it is filled with 0 to the length of 4 sectors and written to the standard output stdout.

Finally process the system file. This file is generated using the GCC compiler, so its execution header format is GCC type, the same as the a.out format defined by linux. After confirming that the execution entry point is 0, the data is written to the standard output stdout. If the code and data size exceeds 128 KB, an error message is displayed. The resulting kernel Image file format is shown in Figure 16-2, where the format is:

- The first sector stores the bootsect code, which is exactly 512 bytes long;
- The 4 sectors (2 - 5 sectors) starting from the 2nd sector store the setup code, and the size is no more than 4 sectors;
- The system module is stored starting from the sixth sector, and its length does not exceed the size (128KB) defined on line 37 of build.c.

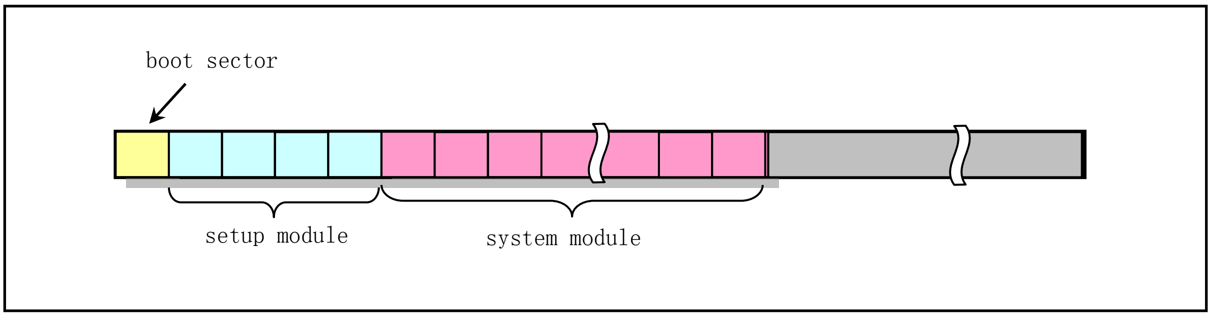


Figure 16-2 The format of Linux kernel on disk

16.1.2 **Code annotation**

Program 16-1 linux/tools/build.c

```
1 /*
2  *  linux/tools/build.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 /*
8  * This file builds a disk-image from three different files:
9  *
10  * - bootsect: max 510 bytes of 8086 machine code, loads the rest
11  * - setup: max 4 sectors of 8086 machine code, sets up system parm
```

```
 * - system: 80386 code for actual system
 *
 * It does some checking that all files are of the correct type, and
 * just writes the result to stdout, removing headers and padding to
 * the right amount. It also writes some system data to stderr.
 */

/*
 * Changes by tytso to allow root device specification
 *
 * Added swap-device specification: Linux 20.12.91
 */

#include <stdio.h>      /* fprintf */
#include <string.h>
#include <stdlib.h>     /* contains exit */
#include <sys/types.h>  /* unistd.h needs this */
#include <sys/stat.h>   // file state information structure
#include <linux/fs.h>
#include <unistd.h>     /* contains read/write */
#include <fcntl.h>      // file operation mode constants

#define MINIX_HEADER 32          // The MINIX object file header size is 32 bytes.
#define GCC_HEADER 1024          // The GCC header information size is 1024 bytes.

#define SYS_SIZE 0x3000          // The max size of the system module (SYS_SIZE*16=128KB).

// By default, the Linux root fs device is the first partition of the second hard disk (device
// number 0x0306). This is because when Mr. Linus developed Linux, he used the first hard disk
// as the MINIX system disk and the second hard disk as the Linux root file system disk.
#define DEFAULT_MAJOR_ROOT 3     // major device number - 3, hard disk.
#define DEFAULT_MINOR_ROOT 6     // minor device number - 6, 1st partition of the 2nd disk.

#define DEFAULT_MAJOR_SWAP 0     // swap device number.
#define DEFAULT_MINOR_SWAP 0

/* max nr of sectors of setup: don't change unless you also change
 * bootsect etc */
#define SETUP_SECTS 4            // The maximum size of setup is 4 sectors (2KB).

#define STRINGIFY(x) #x          // Convert x to string type for use in an error display.

//// Display an error message, and terminate the program.
void die(char * str)
{
        fprintf(stderr,"%s\n",str);
        exit(1);
}

//// Display the program usage and exit.
void usage(void)
{
        die("Usage: build bootsect setup system [rootdev] [> image]");
```

```
}

//// The main program begins...
// The program first checks the parameters on the command line for compliance, sets the root
// device number and the swap device number, then reads and processes the bootsect, setup, and
// system module files, respectively, and writes them to the standard output that has been
// redirected to the Image file. .
int main(int argc, char ** argv)
{
        int i,c,id;
        char buf[1024];
        char major_root, minor_root;
        char major_swap, minor_swap;
        struct stat sb;

// (1) First check the actual command line parameters when the build program is executed, and
// set accordingly according to the number of parameters. The build program requires 4 to 6
// parameters. If the number of parameters on the command line does not meet the requirements,
// the program usage is displayed and exited.
// If there are more than 4 parameters on the program command line, if the root device name
// is not "FLOPPY", the status information of the device file is taken, and the major and minor
// device number are taken as the root device number. If the root device is a FLOPPY device,
// set the major and minor device number to 0, indicating that the root device is the current
// boot device.
        if ((argc < 4) || (argc > 6))
                usage();
        if (argc > 4) {
                if (strcmp(argv[4], "FLOPPY")) {
                        if (stat(argv[4], &sb)) {
                                perror(argv[4]);
                                die("Couldn't stat root device.");
                        }
                        major_root = MAJOR(sb.st_rdev);   // Get the device number.
                        minor_root = MINOR(sb.st_rdev);
                } else {
                        major_root = 0;
                        minor_root = 0;
                }
// If there are only 4 parameters, let the major and minor device number be equal to the system
// default root device number.
        } else {
                major_root = DEFAULT_MAJOR_ROOT;
                minor_root = DEFAULT_MINOR_ROOT;
        }
// If there are 6 parameters on the command line, if the last parameter indicating that the
// switching device is not "NONE", the status information of the device file is taken, and the
// major and minor device numbers are taken as the swap device number. If the last parameter
// is "NONE", the major and the minor device number of the swap device are taken as 0, indicating
// that the swap device is the current boot device.
        if (argc == 6) {
                if (strcmp(argv[5], "NONE")) {
                        if (stat(argv[5], &sb)) {
                                perror(argv[5]);
                                die("Couldn't stat root device.");
                        }
                        major_swap = MAJOR(sb.st_rdev);
                        minor_swap = MINOR(sb.st_rdev);
                } else {
                        major_swap = 0;
                        minor_swap = 0;
                }
// If there are no 6 parameters but 5, it means that there is no swap device name on the command
// line. So let the swap device major and minor device numbers equal the system default swap
// device number.
        } else {
                major_swap = DEFAULT_MAJOR_SWAP;
                minor_swap = DEFAULT_MINOR_SWAP;
        }

// Next, the major and minor device numbers of the root device selected and the major and minor
// device numbers of the swap device are displayed on the standard error terminal. If the major
// device number is not 2 (floppy disk) or 3 (hard disk), nor is it 0 (system default device),
// an error message is displayed and exits. The standard output of the terminal is redirected
// to the file 'Image', so it is used to output the saved kernel code and data to generate a
// kernel image file.
        fprintf(stderr, "Root device is (%d, %d)\n", major_root, minor_root);
        fprintf(stderr, "Swap device is (%d, %d)\n", major_swap, minor_swap);
        if ((major_root != 2) && (major_root != 3) &&
            (major_root != 0)) {
                fprintf(stderr, "Illegal root device (major = %d)\n",
                        major_root);
                die("Bad root device --- major #");
        }
        if (major_swap && major_swap != 3) {
                fprintf(stderr, "Illegal swap device (major = %d)\n",
                        major_swap);
                die("Bad root device --- major #");
        }
// (2) The following begins to read the contents of each file and perform the corresponding
// copy processing. First initialize the 1KB buffer, then open the file specified by parameter
// 1 (bootsect) in read-only mode, and read the 32-byte MINIX execution header structure (see
// the list below) into the buffer buf.
        for (i=0;i<sizeof buf; i++) buf[i]=0;
        if ((id=open(argv[1],O_RDONLY,0))<0)
                die("Unable to open 'boot'");
        if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
                die("Unable to read header of 'boot'");
// Next, check if the bootsect is a valid MINIX executable file based on the MINIX header
// structure. If so, the 512-byte boot sector code and data are read from the file. The value
// "0x04100301" means: 0x0301 - MINIX head magic value a_magic; 0x10 - executable flag a_flag;
// 0x04 - machine type a_cpu, Intel 8086 machine code.
// A series of checks are then performed on the header information. Check if the header size
// field a_hdrlen is correct (32 bytes) (The last three bytes are just useless, it is 0); confirm
// whether the data segment size a_data field (long) content is 0; confirm whether the heap
// a_bss field (long) is 0; confirm whether the execution point a_entry field (long) is 0;
// confirms whether the symbol table size field a_sym is 0.
```

```
        if (((long *) buf)[0]!=0x04100301)
                die("Non-Minix header of 'boot'");
        if (((long *) buf)[1]!=MINIX_HEADER)
                die("Non-Minix header of 'boot'");
        if (((long *) buf)[3]!=0)
                die("Illegal data segment in 'boot'");
        if (((long *) buf)[4]!=0)
                die("Illegal bss in 'boot'");
        if (((long *) buf)[5] != 0)
                die("Non-Minix header of 'boot'");
        if (((long *) buf)[7] != 0)
                die("Illegal symbol table in 'boot'");
// After reading the actual code and data in the file under the condition that the above judgments
// are correct, the number of read bytes should be 512 bytes. Because the bootsect file contains
// the boot sector code and data for one sector, and the last 2 bytes should be the bootable
// flag 0xAA55.
        i=read(id,buf,sizeof buf);
        fprintf(stderr,"Boot sector %d bytes.\n",i);
        if (i != 512)
                die("Boot block must be exactly 512 bytes");
        if ((*(unsigned short *)(buf+510)) != 0xAA55)
                die("Boot block hasn't got boot flag (0xAA55)");
// Then modify the contents of the buffer, store the swap device number at the offset of 506,
// 507, and store the root device number at the offset of 508, 509.
        buf[506] = (char) minor_swap;
        buf[507] = (char) major_swap;
        buf[508] = (char) minor_root;
        buf[509] = (char) major_root;
// Next, write the 512 bytes of data to the standard output stdout and close the bootsect file.
// In linux/Makefile, the build program redirects the standard output to the kernel image
// filename Image using the ">" indicator, so the boot sector code and data are written to the
// first 512 bytes of the Image.
        i=write(1,buf,512);
        if (i!=512)
                die("Write call failed");
        close (id);

// (3) The file (setup) specified by parameter 2 is opened in read-only mode, and the contents
// of the 32-byte MINIX execution file header are read to the buffer buf. Next, according to
// the MINIX header structure, it is checked whether the setup is a valid MINIX execution file.
// If so, a series of checks are performed on the header information. It is handled in the same
// way as above.
        if ((id=open(argv[2],O_RDONLY,0))<0)
                die("Unable to open 'setup'");
        if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
                die("Unable to read header of 'setup'");
        if (((long *) buf)[0]!=0x04100301)
                die("Non-Minix header of 'setup'");
        if (((long *) buf)[1]!=MINIX_HEADER)            // header size (32 bytes)
                die("Non-Minix header of 'setup'");
        if (((long *) buf)[3]!=0)                       // data size field a_data
                die("Illegal data segment in 'setup'");
        if (((long *) buf)[4]!=0)                       // a_bss field.
```

```
                die("Illegal bss in 'setup'");
        if (((long *) buf)[5] != 0)                    // a_entry point.
                die("Non-Minix header of 'setup'");
        if (((long *) buf)[7] != 0)
                die("Illegal symbol table in 'setup'");
// The subsequent actual code and data in the file are read under the condition that the above
// checks are correct, and written to the terminal standard output. At the same time, the length
// of the writing is counted, and the setup file is closed after the operation ends. Then check
// the code and data size of the write operation, the value can not be greater than (SETUP_SECTS
// * 512) bytes, otherwise you have to re-edit the number of sectors occupied by the setup set
// in the build, bootsect and setup programs, and recompile the kernel . If everything is ok,
// the actual length value of the setup is displayed.
        for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
                if (write(1,buf,c)!=c)
                        die("Write call failed");
        close (id);                                   // close setup file.
        if (i > SETUP_SECTS*512)
                die("Setup exceeds " STRINGIFY(SETUP_SECTS)
                        " sectors - rewrite build/boot/setup");
        fprintf(stderr,"Setup is %d bytes.\n",i);
// After clearing the buffer buf, check the difference between the actual write setup length and
// (SETUP_SECTS * 512). If the setup size is smaller than the size (4 * 512 bytes), fill the
// setup with NULL characters to 4 *512 bytes.
        for (c=0 ; c<sizeof(buf) ; c++)
                buf[c] = '\0';
        while (i<SETUP_SECTS*512) {
                c = SETUP_SECTS*512-i;
                if (c > sizeof(buf))
                        c = sizeof(buf);
                if (write(1,buf,c) != c)
                        die("Write call failed");
                i += c;
        }

// (4) Start processing the system module file below. This file is compiled with gas/gcc and
// therefore has the GNU a.out object file format.
// First open the system module file in read-only mode, and read the a.out format header structure
// information (1KB size). After confirming that system is a valid a.out format file, write all
// subsequent data of the file to the standard output (Image file) and close the file. Then show
// the size of the system. If the system code and data size exceed the SYS_SIZE section (128KB
// bytes), an error message is displayed and exits. If there is no error, it returns 0, indicating
// normal exit.
        if ((id=open(argv[3],O_RDONLY,0))<0)
                die("Unable to open 'system'");
        if (read(id,buf,GCC_HEADER) != GCC_HEADER)
                die("Unable to read header of 'system'");
        if (((long *) buf)[5] != 0)               // entry location should be 0.
                die("Non-GCC header of 'system'");
        for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
                if (write(1,buf,c)!=c)
                        die("Write call failed");
        close(id);
        fprintf(stderr,"System is %d bytes.\n",i);
        if (i > SYS_SIZE*16)
                die("System is too big");
        return(0);
}

```

16.1.3 Information

16.1.3.1 MINIX module and executable header data structure

The header structure of the modules and executables generated by MINIX's compiler and linker is as follows:

```
struct  exec {
  unsigned char a_magic[2];     // Magic number, should be 0x0301.
  unsigned char a_flags;        // Flags (see below).
  unsigned char a_cpu;          // Machine CPU identifier.
  unsigned char a_hdrlen;       // Reserved header size, 32 or 48 bytes.
  unsigned char a_unused;       // Reserved.
  unsigned short a_version;     // Version information (not used currently).
  long          a_text;         // Code section size (in bytes).
  long          a_data;         // Data section size (in bytes).
  long          a_bss;          // Stack size (in bytes).
  long          a_entry;        // Execute entry point.
  long          a_total;        // Total amount of memory allocated.
  long          a_syms;         // Symbol table size.
      // The structure ends here if the header size is 32 bytes.
  long          a_trsize;       // Code section relocation table size.
  long          a_drsize;       // Data section relocation table size.
  long          a_tbase;        // Code section relocation base address.
  long          a_dbase;        // Data section relocation base address.
};
```

Among them, the flag field a_flags in the MINIX execution file header is defined as:

```
    A_UZP    0x01               // Unmapped 0 pages (pages).
    A_PAL    0x02               // Ajiusted at the page boundary.
    A_NSYM   0x04               // New type symbol table.
    A_EXEC   0x10               // Executable file.
    A_SEP    0x20               // The code and data are separate (I and D are independent).
```

The CPU identification number field a_cpu can be:

```
    A_NONE   0x00               // Unknown.
    A_I8086  0x04               // Intel i8086/8088.
    A_M68K   0x0B               // Motorola m68000.
    A_NS16K  0x0C               // National Semiconductor Co. 16032.
    A_I80386 0x10               // Intel i80386.
    A_SPARC  0x17               // Sun SPARC.
```

The above MINIX execution header structure exec is similar to the a.out format header structure used by the Linux 0.12 system. See the linux/include/a.out.h file for the header structure and related information of the Linux a.out format executable file.

16.2 Summary

This chapter details the build tool build.c given in the kernel source tree. This program is mainly used to modify and combine the kernel modules to generate a bootable kernel boot image file. So far, we have completed a detailed description and comments on all source code files in the kernel.

In order to gain a deeper understanding of the kernel's operating mechanism, in the next chapter we use the Bochs simulation program to detail the test methods in which the Linux 0.12 operating system is set up and running, and give specific test steps for some experiments.

17 Experimental Environment Settings and Usage

In order to learn the working principle of the Linux 0.1x kernel, this chapter introduces the experimental method of using the PC simulation software and running the Linux 0.1x system on the real computer. These include the kernel compilation process, file access and copying in the simulation environment, how to make the boot disk and root file system, and how to use the Linux 0.1x system. Finally, we also introduced how to make a small number of syntax changes to the kernel code, so that it can successfully pass the compilation process under the existing RedHat system (gcc 3.x), and make the corresponding kernel image file.

Before we start the experiment, we first need to prepare some useful tools. If we are going to experiment on the Windows platform, we need to have the following software ready:

- Bochs 2.6.x open source PC simulation package (https://sourceforge.net/projects/bochs/);
- Notepad++ Editor. Used to edit binary files (https://sourceforge.net/projects/notepad-plus/);
- HxD hex editor. Used to edit binary or disk files, even in-memory data (https://mh-nexus.de/en/hxd/);
- WinImage DOS format floppy image file editing software (http://www.winimage.com/).

If you are experimenting with modern Linux systems (such as Redhat, Ubuntu, etc.), then we usually only need to install the Bochs package to perform the simulation. Other operations in the experiment can be done using the common tools of the Linux system.

The best way to run a Linux 0.1x system is to use PC emulation software. There are currently four popular PC simulation software in the world: VMware's VMware Workstation software, Oracle's VirtualBox open source software, Microsoft's Virtual PC and open source software Bochs (the pronunciation is the same as "box"). They can fully virtualize and simulate the operation of a PC. These types of software can virtualize or emulate an Intel x86 hardware environment, allowing us to run a variety of other "customer" operating systems on the platform on which the software is running.

In terms of scope of use and operational performance, the four simulation softwares still have some differences. Bochs uses software to simulate the entire hardware environment (CPU instructions) of the x86 CPU-based PC and its peripherals, so it can be easily ported to many operating systems or platforms with different architectures. Because it mainly uses software simulation technology, its running performance and speed are much slower than other simulation software. The performance of Virtual PC software is between Bochs and VMware Workstation (or VirtualBox). It emulates most of the x86 instructions, while other parts are implemented using virtual technology. VMware Workstation and VirtualBox only simulate some I/O capabilities, while all other parts are executed directly on x86 real-time hardware. That is to say, when the guest operating system is required to execute an instruction, VMware or VirtualBox does not execute this instruction by simulation, but simply "passes" this instruction directly to the hardware of the actual system. So VMware and VirtualBox are the best in terms of speed and performance in these software. Of course, other simulation software, such as Qemu, can also have high simulation performance.

From an application perspective, VMware Workstation and VirtualBox should be a good choice if the simulation environment is primarily used to run applications. But if you need to develop some low-level system software (such as operating system development and debugging, compiler system development, etc.), then

Bochs is a good choice. With Bochs, you can know the specific state and precise timing of the executed program in the simulated hardware environment, rather than the actual hardware system execution. This is why many operating system developers are more inclined to use Bochs. This chapter describes how to run Linux 0.1x using the Bochs simulation environment. Currently, the name of the Bochs website is http://sourceforge.net/projects/bochs/. You can download the latest release of the Bochs software from above, or you can download the image files of many ready-to-run systems.

17.1 Bochs Simulation Software

Bochs is a program that fully emulates an Intel 80X86 computer. It can be configured to emulate Intel's 80386, 486, Pentium or above new CPU processors. Throughout the execution phase, Bochs simulates all execution instructions, including emulating all device modules of standard PC peripherals. Since Bochs simulates the entire PC environment, the software executing in it "thinks" that it is running on a real machine. This fully simulated approach allows us to run a large number of software systems without modification in Bochs.

Bochs is a software system developed by Kevin Lawton in 1994 using the C++ language. The system is designed to run in hardware environments such as Intel 80X86, PPC, Alpha, Sun, and MIPS. Regardless of the hardware platform on which the host is running, Bochs can still simulate the Intel hardware platform of the Intel 80X86 CPU. This feature is not available in several other simulation software. In order to perform any activity on the machine being simulated, Bochs needs to interact with the host operating system. When a key is pressed in the Bochs display window, a keystroke event is sent to the keyboard device processing module. When the simulated machine needs to perform a read operation from the simulated hard disk, Bochs will perform a read operation on the hard disk image file on the host.

The installation of Bochs software is very convenient. You can download the Bochs installation package directly from http://bochs.sourceforge.net. If the computer operating system you are using is Windows, the installation process is exactly the same as normal software. After the Bochs software is installed, it will generate a directory on the C: 'C:\Program Files\Bochs-2.6\' (where the version number varies with different versions). If your system is RedHat or another Linux system, you can download the Bochs RPM package and install it as follows:

```
user$ su
Password:
root# rpm -i bochs-2.6.i386.rpm
root# exit
user$ _
```

You need root privileges when installing Bochs, otherwise you will have to recompile the Bochs system in your own directory. In addition, Bochs needs to run in the X11 environment, so you must have the X Window System installed on your Linux system to use Bochs. After installing Bochs, it is recommended to test and familiarize yourself with the Bochs system using the Linux dlx demo system included with the Bochs package. You can also download some of the other Linux image files from the Bochs website to do some experimentation. We recommend downloading the SLS Linux emulation system package (sls-0.99pl.tar.bz2) on the Bochs website as an auxiliary platform for creating Linux 0.1x emulation systems. When making new hard disk image files, we can use these systems to partition and format the hard disk image files. The image file for this SLS

Linux system can also be downloaded directly from oldlinux.org: http://oldlinux.org/Linux.old/bochs/sls-1.0.zip. After extracting the downloaded file, go to its directory and double-click the configuration file name bochsrc.bxrc to let Bochs run the SLS Linux system. If the configuration file name does not have the suffix .bxrc, please modify it yourself. For example, the original name bochsrc was modified to bochsrc.bxrc.

For instructions on recompiling the Bochs system or installing Bochs on other hardware platforms, please refer to the instructions in the Bochs User Manual.

