MARCUS BENNETT

THE BELL SYSTEM TECHNICAL JOURNAL

JULY–AUGUST 1978
VOL. 57, NO. 6, PART 2

ISSN0005–8580

UNIX TIME-SHARING SYSTEM

(Contents continued on outside back cover)

THE BELL SYSTEM TECHNICAL JOURNAL

THE BELL SYSTEM TECHNICAL JOURNAL is published monthly, except for the May-June and July-August combined issues, by the American Telephone and Telegraph Company, J. D. deButts, Chairman and Chief Executive Officer; C. L. Brown, President; W. G. Burns, Vice President and Treasurer; F. A. Hutson, Jr., Secretary. Editorial enquiries should be addressed to the Editor, The Bell System Technical Journal, Bell Laboratories, 600 Mountain Ave., Murray Hill, N.J. 07974. Checks for subscriptions should be made payable to The Bell System Technical Journal and should be addressed to Bell Laboratories, Circulation Group, Whippany Road, Whippany, N.J. 07981. Subscriptions $20.00 per year; single copies $2.00 each. Foreign postage $1.00 per year; 15 cents per copy. Printed in U.S.A. Second-class postage paid at New Providence, New Jersey 07974 and additional mailing offices.

THE BELL SYSTEM TECHNICAL JOURNAL

DEVOTED TO THE SCIENTIFIC AND ENGINEERING
ASPECTS OF ELECTRICAL COMMUNICATION

Volume 57 July–August 1978 Number 6, Part 2

UNIX Time-Sharing System:

Preface

By T. H. CROWLEY

(Manuscript received April 18, 1978)

Since 1962, *The Bell System Technical Journal* has published over 90 articles on computer programming. Although that number is not insignificant, it is only about 6 percent of all the articles published in the B.S.T.J. during that period. Publications in the B.S.T.J. tend to reflect the amount of activity in many areas of technology at Bell Laboratories, but that has certainly not been true for computer programming work. Better indicators of the importance of programming for current Bell Laboratories work are the following:

(*i*) 25 percent of the technical staff spent more than 50 percent of their time on programming, or related work, in 1977.
(*ii*) 25 percent of the professional staff recruited in 1977 majored in computer science.
(*iii*) 40 percent of the employees entering the Bell Laboratories Graduate Study Program in 1977 are majoring in computer science.

Programming activities under way at Bell Laboratories cover a very broad spectrum. They range from basic research on compiler-

generating techniques to the maintenance of Bell Laboratories-developed programs now in routine use at operating telephone companies. They include writing of real-time control programs for switching systems, development of time-shared text editing facilities, and design of massive data-base systems. They involve work on microprocessors, minicomputers, and maxicomputers. They extend from the design of sophisticated programming tools to be used only by experts to the delivery of program products to be used by clerks in operating telephone companies. They include programming for computers made by all the major computer hardware vendors as well as programming for special-purpose computers designed at Bell Laboratories and built by the Western Electric Company.

Because computer science is still in an early stage of development, no well-formulated theoretical structure exists around which problems can be defined and results organized. "Elegance" is of prime importance, but is not easily defined or described. Reliability and maintainability are important, but they also are neither precisely defined nor easily measured.

No single issue of the B.S.T.J. can characterize all of Bell Laboratories software activities. However, by using the UNIX* operating system as a central theme, it has been possible to assemble a number of related articles that do provide some idea of the importance of computer programming to Bell Laboratories. The original design of the UNIX system was an elegant piece of work done in the research area, and that design has proven useful in many applications. The range of applications described here typifies much of Bell Laboratories software work with the notable omissions of real-time programming for switching control systems and the design of very large data-base systems. Given the growing importance of computers to the Bell System and the growing importance of programming to the use of computers, it is certain that computer programming will continue to grow in importance at Bell Laboratories.