17.1.1 Setting up the Bochs system

In order to run an operating system in Bochs, we need at least some of the following resources or information:

- Bochs and bochsdbg executable files;
- BIOS image files (commonly referred to as 'BIOS-bochs-latest');
- VGABIOS image files (eg 'VGABIOS-lgpl-latest');
- at least one boot image file (floppy, hard disk or CDROM image) file).

Bochs.exe is the executable for the Bochs system. If you need to track and debug the program in Bochs, you also need bochsdbg.exe to execute the program. The BIOS and VGABIOS are image files emulating the ROM BIOS of the PC and the BIOS software in the display card, respectively. In addition, we also need the boot image file of the system to be used for emulation. These files need to work together, so we need to set some environment parameters for the simulation before running the Bochs program. These parameters can be passed to the Bochs executable on the command line, but we usually use a textual configuration file (file suffix .bxrc, such as Sample.bxrc) to set the running parameters for a specific application. The following describes how to set up the Bochs configuration file.

17.1.2 *.bxrc Configuration File

Bochs uses the information in the configuration file to find the disk image file used, the configuration of the operating environment peripherals, and other virtual machine setup information. Each simulated system needs to set a corresponding configuration file. If the installed Bochs system is 2.1 or later, the Bochs system will automatically recognize the configuration file with the suffix '.bxrc' and automatically start the Bochs system when you double-click the file icon. For example, we can take the configuration file name as 'bochsrc-0.12.bxrc'. In the Bochs installation home directory (usually C:\Program Files\Bochs-2.6\) there is a template configuration file named 'bochsrc-sample.txt' which lists all available parameters with Detailed explanation. Here are a few of the parameters that are often modified in our experiments.

1. megs

Used to set the memory capacity of the simulated system. The default is 32MB. For example, if you want to set up your simulated machine to have a 128MB system, you need to include the following line in your configuration file:

```
megs: 128
```

2. floppya（floppyb）

Floppya represents the first floppy drive, and floppyb represents the second one. If you need to boot the system from a floppy disk, then floppya needs to point to a bootable disk. If you want to use a disk image file, then we will write the name of the disk image file after this option. In many operating systems, Bochs can directly read and write the floppy disk drive of the host system. To access the disks in these actual drives, use the device name (Linux system) or drive letter (Windows system). You can also use status to indicate the insertion status of the disk: 'ejected' means not inserted, and 'inserted' means the disk is inserted. Here are a few examples where all the disks are inserted. If there are several rows of parameters with the same name in the configuration file, only the last row of parameters will work.

```
floppya: 1_44=/dev/fd0, status=inserted                  # Access to 1.44MB A drive under Linux.
floppya: 1_44=b:, status=inserted                        # Access to 1.44MB B drive under Win.
floppya: 1_44=bootimage.img, status=inserted             # Use imagefile bootimage.img.
floppyb: 1_44=..\Linux\rootimage.img, status=inserted    # Use image ..\Linux\rootimage.img.
```

3. ata0、ata1、ata2、ata3

These four parameter names are used to start up to four ATA channels in the simulated system. For each enabled channel, two IO base addresses and one interrupt request number must be specified. By default only ata0 is enabled and the parameters default to the values shown below:

```
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15
ata2: enabled=1, ioaddr1=0x1e8, ioaddr2=0x3e0, irq=11
ata3: enabled=1, ioaddr1=0x168, ioaddr2=0x360, irq=9
```

4. ata0-master（ata0-slave）

The ata0-master is used to indicate the first ATA device (hard disk or CDROM, etc.) connected to the first ATA channel (0 channel) in the simulated system; the ata0-slave indicates the second ATA device connected to the first channel. An example is shown below, where the options for device configuration are as shown in Table 17–1.

```
ata0-master: type=disk, path=hd.img, mode=flat, cylinders=306, heads=4, spt=17, translation=none
ata1-master: type=disk, path=2G.cow, mode=vmware3, cylinders=5242, heads=16, spt=50, translation=echs
ata1-slave:  type=disk, path=3G.img, mode=sparse, cylinders=6541, heads=16, spt=63, translation=auto
ata2-master: type=disk, path=7G.img, mode=undoable, cylinders=14563, heads=16, spt=63, translation=lba
ata2-slave:  type=cdrom, path=iso.sample, status=inserted
ata0-master: type=disk, path="hdc-large.img", mode=flat, cylinders=487, heads=16, spt=63
ata0-slave:  type=disk, path="..\hdc-large.img", mode=flat, cylinders=121, heads=16, spt=63
```

Table 17-1 Device configuration options

Options	Description	Available value
type	Connected device type	[disk | cdrom]
path	Image file path name	
mode	Image file type, valid only for disk	[flat | concat | external | dll | sparse | vmware3 | undoable | growing | volatile]

cylinders	Valid only for disk	
heads	Valid only for disk	
spt	Valid only for disk	
status	Valid only for disk	[inserted | ejected]
biosdetect	Bios detect type	[none | auto], Only valid for disk on ata0 [cmos]
translation	The type of bios convertion (int13), valid only for disk	[none | lba | large | rechs | auto]
mode	Confirm the string returned by the device ATA command	

When configuring an ATA device, you must specify the type of the connected device, which can be ‘disk’ or ‘cdrom’. You must also specify the pathname of the device. The "pathname" can be a hard disk image file, an iso file of the CDROM, or a CDROM drive pointing directly to the system. In Linux systems, system devices can be used as Bochs hard drives, but for security reasons, direct use of physical hard disks on the system is not recommended under Windows.

For devices of type 'disk', the options 'path', 'cylinders', 'heads' and 'spt' are required. For devices of type 'cdrom', the option 'path' is required.

The disk translation scheme (implemented in the traditional int13 bios function and used for old os like DOS) can be defined as:

- none： No need to translate, suitable for hard disks with a capacity less than 528MB (1032192 sectors);
- large： Standard bit shift algorithm for hard disks with a capacity of less than 4.2 GB (8257536 sectors);
- rechs： The modified shift algorithm uses a pseudo-physical hard disk parameter of 15 heads for hard disks with a capacity less than 7.9 GB (15482880 sectors);
- lba： Standard lba-assisted algorithm. Suitable for hard drives with a capacity less than 8.4GB (16,450,560 sectors);
- auto： Automatically select the best conversion scheme (should be changed if system does not start).

The mode option is used to explain how to use the hard disk image file. It can be one of the following modes:

- flat： a flat sequential file;
- concat： Multiple files;
- external： Dedicated by the developer, specified by the C++ class;
- dll： Dedicated by the developer, used by the DLL;
- sparse： Stackable, identifiable, retractable;
- vmware3： Support vmware3 hard disk format;
- undoable： a flat file with a confirmed redo log;
- growing： Capacity scalable image file;
- volatile： A flat file with a variable redo log.

The default values for the above options are:

```
mode=flat, biosdetect=auto, translation=auto, model=”Generic 1234”
```

5. boot

'boot' is used to define the drive in the emulated machine for boot. It can be specified as a floppy disk, hard disk or CDROM, or drive letters 'c' and 'a'. Examples are as follows:

```
boot: a
boot: c
boot: floppy
boot: disk
boot: cdrom
```

6. cpu

The 'cpu' is used to define the parameters of the CPU that is simulated in the simulation system. This option can take four parameters: COUNT, QUANTUM, RESET_ON_TRIPLE_FAULT, and IPS.

Where 'COUNT' is used to indicate the number of processors emulated in the system. When the Bochs package is compiled with the SMP support option, Bochs currently supports up to 8 simultaneous threads. However, if the compiled Bochs does not support SMP, COUNT can only be set to 1.

'QUANTUM' is used to specify the maximum number of instructions that can be executed before switching from one processor to another. This option is also only available for Bochs programs that support SMP.

'RESET_ON_TRIPLE_FAULT' is used to specify that the CPU needs to perform a reset operation instead of just a panic when a triple error occurs in the processor.

IPS specifies the number of instructions per second to be simulated. This is the IPS value that Bochs runs on the host system. This value affects many events related to time in the simulation system. For example, changing the IPS value will affect the rate of VGA updates and other simulation system evaluations. It is therefore necessary to set this value depending on the host performance used. Refer to Table 17-2 for settings. For example:

```
cpu: count=1, ips=50000000, reset_on_triple_fault=1
```

Table 17-2 Examples IPS settings

Bochs version	**Speed**	**Machine / Compiler**	**Typical IPS**
2.4.6	3.4Ghz	Intel Core i7 2600 with Win7x64/g++ 4.5.2	85 to 95 MIPS
2.3.7	3.2Ghz	Intel Core 2 Q9770 with WinXP/g++ 3.4	50 to 55 MIPS
2.3.7	2.6Ghz	Intel Core 2 Due with WinXP/g++ 3.4	38 to 43 MIPS
2.2.6	2.6Ghz	Intel Core 2 Due with WinXP/g++ 3.4	21 – 25 MIPS
2.2.6	2.1Ghz	Athlon XP with Linux 2.6/g++ 3.4	12 – 15 MIPS

7. log

Specifying the path name of 'log' allows Bochs to log some log information during execution. If the system running in Bochs does not work properly, you can refer to the information to find out the basic reason. The log is usually set to:

```
 log: bochsout.txt
```

17.2 Running Linux 0.1x system in Bochs

To run a Linux operating system, we need a root filesystem (root fs) in addition to the kernel. The root file system is usually an external device that stores the necessary files (such as system configuration files and device files) and data files for Linux system runtime. In modern Linux operating systems, the kernel image file (bootimage) is stored in the root file system. The system boot initiator loads the kernel execution code into memory from this root file system device for execution.

However, the kernel image file and the root file system are not required to be stored on the same device, that is, they are not necessarily required to be placed in a floppy disk or in the same partition of the hard disk. For the case of using only floppy disks, due to the limitation of floppy disk capacity, the kernel image file and the root file system are usually placed on two separate disks. The floppy disk that stores the bootable kernel image file is called the kernel boot disk. (bootimage); The floppy disk that holds the root file system is called the root file system image file (rootimage). Of course, we can also load the kernel image file from the floppy disk, and at the same time use the root file system in the hard disk, or let the system boot the system directly from the hard disk, that is, load the kernel image file from the root file system of the hard disk and use the root file system in the hard disk.

This section describes how to run several Linux 0.1x systems that have been set up in Bochs and explains the settings of several main parameters in the relevant configuration files. First we download the following Linux 0.1x system package from the website to the desktop of the computer:

```
   http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.zip
```

The last 6 digits in the package name are date information. You should usually choose the latest package with the latest download date. After the download is complete, you can use a general decompression program such as unzip, 7-zip or rar to decompress it. Note that you need about 250MB of disk space to unzip this file.

17.2.1 Description of the files in the package

When the linux-0.12-080324.zip file is unzipped, a directory named linux-0.12-080324 is generated. After entering the directory, we can see that there are about 20 files below.

```
[root@www linux-0.12-080324]# ls -o -g
total 256916
-rw-r--r--   1   3078642 Mar 24 10:49 bochs-2.3.6-1.i586.rpm
-rw-r--r--   1   3549736 Mar 24 10:48 Bochs-2.3.6.exe
-rw-r--r--   1     15533 Mar 24 18:04 bochsout.txt
```

```
-rw-r--r--   1      1774 Mar 24 20:13 bochsrc-0.12-fd.bxrc
-rw-r--r--   1      5903 Mar 24 17:56 bochsrc-0.12-hd.bxrc
-rw-r--r--   1     35732 Dec 24 20:15 bochsrc-sample.txt
-rw-r--r--   1    150016 Mar  6  2004 bootimage-0.12-fd
-rw-r--r--   1    154624 Aug 27  2006 bootimage-0.12-hd
-rw-r--r--   1        68 Mar 24 12:21 debug.bat
-rw-r--r--   1   1474560 Mar 24 15:27 diska.img
-rw-r--r--   1   1474432 Aug 27  2006 diskb.img
-rw-r--r--   1      7917 Mar 24 11:32 linux-0.12-README
-rw-r--r--   1   1474560 Mar 24 17:03 rootimage-0.12-fd
-rw-r--r--   1 251338752 Mar 24 18:04 rootimage-0.12-hd
-rw-r--r--   1     21253 Mar 13  2004 SYSTEM.MAP
[root@www linux-0.12-080324]#
```

This package contains two Bochs installers, two Bochs .bxrc configuration files, two bootimage files containing kernel code, a floppy disk and a hard disk root file system (rootimage) file, and other files. 其中 The README file briefly describes the purpose of each file. Here we will explain in more detail the purpose of each file.

- Bochs-2.3.6-1.i586.rpm is the Bochs installer for the Linux operating system. You can re-download the latest program.
- Bochs-2.3.6.exe is the Bochs installer for the Windows operating system platform. The latest version of the Bochs software can be downloaded from http://sourceforge.net/projects/bochs/. As Bochs continues to improve, some newer versions may cause compatibility issues. This needs to be resolved by modifying the .bxrc configuration file, and some issues need to be resolved by modifying the Linux 0.1x kernel code.
- bochsout.txt is a log file that is automatically generated when the Bochs system is running. It contains various status information for the Bochs runtime. When running Bochs has problems, you can check the contents of this file to preliminarily determine the cause of the problem.
- bochsrc-0.12-fd.bxrc is the configuration file that allows the system to boot from a floppy disk. This configuration file is used to boot the Linux 0.12 system from the Bochs Virtual A drive (/dev/fd0), ie the kernel image file is set in virtual disk A and the subsequent root file system is required to be inserted into the current virtual boot drive. During the boot process it will ask us to "insert" the root filesystem disk (rootimage-0.12-fd) in the virtual A drive. The kernel image and boot file used by this configuration file is bootimage-0.12-fd. After Bochs is properly installed, double-click this configuration file to run the configured Linux 0.12 system.
- bochsrc-0.12-hd.bxrc is also a configuration file set to boot from drive A, but will use the root file system in the hard drive image file (rootimage-0.12-hd). This configuration file is booted using bootimage-0.12-hd. Similarly, after properly installing Bochs, double-click on this configuration file to run the configured Linux 0.12 system.
- bootimage-0.12-fd is an image file generated by the compiled kernel. It contains the code and data for the entire kernel, including the code for the floppy boot sector. You can run the configured Linux 0.12 system by double-clicking on the relevant configuration file.
- bootimage-0.12-hd is the kernel image file used to use the root file system on the virtual hard disk, that is, the root file system device number of the 509th and 510th bytes of the file has been set to the 1st partition of the C hard disk (/dev /hd1), the device number is 0x0301.
- debug.bat is a batch program that starts the Bochs debugging function on the Windows platform.

Please note that you may need to modify the pathname based on the specific directory where Bochs is installed. In addition, the Bochs system installed and running on Linux systems by default does not include debugging features. You can debug directly using the gdb program on Linux. If you still want to take advantage of Bochs' debugging features, then you need to download the source code of Bochs and customize it yourself.

- diska.img and diskb.img are two floppy image files in DOS format. It contains some utilities. In Linux 0.12 you can use the command mcopy and other commands to access these two image files. Of course, you need to dynamically "insert" the corresponding "floppy disk" before accessing. When you double-click the bochsrc-0.12-fd.bxrc or bochsrc-0.12--hd.bxrc configuration file to run the Linux 0.12 system, the B drive is configured to be "inserted" with the diskb.img disk.
- rootimage-0.12-hd is the virtual hard disk image file mentioned above, which contains 3 partitions. The first partition is a MINIX file system type 1.0 root file system, and the other two partitions are also MINIX 1.0 file system types, and some source code files for testing are stored. You can load and use these spaces by using the mount command.
- rootimage-0.12-fd is the root file system on the floppy disk. This root file system disk is used when running the Linux 0.12 system using the bochsrc-0.12-fd configuration file.
- The SYSTEM.MAP file is the kernel memory storage location information file generated when the Linux 0.12 kernel is compiled. The contents of this file are very useful when debugging the kernel.

17.2.2 Installing the Bochs

The bochs-2.3.6-1.i586.rpm file in the package is the Bochs installer used under Linux. Bochs-2.3.6.exe is the Bochs installer on the Windows operating system. The latest version of the Bochs software is always available at the following website location:

```
    http://sourceforge.net/projects/bochs/
```

If we are experimenting with a Linux system, you can install the Bochs software by running the rpm command on the command line or by double-clicking on the first file in the above package in the X window:

```
rpm -i bochs-2.3.6-1.i586.rpm
```

If you are on a Windows system, simply double-click the Bochs-2.3.6.exe file icon to install the Bochs system. After the installation, please modify the contents of the batch file debug.bat according to the specific directory of the installation. In addition, in the following experimental procedures and examples, we mainly introduce the use of Bochs on the Windows platform.

17.2.3 Running the Linux 0.1x System

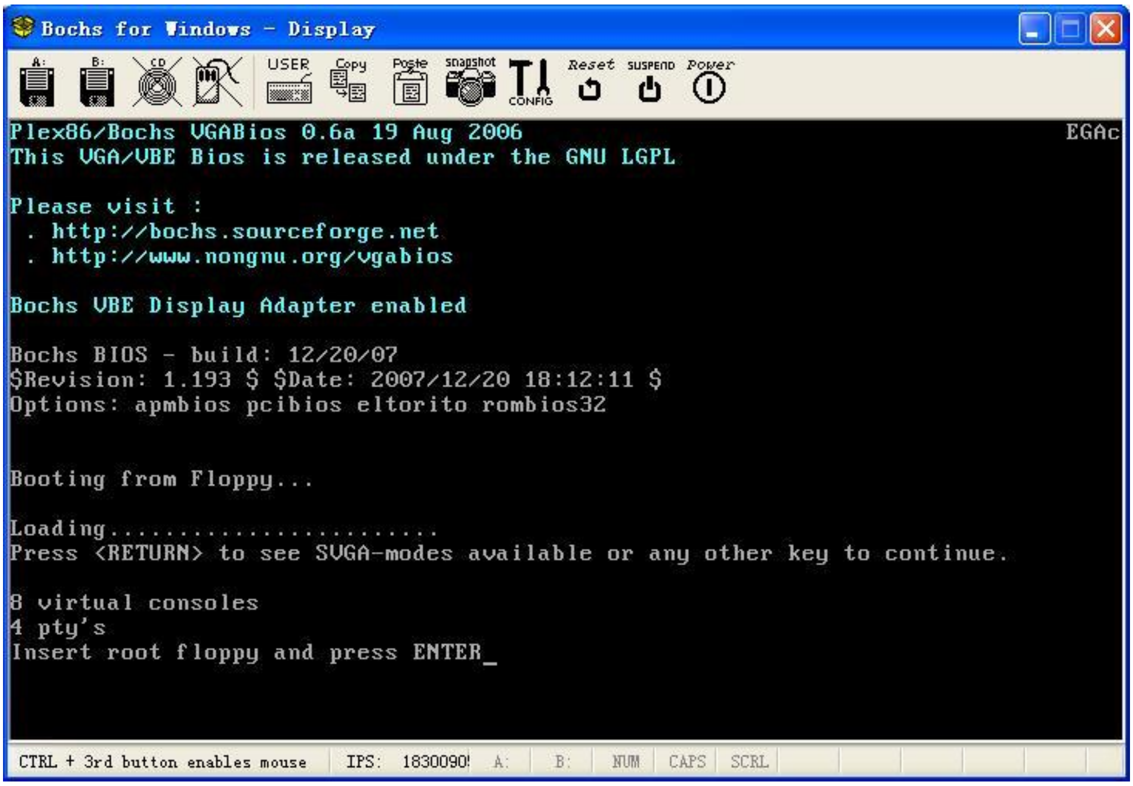
Running a Linux 0.1x system in Bochs is very simple. Once the Bochs software is properly installed, you can simply double-click on the appropriate Bochs configuration file (*.bxrc) to get started. The PC environment for runtime simulation has been set up in each configuration file. You can modify these files with any text editor. To run a Linux 0.12 system, the corresponding configuration files usually only need to include the following lines of information:

```
romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
megs: 16
floppya: 1_44="bootimage-0.12-hd", status=inserted
ata0-master: type=disk, path="rootimage-0.12-hd", mode=flat, cylinders=487, heads=16, spt=63
boot: a
```

The first two lines indicate the ROM BIOS and VGA display card ROM program of the simulated PC, and generally do not need to be modified. Line 3 indicates the physical memory capacity of the PC, which is set to 16MB. Because the default Linux 0.12 kernel only supports up to 16MB of memory, the big settings don't work either. The parameter floppya specifies that the floppy disk drive A of the simulated PC uses the 1.44MB disk type, and has been set to use the bootimage-0.12-fd floppy image file, and is in the inserted state. The corresponding floppyb can be used to indicate the floppy image file used or inserted in the B drive. The parameter ata-master is used to specify the virtual hard disk capacity and hard disk parameters attached to the simulated PC. For the specific meaning of these hard disk parameters, please refer to the previous description. In addition, ata0-slave can be used to specify the image file and parameters used by the second virtual hard disk. The last 'boot' is used to specify the boot drive that can be set to boot from the A drive or from the C drive (hard drive). Here we set it to boot from the A drive (a).

1. Run the Linux 0.12 system using the bochsrc-0.12-fd.bxrc file.

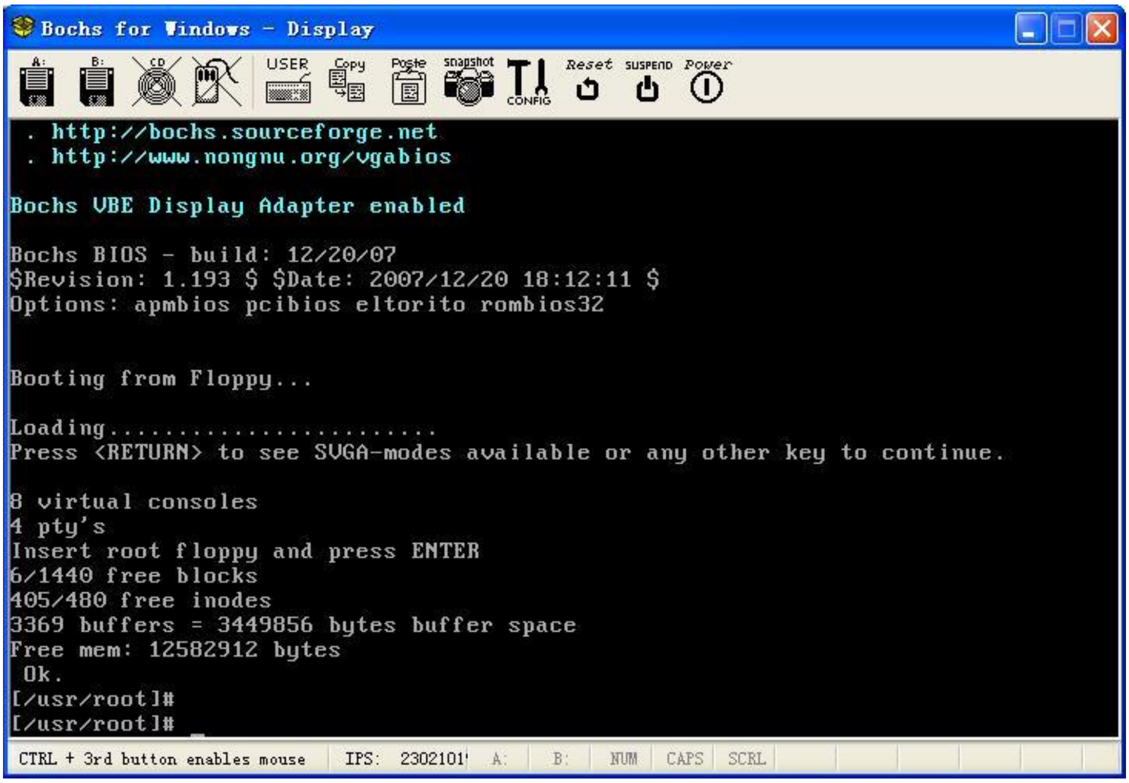
That is, boot the Linux 0.12 system from the floppy disk and use the root file system in the current drive. This way of running a Linux 0.12 system uses only two floppy disks: bootimage-0.12-fd and rootimage-0.12-fd. The contents of the several lines of configuration files listed above are the basic settings in bochsrc-0.12-fd.bxrc, just the boot image file is replaced by bootimage-0.12-fd. When you double-click this configuration file to run the Linux 0.12 system, a message will appear in the main window of Bochs display, as shown in Figure 17-1.



```
Plex86/Bochs VGABios 0.6a 19 Aug 2006                                    EGAc
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 12/20/07
$Revision: 1.193 $ $Date: 2007/12/20 18:12:11 $
Options: apmbios pcibios eltorito rombios32


Booting from Floppy...

Loading........................
Press <RETURN> to see SVGA-modes available or any other key to continue.

8 virtual consoles
4 pty's
Insert root floppy and press ENTER_
```

Figure 17-1 Boot from a floppy disk and use the root fs in a floppy disk

Since bochsrc-0.12-fd.bxrc configures the Linux 0.12 runtime to boot from drive A, and the kernel image file bootimage-0.12-fd will require the root file system to be in the drive currently being booted (disk A). , so the kernel will display a message asking us to "remove" the kernel boot image file bootiamge-0.12-fd and "insert in" the root file system. At this point we can use the A disk icon at the top left of the window to "replace" the A disk. Click this icon and change the original image file name (bootimage-0.12-fd) to rootimage-0.12-fd, so that we have completed the floppy disk replacement operation. After clicking the "OK" button to close the dialog window, press the Enter key to let the kernel load the root file system on the floppy disk, and finally the command prompt line appears, as shown in Figure 17-2.



```
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 12/20/07
$Revision: 1.193 $ $Date: 2007/12/20 18:12:11 $
Options: apmbios pcibios eltorito rombios32


Booting from Floppy...

Loading.......................
Press <RETURN> to see SVGA-modes available or any other key to continue.

8 virtual consoles
4 pty's
Insert root floppy and press ENTER
6/1440 free blocks
405/480 free inodes
3369 buffers = 3449856 bytes buffer space
Free mem: 12582912 bytes
 Ok.
[/usr/root]#
[/usr/root]# _
```

Figure 17-2 "Replace" the floppy disk and press Enter to continue running

2.   2. Run the Linux 0.12 system using the bochsrc-0.12-hd.bxrc file

The configuration file will load the Linux 0.12 kernel image file bootimage-0.12-hd from the boot floppy disk (A disk) and use the root file system in the first partition of the hard disk image file rootimage-0.12-hd. Because the 509th and 510th bytes in the bootimage-0.12-hd file have been set to the device number 0x0301 of the first partition of the C drive (ie 0x01, 0x03), the kernel will automatically start loading the root file system from the first partition of the virtual C drive when the kernel is initialized. At this point, double-click the bochsrc-0.12-hd.bxrc file name to run the Linux 0.12 system directly, and you will get a screen similar to Figure 17-2.

17.3 Access Information in a Disk Image File

Bochs uses disk image files to emulate external storage devices in the simulated system. All files in the simulated operating system are saved in the image file in the format of a floppy disk or hard disk device. This brings about the problem of exchanging information between the host operating system and the simulated system in Bochs. Although the Bochs system can be configured to run directly using physical devices such as a host's floppy disk drive, CDROM drive, etc., it is cumbersome to utilize such an information exchange method. Therefore, it is best to directly read and write the information in the Image file. If you need to add a file to the simulated operating system, save the file in the Image file; if you want to get the file, read it from the Image file. However, since the information stored in the Image file is not only stored in the corresponding floppy disk or hard disk format, but also stored in a certain file system format. Therefore, the program that accesses the Image file must be able to recognize the file system in it to operate. For the purposes of this chapter, we need some tools to identify the MINIX and/or DOS file system formats in the Image file.

In general, if the file size exchanged with the simulated system is small, we can use the floppy Image file as the exchange medium. If there are large batches of files that need to get from the simulated system or put into the simulatied system, then we can use the existing Linux system to mount the image file. Below we discuss several methods that can be used from these two aspects.

- Use the disk image tool to access information (small files or split files) in the floppy image file;
- Use the loop device to access the hard disk image file in Linux. (a large amount exchange);
- Use iso format file for information exchange (a large amount exchange).

17.3.1 Using WinImage Software

By using a floppy Image file, we can exchange a small amount of files with the simulated system. The prerequisite is that the simulated system supports reading and writing DOS format floppy disks, for example, by using the mtools software. mtools is a program for accessing files in the MSDOS file system in a UNIX-like system. The software implements common MSDOS commands such as copy, dir, cd, format, del, md, and rd. Adding the letter m to the name of these commands is the corresponding command in mtools. The specific operation method will be described below by way of example.

Before reading/writing a file, you first need to prepare a 1.44MB Image file (the file name is assumed to be diskb.img) according to the method described above, and modify the boochs.bxrc configuration file of Linux 0.12. Add the following line to the floppya parameter:

```
    floppyb: 1_44=″diskb.img″, status=inserted
```

That is, the second 1.44MB floppy disk device is added to the simulated system, and the image file name used by the device is diskb.img.

If you want to take a file (hello.c) from the Linux 0.12 system, you can now start the Linux 0.12 system by double-clicking the configuration file icon. After entering the Linux 0.12 system, use the DOS floppy disk read and write tool mtools to write the hello.c file to the second floppy image. If the floppy Image was created using Bochs or has not been formatted, you can use the mformat b: command to format it first.

```
[/usr/root]# mcopy hello.c b:
Copying HELLO.C
[/usr/root]# mdir b:
 Volume in drive B has no label
 Directory for B:/

HELLO    C            74    4-30-104   4:47p
       1 File(s)     1457152 bytes free
[/usr/root]# _
```

Now exit the Bochs system and open the diskb.img file with WinImage. There will be a hello.c file in the WinImage main window. Use the mouse to select the file and drag it to the desktop, which completes the entire operation of fetching the file. If you need to put a file into the simulated system, the steps are exactly the opposite. Also note that WinImage can only access and manipulate disk files with DOS format, it can not access disk files in other formats such as MINIX file system.

17.3.2 Using an Existing Linux System

Existing Linux systems (such as Redhat) have access to a variety of file systems, including the use of loop devices to access file systems stored in image files. For the floppy Image file, we can use the mount command to load the file system in the Image for read/write access. For example, we need to access the file in rootimage-0.12, then just execute the following command.

```
[root@plinux images]# mount -t minix rootimage-0.12 /mnt -o loop
[root@plinux images]# cd /mnt
[root@plinux mnt]# ls
bin  dev  etc  root  tmp  usr
[root@plinux mnt]# _
```

The "-t minix" option of the mount command indicates that the file system type being read is MINIX, and the "-o loop" option indicates that the file system is loaded by the loop device. If you need to access the DOS format floppy image file, just replace the file type option "minix" in the mount command with "msdos".

If you want to access the hard disk Image file, the operation process is different from the above. Since the floppy Image file generally contains an image of a complete file system, the file system in the floppy disk image can be directly loaded using the mount command, but the hard disk Image file usually contains partition information, and the file system is created in each partition. So we need to load the required partition first, and then we can treat the partition as a complete "big" floppy disk.

Therefore, in order to access the information in a partition of a hard disk Image file, we need to first understand the partition information in the image file to determine the starting sector offset position of the partition to be accessed in the Image file. For the partition information in the hard disk Image file, we can use the fdisk command to view it in the simulation system, or use the method described here. Here, the image file rootimage-0.12-hd.img included in the following package is taken as an example to illustrate the method of accessing the file system in the first partition.

http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.zip

Here you need to use the loop device settings and control commands losetup. This command is mainly used to associate a normal file or a block device with a loop device, or to release a loop device and query the status of a loop device. For a detailed description of this command, please refer to the Linux online manual page.

First execute the following command to associate the rootimage-0.12-hd file with loop1, and use the fdisk command to view the partition information.

```
[root@www linux-0.12-080324]# losetup /dev/loop1 rootimage-0.12-hd
[root@www linux-0.12-080324]# fdisk /dev/loop1
Command (m for help): x
Expert command (m for help): p
Disk /dev/loop1: 16 heads, 63 sectors, 487 cylinders

Nr AF  Hd Sec  Cyl  Hd Sec  Cyl     Start      Size   ID
```

```
 1 80   1   1    0  15  63  130         1     132047 81
 2 00   0   1  131  15  63  261    132048     132048 81
 3 00   0   1  262  15  63  392    264096     132048 81
 4 00   0   1  393  15  63  474    396144      82656 82
Expert command (m for help): q

[root@www linux-0.12-080324]# _
```

As can be seen from the partition information given by fdisk above, the Image file contains 3 MINIX partitions (ID=81) and 1 swap partition (ID=82). If we need to access the contents of the first partition, write down the starting sector number of the partition (that is, the contents of the 'Start' column in the partition table). If you need to access the hard disk Image of another partition, then you just need to remember the starting sector number of the relevant partition.

Next, we first use the "-d" option of losetup to unlink the rootimage-0.12-hd file from loop1 and then re-associate it to the beginning of the first partition of the image file. This requires the use of the "-o" option of losetup, which indicates the associated starting byte offset position. As can be seen from the above partition information, the starting offset position of the first partition here is 1 * 512 bytes. Therefore, after re-associating the first partition with loop1, we can use the mount command to access the files.

```
[root@www linux-0.12-080324]# losetup -d /dev/loop1
[root@www linux-0.12-080324]# losetup -o 512 /dev/loop1 rootimage-0.12-hd
[root@www linux-0.12-080324]# mount -t minix /dev/loop1 /mnt
[root@www linux-0.12-080324]# cd /mnt
[root@www mnt]# ls
bin  etc  home   MCC-0.12  mnt1    root  usr
dev  hdd  image  mnt       README  tmp   vmlinux
[root@www mnt]# _
```

After the access to the file system in the partition is over, finally unmount the file system and disassociate.

```
[root@www mnt]# cd
[root@www ~]# umount /dev/loop1
[root@www ~]# losetup -d /dev/loop1
[root@www ~]# _
```

17.4 Compiling and running the simple kernel

A simple multitasking kernel sample program is given in the previous chapter on the 80386 protection mode and its programming in Chapter 4, which we call the Linux 0.00 system. It contains two tasks running on privilege level 3, which will cycle through the characters A and B on the screen and perform task switching operations under clock timing control. The packages that have been configured to run in the Bochs simulation environment are given on the book's website:

```
    http://oldlinux.org/Linux.old/bochs/linux-0.00-050613.zip
    http://oldlinux.org/Linux.old/bochs/linux-0.00-041217.zip
```

We can download any of the above files to experiment. The program given in the first package is the same as described here. The program in the second package is slightly different (the kernel head code runs directly at the 0x10000 address), but the principle is exactly the same. Here we will use the program in the first software package as an example to illustrate the experiment. The software of the second package is left to the reader for experimental analysis. After unpacking the linux-0.00-050613.zip package with a decompression software, a linux-0.00 subdirectory will be generated in the current directory. We can see that this package contains the following files:

```
1.  linux-0.00.tar.gz      - Compressed source file;
2.  linux-0.00-rh9.tar.gz  - Compressed source file;
3.  Image                  - Kernel boo image file;
4.  bochsrc-0.00.bxrc      - Bochs configuration file;
5.  rawrite.exe            - The program to write an Image to a floppy disk under Windows.
6.  README                 - Package documentation;
```

The first file, linux-0.00.tar.gz, is a compressed source file for the kernel example. It can be compiled in the Linux 0.12 system to generate a kernel image file. The second is also a compressed source file for the kernel example, but the source program can be compiled under RedHat 9 Linux. The third file Image is a 1.44MB floppy image file of the runnable code compiled by the source program. The fourth file, bochsrc-0.00.bxrc, is the Bochs configuration file used when running in the Bochs environment. If you have installed the PC emulation software Bochs on your system, you can run the kernel code in the Image by double-clicking on the bochsrc-0.00.bxrc filename. The fifth is a utility program under DOS or Windows for writing image files to a floppy disk. We can run the RAWRITE.EXE program directly and write the kernel image file here to a 1.44MB floppy disk to run.

The source code for the kernel example given above is included in the linux-0.00-tar.gz file. Decompressing this file will generate a subdirectory containing the source files, including a makefile in addition to the boot.s and head.s programs. The beginning part of the 'boot' file generated by as86/ld86 compilation and linking contains 32 bytes of MINIX executable file header information, and the beginning part of the 'head' file compiled and linked by as/ld includes 1024 bytes of a.out header data, so when making the kernel 'Image' file, we need to remove the header information. We can use two 'dd' commands to remove the header information and then combine them into a kernel image Image file.

The Image file is generated by executing the make command directly in the source code directory. If you have already executed the make command, then execute 'make clean' first and then execute the make command.

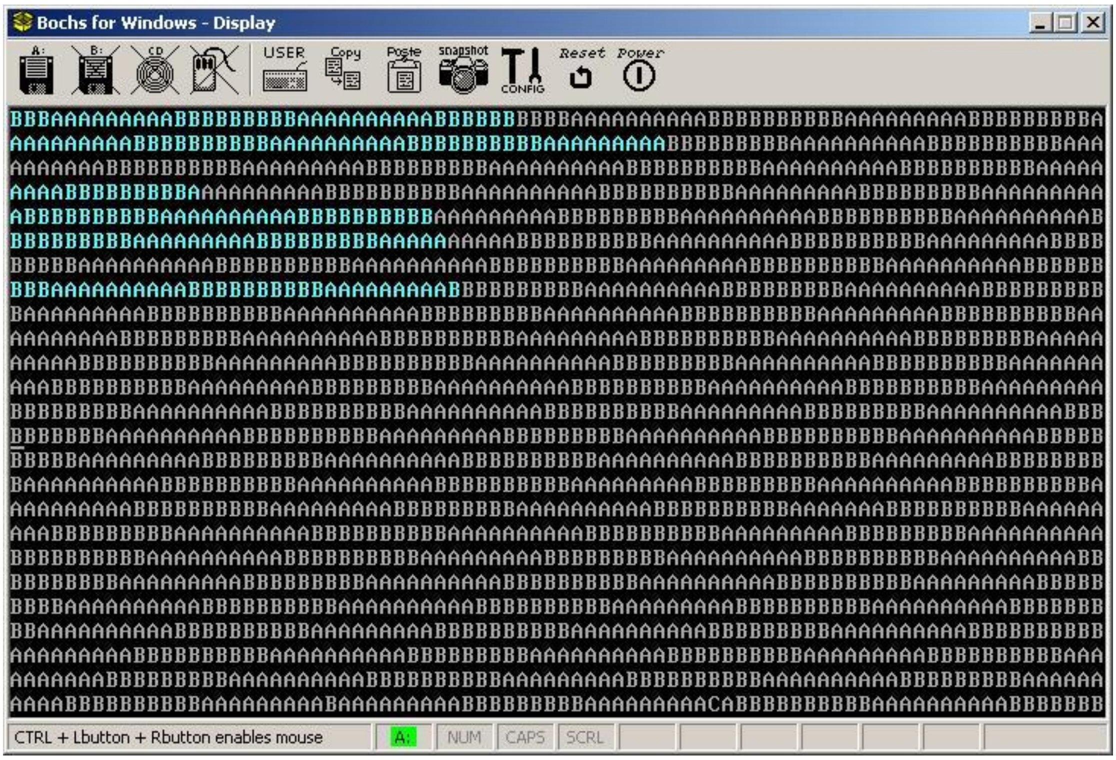
```
[/usr/root/linux-0.0]# ls -l
total 9
-rw-------   1 root     root          487 Jun 12 19:25 Makefile
-rw-------   1 root     4096         1557 Jun 12 18:55 boot.s
-rw-------   1 root     root         5243 Jun 12 19:01 head.s

[/usr/root/linux-0.0]# make
```

```
as86 -0 -a -o boot.o boot.s
ld86 -0 -s -o boot boot.o
gas -o head.o head.s
gld -s -x -M head.o  -o system > System.map

dd bs=32 if=boot of=Image skip=1
16+0 records in
16+0 records out

dd bs=512 if=system of=Image skip=2 seek=1
16+0 records in
16+0 records out
[/usr/root/linux-0.0]#
```

To copy the Image file to the A disk image file, we can execute the command 'make disk' as follows. However, before executing this command, if you are compiling the Linux 0.12 system under Bochs, please copy and save your boot image disk file (for example, bootimage-0.12-hd), so that you can restore the Linux 0.12 system after the test. Image file.

```
[/usr/root/linux-0.0]# ls
Image       System.map  boot.o      head.o      system
Makefile    boot        boot.s      head.s
[/usr/root/linux-0.0]# make disk
dd bs=8192 if=Image of=/dev/fd0
1+1 records in
1+1 records out
sync;sync;sync
[/usr/root/linux-0.0]#
```

To run this simple kernel example, we can use the mouse to directly click the RESET icon on the Bochs window. Its operation is shown in the figure below. After that, if you want to resume running the Linux 0.12 system, please overwrite the startup file with the image file you just saved.


Bochs for Windows - Display
A:
B:
CD
USER
Copy
Paste
snapshot
CONFIG
Reset
Power
CTRL + Lbutton + Rbutton enables mouse
A:
NUM
CAPS
SCRL

```
BBBAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBA
AAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAA
AAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBAAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAA
AAAABBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAA
ABBBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAAAB
BBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAABBBB
BBBBAAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAAAABBBBB
BBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBB
BAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBBA
AAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAAABBBBBBBBBBAAAA
AAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBAAAAAAAAAAABBBBBBBBAAAAAAAAAAABBBBBBBBBBAAAAAAA
AAABBBBBBBBBBAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAA
BBBBBBBBAAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBB
BBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBAAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBB
BBBBBAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBB
BAAAAAAAAAABBBBBBBBBAAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBAAAAAAAAAAABBBBBBBBBBA
AAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBAAAAAAAAAAABBBBBBBBBAAAAAAAABBBBBBBBBBBAAAAA
AAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAA
BBBBBBBBBAAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBAAAAAAAAAAABBBBBBBBBBAAAAAAAAAABB
BBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAAAABBBB
BBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBBAAAAAAAAABBBBBBBBBAAAAAAAAAAABBBBBBB
BBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBB
AAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBAAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBAAA
AAAAAAABBBBBBBBAAAAAAAAAAABBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAA
AAAABBBBBBBBBBAAAAAAAAABAAAAAAAABBBBBBBBBBAAAAAAAAACABBBBBBBBBBAAAAAAAAAABBBBBBB
```

Figure 17-3 Screen display of simple kernel runtime

17.5 Using Bochs to Debug the Kernel

Bochs has very powerful operating system kernel debugging capabilities, which is one of the main reasons why Bochs was chosen as our preferred experimental environment. For a description of Bochs debugging features, see Section 17.2 above, which is based on the Linux 0.12 kernel to illustrate the basic methods of Bochs debugging operations in the Windows environment.

17.5.1Running the Bochs Debugger

Assume that the Bochs system has been installed in the directory "C:\Program Files\Bochs-2.3.6\", and the Bochs configuration file name for the Linux 0.12 system is "bochsrc-0.12-hd.bxrc". Now we create a simple batch file run.bat in the directory containing the kernel Image file, which reads as follows:

```
"C:\Program Files\Bochs-2.3.6\bochsdbg" -q -f bochsrc-0.12-hd.bxrc
```

Among them, bochsdbg is the debugging execution program of Bochs. The parameter "-q" means quick start (skip configuration interface), and the parameter "-f" means followed by a configuration file name. If the parameter is "-h", the program will display help information for all optional parameters. Run the batch command to enter the debugging environment. At this point, the main display window of Bochs is blank, and the control window will display the following similar content:

```
C:\Linux-0.12>"C:\Program Files\Bochs-2.3.6\bochsdbg" -q -f bochsrc-0.12-hd.bxrc
00000000000i[APIC?] local apic in  initializing
========================================================================
                       Bochs x86 Emulator 2.3.6
                Build from CVS snapshot on December 24, 2007
========================================================================
00000000000i[     ] reading configuration from bochsrc-hd-new.bxrc
00000000000i[     ] installing win32 module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
Next at t=0
(0) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
<bochs:1>
```

At this point the Bochs debug system is ready to start running and the CPU execution pointer has been pointed to the instruction at address 0x000fffff0 in the ROM BIOS. Where "<bochs:1>" is the command line input prompt, where the number indicates the current command serial number. By typing the "help" command after the command prompt, we can list the basic commands for debugging the system. To find out how to use a command in detail, type the 'help' command followed by a specific command enclosed in single quotes, for example: " help 'vbreak' ". See below.

```
<bochs:1> help
help - show list of debugger commands
help ’command’- show short command description
-*- Debugger control -*-
    help, q|quit|exit, set, instrument, show, trace-on, trace-off,
    record, playback, load-symbols, slist
-*- Execution control -*-
    c|cont, s|step|stepi, p|n|next, modebp
-*- Breakpoint management -*-
    v|vbreak, lb|lbreak, pb|pbreak|b|break, sb, sba, blist,
    bpe, bpd, d|del|delete
-*- CPU and memory contents -*-
    x, xp, u|disas|disassemble, r|reg|registers, setpmem, crc, info, dump_cpu,
    set_cpu, ptime, print-stack, watch, unwatch, ?|calc
<bochs:2> help ’vbreak’
help vbreak
vbreak seg:off - set a virtual address instruction breakpoint
<bochs:3>
```

Some of the more commonly used commands are listed below. A complete list of all debug commands can be found in Bochs' own help file (internal-debugger.html) or refer to the online help information ("help" command).