* UNIX is a trademark of Bell Laboratories.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

Foreword

by M. D. McILROY, E. N. PINSON, and B. A. TAGUE
(Manuscript received March 17, 1978)

Intelligence ... is the faculty of making artificial objects, especially tools to make tools. — Bergson

UNIX is a trademark for a family of computer operating systems developed at Bell Laboratories. Over 300 of these systems, which run on small to large minicomputers, are used in the Bell System for program development, for support of telephone operations, for text processing, and for general-purpose computing; even more have been licensed to outside users. The papers in this issue describe highlights of the UNIX family, some important uses, and some UNIX software tools. They also attempt to convey a feeling for the particular style or outlook on program design that is both manifest in UNIX software and promoted by it.

The UNIX story begins with Ken Thompson's work on a cast-off PDP-7 minicomputer in 1969. He and the others who soon joined him had one overriding objective: to create a computing environment where they themselves could comfortably and effectively pursue their own work—programming research. The result is an operating system of unusual simplicity, generality, and, above all, intelligibility. A distinctive software style has grown upon this base. UNIX software works smoothly together; elaborate computing tasks are typically composed from loosely coupled small parts, often software tools taken off the shelf.

The growth and flowering of UNIX as a highly effective and reliable

time-sharing system are detailed in the prizewinning ACM paper by Ritchie and Thompson that has been updated for this volume. That paper describes the operating system proper and lists the important utility programs that have been adopted by descendant systems as well. There is no more concise summary of the UNIX time-sharing system than the oft-quoted passage from Ritchie and Thompson:

> It offers a number of features seldom found even in larger operating systems, including
> (*i*) A hierarchical file system incorporating demountable volumes,
> (*ii*) Compatible file, device, and inter-process I/O,
> (*iii*) The ability to initiate asynchronous processes,
> (*iv*) System command language selectable on a per-user basis,
> (*v*) Over 100 subsystems including a dozen languages.

Implementation details are covered in a separate paper by Thompson. Matters of efficiency and design philosophy are considered in a retrospective paper by Ritchie.

The most visible system interface is the "shell," or command language interpreter, through which other programs are called into execution singly or in combination. The shell, described by Bourne, is actually a very high level programming language that talks about programs and files. Particularly noteworthy are its notations for input-output connections. By making it easy to combine programs, the shell fosters small, coherent software modules.

The UNIX system and most software that runs under it are programmed in the general-purpose procedural language C. C provides almost the full capability of popular instruction sets in a setting of structured code, structured data, and modular compilation. C is easy to write and (when well-written) easy to read. The language and the philosophy behind it are covered by Ritchie, Johnson, Lesk, and Kernighan.

Until mid-1977, the UNIX operating system and its variants ran only on computers of the Digital Equipment Corporation PDP-11 family. In an interesting exercise in portability, Johnson and Ritchie exploited the machine-independence of C to move the operating system and the bulk of its software to a quite different Interdata machine. Careful parameterization and some repackaging have made it possible to use largely identical source code for both machines.

Variations

Three papers by Bayer, Lycklama, and Christensen describe

variations on the UNIX operating system that were developed to accommodate real-time processing, microprocessor systems, and laboratory support applications. They were motivated by the desire to retain the benefits of the UNIX system for program development while offering different trade-offs to the user in real-time response, hardware requirements, and resource management for production programs. Many UNIX utilities—especially those useful for writing programs and processing text—will run under any of these variant systems without change.

The MERT operating system (Lycklama and Bayer) provides a generalized kernel that permits extensive interprocess communication and direct user control of peripherals, scheduling, and storage management. Applications with stringent requirements for real-time response, and even different operating systems (in particular, UNIX) can be operated simultaneously under the MERT kernel.

The microprocessor version of the UNIX operating system (Lycklama) and the Satellite Processing System that shares process execution between one big and one tiny machine (Lycklama and Christensen) involve other trade-offs between efficiency and resource requirements. Both also may be looked upon as vehicles for applications in which one wishes to delegate some sticky part of the job—frequently involving real-time demands—to a dedicated machine. The application described later in the issue by Wonsiewicz, Storm, and Sieber is a particularly interesting example involving UNIX, the microprocessor system, and the Satellite Processing System.