1. Execute control commands. Control single or multi-step execution of instructions.

```
c                 Continuous execution
```

stepi [count]	Execute 'count' instructions, the default is 1.
si [count]	lbid.
step [count]	lbid.
s [count]	lbid.
p	Similar to 's', but with interrupt and function instructions as single-step execution, that is, single-step execution of the interrupt or functions.
n (or next)	lbid.
Ctrl-C	Stop execution and go back to the command line prompt.
Ctrl-D	Exit Bochs if you type the command at an empty command line prompt.
quit, q	Exit debugging.

2. Breakpoint setting commands. Where 'seg', 'off' and 'addr' can be hexadecimal numbers starting with '0x', or decimal numbers or octal numbers starting with '0'.

vbreak seg:off	Set the instruction breakpoint on the virtual address.
vb seg:off	lbid.
lbreak addr	Set the instruction breakpoint on the linear address.
lb addr	lbid.
pbreak [*] addr	Set the instruction breakpoint on the physical address. Where '*' is an option for compatibility with GDB.
pb [*] addr	lbid.
break [*] addr	lbid.
b [*] addr	lbid.
info break	Displays the status of all current breakpoints.
delete n	Delete a breakpoint.
del n	lbid.
d n	lbid.

3. Memory operation commands

x /nuf addr	Check the memory contents at the linear address 'addr'. If 'addr' is not specified, the default is the next address.
xp /nuf addr	Check the memory contents at the physical address 'addr'.

The optional parameters 'n', 'u' and 'f' can be:

n The number of memory units to display. The default value is 1.

u Indicates the unit size, the default selection is the following 'w':
b(Bytes),1 byte; h(Halfwords),2 bytes; w(Words),4 bytes; g(Giantwords), 8 bytes.
Note: These abbreviations are different from those of Intel, mainly to be consistent with the GDB debugger representation.

f Display format, the default selection is 'x':
x (hex); d (decimal); u (unsigned); o (octal);t (binary);
c (char) Display chars. If it cannot be displayed as a char, the code is displayed directly.

crc addr1 addr2	Displays the CRC checksum of physical memory from addr1 to addr2.
info dirty	Displays the physical memory page that has been modified since the last time this command was executed. Only the first 20 bytes of the page are displayed.

4. Information display and CPU register operation commands

info program	Displays the execution status of the program.

```
info registers Displays the CPU registers (no floating point registers).
info break     Displays the current breakpoint setting status.
set $reg = val Modify the contents of the CPU register (except segment and flag register).
               For example, set $eax = 0x01234567; set $edx = 25
dump_cpu       Displays all status information of the CPU.
set_cpu        Set all status information of the CPU.
```

The content format displayed by the "dump_cpu" and "set_cpu" commands is:

```
    "eax:0x%x\n"
    "ebx:0x%x\n"
    "ecx:0x%x\n"
    "edx:0x%x\n"
    "ebp:0x%x\n"
    "esi:0x%x\n"
    "edi:0x%x\n"
    "esp:0x%x\n"
    "eflags:0x%x\n"
    "eip:0x%x\n"
    "cs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
    "ss:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
    "ds:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
    "es:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
    "fs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
    "gs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
    "ldtr:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
    "tr:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"
    "gdtr:base=0x%x, limit=0x%x\n"
    "idtr:base=0x%x, limit=0x%x\n"
    "dr0:0x%x\n"
    "dr1:0x%x\n"
    "dr2:0x%x\n"
    "dr3:0x%x\n"
    "dr4:0x%x\n"
    "dr5:0x%x\n"
    "dr6:0x%x\n"
    "dr7:0x%x\n"
    "tr3:0x%x\n"
    "tr4:0x%x\n"
    "tr5:0x%x\n"
    "tr6:0x%x\n"
    "tr7:0x%x\n"
    "cr0:0x%x\n"
    "cr1:0x%x\n"
    "cr2:0x%x\n"
    "cr3:0x%x\n"
    "cr4:0x%x\n"
    "inhibit_int:%u\n"
    "done\n"
```

among them:

- 's' means a selector;
- 'dl' is the low 4-byte of the segment descriptor in the selector shadow register;
- 'dh' is the high 4-byte of the segment descriptor in the selector shadow register;
- 'valid' indicates whether a valid shadow descriptor is being stored in the segment register;
- 'inhibit_int' is an instruction delay interrupt flag. If set, it means that the instruction that has just been executed by the previous one is an instruction that delays the CPU from accepting the interrupt (for example, STI, MOV SS);

In addition, when the "set_cpu" command is executed, any error will be reported using the format "Error: ...". These error messages may appear after each input line or after the last "done" is displayed. If the "set_cpu" command is successfully executed, the command will display "OK" to end the command.

5. Disassembly command

```
disassemble start end    Disassemble instructions within a given linear address range.
disas
u
```

The following are some of the new commands from Bochs, but the commands involving file names in the Windows environment may not work properly.

- record *filename* -- Write your input command sequence to the file 'filename' during execution. The file will contain lines of the form "%s %d %x". The first parameter is the event type; the second is the timestamp; the third is the data of the related event.
- playback *filename* -- The execution command is played back using the contents of the file 'filename'. You can also type other commands directly in the control window. Each event in the file will be played back, and the playback time will be counted relative to the time the command was executed.
- print-stack [num words] -- Display num 16-bit words at the top of the stack. The default value of num is 16. When the base address of the stack segment is 0, the command can be used normally only in protected mode.
- load-symbols [global] *filename* [offset] -- Load symbol information from the file 'filename'. If the keyword global is given, then all symbols will be visible in the context before the symbol was loaded. The 'offset' (default is 0) is added to each symbol item. The symbol information is loaded into the context of the currently executing code. The format of each line in the symbol file filename is "%x %s". The first value is the address and the second is the symbol name.

In order for Bochs to directly simulate execution to the beginning of the Linux bootloader, we can first set a breakpoint at 0x7c00 using the breakpoint command and then let the system continue to run to 0x7c00 to stop. The sequence of commands executed is as follows:

```
<bochs:3> vbreak 0x0000:0x7c00
<bochs:4> c
(0) Breakpoint 1, 0x7c00 (0x0:0x7c00)
Next at t=4409138
(0) [0x00007c00] 0000:7c00 (unk. ctxt): mov ax, 0x7c0            ; b8c007
<bochs:5>
```

At this point, the CPU executes the first instruction at the beginning of the boot.s program, and the Bochs main window will display some information such as "Boot From floppy...". Now we can follow the debugger by stepping through the command 's' or 'n' (not tracking into the subroutine). Bochs' breakpoint setting commands, disassembly commands, information display commands, etc. can be used to assist in our debugging operations. Here are some examples of common commands:

```
<bochs:8> u /10                                   # Disassemble 10 instructions.
00007c00: (                   ): mov ax, 0x7c0             ; b8c007
00007c03: (                   ): mov ds, ax                ; 8ed8
00007c05: (                   ): mov ax, 0x9000            ; b80090
00007c08: (                   ): mov es, ax                ; 8ec0
00007c0a: (                   ): mov cx, 0x100             ; b90001
00007c0d: (                   ): sub si, si                ; 29f6
00007c0f: (                   ): sub di, di                ; 29ff
00007c11: (                   ): rep movs word ptr [di], word ptr [si] ; f3a5
00007c13: (                   ): jmp 9000:0018             ; ea18000090
00007c18: (                   ): mov ax, cs                ; 8cc8
<bochs:9> info r                                 # View the contents of the current register
eax            0xaa55           43605
ecx            0x110001         1114113
edx            0x0              0
ebx            0x0              0
esp            0xfffe           0xfffe
ebp            0x0              0x0
esi            0x0              0
edi            0xffe4           65508
eip            0x7c00           0x7c00
eflags         0x282            642
cs             0x0              0
ss             0x0              0
ds             0x0              0
es             0x0              0
fs             0x0              0
gs             0x0              0
<bochs:10> print-stack                           # Display the current stack
   0000fffe [0000fffe]  0000
   00010000 [00010000]  0000
   00010002 [00010002]  0000
   00010004 [00010004]  0000
   00010006 [00010006]  0000
   00010008 [00010008]  0000
   0001000a [0001000a]  0000
...
<bochs:11> dump_cpu                              # Displays all registers in the CPU.
eax:0xaa55
ebx:0x0
ecx:0x110001
edx:0x0
ebp:0x0
esi:0x0
```

```
edi:0xffe4
esp:0xfffe
eflags:0x282
eip:0x7c00
cs:s=0x0, dl=0xffff, dh=0x9b00, valid=1        # s-selector;dl,dh - low & high 4-byte of desc.
ss:s=0x0, dl=0xffff, dh=0x9300, valid=7
ds:s=0x0, dl=0xffff, dh=0x9300, valid=1
es:s=0x0, dl=0xffff, dh=0x9300, valid=1
fs:s=0x0, dl=0xffff, dh=0x9300, valid=1
gs:s=0x0, dl=0xffff, dh=0x9300, valid=1
ldtr:s=0x0, dl=0x0, dh=0x0, valid=0
tr:s=0x0, dl=0x0, dh=0x0, valid=0
gdtr:base=0x0, limit=0x0
idtr:base=0x0, limit=0x3ff
dr0:0x0
dr1:0x0
dr2:0x0
dr3:0x0
dr6:0xffff0ff0
dr7:0x400
tr3:0x0
tr4:0x0
tr5:0x0
tr6:0x0
tr7:0x0
cr0:0x60000010
cr1:0x0
cr2:0x0
cr3:0x0
cr4:0x0
inhibit_mask:0
done
<bochs:12>
```

Since the 32-bit code of the Linux 0.1X kernel is stored from the absolute physical address 0, if you want to execute directly to the beginning of the 32-bit code (that is, at the beginning of the head.s program), we can set a breakpoint at linear address 0x0000, and run the command 'c' to execute to that location.

In addition, when you press the Enter key directly at the command prompt, the previous command will be executed repeatedly; pressing the up arrow will display the previous command. Please refer to the 'help' command for other commands.

17.5.2 Locating Variables or Data Structures in the Kernel

A 'system.map' file is generated when the kernel is compiled. This file lists the global variables in the kernel Image (bootimage) file and the offset address locations of the local variables in each module. After the kernel is compiled, you can use the file export method described above to extract the 'system.map' file to the host environment (windows). See Chapter 3 for the detailed purpose and role of the 'system.map' file. Some of the contents of the 'system.map' sample file are shown below. Using this file, we can quickly locate a variable or jump to the specified function code in the Bochs debug system.

```
...
Global symbols:

  _dup: 0x16e2c
  _nmi: 0x8e08
  _bmap: 0xc364
  _iput: 0xc3b4
  _blk_dev_init: 0x10ed0
  _open: 0x16dbc
  _do_execve: 0xe3d4
  _con_init: 0x15ccc
  _put_super: 0xd394
  _sys_setgid: 0x9b54
  _sys_umask: 0x9f54
  _con_write: 0x14f64
  _show_task: 0x6a54
  _buffer_init: 0xd1ec
  _sys_settimeofday: 0x9f4c
  _sys_getgroups: 0x9edc
...
```

Similarly, since the 32-bit code of the Linux 0.1X kernel is stored from the absolute physical address 0, the offset position of the global variable in 'system.map' is the linear address position in the CPU. So we can set breakpoints directly at the variable or function name location of interest and let the program execute continuously to the specified location. For example, if we want to debug the function buffer_init(), we can see that it is located at 0xd1ec from the 'system.map' file. At this point we can set a linear address breakpoint there and execute the command 'c' to let the CPU execute to the beginning of the specified function, as shown below.

```
<bochs:12> lb 0xd1ec                              # Set a linear address breakpoint.
<bochs:13> c                                      # Continuous execution.
(0) Breakpoint 2, 0xd1ec in ?? ()
Next at t=16689666
(0) [0x0000d1ec] 0008:0000d1ec (unk. ctxt): push ebx                 ; 53
<bochs:14> n                                      # Next instruction.
Next at t=16689667
(0) [0x0000d1ed] 0008:0000d1ed (unk. ctxt): mov eax, dword ptr ss:[esp+0x8] ; 8b442408
<bochs:15> n                                      # Next instruction.
Next at t=16689668
(0) [0x0000d1f1] 0008:0000d1f1 (unk. ctxt): mov edx, dword ptr [ds:0x19958] ; 8b1558990100
<bochs:16>
```

Program debugging is a skill that requires more practice to make it happen. Some of the basic commands described above need to be combined to give you the flexibility to observe the overall environment of kernel code execution.

17.6 Creating a Disk Image File

A disk image file is a complete image of the data on a floppy disk or hard disk and is saved as a file. The format of the information stored in the disk image file is exactly the same as the format of the information stored on the real disk. An empty disk image file is a file with the same capacity as the disk we created but with a content of all 0s. These empty image files are like new floppy disks or hard disks that have just been purchased, and they need to be partitioned or / and formatted before they can be used.

Before making a disk image file, we first need to determine the capacity of the image file we created. For floppy image files, the capacity of various specifications (1.2MB or 1.44MB) is fixed. So here is how to determine the capacity of the hard disk image file you need. The structure of a conventional hard disk consists of stacked metal discs. The upper and lower sides of each disc are used to store data, and the entire surface is divided into tracks by concentric circles, or Cylinders. A head is required on each side of each disc to read and write data on the disc. As the disc rotates, the head only needs to be moved radially to move over any track, thereby providing access to all valid positions on the surface of the disc. Each track is divided into sectors, and the sector size is generally composed of 256 - 1024 bytes. For most systems, the sector size is typically 512 bytes. A typical hard disk structure is shown in Figure 17-4.

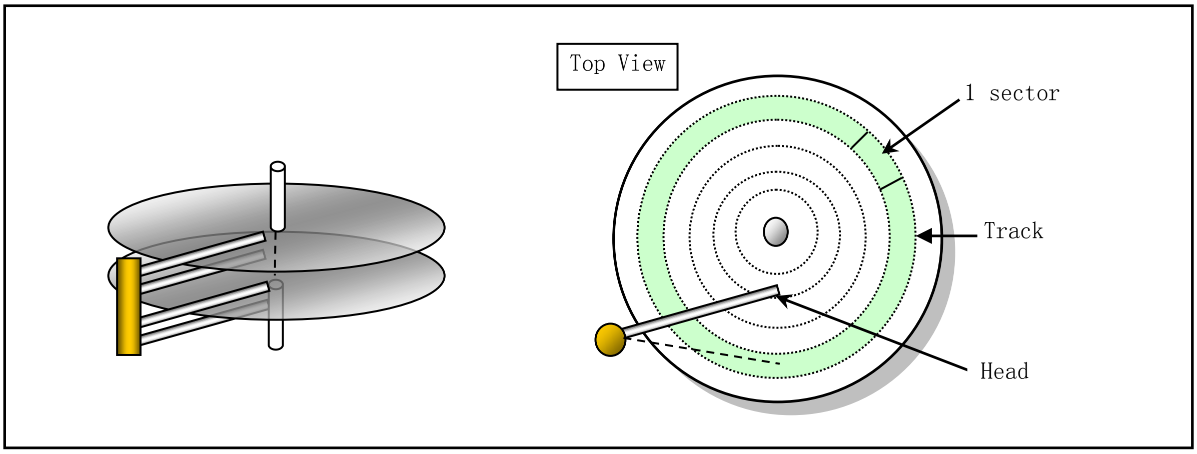


Figure 17-4 Typical hard disk internal structure

The figure shows a hard disk structure with two metal disks with four physical heads. The maximum number of cylinders contained is determined at the time of production. When the hard disk is partitioned and formatted, the magnetic media on the surface of the disk is initialized to data in a specified format such that each track (or cylinder) is divided into a specified number of sectors. Therefore the total number of sectors of this hard disk is:

Total sectors = number of physical tracks * number of physical heads * number of sectors per track

The parameters such as the track and head used in an operating system are different from those of the actual physical parameters in a hard disk, which are called logical parameters. However, the total number of sectors calculated by these parameters is definitely the same as that calculated by the physical parameters of the hard disk. Since the performance and capacity of the hardware device are not developed so rapidly when designing the PC system, some of the ROM BIOS's representation of the hard disk parameters are too small to meet the physical parameters of the actual hard disk. Therefore, the current remedy commonly used in operating systems or machine BIOS is to properly adjust the number of tracks, the number of heads, and the number of sectors per track to ensure compatibility and parameter representation constraints while ensuring that the total number of sectors of the hard disk is constant. The "translation" option in the Bochs configuration file for hard disk device parameters is also set for this purpose.

When we made the hard disk Image file for the Linux 0.1X system, considering the small amount of its own code, and the maximum capacity of the MINIX 1.5 file system used is 64MB, the maximum size of each hard disk partition can only be 64MB. In addition, the Linux 0.1X system does not yet support extended partitions, so for a hard disk Image file, there are up to 4 partitions. Therefore, the maximum capacity of a hard disk Image file that can be used by a Linux 0.1X system is 64 x 4 = 256MB. In the following description, we will illustrate the creation of a hard disk Image file with 4 partitions and 60MB per partition.

For a USB flash disk, we can treat it as a hard disk. For a floppy disk, we can think of it as a non-partitioned ultra-small hard disk with a fixed number of tracks (number of cylinders), number of heads, and number of sectors per track. For example, a floppy disk with a capacity of 1.44 MB is 80 tracks, 2 heads and 18 sectors per track, and 512 bytes per sector. The total number of sectors is 2880 and the total capacity is 80 x 2 x 18 x 512 = 1474560 bytes. Therefore, all the methods described below for making hard disk image files can be used to create USB disk and floppy image files. For the convenience of the description, we refer to all disk image files as Image files unless otherwise specified.

17.6.1 Using Bochs' own Image Creation Tool

The Bochs system comes with an disk Image creation tool 'bximage.exe', which can be used to create empty Image files for floppy disks and hard disks. When you run it and the Image creation interface appears, the program will first prompt you to select the type of Image you want to create (hard disk hd or floppy disk fd). If you create a hard disk, you will also be prompted to enter the mode type of the hard disk image, usually only need to select its default value 'flat'. Then enter the image size you need to create. The program will display the corresponding hard disk parameter values: number of cylinders (number of tracks), number of heads, and number of sectors per track, and ask for the name of the image file. After the Image file is generated, the program displays a configuration message for setting the hard disk parameters in the Bochs configuration file. You can write down this information and edit it into the configuration file. Below is the process of creating a 256MB hard drive Image file.

```
========================================================================
                                bximage
                  Disk Image Creation Tool for Bochs
        $Id: bximage.c,v 1.19 2006/06/16 07:29:33 vruppert Exp $
========================================================================
Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd]

What kind of image should I create?
Please type flat, sparse or growing. [flat]

Enter the hard disk size in megabytes, between 1 and 32255
[10] 256

I will create a 'flat' hard disk image with
  cyl=520
  heads=16
  sectors per track=63
  total sectors=524160
  total size=255.94 megabytes

What should I name the image?
[c.img] hdc.img
Writing: [] Done.
I wrote 268369920 bytes to (null).

The following line should appear in your bochsrc:
  ata0-master: type=disk, path="hdc.img", mode=flat, cylinders=520, heads=16, spt=63

Press any key to continue
```

If you already have a hard disk Image file with the required capacity, you can also copy the file directly to generate another Image file, and then you can process the file according to your own requirements. The process for creating a floppy Image file is similar to the above, except that you will be prompted to select a floppy disk type. Similarly, if you already have other floppy Image files, you can use the direct copy method.

17.6.2 Creating an Image File Using the dd Command on a Linux

The dd command is a command line tool on Linux systems, mainly used to copy files and convert file data formats. As explained above, the Image file just created is an empty file with all contents 0, but its capacity is consistent with the requirements. So we can first calculate the number of sectors of the Image file that requires the capacity, and then use the 'dd' command to generate the corresponding Image file.

For example, we want to create a hard disk Image file with a cylinder number of 520, a head count of 16, and a sector number of 63 per track. The total number of sectors is: 520 * 16 * 63 = 524160, then the command is:

```
  dd if=/dev/zero of=hdc.img bs=512 count=524160
```

The parameter ’if’ is the name of the copied input file, ’/dev/zero’ is the device file that can generate 0 value bytes; ’of’ is the output file name generated; ’bs’ specifies the size of the copied data block; ’count’ is the number of data blocks copied. For a 1.44MB floppy Image file, the number of sectors is 2880, so the command is:

```
  dd if=/dev/zero of=diska.img bs=512 count=2880
```

17.6.3 Creating a floppy disk image file in DOS format using WinImage

WinImage is a DOS format Image file access and creation tool. After associating with the software, double-click the icon of the DOS floppy Image file and you can browse, delete or add files to it. In addition, it can also be used to browse the contents of the CDROM iso file. When you use WinImage to create a floppy disk Image, you can generate an Image file with DOS format. Methods as below:

a) Run WinImage. Select the "Options->Settings" menu and select the Image Settings page. Set Compression to "None" (that is, pull the indicator to the far left);
b) Create an Image file. Select the menu "File->New" and a floppy disk selection box will pop up.

Please choose a format with a capacity of 1.44MB;

c) Select the boot sector property menu item "Image->Boot Sector properties" and click the MS-DOS button in the dialog box;

d) Save the file.

Note that you must select "All files (*.*)" in the "Save Type" dialog box. Otherwise, the created Image file will contain some WinImage information, which will cause the Image file to not work properly under Bochs. We can determine if the newly created Image meets the requirements by looking at the file size. The standard 1.44MB floppy disk should have a capacity of 1474560 bytes. If the new Image file size is larger than this value, please re-create or use the binary editor such as Notepad++ (requires Hex-Editor plugin) to delete the extra bytes. The method of deleting the operation is as follows:

- Open the Image file with Notepad++ and run the plugin Hex-Editor. According to the 511, 512 bytes of the disk image file are 55, AA two hexadecimal numbers, we push back 512 bytes, delete all the previous bytes. At this point, for a disk using MSDOS 5.0 as the boot, the first few bytes of the file should be similar to "EB 3C 90 4D ...".
- Then pull down the right scroll bar and move to the end of the img file. Delete all data after "...F6 F6 F6". Usually it is to delete all data starting from 0x168000. The last line when the operation is completed should be a complete line "F6 F6 F6...". Save and exit to use the Image file.

17.7 Making a Root File System

The goal of this section is to create a root file system on the hard disk. Although the floppy disk and the hard disk root file system Image file can be downloaded from the oldlinux.org website, the creation process is described in detail here for your reference. In the process of establishing, you can also refer to the installation article written by Mr. Linus: INSTALL-0.11. Before making the root file system disk, we first download the rootimage-0.12 and bootimage-0.12 image files (please download the latest files):

```
http://oldlinux.org/Linux.old/images/bootimage-0.12-20040306
http://oldlinux.org/Linux.old/images/rootimage-0.12-20040306
```

Modify these two file names into easy-to-remember names bootimage-0.12 and rootimage-0.12, and create a subdirectory named Linux-0.12 for them. During the making process, we need to copy some of the executables in the rootimage-0.12 floppy disk and boot the simulation system using the bootimage-0.12 boot disk. So before you start working on the root file system, you first need to confirm that you have been able to run the minimum Linux system consisting of these two floppy Image files.

17.7.1 Root File System and Root File Device

When the Linux boot starts, the default file system containing the root directory is the root file system. The following subdirectories and files are generally included in the root directory:

```
♦ etc/      This directory mainly contains some configuration files, such as 'rc' file;
♦ dev/      Contains device special files for operating the device with files;
```

♦ bin/	Store system execution programs. Such as sh, mkfs, fdisk, etc.;
♦ usr/	Store library functions, manuals, and other files;
♦ usr/bin/	Store commands commonly used by users;
♦ var/	Used to store data when the system is running or for information such as logs.

The device that holds the file system is the file system device. For example, for the Windows operating system, the hard disk C drive is the file system device, and the files stored on the hard disk according to certain rules constitute the file system. Windows can have file systems in formats such as NTFS or FAT32, while the file system supported by the Linux 0.1X kernel is the MINIX 1.0 file system.

When the Linux boot disk loads the root file system, the root file system is loaded from the specified device according to the root file system device number in a word (ROOT_DEV) at the 509th and 510th bytes of the boot sector on the boot disk. If the device number is 0, it means that the root file system needs to be loaded from the current drive where the boot disk is located. If the device number is a hard disk partition device number, the root file system is loaded from the specified hard disk partition. The hard disk device numbers supported by the Linux 0.1X kernel are shown in Table 17-3.

Table 17-3 Hard disk logical device number

Device nr	Device file	Description
0x0300	/dev/hd0	Represents the entire first hard driv
0x0301	/dev/hd1	The first partition of the first disk
0x0302	/dev/hd2	The second partition of the first disk
0x0303	/dev/hd3	The third partition of the first disk
0x0304	/dev/hd4	The fourth partition of the first disk
0x0305	/dev/hd5	Represents the entire second hard driv
0x0306	/dev/hd6	The first partition of the second disk
0x0307	/dev/hd7	The second partition of the second disk
0x0308	/dev/hd8	The third partition of the second disk
0x0309	/dev/hd9	The fourth partition of the second disk

If the device number is a floppy device number, the kernel will load the root file system from the floppy drive specified by the device number. The floppy device numbers used in the Linux 0.1X kernel are shown in Table 17-4. For the calculation method of the floppy disk drive device number, please refer to the description after the floppy.c program in Chapter 9.

Table 17-4 Floppy drive logic device number

Device Nr	Device file	Description
0x0208	/dev/at0	1.2MB A drive
0x0209	/dev/at1	1.2MB B drive
0x021c	/dev/fd0	1.44MB A drive
0x021d	/dev/fd1	1.44MB B drive

17.7.2 Creating a File System

For the hard disk Image file created above, we must also partition and create a file system on it before it can be used. The usual practice is to attach the hard disk Image file to be processed to the existing simulation system (such as SLS Linux mentioned above) under Bochs , and then use the commands in the simulation system to process the new Image file. The following assumes that you have installed the SLS Linux emulation system and it is stored in a subdirectory named SLS-Linux. We use it to partition the 256MB hard disk image file hdc.img created above and create a MINIX file system on it. We will create a partition in this Image file and create a MINIX file system. The steps we performed are as follows:

1. Create a subdirectory named Linux-0.12 in the SLS-Linux directory and move the hdc.img file to it.
2. Go to the SLS-Linux directory and edit the Bochs configuration file 'bochsrc.bxrc' for the SLS Linux system. Add the configuration parameter line of our hard disk Image file under the option 'ata0-master':

```
    ata0-slave:type=disk, path=..\Linux-0.12\hdc.img, cylinders=520, heads=16, spt=63
```

3. Exit the editor. Double-click the icon for the 'bochsrc.bxrc' file to run the SLS Linux emulation system. Type 'root' at the Login prompt and press Enter. If Bochs does not work properly at this time, generally because the configuration file information is incorrect, please re-edit the configuration file.
4. Use fdisk to create 1 partition in the hdc.img file. Below is the sequence of commands to create the first partition. The process of creating another three partitions is similar. Since the partition type established by SLS Linux by default is 81 type (Linux/MINIX) that supports the MINIX2.0 file system, you need to use the fdisk t command to change the type to 80 (Old MINIX) type. Please note here that we have hooked hdc.img to the second hard drive under the SLS Linux system. According to the Linux 0.1X hard disk naming rules, the overall device name of the hard disk should be /dev/hd5. However, since the Linux kernel version 0.95, the naming rules for the hard disk have been changed to the currently used rules, so the device name of the second hard disk under SLS Linux is /dev/hdb.

```
[/]# fdisk /dev/hdb
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-520): 1
Last cylinder or +size or +sizeM or +sizeK (1-520): +63M

Command (m for help): t
Partition number (1-4): 1
Hex code (type L to list codes): L
 0  Empty            8  AIX            75  PC/IX          b8  BSDI swap
 1  DOS 12-bit FAT   9  AIX bootable   80  Old MINIX      c7  Syrinx
 2  XENIX root       a  OPUS           81  Linux/MINIX    db  CP/M
 3  XENIX user      40  Venix          82  Linux swap     e1  DOS access
 4  DOS 16-bit <32M 51  Novell?        83  Linux extfs    e3  DOS R/O
 5  Extended        52  Microport      93  Amoeba         f2  DOS secondary
 6  DOS 16-bit >=32 63  GNU HURD       94  Amoeba BBT     ff  BBT
 7  OS/2 HPFS       64  Novell         b7  BSDI fs
Hex code (type L to list codes): 80

Command (m for help): p
Disk /dev/hdb: 16 heads, 63 sectors, 520 cylinders
Units = cylinders of 1008 * 512 bytes
   Device Boot  Begin   Start     End  Blocks   Id  System
```

```
/dev/hdb1          1       1     129   65015+  80  Old MINIX

Command (m for help):w
The partition table has been altered.
Please reboot before doing anything else.
[/]#
```

5. Remember the number of data blocks in this partition (here is 65015), which is used when creating the file system. When the partition is set up, it is necessary to restart the system as usual, so that the SLS Linux kernel can correctly identify the newly added partition.
6. After entering the SLS Linux emulation system again, we use the mkfs command to create the MINIX file system on the first partition we just created. The commands and information are as follows. This creates a partition with 64,000 data blocks (one block of data is 1 KB).

```
[/]# mkfs /dev/hdb1 64000
21333 inodes
64000 blocks
Firstdatazone=680 (680)
Zonesize=1024
Maxsize=268966912
[/]#
```

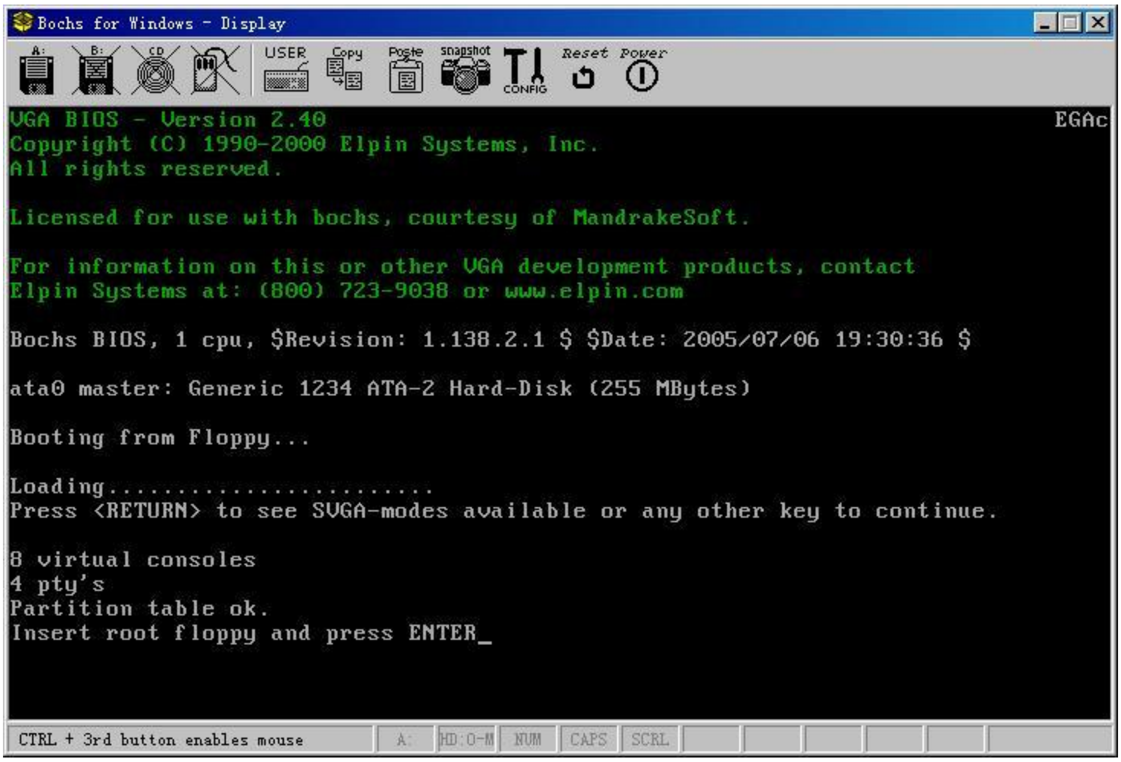
At this point, we have completed the work of creating a file system in the first partition of the hdc.img file. Of course, creating a file system can also be established when running the root file system on a Linux 0.12 floppy disk. Now we can build this partition into a root file system.

17.7.3 Bochs Configuration File for Linux-0.12

When running a Linux 0.12 system in Bochs, the following configuration is usually required in the configuration file bochsrc.bxrc.

```
romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
megs: 16
floppya: 1_44="bootimage-0.12", status=inserted
ata0-master: type=disk, path="hdc.img", mode=flat, cylinders=520, heads=16, spt=63
boot: a
log: bochsout.txt
panic: action=ask
#error: action=report
#info: action=report
#debug: action=ignore
ips: 1000000
mouse: enabled=0
```

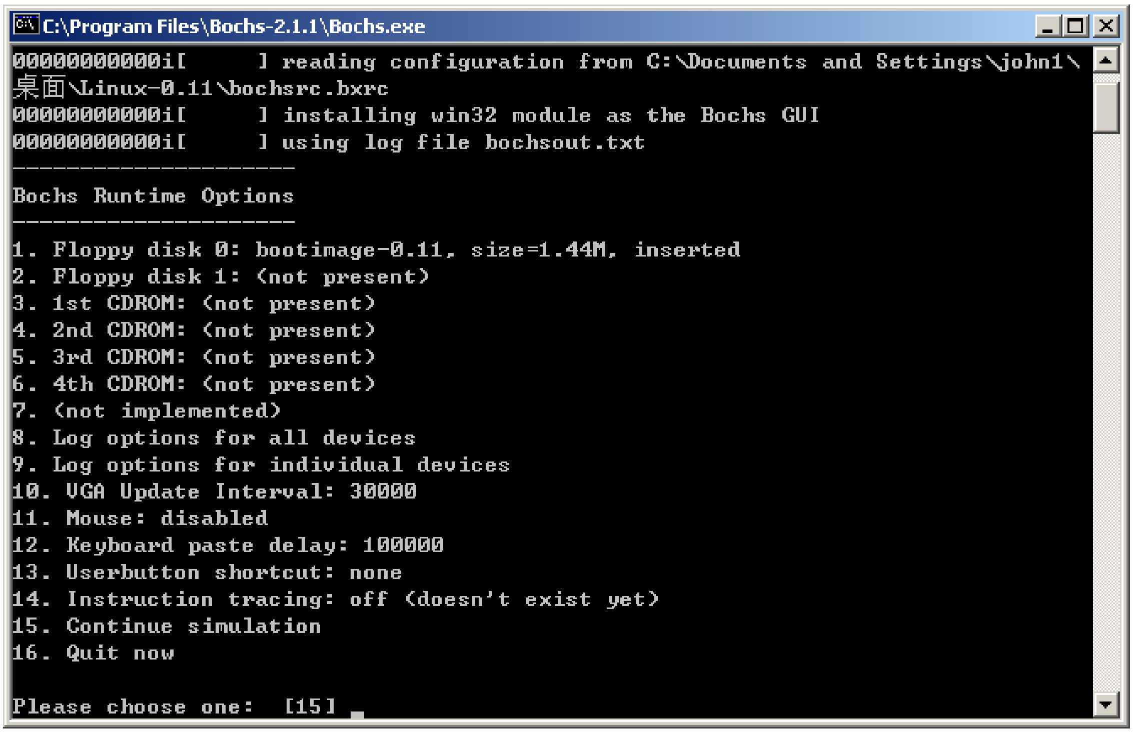
We can copy the Bochs configuration file bochsrc.bxrc from SLS Linux into the Linux-0.12 directory and modify it to the same content as above. Special attention should be paid to 'floppya', 'ata0-master' and 'boot'. These three parameters must be consistent with the above. Now we double click on this configuration file with the mouse. First, the screen in Figure 17-5 should appear in the Bochs display window.



```
VGA BIOS - Version 2.40
Copyright (C) 1990-2000 Elpin Systems, Inc.
All rights reserved.

Licensed for use with bochs, courtesy of MandrakeSoft.

For information on this or other VGA development products, contact
Elpin Systems at: (800) 723-9038 or www.elpin.com

Bochs BIOS, 1 cpu, $Revision: 1.138.2.1 $ $Date: 2005/07/06 19:30:36 $

ata0 master: Generic 1234 ATA-2 Hard-Disk (255 MBytes)

Booting from Floppy...

Loading........................
Press <RETURN> to see SVGA-modes available or any other key to continue.

8 virtual consoles
4 pty's
Partition table ok.
Insert root floppy and press ENTER_
```

Figure 17-5 Bochs system running window

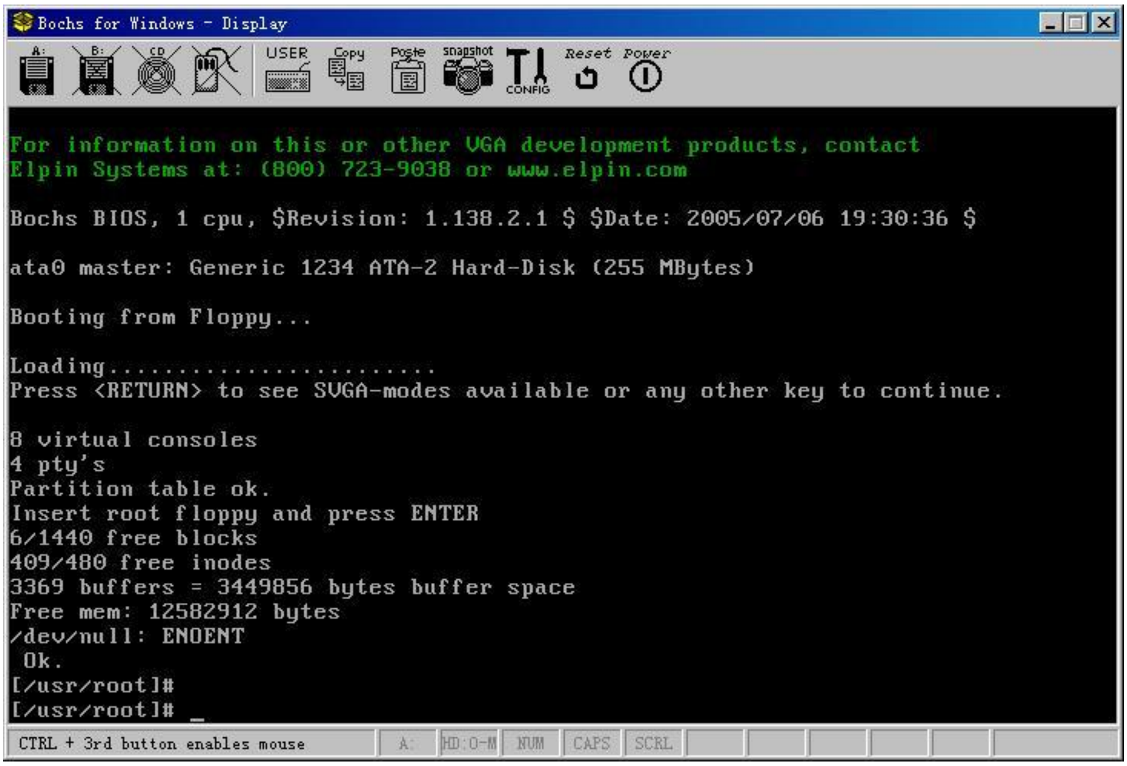
At this point, you should click the A: floppy disk icon on the window menu bar, and configure the A disk as the rootimage-0.12 file in the dialog box. Or use the Bochs configuration window to set it up by clicking the 'CONFIG' icon on the menu bar to enter the Bochs settings window (you need to click the mouse to bring the window to the front). The contents of the setting window display are shown in Figure 17-6.



```
00000000000i[     ] reading configuration from C:\Documents and Settings\john1\
桌面\Linux-0.11\bochsrc.bxrc
00000000000i[     ] installing win32 module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
---------------------
Bochs Runtime Options
---------------------
1. Floppy disk 0: bootimage-0.11, size=1.44M, inserted
2. Floppy disk 1: (not present)
3. 1st CDROM: (not present)
4. 2nd CDROM: (not present)
5. 3rd CDROM: (not present)
6. 4th CDROM: (not present)
7. (not implemented)
8. Log options for all devices
9. Log options for individual devices
10. VGA Update Interval: 30000
11. Mouse: disabled
12. Keyboard paste delay: 100000
13. Userbutton shortcut: none
14. Instruction tracing: off (doesn't exist yet)
15. Continue simulation
16. Quit now

Please choose one:  [15] _
```

Figure 17-6 Bochs system configuration window

Modify the floppy disk setting of item 1 to point to the rootimage-0.12 disk. Then press the Enter key continuously until the last line of the setup window displays 'Continuing simulation'. At this point, switch to the Bochs Run window and click Enter to formally enter the Linux 0.12 system, as shown in Figure 17-7.



```
For information on this or other VGA development products, contact
Elpin Systems at: (800) 723-9038 or www.elpin.com

Bochs BIOS, 1 cpu, $Revision: 1.138.2.1 $ $Date: 2005/07/06 19:30:36 $

ata0 master: Generic 1234 ATA-2 Hard-Disk (255 MBytes)

Booting from Floppy...

Loading.........................
Press <RETURN> to see SVGA-modes available or any other key to continue.

8 virtual consoles
4 pty's
Partition table ok.
Insert root floppy and press ENTER
6/1440 free blocks
409/480 free inodes
3369 buffers = 3449856 bytes buffer space
Free mem: 12582912 bytes
/dev/null: ENOENT
 Ok.
[/usr/root]#
[/usr/root]# _
```

Figure 17-7 Linux 0.12 system running in Bochs

17.7.4 Establishing a Root File System in hdc.img

Since the floppy disk capacity is too small, if you want the Linux 0.12 system to really do something, you need to create a root file system on the hard disk (here, the hard disk Image file). In the previous section, we have created a 256MB hard disk image file hdc.img, and it is already connected to the running Bochs environment, so a message about the hard disk appears in the figure above:

```
    “ata0 master: Generic 1234 ATA-2 Hard-Disk (255 Mbytes)”
```

If you do not see this message, your Linux 0.12 configuration file is not set correctly. Please re-edit the bochsrc.bxrc file and re-run the Bochs system until the same screen as above appears. Now, we have previously created the MINIX filesystem on the first partition of hdc.img. If you haven't built it yet or want to try it again, then type the command to create a 64MB file system:

```
[/usr/root]# mkfs /dev/hd1 64000
```

Now we can start loading the file system on the hard disk. Execute the following command to load the new file system into the /mnt directory.

```
[/usr/root]# cd /
[/]# mount /dev/hd1 /mnt
[/]#
```

After loading the file system on the hard disk partition, we can copy the root file system on the floppy disk to the hard disk. Please execute the following command:

```
[/]# cd /mnt
[/mnt]# for i in bin dev etc usr tmp
> do
> cp +recursive +verbose /$i $i
> done
```

At this point, all the files on the floppy root file system will be copied to the file system on the hard disk. A lot of information similar to the following will appear during the copy process:

```
/usr/bin/mv -> usr/bin/mv
/usr/bin/rm -> usr/bin/rm
/usr/bin/rmdir -> usr/bin/rmdir
/usr/bin/tail -> usr/bin/tail
```

```
/usr/bin/more -> usr/bin/more
/usr/local -> usr/local
/usr/root -> usr/root
/usr/root/.bash_history -> usr/root/.bash_history
/usr/root/a.out -> usr/root/a.out
/usr/root/hello.c -> usr/root/hello.c
/tmp -> tmp
[/mnt]# _
```

Now that you have built a basic root filesystem on your hard drive. You can view it anywhere in the new file system, then unmount the hard disk file system and type 'logout' or 'exit' to exit the Linux 0.12 system. The following messages will be displayed:

```
[/mnt]# cd /
[/]# umount /dev/hd1
[/]# logout

child 4 died with code 0000
[/usr/root]# _
```