Software Tools

Perhaps the most widely used UNIX programs are the utilities for the editing, transformation, analysis, and publication of text of all sorts. Indeed, the text-processing utilities covered by Kernighan, Lesk, and Ossanna were used to produce this issue of the B.S.T.J. Some more unusual applications that become possible where text processors and plenty of text are ready at hand are described by McMahon, Morris, and Cherry.

UNIX utilities are usually thought of as tools—sharply honed programs that help with generic data processing tasks. Tools were often invented to help with the development of UNIX programs and were continually improved by much trial, error, discussion, and redesign, as was the operating system itself. Tools may be used in combination to perform or construct specific applications.

Sophisticated tools to make tools have evolved. The basic typesetting programs nroff and troff covered by Kernighan, Lesk, and Ossanna help experts define the layouts for classes of documents; the resulting packages exhibit only what is needed for one particular type of document and are easy for nonspecialists to use. Johnson and Lesk describe Yacc and Lex, tools based in formal language theory that systematize the construction of compiler "front ends." Language processors built with the aid of these tools are typically more precisely defined and freer from error than hand-built counterparts.

The UNIX system was originally designed to help build research software. What worked well in a programming laboratory also worked well on modest projects to develop minicomputer-based systems in support of telephone company operations. Such projects are treated in the final group of papers and are more fully introduced by Luderer, Maranzano, and Tague. The strengths of this environment proved equally attractive to large programming projects building applications for large computers with operating systems that were less tractable for program development. The PWB/UNIX extensions discussed by Dolotta, Haight, and Mashey provide such projects with a "front end" for comfortable and effective program development and documentation, together with administrative tools to handle massive projects.

Style

A number of maxims have gained currency among the builders and users of the UNIX system to explain and promote its characteristic style:

(*i*) Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features."

(*ii*) Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.

(*iii*) Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.

(*iv*) Use tools in preference to unskilled help to lighten a

programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

Illustrations of these maxims are legion:

(*i*) Surprising to outsiders is the fact that UNIX compilers produce no listings: printing can be done better and more flexibly by a separate program.

(*ii*) Unexpected uses of files abound: programs may be compiled to be run and also typeset to be published in a book from the same text without human intervention; text intended for publication serves as grist for statistical studies of English to help in data compression or cryptography; mailing lists turn into maps. The prevalence of free-format text, even in "data" files, makes the text-processing utilities useful for many strictly data processing functions such as shuffling fields, counting, or collating.

(*iii*) The UNIX system and the C language themselves evolved by deliberate steps from early working models that had at most a few man-months invested in them. Both have been fully recoded several times by the same people who designed them, with as much mechanical aid as possible.

(*iv*) The use of tools instead of labor is nicely illustrated by typesetting. When a paper needs a new layout for some reason, the typographic conventions for paragraphs, subheadings, etc. are entered in one place, then the paper is run off in the new shape without retyping a single word.

To many, the UNIX systems embody Schumacher's dictum, "Small is beautiful." On the other hand it has been argued by Brooks in *The Mythical Man Month,* for example, that small is unreal; the working of a handful of people doesn't extrapolate to the world of big jobs. We agree only in part, for the present volume demonstrates with unusual force another important factor: intelligently applied computing technology can compress jobs that used to be big to manageable size. The first system had only about 5 man-years' work in it (including operating system, assembler, Fortran, and many other utilities) when it began to be used for Bell System projects. It was, to be sure, a taut package that lacked the gamut of libraries, languages, and support for peripheral equipment typical of a large commercial system. But the base was unusually

pliable and responsive; new facilities usually could be added with much less work than is required by corresponding features in other systems.

The UNIX operating system, the C programming language, and the many tools and techniques developed in this environment are finding extensive use within the Bell System and at universities, government laboratories, and other commercial installations. The style of computing encouraged by this environment is influencing a new generation of programmers and system designers. This, perhaps, is the most exciting part of the UNIX story, for the increased productivity fostered by a friendly environment and quality tools is essential to meet ever-increasing demands for software. UNIX is not the end of the road in operating system innovations, but it has been a significant step that Bell Laboratories people are proud to have originated.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

The UNIX Time-Sharing System†

by D. M. RITCHIE and K. THOMPSON
(Manuscript received April 3, 1978)

UNIX is a general-purpose, multi-user, interactive operating system for the larger Digital Equipment Corporation PDP-11 and the Interdata 8/32 computers. It offers a number of features seldom found even in larger operating systems, including*

(i) A hierarchical file system incorporating demountable volumes,
(ii) Compatible file, device, and inter-process I/O,
(iii) The ability to initiate asynchronous processes,
(iv) System command language selectable on a per-user basis,
(v) Over 100 subsystems including a dozen languages,
(vi) High degree of portability.