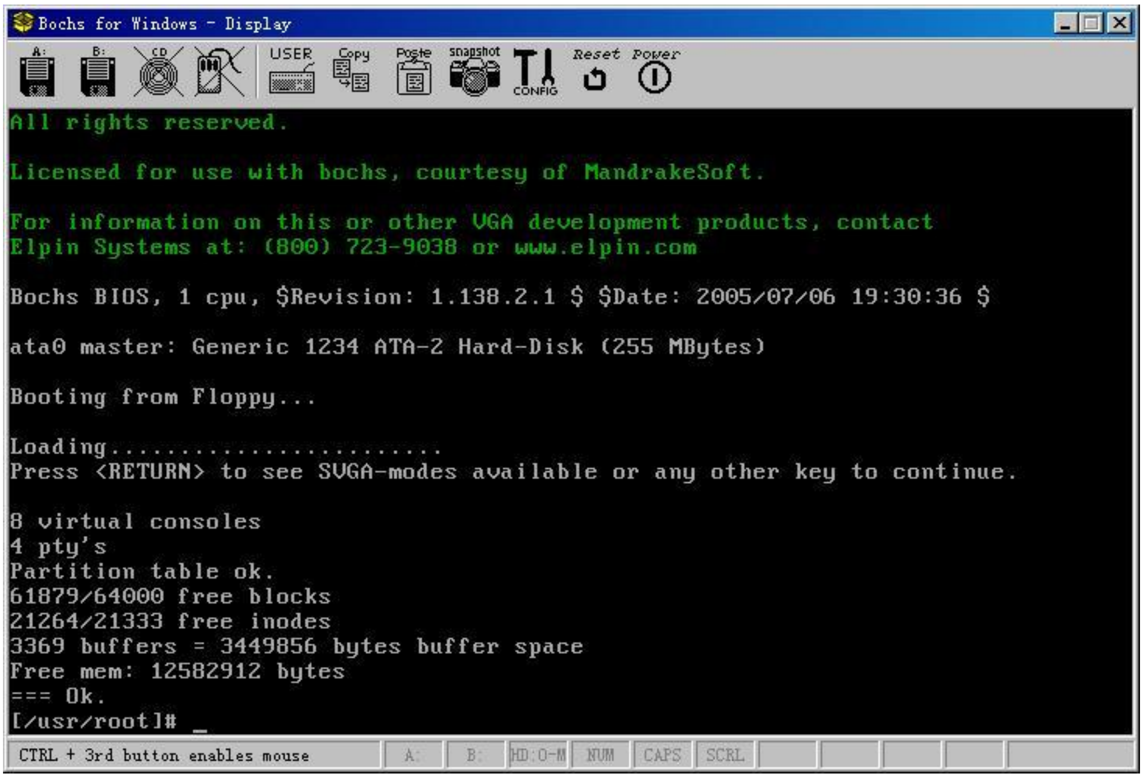
17.7.5 Using the Root File System on the Hard Disk Image

Once you have created a filesystem on your hard disk image file, you can have Linux 0.12 launch it as the root filesystem. This can be done by modifying the contents of the 509th, 510th byte (0x1fc, 0x1fd) of the bootimage-0.12 file. Please follow the steps below:

1. First copy the two files bootimage-0.12 and bochsrc.bxrc to generate the bootimage-0.12-hd and bochsrc-hd.bxrc files.
2. 2. Edit the bochsrc-hd.bxrc configuration file, change the file name on the 'floppya:' option to 'bootimage-0.12-hd', and save it;
3. 3. Edit the bootimage-0.12-hd binary with Notepad++ or any other binary editor and modify the 509th and 510th bytes (ie 0x1fc, 0x1fd). The original value should be 00, 00, modified to 01, 03, indicating that the root file system device is on the first partition of the hard disk Image, and then save and exit. If you have the file system installed on another partition, you will need to modify the first byte to correspond to your partition.

```
000001f0h: 00 00 00 00 00 00 00 00 00 00 00 00 01 03 55 AA ; ..............U?
```

Now you can double-click on the icon of the bochsrc-hd.bxrc file. The Bochs system should quickly enter the Linux 0.12 system and display the graphics in Figure 17-8.



```
All rights reserved.

Licensed for use with bochs, courtesy of MandrakeSoft.

For information on this or other VGA development products, contact
Elpin Systems at: (800) 723-9038 or www.elpin.com

Bochs BIOS, 1 cpu, $Revision: 1.138.2.1 $ $Date: 2005/07/06 19:30:36 $

ata0 master: Generic 1234 ATA-2 Hard-Disk (255 MBytes)

Booting from Floppy...

Loading........................
Press <RETURN> to see SVGA-modes available or any other key to continue.

8 virtual consoles
4 pty's
Partition table ok.
61879/64000 free blocks
21264/21333 free inodes
3369 buffers = 3449856 bytes buffer space
Free mem: 12582912 bytes
=== Ok.
[/usr/root]# _
```

Figure 17-8 Use the file system in the hard disk image file

17.8 Compile Kernel on Linux 0.12 System

The author has reorganized a Linux 0.12 system package with the gcc 1.40 build environment. The system is set up to run under the Bochs simulation system and the corresponding bochs configuration file has been configured. This package is available from the following address:

```
    http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.zip
```

The package contains a README file that explains the purpose and use of all the files in the package. If you have a bochs system installed on your system, you can run this Linux 0.12 by simply double-clicking the icon of the configuration file bochsrc-0.12-hd.bxrc. It uses the hard disk Image file as the root file system. After running the system, type the 'make' command in the /usr/src/linux directory to compile the Linux 0.12 kernel source code and generate the boot image file 'Image'. If you need to output this Image file, you can first backup the bootimage-0.12-hd file, and then use the following command to replace bootimage-0.12-hd with the new boot file. Now restart Bochs directly, you can use the bootimage-0.12-hd generated by the new compiler to boot the system.

```
[/usr/src/linux]# make
[/usr/src/linux]# dd bs=8192 if=Image of=/dev/fd0
[/usr/src/linux]# _
```

You can also use the mtools command to write the newly generated Image file to the second floppy image file 'diskb.img', and then use the tool software WinImage to extract the 'Image' file in 'diskb.img'.

```
[/usr/src/linux]# mdir a:
Probable non-MSDOS disk
mdir: Cannot initialize ’A:’
[/usr/src/linux]# mcopy Image b:
Copying IMAGE
[/usr/src/linux]# mcopy System.map b:
Copying SYSTEM.MAP
[/usr/src/linux]# mdir b:
Volume in drive B is B.
 Directory for B:/
GCCLIB-1 TAZ      934577    3-29-104   7:49p
IMAGE             121344    4-29-104  11:46p
SYSTEM   MAP       17162    4-29-104  11:47p
README               764    3-29-104   8:03p
       4 File(s)     382976 bytes free
[/usr/src/linux]# _
```

If you want to use the new boot image file with the root file system rootimage-0.12 on the floppy disk, first edit the Makefile before compiling, and comment out the 'ROOT_DEV=' line with '#'.

It is usually done very smoothly when compiling the kernel. A possible problem is that the gcc compiler does not recognize the option '-mstring-ins'. This option is an extended experimental parameter implemented by Linus for the gcc 1.40 compiler compiled by itself. It is used to optimize the gcc when generating string instructions. In order to solve this problem, you can directly delete this option in all Makefiles and recompile the kernel. Another possible problem is that the 'gar' command cannot be found. In this case, you can directly link or copy/rename 'ar' under /usr/local/bin/ to 'gar'.

17.9 Compile kernel under Redhat system

The original Linux operating system kernel was cross-compiled and developed on the Minix-i386, an extended version of the Minix 1.5.10 operating system. The Minix 1.5.10 operating system was released by Prentice Hall Publishing Company along with the first edition of A.S. Tanenbaum's "Design and Implementation of Minix". Although this version of Minix can run on the 80386 and its compatible microcomputers, it does not take advantage of the 80386 32-bit protection mechanism. In order to develop 32-bit operating systems on the system, Linus used Bruce Evans' patch to upgrade it to MINIX-386, and ported GNU series development tools gcc, gld, emacs, bash, etc. on to the Minix-386. On this platform, Linus cross-compile and develop kernels of versions 0.01, 0.03, 0.11, and 0.12. According to some articles in the Linux mailing list, the author has established a development platform similar to Mr. Linus's then, and successfully compiled the early version of Linux kernel.

However, since Minix 1.5.10 is outdated and the development platform is very cumbersome, here is a brief introduction to how to modify the Linux 0.12 kernel source code so that it can be compiled in the current RedHat system compilation environment, and generate a bootimage file bootimage that can be run. Readers can run it on a regular PC or in virtual machine software such as Bochs. Only the main modification aspects are given here. All the modifications can use the tool diff to compare the modified and unmodified code to find out the difference. If the unmodified code is in the linux/ directory and the modified code is in linux-mdf/, you need to execute the following command:

```
diff -r linux linux-mdf > dif.out
```

The file 'dif.out' contains all the modified places in the source code. The Linux 0.1X kernel source code that has been modified and can be compiled under RedHat 9 can be downloaded from the following address:

```
http://oldlinux.org/Linux.old/kernel/linux-0.11-040327-rh9.tar.gz
http://oldlinux.org/Linux.old/kernel/linux-0.11-040327-rh9.diff.gz
http://oldlinux.org/Linux.old/kernel/linux-0.11-060617-gcc4-diff.gz
http://oldlinux.org/Linux.old/kernel/linux-0.11-060618-gcc4.tar.gz
http://oldlinux.org/Linux.old/kernel/linux-0.12-080328-gcc4-diff.gz
http://oldlinux.org/Linux.old/kernel/linux-0.12-080328-gcc4.tar.gz
```

When booting with the compiled boot image file, the following information should appear on the screen:

```
Booting from Floppy...
Loading system ...

Insert root floppy and press ENTER
```

Note that if there is no response after displaying "Loding system...", this means that the kernel does not recognize the hard disk controller subsystem in the computer. At this point, you can use VirtualBox, VMware, bochs and other virtual machine software to test. When you are asked to insert the root file system disk, if you press the Enter key directly, the following information that cannot be loaded the root file system will be displayed and it will crash. To run the Linux 0.1X operating system completely, you need a matching root file system, which can be downloaded from the oldlinux.org website.

17.9.1 Modifying the Makefile

In the Linux 0.1X kernel source directory, almost every subdirectory includes a Makefile, which needs to be modified as follows:

a. Rename 'gas' to 'as', 'gld' to 'ld'. Because now 'gas' and 'gld' have been directly renamed to 'as' and 'ld'.
b. 'as' (original gas) has no need to use the '-c' option, so the -c compilation option needs to be removed from the Makefile in the kernel home directory Linux.
c. c. Remove gcc's compile flag options: '-fcombine-regs', '-mstring-insns', and these two options in the Makefile in all subdirectories. The '-fcombine-regs' option was not found in the 1994 gcc manual, and '-mstring-insns' is an option that Linus added to gcc modifications, so this optimization option is definitely not included in your gcc.
d. In the gcc compile option, add the '-m386' option. In this way, the kernel image file compiled under RedHat 9 will not contain the instructions of the CPU of 80486 and above, so the kernel can run on the 80386 machine.

17.9.2 Modifying Comments in the Assembly Language Programs

The as86 compiler does not recognize the c comment statement, so you need to use the '!' to comment out the C comment in the boot/bootsect.s file.

17.9.3 Modifying the align value of the memory alignment statement

The use of the 'align' statement has changed in the three assembly language programs in the boot directory. The value after the original 'align' refers to the power value of the memory location, but now it needs to directly give the value of the integer address. Therefore, the original statement:

```
.align 3
```

Need to be modified to (2^3=8):

```
.align 8
```

17.9.4 Modifying Inline Macro Assembly Language Programs

Due to the continuous improvement of the as assembler, it is now more and more automated, so there is no need to manually specify the CPU register to be used for a variable. Therefore all '__asm__("ax")' in the kernel code needs to be removed. For example, on lines 20 and 26 of the fs/bitmap.c file, on line 68 of the fs/namei.c file.

In the inline assembly code, it is also necessary to remove all declarations that are invalid for the contents of the register (the registers that will be modified). For example, line 84 in include/string.h:

```
:″si″,″di″,″ax″,″cx″);
```

All registers need to be removed, leaving only the colon and the right parenthesis: ":);".

Sometimes there are some problems with this modification. Since gcc sometimes optimizes the program according to the above statement, in some places, deleting the contents of the register that will be modified will cause a gcc optimization error. Therefore, some places in the program code need to retain some of these declarations depending on the situation, such as line 342 in the include/string.h file memcpy() definition.

17.9.5 Reference representation of C variables in assembly statements

The assembler used in the development of the Linux 0.1X kernel needs to add the underscore character '_' to the variable name when referring to the C variables. The current gcc compiler can directly recognize the C variables referenced in these assemblies, so Remove the underscore before all C variables in the assembler (including embedded assembly statements). For example, the statement at line 15 in the boot/head.s program:

```
.globl _idt,_gdt,_pg_dir,_tmp_floppy_area
```

Need to be changed to:

```
.globl idt,gdt,pg_dir,tmp_floppy_area
```

The variable name "_stack_start" on the 31st line statement needs to be modified to "stack_start".

17.9.6 Debug Display Function in Protected Mode

Before entering protected mode, you can use the int 0x10 call in the ROM BIOS to display information on the screen. However, after entering the protection mode, these interrupt calls cannot be used. In order to understand the internal data structure and state of the kernel in a protected mode environment, we can use the following function check_data32() to display the kernel data (previously provided by a friend 'notrump' on the oldlinux.org forum). Although there is a printk() display function in the kernel, it needs to call tty_write(), which is not available when the kernel is not fully functional.

After entering protected mode, this check_data32() function can print what you are interested in on the screen. Whether the page function is enabled or not does not affect the use. Because the virtual memory in 4M just uses the first page table directory entry, and the page table directory starts from physical address 0, plus the kernel data segment base address is 0, so in the 4M range, the addresses of virtual memory, linear memory and physical memory are the same. Mr. Linus may have considered this in the first place, and feels that this setting is more convenient to use.

```
/*
* Purpose: Display a 32-bit integer in hexadecimal on the screen.
* Params:  value -- the integer to display.
*          pos -- The screen position, in units of 16 chars wide, for example, 2, which means
*                 that the display starts at the width of 32 chars from the upper left corner.
* Return: None.
* If you want to use it in an assembly language program, make sure that the function is compiled
* and linked into the kernel. The usage in gcc assembly is as follows:
* pushl pos              // 'pos' should be replaced with your actual data, such as pushl $4
* pushl value            // 'pos' and 'value' can be any legal addressing method.
* call  check_data32
*/
inline void check_data32(int value, int pos)
{
__asm__ __volatile__(             // %0 -the value to be displayed; EBX -screen position pos.
"shl     $4, %%ebx\n\t"            // Multiply the pos by 16, plus VGA memory start address,
"addl    $0xb8000, %%ebx\n\t"      // get the position from top left of the screen in EBX.
"movl    $0xf0000000, %%eax\n\t"   // Set a 4-bit mask.
"movb    $28, %%cl\n"              // Set the initial right shift bit value.
"1:\n\t"
"movl    %0, %%edx\n\t"            // Put the displayed value to EDX.
"andl    %%eax, %%edx\n\t"         // Take 4 bits specified by EAX in EDX.
"shr     %%cl, %%edx\n\t"          // Shift 28 bits to right, EDX is the value of 4 bits taken.
"add     $0x30, %%dx\n\t"          // Convert this value to ASCII code.
"cmp     $0x3a, %%dx\n\t"          // If less than 10, jumps forward to label  2.
"jb2f\n\t"
"add     $0x07, %%dx\n"            // Otherwise add 7 and convert the value to A-F.
"2:\n\t"
"add    $0x0c00, %%dx\n\t"          // Set the display attributes.
"movw   %%dx, (%%ebx)\n\t"          // Put this value in the display memory.
"sub    $0x04, %%cl\n\t"            // Prepare the next hex number, the number of shifts minus 4.
"shr    $0x04, %%eax\n\t"           // The mask is shifted to the right by 4 bits.
"add    $0x02, %%ebx\n\t"           // Update the display memory position.
"cmpl   $0x0, %%eax\n\t"            // The mask has moved out of the right (8 hexs displayed)?
"jnz1b\n"                           // No, there still numbers to be displayed, jump to lable 1.
::"m"(value), "b"(pos));
}
```

17.10 Integrated Boot Disk and Root FS

This section explains how to make an integrated disk image file that is a combination of a kernel boot image file and a root file system. Its main purpose is to understand the working principle of the Linux 0.1X kernel memory virtual disk, and further understand the concept of the boot disk and the root file system disk, and deepen the understanding of the kernel/blk_drv/ramdisk.c program running method. In fact, the boot module, kernel module, and file system module image structure stored in Flash in a typical embedded system is similar to the integrated disk here.

Below we take the process of making an integrated disk using the Linux 0.11 kernel as an example to illustrate the making process. As an exercise, readers use the 0.12 kernel to implement a similar integrated disk. Before making this integrated disk, we need to first download or prepare the following experimental software (the latter two are used for the building of the 0.12 kernel integrated disk):

```
    http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-040923.zip
    http://oldlinux.org/Linux.old/images/rootimage-0.11-for-orig
    http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.zip
    http://oldlinux.org/Linux.old/images/rootimage-0.12-20040306
```

'linux-0.11-devel' is a Linux 0.11 system with a development environment running under Bochs. The 'rootimage-0.11' is a Linux 0.11 root file system in a 1.44MB floppy image file. The suffix 'for-orig' refers to the kernel boot image file that is compiled for the unmodified Linux 0.11 kernel source code. Of course, the "unmodified" mentioned here means that there has not been any major changes to the kernel, because we still need to modify the compiled configuration file Makefile to compile the kernel code containing the memory virtual disk.

17.10.1 Integrated disk building principle

Usually we need two disks when booting Linux 0.1X system using floppy disk (here, "disk" refers to the image file corresponding to the floppy disk): one is the kernel boot disk and the other is the root file system disk. This requires two disks to boot the system to run a basic Linux system, and the root file system disk must remain in the floppy disk drive during runtime. The integrated disk we describe here is a combination of the contents of the kernel boot disk and a basic root file system disk on one disk. This way we can boot a Linux

0.1X system to a command prompt using only one integrated disk. The integrated disk is actually a kernel boot disk with a root file system.

In order to run the integrated disk system, the function of the memory virtual disk (RAMDISK) needs to be turned on in the kernel code on the disk. The root file system on the integrated disk can be loaded into the virtual disk in memory so that the two floppy drives on the system can be freed for mounting other file system disks or for other purposes. Below we will introduce in detail the principles and steps of making an integrated disk on a 1.44MB disk.

17.10.1.1 Principle of the boot process

The Linux 0.1X kernel will determine whether the virtual disk area should be reserved in the system physical memory according to the RAMDISK option set in the compile-time Makefile. If RAMDISK is not set (ie its size is 0), the kernel will load the root file system from the floppy disk or hard disk according to the device number of the root file system set by ROOT_DEV, and perform the general startup process when there is no virtual disk.

If the size of the RAMDISK is defined in its linux/Makefile when compiling the Linux 0.1X kernel source code, the kernel code will first try to detect the boot disk from the 256th disk block after booting and initializing the RAMDISK memory area. Is there a root file system? The detection method is to check if there is a valid file system super block in the 257th disk block. If so, the file system is loaded into the RAMDISK area and used as the root file system. So we can use a boot disk that integrates the root file system to boot the system to the shell command prompt. If a valid root file system is not stored on the boot disk at the specified disk block location (starting with the 256th disk block), the kernel will prompt to insert the root file system disk. After the user presses the Enter key to confirm, the kernel reads the root file system on the independent disk and reads it into the virtual disk area of the memory. This detection and loading process can be seen in Figure 9-7.

17.10.1.2 Structure of the integrated disk

For the Linux 0.1X kernel, the size of the code plus data segment is very small, and the size is about 120KB to 160KB. In the initial stage of developing Linux system, even considering the kernel extension, Mr. Linus still thinks that the size of the kernel will not exceed 256KB, so you can store a basic root file system at the beginning of the 256th disk block of the 1.44MB boot disk. Thereby combined to form an integrated disk. A schematic diagram of a boot disk (ie, an integrated disk) with a basic root file system added is shown in Figure 17-9. For the detailed structure of the file system, please refer to the description in the file system chapter.

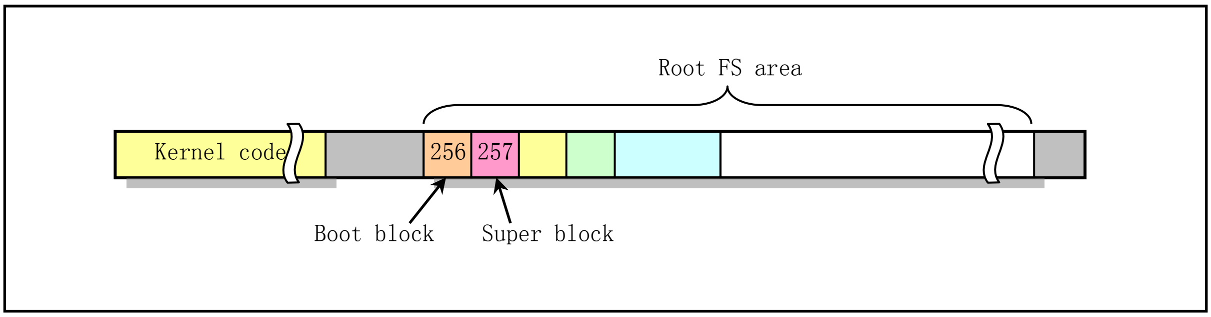


Figure 17-9 Integrated disk code structure

As mentioned above, the location and size of the root file system placement on the integrated disk is primarily related to the length of the kernel and the size of the defined RAMDISK area. Mr. Linus defines in the ramdisk.c program that the root file system's starting placement position is the beginning of the 256th disk block. For the Linux 0.1X kernel, the compiled kernel image file (ie the boot disk Image file) is about 120KB to 160KB. Therefore, putting the root file system at the beginning of the 256th disk block of the disk is certainly no problem, only a little wasted disk space. There are still a total of 1440 - 256 = 1184 KB space available for the root file system. Of course, we can also adjust the starting disk block location of the root file system according to the specific compiled kernel size. For example, we can modify the value of the 'block' of line 75 of ramdisk.c to 130, so that the starting position of the root file system is moved backwards to free up more disk space for the root file system on the disk.

17.10.2 Integrated disk construction process

Without changing the default disk block location in the kernel program ramdisk.c, we assume that the root file system on the integrated disk needs to be 1024KB (maximum of 1184KB). The main idea of making an integrated disk is to first create a 1.44MB empty Image disk file, and then copy the newly compiled kernel image file with RAMDISK function to the beginning of the disk. Then copy the customized file system with a size of 1024KB or less to the beginning of the 256th disk block of the disk. The specific building steps are as follows.