This paper discusses the nature and implementation of the file system and of the user command interface.

I. INTRODUCTION

There have been four versions of the UNIX time-sharing system. The earliest (circa 1969-70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unpro-

† Copyright 1974, Association for Computing Machinery, Inc., reprinted by permission. This is a revised version of an article that appeared in Communications of the ACM, *17*, No. 7 (July 1974), pp. 365-375. That article was a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973.

* UNIX is a trademark of Bell Laboratories.

tected PDP-11/20 computer. The third incorporated multiprogramming and ran on the PDP-11/34, /40, /45, /60, and /70 computers; it is the one described in the previously published version of this paper, and is also the most widely used today. This paper describes only the fourth, current system that runs on the PDP-11/70 and the Interdata 8/32 computers. In fact, the differences among the various systems is rather small; most of the revisions made to the originally published version of this paper, aside from those concerned with style, had to do with details of the implementation of the file system.

Since PDP-11 UNIX became operational in February, 1971, over 600 installations have been put into service. Most of them are engaged in applications such as computer science education, the preparation and formatting of documents and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: it can run on hardware costing as little as $40,000, and less than two man-years were spent on the main system software. We hope, however, that users find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the operating system proper, some major programs available under UNIX are

C compiler
Text editor based on QED[1]
Assembler, linking loader, symbolic debugger
Phototypesetting and equation setting programs[2,3]
Dozens of languages including Fortran 77, Basic, Snobol, APL, Algol 68, M6, TMG, Pascal

There is a host of maintenance, utility, recreation and novelty programs, all written locally. The UNIX user community, which numbers in the thousands, has contributed many more programs and languages. It is worth noting that the system is totally self-supporting. All UNIX software is maintained on the system; likewise, this paper and all other documents in this issue were generated and formatted by the UNIX editor and text formatting programs.

II. HARDWARE AND SOFTWARE ENVIRONMENT

The PDP-11/70 on which the Research UNIX system is installed is a 16-bit word (8-bit byte) computer with 768K bytes of core memory; the system kernel occupies 90K bytes about equally divided between code and data tables. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the software mentioned above can require as little as 96K bytes of core altogether. There are even larger installations; see the description of the PWB/UNIX systems,[4,5] for example. There are also much smaller, though somewhat restricted, versions of the system.[6]

Our own PDP-11 has two 200-Mb moving-head disks for file system storage and swapping. There are 20 variable-speed communications interfaces attached to 300- and 1200-baud data sets, and an additional 12 communication lines hard-wired to 9600-baud terminals and satellite computers. There are also several 2400- and 4800-baud synchronous communication interfaces used for machine-to-machine file transfer. Finally, there is a variety of miscellaneous devices including nine-track magnetic tape, a line printer, a voice synthesizer, a phototypesetter, a digital switching network, and a chess machine.

The preponderance of UNIX software is written in the above-mentioned C language.[7] Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system was about one-third greater than that of the old. Since the new system not only became much easier to understand and to modify but also included many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we consider this increase in size quite acceptable.

III. THE FILE SYSTEM

The most important role of the system is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary files

A file contains whatever information the user places on it, for example, symbolic or binary (object) programs. No particular

structuring is expected by the system. A file of text consists simply of a string of characters, with lines demarcated by the newline character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure; for example, the assembler generates, and the loader expects, an object file in a particular format. However, the structure of files is controlled by the programs that use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the **root** directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the **root**. Other system directories contain all the programs provided for general use; that is, all the *commands*. As will be seen, however, it is by no means necessary that a program reside in one of these directories for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name,* which is a sequence of directory names separated by slashes, "/", and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name **/alpha/beta/gamma** causes the system to search the root for directory **alpha**, then to search **alpha** for **beta**, finally to find **gamma** in **beta**. **gamma** may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root itself.

A path name not starting with "/" causes the system to begin the search in the user's current directory. Thus, the name **alpha/beta** specifies the file named **beta** in subdirectory **alpha** of the current directory. The simplest kind of name, for example, **alpha**, refers to a file that itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same non-directory file may appear in several directories under possibly different names. This feature is called *linking;* a directory entry for a file is sometimes called a link. The UNIX system differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name "." in each directory refers to the directory itself. Thus a program may read the current directory under the name "." without knowing its complete path name. The name ".." by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries "." and "..", each directory must appear as an entry in exactly one other directory, which is its parent. The reason for this is to simplify the writing of programs that visit subtrees of the directory structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special files

Special files constitute the most unusual feature of the UNIX file system. Each supported I/O device is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory **/dev**, although a link may be made to one of these files just as it may to an ordinary file. Thus, for example, to write on a magnetic tape one may write on the file **/dev/mt**. Special files exist for each communication line, each disk, each tape drive, and for physical main memory. Of course, the active disks and the memory special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally,

special files are subject to the same protection mechanism as regular files.

3.4 Removable file systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a **mount** system request with two arguments: the name of an existing ordinary file, and the name of a special file whose associated storage volume (e.g., a disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of **mount** is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, **mount** replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the **mount**, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on a small partition of one of our disk drives, while the other drive, which contains the user's files, is mounted by the system initialization sequence. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping that would otherwise be required to assure removal of the links whenever the removable volume is dismounted.

3.5 Protection

Although the access control scheme is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of ten protection bits. Nine of these specify independently read, write, and execute permission for the owner of the file, for other members of his group, and for all remaining users.

If the tenth bit is on, the system will temporarily change the user

identification (hereafter, user ID) of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program that calls for it. The set-user-ID feature provides for privileged programs that may use files inaccessible to other users. For example, a program may keep an accounting file that should neither be read nor changed except by the program itself. If the set-user-ID bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully written commands that call privileged system entries. For example, there is a system entry invokable only by the "super-user" (below) that creates an empty directory. As indicated above, directories are expected to have entries for "." and "..". The command which creates a directory is owned by the super-user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for "." and "..".

Because anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available without administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed by "Aleph-null."[8]

The system recognizes one particular user ID (that of the "super-user") as exempt from the usual constraints on file access; thus (for example), programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between "random" and "sequential" I/O, nor is any logical record size imposed by the system. The size of an ordinary file is determined by the number of bytes written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O, some of the basic calls are summarized below in an anonymous language that will indicate the required parameters without getting into the underlying complexities. Each call to the system may potentially result in an error

into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry found thereby (the file's *i-node*) contains the description of the file:

(*i*) the user and group-ID of its owner
(*ii*) its protection bits
(*iii*) the physical disk or tape addresses for the file contents
(*iv*) its size
(*v*) time of creation, last use, and last modification
(*vi*) the number of links to the file, that is, the number of times it appears in a directory
(*vii*) a code indicating whether the file is a directory, an ordinary file, or a special file.

The purpose of an **open** or **create** system call is to turn the path name given by the user into an i-number by searching the explicitly or implicitly named directories. Once a file is open, its device, i-number, and read/write pointer are stored in a system table indexed by the file descriptor returned by the **open** or **create**. Thus, during a subsequent call to read or write the file, the descriptor may be easily related to the information necessary to access the file.

When a new file is created, an i-node is allocated for it and a directory entry is made that contains the name of the file and the i-node number. Making a link to an existing file involves creating a directory entry with the new name, copying the i-number from the original file entry, and incrementing the link-count field of the i-node. Removing (deleting) a file is done by decrementing the link-count of the i-node specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the i-node is de-allocated.