17.10.2.1 Recompiling the kernel

Recompile the kernel Image file with the RAMDISK definition, assuming the RAMDISK area is set to 2048KB. The method is to run the linux-0.1X system in Bochs. Edit the /usr/src/linux/Makefile file and modify the following settings line:

```
RAMDISK = -DRAMDISK = 2048
ROOT_DEV = FLOPPY
```

Then recompile the kernel source code to generate a new kernel image file:

```
make clean; make
```

17.10.2.2 Making a Temporary Root File System

Make a root file system Image file with a size of 1024KB, and now assume its file name is 'rootram.img'. The Bochs system is run during the building process using a configuration file (bochsrc-hd.bxrc) with a hard disk Image. The building method is as follows:

(1) Make an empty Image file of 1024KB in size using the method described earlier in this chapter. The name of the file is specified as 'rootram.img'. You can use the following command to generate under the current Linux system:

```
    dd bs=1024 if=/dev/zero of=rootram.img count=1024
```

(2) Run the linux-0.1X system in Bochs. Then configure the driver disks in the main Bochs window: disk A is rootimage-0.1X (0.11 kernel is rootimage-0.11-orign); disk B is rootram.img.

(3) Use the following command to create an empty file system of size 1024KB on the rootram.img disk. Then mount the A and B disks to the /mnt and /mnt1 directories respectively. If the directory /mnt1 does not exist, you can create one.

```
mkfs /dev/fd1 1024
mkdir /mnt1
mount /dev/fd0 /mnt
mount /dev/fd1 /mnt1
```

(4) Use the 'cp' command to selectively copy /mnt files from rootimage-0.1X to the /mnt1 directory and create a root filesystem in /mnt1. If you encounter any error message, then the content is usually more than 1024KB. First reduce the files in /mnt/ to meet the capacity requirements of no more than 1024KB. We can remove some files under /bin and /usr/bin to achieve this. Regarding capacity, we can use the 'df' command to view it. For example, the files we can choose to keep are the following:

```
[/mnt/bin]# ll
total 495
-rwx--x--x   1 root     root        29700 Apr 29 20:15 mkfs
-rwx--x--x   1 root     root        21508 Apr 29 20:15 mknod
-rwx--x--x   1 root     root        25564 Apr 29 20:07 mount
-rwxr-xr-x   1 root     root       283652 Sep 28 10:11 sh
-rwx--x--x   1 root     root        25646 Apr 29 20:08 umount
-rwxr-xr-x   1 root     4096       116479 Mar  3  2004 vi
[/mnt/bin]# cd /mnt/usr/bin
[/mnt/usr/bin]# ll
total 364
-rwxr-xr-x   1 root     root        29700 Jan 15  1992 cat
-rwxr-xr-x   1 root     root        29700 Mar  4  2004 chmod
-rwxr-xr-x   1 root     root        33796 Mar  4  2004 chown
-rwxr-xr-x   1 root     root        37892 Mar  4  2004 cp
-rwxr-xr-x   1 root     root        29700 Mar  4  2004 dd
-rwx--x--x   1 root     4096        36125 Mar  4  2004 df
-rwx--x--x   1 root     root        46084 Sep 28 10:39 ls
-rwxr-xr-x   1 root     root        29700 Jan 15  1992 mkdir
-rwxr-xr-x   1 root     root        33796 Jan 15  1992 mv
-rwxr-xr-x   1 root     root        29700 Jan 15  1992 rm
-rwxr-xr-x   1 root     root        25604 Jan 15  1992 rmdir
[/mnt/usr/bin]#
```

(5) Then use the following command to copy the file. In addition, you can modify the contents of /mnt/etc/fstab and /mnt/etc/rc as needed. At this point, we have created a file system with a size of 1024KB or less in fd1(/mnt1/).

```
cd /mnt1
for i in bin dev etc usr tmp
do
cp +recursive +verbose /mnt/$i $i
done
sync
```

(6) Use the 'umount' command to unmount the filesystems on /dev/fd0 and /dev/fd1, then use the 'dd' command to copy the filesystem from /dev/fd1 to the Linux-0.1X system and create a name called rootram- 0.1X root file system Image file:

```
dd bs=1024 if=/dev/fd1 of=rootram-0.1X count=1024
```

At this time, in the Linux-0.1X system under Bochs, we have a newly compiled kernel image file /usr/src/linux/Image and a simple root file system image file rootram-0.1X with a capacity of 1024KB or less.

17.10.2.3 Creating an Integrated Disk

Now combine the above two image files to create an integrated disk. Modify the A disk configuration in the main Bochs window and set it to the previously prepared 1.44MB image file named bootroot-0.1X. Then execute the following commands:

```
dd bs=8192 if=/usr/src/linux/Image of=/dev/fd0
dd bs=1024 if=rootram-0.1X of=/dev/fd0 seek=256
sync;sync;sync;
```

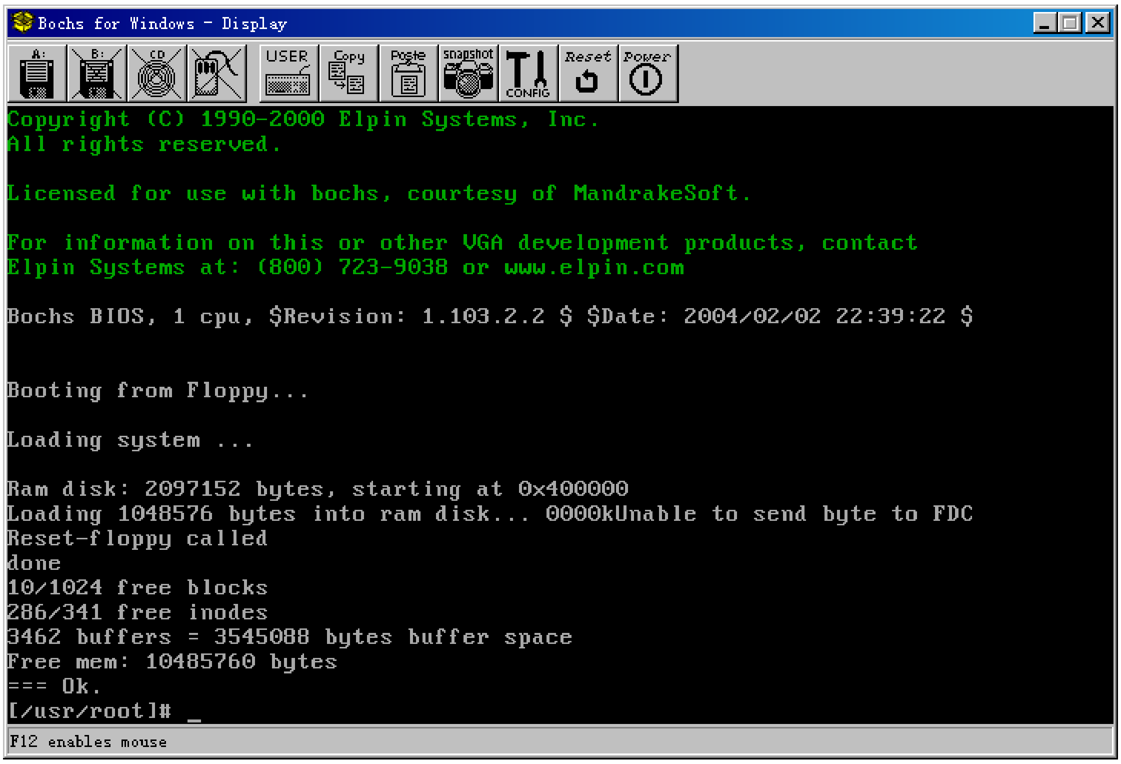
The option 'bs=1024' means that the size of the definition buffer is 1KB; 'seek=256' means that the first 256 disk blocks are skipped when the output file is written. Then exit the Bochs system. At this point, we get a running integrated disk image file bootroot-0.1X in the current directory of the host.

17.10.3 Running the Integrated Disk System

First, let's make a simple Bochs configuration file, bootroot-0.1X.bxrc, for the integrated disk. The main settings are:

```
floppya: 1_44=bootroot-0.1X
```

Then double-click the configuration file with the mouse to run the Bochs system. At this point, the results should be as shown in Figure 17-10.



```
Copyright (C) 1990-2000 Elpin Systems, Inc.
All rights reserved.

Licensed for use with bochs, courtesy of MandrakeSoft.

For information on this or other VGA development products, contact
Elpin Systems at: (800) 723-9038 or www.elpin.com

Bochs BIOS, 1 cpu, $Revision: 1.103.2.2 $ $Date: 2004/02/02 22:39:22 $


Booting from Floppy...

Loading system ...

Ram disk: 2097152 bytes, starting at 0x400000
Loading 1048576 bytes into ram disk... 0000kUnable to send byte to FDC
Reset-floppy called
done
10/1024 free blocks
286/341 free inodes
3462 buffers = 3545088 bytes buffer space
Free mem: 10485760 bytes
=== Ok.
[/usr/root]# _
```

Figure 17-10 Integrated disk running interface

In order to facilitate the experiment, you can also download the integrated disk software of 0.11 kernel that is ready and can run immediately from the following website:

```
http://oldlinux.org/Linux.old/bochs/bootroot-0.11-040928.zip
```

17.11 Debugging Kernel Code with GDB and Bochs

This section explains how to use the Bochs emulation environment and gdb tools to debug Linux 0.1X kernel source code on existing Linux systems such as RedHat or Fedora. Before using this method, the X window system should already be installed on the existing Linux system. Since the Bochs executable in the RPM installation package provided by the Bochs website does not have the 'gdbstub' module that communicates with the gdb debugger, we need to download the Bochs source code to compile the running program with this module.

The 'gdbstub' module allows the Bochs program to listen for commands from gdb on the local 1234 network port and send command execution results to gdb. So we can use gdb to debug the C language level of the Linux 0.1X kernel. Of course, the Linux0.1X kernel also needs to be recompiled with the '-g' option to have the generated kernel code with debugging information.

17.11.1 Compiling a Bochs System with gdbstub

The Bochs User Manual describes how to compile the Bochs system yourself. Here we also give the methods and steps to compile the Bochs system with gdbstub. First download the latest Bochs system source code from the following website (for example: bochs-2.6.tar.gz):

```
    http://sourceforge.net/projects/bochs/
```

Decompressing the package with 'tar' will generate a bochs-2.6 subdirectory in the current directory. After entering this subdirectory, run the configuration program 'configure' with the option "--enable-gdb-stub", then run 'make' and 'make install' as shown below:

```
[root@plinux bochs-2.2]# ./configure --enable-gdb-stub
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
...
[root@plinux bochs-2.2]# make
[root@plinux bochs-2.2]# make install
```

If we encounter some problems when running './configure' and cannot generate the Makefile used for compilation, this is usually caused by not installing the X window development environment software or related library files. At this point we must first install the necessary software and then recompile Bochs.

17.11.2 Compiling the Linux 0.1X Kernel with Debug Information

By linking Bochs' simulation runtime environment to the gdb symbolic debugging tool, we can either use the kernel module with debugging information compiled under Linux 0.1X system to debug, or use the 0.1X kernel module compiled in RedHat environment to debug. In both environments, all Makefiles in the 0.1X kernel source directory need to be modified by adding the '-g' option to the compile flag line and removing the '-s' option on the link flag line:

```
LDFLAGS = -M -x                                      // Remove ’-s ’ flag.
CFLAGS  =-Wall -O -g -fomit-frame-pointer \          // Add ’-g ’ flag.
```

After entering the kernel source directory, we can use the 'find' command to find all the following Makefiles that need to be modified:

```
[root@plinux linux-0.1X]# find ./ -name Makefile
./fs/Makefile
./kernel/Makefile
./kernel/chr_drv/Makefile
./kernel/math/Makefile
./kernel/blk_drv/Makefile
./lib/Makefile
./Makefile
./mm/Makefile
[root@plinux linux-0.1X]#
```

In addition, since the kernel code module compiled at this time contains debugging information, the system module size may exceed the default maximum value of the write kernel code image file SYSSIZE = 0x3000 (defined in line 7 of the boot/bootsect.s file). At this point, we can modify the rules of the Image file generated in the Makefile in the root directory of the source code by removing the symbol information in the kernel module 'system' and then writing it to the Image file. The original 'system' module with the symbol information is reserved for use by the gdb debugger. Note that the implementation command for the target in the Makefile needs to start with a tab.

```
Image: boot/bootsect boot/setup tools/system tools/build
        cp -f tools/system system.tmp
        strip system.tmp
        tools/build boot/bootsect boot/setup system.tmp $(ROOT_DEV) $(SWAP_DEV) > Image
        rm -f system.tmp
        sync
```

Of course, we can also modify the SYSSIZE value in boot/bootsect.s and tools/build.c to 0x8000 to handle this situation.

17.11.3 Debugging methods and steps

Below we explain the debugging methods and steps according to the kernel code compiled on a modern Linux system (such as RedHat or Fedora) and compiled on a Linux 0.1X system running in Bochs. The debugging methods and steps of the Linux 0.11 kernel code are given below. The debugging method and steps of the 0.12 kernel are exactly the same.

17.11.3.1 Debugging the Linux 0.11 kernel compiled on modern Linux

Assuming that our Linux 0.11 kernel source root directory is linux-rh9-gdb/, we first modify all Makefiles in this directory according to the above method, then create a Bochs configuration file in it and download a root file system image file that supporting the kernel. We can also download the following packages that have been set up directly from the website to do the experiment:

```
    http://oldlinux.org/Linux.old/bochs/linux-0.11-gdb-rh9-050619.tar.gz
```

After unpacking this package with the command "tar zxvf linux-gdb-rh9-050619.tar.gz", you can see that it contains the following files and directories:

```
[root@plinux linux-gdb-rh9]# ls -l
total 1600
-rw-r--r--    1 root     root        18055 Jun 18 15:07 bochsrc-fd1-gdb.bxrc
drwxr-xr-x   10 root     root         4096 Jun 18 22:55 linux
-rw-r--r--    1 root     root      1474560 Jun 18 20:21 rootimage-0.11-for-orig
-rwxr-xr-x    1 root     root           35 Jun 18 16:54 run
[root@plinux linux--gdb-rh9]#
```

The first file 'bochsrc-fd1-gdb.bxrc' is the Bochs configuration file, in which the file system image file 'rootimage-0.11-for-orig' has been set to be inserted in the second "floppy drive". The main difference between this Bochs configuration file and other Linux 0.1X configuration files is that the following line is added to the front of the file, indicating that when Bochs runs with this configuration file, it will listen for commands from the gdb debugger on the local network port 1234:

```
gdbstub: enabled=1, port=1234, text_base=0, data_base=0, bss_base=0
```

The second item of linux/ is the Linux 0.11 source code directory, which contains the kernel source code files that have been modified for all Makefiles. The third file 'rootimage-0.11-for-orig' is the root file system image file that is associated with this kernel code. The fourth file 'run' is a simple script that contains a line of Bochs startup command. The basic steps to run this experiment are as follows:

1. Open two terminal windows under the X window system;
2. In one of the terminal windows, switch the working directory to the linux-gdb-rh9/ directory and run the program './run'. At this point, a message waiting for gdb to connect is displayed: "Wait for gdb connection On localhost:1234", and the system will create a Bochs main window (no content at this time);
3. In another terminal window, we switch the working directory to the kernel source directory linux-gdb-rh9/linux/ and run the command: "gdb tools/system";
4. Type the command "break main" and "target remote localhost:1234" in the window where gdb is run. At this time, gdb will display the information that has been connected to Bochs.
5. Execute the command "cont" in the gdb environment. After a while, gdb will show that the program stops at the main() function of init/main.c.

After that we can use the gdb command to observe the source code and debug the kernel. For example, we can use the 'list' command to observe the source code, use the 'help' command to get online help information, use 'break' to set other breakpoints, use 'print/set' to display/set some variable values, use ' Next/step' to perform single-step debugging, use the 'quit' command to exit gdb, and so on. Please refer to the gdb manual for the specific usage of gdb. Below are some examples of commands that run gdb and execute in it.

```
[root@plinux linux]# gdb tools/system             // Start gdb to execute the system module.
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) break main                                  // Set a breakpoint at the main() function.
Breakpoint 1 at 0x6621: file init/main.c, line 110.
(gdb) target remote localhost:1234                // Connecting to Bochs.
Remote debugging using localhost:1234
0x0000fff0 in sys_mkdir (pathname=0x0, mode=0) at namei.c:481
481     namei.c: No such file or directory.
        in namei.c
(gdb) cont                                        // Continue execution until the breakpoint.
```

```
Continuing.
Breakpoint 1, main () at init/main.c:110            // Stop running at the breakpoint.
110             ROOT_DEV = ORIG_ROOT_DEV;
(gdb) list                                          // View the source code.
105     {                       /* The startup routine assumes (well, ...) this */
106     /*
107      * Interrupts are still disabled. Do necessary setups, then
108      * enable them
109      */
110             ROOT_DEV = ORIG_ROOT_DEV;
111             drive_info = DRIVE_INFO;
112             memory_end = (1<<20) + (EXT_MEM_K<<10);
113             memory_end &= 0xfffff000;
114             if (memory_end > 16*1024*1024)
(gdb) next                                          // Single step execution.
111             drive_info = DRIVE_INFO;
(gdb) next                                          // Single step.
112             memory_end = (1<<20) + (EXT_MEM_K<<10);
(gdb) print /x ROOT_DEV                             // Print the variable ROOT_DEV.
$3 = 0x21d                                          // The second floppy device number.
(gdb) quit                                          // Exit gdb.
The program is running.  Exit anyway? (y or n) y
[root@plinux linux]#
```

When debugging kernel source code in gdb, sometimes the problem that the source program does not find is displayed. For example, gdb sometimes displays "memory.c: No such file or directory". This is because when compiling mm/memory.c and other files, the Makefile indicates that the ld linker has linked the file module under mm/ to generate a relocatable module 'mm.o', and in the source code root directory linux / Down, it is again used as the input module for ld. Therefore, we can copy these files to the linux/ directory and re-execute the kernel debugging operation.

17.11.3.2 Debug the 0.1X kernel compiled on Linux 0.1X system

In order to debug a kernel compiled on a 0.1X system in a modern Linux operating system such as RedHat, we need to copy the entire 0.1X kernel source directory to the Redhat system after modifying and compiling the kernel image file Image. Then follow the similar steps above. We can use the linux-0.1X environment described above to compile the kernel, then compress the kernel source tree containing the Image file, then use the mcopy command to write to the second floppy image file of Bochs, and finally use WinImage software or the mount command to extract the compressed file. The basic steps of the process of compiling and extracting files are given below.

1. Run Linux-0.1X under Bochs, enter the directory /usr/src/, and create the directory 'linux-gdb ';
2. Use the command to copy the entire 0.1X kernel source tree first: "cp -a linux linux-gdb/". Then enter the linux-gdb/linux/ directory, modify all Makefiles as described above, and compile the kernel;
3. Go back to the /usr/src/ directory and use the 'tar' command to compress the linux-gdb/ directory to get the 'linux-gdb.tgz' file.
4. Copy the compressed file to the second floppy (b drive) image file: "mcopy linux-gdb.tgz b:". If the b disk space is not enough, please use the delete file command "mdel b: file name" to make some space on the b disk.
5. If the host environment is a Windows operating system, then use WinImage to extract the compressed

file in the b disk image file and put it into the Redhat system through the FTP or other methods; if the host environment is originally Redhat or other modern Linux system, then Use the 'mount' command to load the b disk image file and copy the compressed kernel file from it.

6. Decompressing the copied compressed file on the modern Linux system will generate a linux-gdb/ directory containing the 0.1X kernel source tree. Go to the linux-gdb/ directory and create the bochs configuration file 'bochsrc-fd1-gdb.bxrc'. You can also take the contents of the 'bochsrc-fdb.bxrc' configuration file from the 'linux-0.11-devel' package and add the 'gdbstub' parameter line yourself. Then download the 'rootimage-0.11' root file system floppy image file from the oldlinux.org website, which is also saved in the linux-gdb/ directory.

After that we can continue to perform the source code debugging experiment according to the steps in the previous section. Below is an example of the above steps. Let's assume that the host environment is a Redhat system and run the Linux 0.11 system in Bochs.

```
[/usr/root]# cd /usr/src                          // Enter the source code directory.
[/usr/src]# mkdir linux-gdb                       // Ceate directory linux-gdb/
[/usr/src]# cp -a linux linux-gdb/                // Copy source code to linux-ddb/
[/usr/src]# cd linux-gdb/linux
[/usr/src/linux-gdb/linux]# vi Makefile           // Modify the Makefiles.
...
[/usr/src/linux-gdb/linux]# make clean; make      // Compling the kernel.
...
[/usr/src/linux-gdb/linux]# cd ../../
[/usr/src]# tar zcvf linux-gdb.tgz linux-gdb       // Create compressed file.
...
[/usr/src]# mdir b:                               // Check the contents of b disk.
 Volume in drive B is Bt
 Directory for B:/
LINUX-GD TGZ      827000    6-18-105  10:28p
TPUT     TAR      184320    3-09-132   3:16p
LILO     TAR      235520    3-09-132   6:00p
SHOELA~1 Z        101767    9-19-104   1:24p
SYSTEM   MAP       17771   10-05-104  11:22p
       5 File(s)      90624 bytes free
[/usr/src]# mdel b:linux-gd.tgz                   // Space not enough, delete some file.
[/usr/src]# mcopy linux-gdb.tgz b:                // Copy the linux-gdb.tgz to b disk.
Copying LINUX-GD.TGZ
[/usr/src]#
```

After closing the Bochs system, we get a compressed file named 'LINUX-GD.TGZ' in the b disk image file. The debug experiment directory can be established by using the following command sequence in the Redhat Linux host environment.

```
[root@plinux 0.11]# mount -t msdos diskb.img /mnt/d4 -o loop,r    // Mount b disk image file.
[root@plinux 0.11]# ls -l /mnt/d4                                 // Check contents.
total 1234
-rwxr-xr-x    1 root     root       235520 Mar  9  2032 lilo.tar
-rwxr-xr-x    1 root     root       723438 Jun 19  2005 linux-gd.tgz
-rwxr-xr-x    1 root     root       101767 Sep 19  2004 shoela~1.z
```

```
-rwxr-xr-x    1 root     root        17771 Oct  5  2004 system.map
-rwxr-xr-x    1 root     root       184320 Mar  9  2032 tput.tar
[root@plinux 0.11]# cp /mnt/d4/linux-gd.tgz .                      // Copy the file.
[root@plinux 0.11]# umount /mnt/d4                                 // Unmount the b disk.
[root@plinux 0.11]# tar zxvf linux-gd.tgz                          // Untar the file.
...
[root@plinux 0.11]# cd linux-gdb
[root@plinux linux-gdb]# ls -l
total 4
drwx--x--x   10 15806    root         4096 Jun 19  2005 linux
[root@plinux linux-gdb]#
```

After that, we also need to create the Bochs configuration file 'bochsrc-fd1-gdb.bxrc' in the linux-gdb/ directory and download the floppy root file system image file 'rootimage-0.11'. For convenience, we can also create a script file 'run' containing only one line of "bochs -q -f bochsrc-fd1-gdb.bxrc" and set the file attribute to executable. In addition, a package for direct debugging experiments has been created for everyone on oldlinux.org, which contains the same content as the package compiled directly under Redhat:

```
    http://oldlinux.org/Linux.old/bochs/linux-0.11-gdb-050619.tar.gz
```

17.12 Summary

This is the last chapter of the book. This chapter describes the experimental operation of Linux 0.1X using the Bochs simulation environment. The basic usage of the Bochs system is given. The method of transferring files between the simulation system and the host system is described in detail. It also gives specific methods and steps for compiling and debugging the Linux 0.1X kernel.

Although the content of this book has ended so far, I hope that readers will regard this as the starting point of the new journey, and begin to further learn and study the new technologies and new functions used in the kernel code of today's Linux system. Thanks again to all the friends who have strong will to accompany the author's poor writings here! I wish you all a happy new journey. Thank you!

References

[1] Intel Co. INTEL 80386 Programmer's Reference Manual 1986, INTEL CORPORATION,1987.

[2] Intel Co. IA-32 Intel Architecture Software Developer's Manual Volume.3:System Programming Guide. http://www.intel.com/, 2005.

[3] James L. Turley. Advanced 80386 Programming Technigues. Osborne McGraw-Hill,1988.

[4] Brian W. Kernighan, Dennis M. Ritchie. The C programming Language. Prentice-Hall 1988.

[5] Leland L. Beck. System Software: An Introduction to Systems Programming,3nd. Addison-Wesley,1997.

[6] Richard Stallman, Using and Porting the GNU Compiler Collection,the Free Software Foundation, 1998.

[7] The Open Group Base Specifications Issue 6 IEEE Std 1003.1-2001, The IEEE and The Open Group.

[8] David A Rusling, The Linux Kernel, 1999. http://www.tldp.org/

[9] Linux Kernel Source Code，http://www.kernel.org/

[10] Digital co.ltd. VT100 User Guide, http://www.vt100.net/

[11] Clark L. Coleman. Using Inline Assembly with gcc. http://oldlinux.org/Linux.old/

[12] John H. Crawford, Patrick P. Gelsinger. Programming the 80386. Sybex, 1988.

[13] FreeBSD Online Manual, http://www.freebsd.org/cgi/man.cgi

[14] Andrew S.Tanenbaum.Operating Systems: Design and Implementation. Prentice-Hall-International Editions. 1990.4

[15] Maurice J. Bach. The Design of the UNIX Operating System. Prentice Hall. 1990

[16] John Lions. Lions' Commentary on UNIX 6th Edition with Source Code. Peer-to-Peer Communications, Inc. 1996

[17] Andrew S. Tanenbaum, Albert S. Woodhull. Operating Systems:Design and Implementation (Second Edition). Prentice Hall. 1997.

[18] Alessandro Rubini, Jonathan. Linux Device Drivers. O'Reilly & Associates. Inc. 2001

[19] Daniel P. Bovet, Marco Cesati. Understanding The Linux Kernel. China Electric Power Press. 2001.

[20] 张载鸿. 微型机(PC 系列)接口控制教程, 清华大学出版社，1992.

[21] 李凤华，周利华，赵丽松. MS-DOS 5.0 内核剖析. 西安电子科技大学出版社，1992.

[22] RedHat 9.0 Online manual. http://www.plinux.org/cgi-bin/man.cgi

[23] W.Richard Stevens. Advanced Programming in the UNIX Environment. China Machine Press. 2000.2

[24] Linux Weekly Edition News. http://lwn.net/

[25] P.J. Plauger. The Standard C Library. Prentice Hall, 1992

[26] Free Software Foundation. The GNU C Library. http://www.gnu.org/ 2001

[27] Chuck Allison. The Standard C Library. C/C++ Users Journal CD-ROM, Release 6. 2003

[28] Bochs simulation system. http://bochs.sourceforge.net/

[29] Brennan "Bas" Underwood. Brennan's Guide to Inline Assembly. http://www.rt66.com/~brennan/

[30] John R. Levine. Linkers & Loaders. http://www.iecc.com/linker/

[31] Randal E. Bryant, David R. O'Hallaron. Computer Systems A programmer's Perspective. Publishing House of Electronics Industry. 2004.3

[32] Intel. Data Sheet: 8254 Programmable Interval Timer. 1993.9

[33] Intel. Data Sheet: 8259A Programmable Interrupt Controller. 1988.12

[34] Intel. Data Sheet: 82077A CHMOS Single-chip Floppy Disk Controller. 1994.5

[35] Robert Love. Linux Kernel Development. China Machine Press. 2004

[36] Adam Chapweske. The PS/2 Keyboard Interface. http://www.computer-engineering.org/

[37] Dean Elsner, Jay Fenlason & friends. Using as: The GNU Assembler. http://www.gnu.org/ 1998

[38] Steve Chamberlain. Using ld: The GNU linker. http://www.gnu.org/ 1998

[39] Michael K. Johnson. The Linux Kernel Hackers' Guide. http://www.tldp.org/ 1995

[40] Richard F. Ferraro. Programmer’s Guide to the EGA, VGA, and Super VGA Cards. 3rd ed. Addison-Wesley, 1995.

Appendix

A1 ASCII Code Table

Decimal	Hex	Character	Decimal	Hex	Character	Decimal	Hex	Character
0	00	NUL	43	2B	+	86	56	V
1	01	SOH	44	2C	,	87	57	W
2	02	STX	45	2D	-	88	58	X
3	03	ETX	46	2E	.	89	59	Y
4	04	EOT	47	2F	/	90	5A	Z
5	05	ENQ	48	30	0	91	5B	[
6	06	ACK	49	31	1	92	5C	\
7	07	BEL	50	32	2	93	5D	]
8	08	BS	51	33	3	94	5E	ˆ
9	09	TAB	52	34	4	95	5F	_
10	0A	LF	53	35	5	96	60	`
11	0B	VT	54	36	6	97	61	a
12	0C	FF	55	37	7	98	62	b
13	0D	CR	56	38	8	99	63	c
14	0E	SO	57	39	9	100	64	d
15	0F	SI	58	3A	:	101	65	e
16	10	DLE	59	3B	;	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6A	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20	(space)	75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	″	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	|
39	27	’	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	˜
41	29	)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

A2 Common C0, C1 Control Characters

Common C0 control characters table

Mnemonic	Code	Actions taken
NUL	0x00	Null -- Ignored when received (not saved in the input buffer).
ENQ	0x05	Enquiry -- Sends a reply message.
BEL	0x07	Bell -- makes a sound.
BS	0x08	Backspace -- Moves the cursor one character position to the left. If the cursor is already on the left edge, there is no action.
HT	0x09	Horizontal Tabulation -- Moves the cursor to the next tab stop. If there is no tab stop on the right side, move to the right edge.
LF	0x0a	Linefeed -- This code causes a carriage return or line feed operation (see linefeed mode).
VT	0x0b	Vertical Tabulation -- acts like LF.
FF	0x0c	Form Feed -- acts like LF.
CR	0x0d	Carriage Return -- Moves the cursor to the left edge of the current line.
SO	0x0e	Shift Out -- Uses the G1 character set selected by the SCS control sequence. G1 can specify one of five character sets.
SI	0x0f	Shift In -- Uses the G0 character set selected by the SCS control sequence. G0 can specify one of five character sets.
DC1	0x11	Device Control 1 -- XON. Let the terminal resume transmission.
DC3	0x13	Device Control 3 -- XOFF. Stop sending all other codes except sending XOFF and XON.
CAN	0x18	Cancel -- If sent during a control sequence, the sequence will not execute and will terminate immediately. The error character is also displayed.
SUB	0x1a	Substitute -- works the same as CAN.
ESC	0x1b	Escape -- Generates an Escape Control Sequence.
DEL	0x7f	Delete -- Ignore when typing (not saved in the input buffer).

Common C1 control characters table

Mnemonic	Code	7B seq.	Actions taken
IND	0x84	ESC D	Index -- The cursor moves down one row in the same column. If the cursor is already on the bottom line, a scrolling operation is performed.
NEL	0x85	ESC H	Next Line -- The cursor moves to the first column of the next line. If the cursor is already on the bottom line, a scrolling operation is performed.
HTS	0x88	ESC E	Horizontal Tab Set -- Sets a horizontal tab stop at the cursor.
RI	0x8d	ESC M	Reverse Index -- The cursor moves one line up the same column. If the cursor is already on the top line, a scrolling operation is performed.
SS2	0x8e	ESC N	Single Shift G2 -- Temporarily uses the G2 character set in GL for the display of the next character. G2 is specified by the Selective Character Set (SCS) control sequence (see the escape sequence and control sequence table in Appendix 3).
SS3	0x8f	ESC O	Single Shift G3 -- Temporarily calls the G3 character set in GL for the display of the next character. G3 is specified by the Selective Character Set (SCS) control sequence (see the

			escape sequence and control sequence table in Appendix 3).
DCS	0x90	ESC P	Device Control String -- Used as the starting qualifier of the device control string.
CSI	0x9b	ESC [	Control Sequence Introducer -- Used as a control sequence leader code.
ST	0x9c	ESC \	String Terminator -- Used as the ending qualifier of the DCS string.

A3 Escape and Control Sequences

Sequence and Name	Description
ESC (Ps or ESC) Ps Select Character Set	Select Character Set (SCS) -- The G0 and G1 character sets can each specify one of five character sets. 'ESC (Ps' specifies the character set used by G0, 'ESC) Ps' specifies the character set used by G1. Parameters Ps: A - UK character set; B - US character set; 0 - graphic character set; 1 - alternative ROM character set; 2 - optional ROM special character set. A terminal can display up to 254 different characters, however the terminal only stores 127 display characters in its ROM. You must install additional character set ROM for the other 127 display characters. At some point, the terminal is able to select 94 characters (one character set). Therefore, the terminal can use one of five character sets, some of which appear in multiple character sets. At any one time, the terminal can use two active character sets. The computer can use the SCS sequence to specify any two character sets as G0 and G1. You can then switch between these two character sets using a single control character. The Shift In - SI (14) control character is used to select the G0 character set, and the Shift Out - SO (15) control character can be used to select the G1 character set. The specified character set will be used as the current character set until the terminal receives another SCS sequence.
ESC [Pn A Cursor up (Terminal <--> Host)	Cursor Up (CUU) -- The CUU control sequence moves the cursor up but the column position is unchanged. The number of moving character positions is determined by parameters. If the parameter is 'Pn', the cursor moves up the 'Pn' line. The cursor is moved up to the top row at most. Note that 'Pn' is an ASCII numeric variable. If you do not select a parameter or the parameter value is 0, the terminal will assume a parameter value of 1.
ESC [Pn B or ESC [Pn e Cursor Down (Terminal <--> Host)	Cursor Down (CUD) -- The CUD control sequence moves the cursor down but the column position is unchanged. The number of moving character positions is determined by parameters. If the parameter is 1 or 0, the cursor moves down 1 line. If the parameter is 'Pn', the cursor moves down the 'Pn' line. The cursor moves down to the bottom line at most.
ESC [Pn C or ESC [Pn a Cursor Forward (Terminal <--> Host)	Cursor Forward (CUF) -- The CUF control sequence moves the current cursor to the right. The number of moving positions is determined by parameters. If the argument is 1 or 0, move 1 character position. If the parameter value is 'Pn', the cursor moves by 'Pn' character positions. The cursor moves up to the right border at most.
ESC [Pn D Cursor Backward (Terminal <--> Host)	Cursor Backward (CUB) -- The CUB control sequence moves the current cursor to the left. The number of moving positions is determined by parameters. If the argument is 1 or 0, move 1 character position. If the parameter value is 'Pn', the cursor moves by 'Pn' character positions. The cursor moves up to the left border at most.
ESC [Pn E Cursor moves down	Cursor Next Line (CNL) -- This control sequence moves the cursor to the first character of the 'Pn' line below.
ESC [Pn F Cursor moves up	Cursor Last Line (CLL) -- This control sequence moves the cursor up to the first character of the 'Pn' line.
ESC [Pn G or ESC [Pn ` Cursor moves in line	Cursor Horizon Absolute (CHA) -- This control sequence moves the cursor to the 'Pn' character position of the current line.
ESC [Pn ; Pn H or	Cursor Position (CUP), Horizontal And Vertical Position (HVP) -- The CUP control sequence moves

ESC [Pn;Pn f Cursor positioning	the current cursor to the position specified by the parameter. The two parameters specify the row and column values, respectively. If the value is 0, it is the same as 1, indicating that one position is moved. In the default condition without parameters, it is equivalent to moving the cursor to the home position (ie ESC [H).
ESC [Pn d Set the row position	Vertical Line Position Absolute -- Moves the cursor to the 'Pn' line of the current column. If you try to move below the last line, the cursor will stay on the last line.
ESC [s Save cursor position	Save Current Cursor Position -- This control sequence has the same effect as DECSC except that the page number displayed on the cursor is not saved.
ESC [u Restore cursor position	Restore Saved Cursor Position -- This control sequence has the same effect as DECRC except that the cursor is still on the same display page and not moved to the display page where the cursor is saved.
ESC D Index	Index (IND) -- This control sequence moves the cursor down one line, but the column number does not change. If the cursor is on the bottom line, it will cause the screen to scroll up one line.
ESC M Reverse index	Reverse Index (RI) -- This control sequence moves the cursor up one line, but the column number does not change. If the cursor is on the top line, it will cause the screen to scroll down one line.
ESC E Move down one line	Next Line (NEL) -- This control sequence will move the cursor to the beginning of the left side of the next line. If the cursor is on the bottom line, it will cause the screen to scroll up one line.
ESC 7 Save cursor	Save Cursor (DECSC) -- This control sequence will cause the cursor position, graphics to be reproduced, and the character set to be saved.
ESC 8 Restore cursor	Restore Cursor (DECRC) -- This control sequence will cause the previously saved cursor position, graphics to be reproduced, and the character set to be restored.
ESC [Ps; Ps; … ; Ps m Set character attributes	Select Graphic Rendition (SGR) - Character re-display and attribute are characteristics that affect the display of a character without changing the character code. The control sequence sets the character display attributes according to the parameters. All characters sent to the terminal in the future will use the attributes specified here until the control sequence reset the attributes of the character again. Parameter 'Ps': 0 - no attribute (default attribute); 1 - bold and bright; 4 - underline; 5 - flashing; 7 - reverse; 22 - non-bold; 24 - no underline; 25 - no flicker; ;30--38 Set foreground color; 39 - Default foreground color (White); 40--48 - Set background color; 49 - Default background color (Black). 30--37 and 40-47 correspond to colors: Black, Red, Green, Yellow, Blue, Magenta, Cyan, White.
ESC [Pn L Insert line	Insert Line (IL) -- This control sequence inserts one or more blank lines at the cursor. The cursor position does not change after the operation is completed. When a blank line is inserted, the line in the scroll area below the cursor moves down. The line scrolling out of the display page is lost.
ESC [Pn M Delete line	Delete Line (DL) -- This control sequence deletes one or more lines from the line where the cursor is located in the scroll area. When the line is deleted, the line below the deleted line in the scroll area moves up, and 1 blank line is added to the bottom line. If 'Pn' is greater than the number of lines remaining on the display page, then this sequence only deletes these remaining lines and does not work outside the scroll area.
ESC [Pn @ Insert character	Insert Character (ICH) -- This control sequence inserts one or more space characters at the current cursor using the normal character attribute. 'Pn' is the number of characters inserted. The default is 1. The cursor will still be at the first inserted space character. The character at the cursor and right border will shift to the right. Characters that exceed the right border will be lost.
ESC [Pn P Delete character	Delete Character (DCH) -- This control sequence deletes 'Pn' characters from the cursor. When a character is deleted, all characters to the right of the cursor are shifted to the left. This will produce a

<table>
<tr><td></td><td>null character at the right border. Its properties are the same as the last left-shift character.</td></tr>
<tr><td>ESC [Ps J
Erase character</td><td>Erase In Display (ED) -- This control sequence erases some or all of the displayed characters, depending on the parameters. The erase operation removes characters from the screen without affecting other characters. The erased characters are discarded. The cursor position does not change when erasing characters or lines. While erasing characters, the attributes of the characters are also discarded. Any entire line erased by this control sequence will return the line to a single character width mode. Parameter 'Ps':
0 - Erase the cursor to all characters at the bottom of the screen; 1 - Erase the top of the screen to the cursor except all characters; 2 - Erase the entire screen.</td></tr>
<tr><td>ESC [Ps K
Erase in line</td><td>Erase In Line (EL) -- Erases some or all of the characters in the line of the cursor according to the parameters. The erase operation removes characters from the screen without affecting other characters. The erased characters are discarded. The cursor position does not change when erasing characters or lines. While erasing characters, the attributes of the characters are also discarded. Parameter 'Ps':
0 - Erases the cursor to all characters at the end of the line; 1 - Erases the left border to all characters at the cursor; 2 - Erases an entire line.</td></tr>
<tr><td>ESC [Pn ; Pn r
Set top &bottom margins</td><td>Set Top and Bottom Margins (DECSTBM) -- This control sequence sets the upper and lower areas of the scroll screen. The scrolling margin is an area on the screen where we can receive new characters by taking away the original characters from the screen. This area is defined by the top and bottom borders of the screen. The first parameter is the first line of the start of the scrolling area, and the second parameter is the last line of the scrolling area. By default it is the entire screen. The smallest scrolling area is 2 lines, ie the top border line must be smaller than the bottom border line. The cursor will be placed in the home position.</td></tr>
<tr><td>ESC [Pn c or
ESC Z
Device attributes
(Terminal <--> Host)</td><td>In response to a host request, the terminal can send a report message. These messages provide the identity (terminal type), cursor position, and terminal operational status. There are two types of reports: device attributes and device status reports. Device Attributes (DA) -- The host sends a device attribute (DA) control sequence (the same as ESC Z) with no parameters or parameter 0. The terminal sends one of the following sequences in response to the host's sequence.
<table>
<tr><th>Terminal optional attribute</th><th>Send sequence</th></tr>
<tr><td>None, VT101</td><td>ESC [?1;0c</td></tr>
<tr><td>Processor option (STP)</td><td>ESC [?1;1c</td></tr>
<tr><td>Advance Video (AVO) VT100</td><td>ESC [?1;2c</td></tr>
<tr><td>AVO and STP</td><td>ESC [?1;3c</td></tr>
<tr><td>Graphic property option (GPO)</td><td>ESC [?1;4c</td></tr>
<tr><td>GPO and STP</td><td>ESC [?1;5c</td></tr>
<tr><td>GPO and AVO, VT102</td><td>ESC [?1;6c</td></tr>
<tr><td>GPO, STP and AVO</td><td>ESC [?1;7c</td></tr>
</table></td></tr>
<tr><td>ESC c
Reset to initial state</td><td>Reset To Initial State (RIS) -- Lets the terminal reset to its initial state, that is, just turned on. All characters received during the reset phase will be lost. There are two ways to avoid this: 1. (Auto XON/XOFF) After the transmission, the host assumes that the terminal has sent XOFF. The host stops sending characters until it receives XON. 2. Delay at least 10 seconds and wait for the terminal reset operation to complete.</td></tr>
</table>

A4 The First Set of Keyboard Scan Code

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
A	1E	9E	9	0A	8A	[	1A	9A
B	30	B0	`	29	89	INSERT	E0,52	E0,D2
C	2E	AE	-	0C	8C	HOME	E0,47	E0,97
D	20	A0	=	0D	8D	PG UP	E0,49	E0,C9
E	12	92	\	2B	AB	DELETE	E0,53	E0,D3
F	21	A1	BKSP	0E	8E	END	E0,4F	E0,CF
G	22	A2	SPACE	39	B9	PG DN	E0,51	E0,D1
H	23	A3	TAB	0F	8F	Up Arrow	E0,48	E0,C8
I	17	97	CAPS	3A	BA	Left Arrow	E0,4B	E0,CB
J	24	A4	Left SHFT	2A	AA	Down Arrow	E0,50	E0,D0
K	25	A5	Left CTRL	1D	9D	Right Arrow	E0,4D	E0,CD
L	26	A6	Left GUI	E0,5B	E0,DB	NUM LOCK	45	C5
M	32	B2	Left ALT	38	B8	KP /	E0,35	E0,B5
N	31	B1	Right SHFT	36	B6	KP *	37	B7
O	18	98	Right CTRL	E0,1D	E0,9D	KP -	4A	CA
P	19	99	Right GUI	E0,5C	E0,DC	KP +	4E	CE
Q	10	90	Right ALT	E0,38	E0,B8	KP ENTER	E0,1C	E0,9C
R	13	93	APPS	E0,5D	E0,DD	KP .	53	D3
S	1F	9F	ENTER	1C	9C	KP 0	52	D2
T	14	94	ESC	01	81	KP 1	4F	CF
U	16	96	F1	3B	BB	KP 2	50	D0
V	2F	AF	F2	3C	BC	KP 3	51	D1
W	11	91	F3	3D	BD	KP 4	4B	CB
X	2D	AD	F4	3E	BE	KP 5	4C	CC
Y	15	95	F5	3F	BF	KP 6	4D	CD
Z	2C	AC	F6	40	C0	KP 7	47	C7
0	0B	8B	F7	41	C1	KP 8	48	C8
1	02	82	F8	42	C2	KP 9	49	C9
2	03	83	F9	43	C3	]	1B	9B
3	04	84	F10	44	C4	;	27	A7
4	05	85	F11	57	D7	’	28	A8
5	06	86	F12	58	D8	,	33	B3
6	07	87	PRNT SCRN	E0,2A, E0,37	E0,B7, E0,AA	.	34	B4
7	08	88	SCROLL	46	C6	/	35	B5
8	09	89	PAUSE	E1,1D,45 E1,9D,C5	无			

Note 1: All values in the table are in hexadecimal.

Note 2: In the table, KP - KeyPad, represents the key on the numeric keypad.

Note 3: The colored parts in the table are all extended keys.