The space on all disks that contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit that depends on the device. There is space in the i-node of each file for 13 device addresses. For nonspecial files, the first 10 device addresses point at the first 10 blocks of the file. If the file is larger than 10 blocks, the 11 device address points to an indirect block containing up to 128 addresses of additional blocks in the file. Still larger files use the twelfth device address of the i-node to point to a double-indirect block naming 128 indirect blocks, each pointing to 128 blocks of the file. If required, the thirteenth device address is a triple-indirect block. Thus files may conceptually grow to $[(10+128+128^2+128^3)\cdot 512)]$ bytes. Once opened, bytes numbered below 5120 can be read with a single disk access; bytes in the

range 5120 to 70,656 require two accesses; bytes in the range 70,656 to 8,459,264 require three accesses; bytes from there to the largest file (1,082,201,088) require four accesses. In practice, a device cache mechanism (see below) proves effective in eliminating most of the indirect fetches.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last 12 device address words are immaterial, and the first specifies an internal *device name,* which is interpreted as a pair of numbers representing, respectively, a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar terminal interfaces.

In this environment, the implementation of the **mount** system call (Section 3.4) is quite straightforward. **mount** maintains a system table whose argument is the i-number and device name of the ordinary file specified during the **mount**, and whose corresponding value is the device name of the indicated special file. This table is searched for each i-number/device pair that turns up while a path name is being scanned during an **open** or **create**; if a match is found, the i-number is replaced by the i-number of the root directory and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is, immediately after return from a **read** call the data are available; conversely, after a **write** the user's workspace may be reused. In fact, the system maintains a rather complicated buffering mechanism that reduces greatly the number of I/O operations required to access a file. Suppose a **write** call is made specifying transmission of a single byte. The system will search its buffers to see whether the affected disk block currently resides in main memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer and an entry is made in a list of blocks to be written. The return from the **write** call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

The system recognizes when a program has made accesses to sequential blocks of a file, and asynchronously pre-reads the next

block. This significantly reduces the running time of most programs while adding little to system overhead.

A program that reads or writes files in units of 512 bytes has an advantage over a program that reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. If a program is used rarely or does no great volume of I/O, it may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example, verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, because it need only scan the linearly organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, because all directory entries for a file have equal status. Charging the owner of a file is unfair in general, for one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. Many installations avoid the issue by not charging any fees at all.

V. PROCESSES AND IMAGES

An *image* is a computer execution environment. It includes a memory image, general register values, status of open files, current directory and the like. An image is the current state of a pseudo-computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in main memory; during the execution of other processes it remains in main memory unless the appearance of an active, higher-priority process forces it to be swapped out to the disk.

The user-memory part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first hardware protection byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the stack pointer fluctuates.

5.1 Processes

Except while the system is bootstrapping itself into operation, a new process can come into existence only by use of the **fork** system call:

```
processid = fork ( )
```

When **fork** is executed, the process splits into two independently executing processes. The two processes have independent copies of the original memory image, and share all open files. The new processes differ only in that one is considered the parent process: in the parent, the returned **processid** actually identifies the child process and is never 0, while in the child, the returned value is always 0.

Because the values returned by **fork** in the parent and child process are distinguishable, each process may determine whether it is the parent or child.

5.2 Pipes

Processes may communicate with related processes using the same system **read** and **write** calls that are used for file-system I/O. The call:

```
filep = pipe ( )
```

returns a file descriptor **filep** and creates an inter-process channel called a *pipe*. This channel, like other open files, is passed from parent to child process in the image by the **fork** call. A **read** using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between

the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although inter-process communication via pipes is a quite valuable tool (see Section 6.2), it is not a completely general mechanism, because the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of programs

Another major system primitive is invoked by

$$\textbf{execute}(\textbf{file}, \textbf{arg}_1, \textbf{arg}_2, \ldots, \textbf{arg}_n)$$

which requests the system to read in and execute the program named by **file**, passing it string arguments $\textbf{arg}_1$, $\textbf{arg}_2$, ..., $\textbf{arg}_n$. All the code and data in the process invoking **execute** is replaced from the **file**, but open files, current directory, and inter-process relationships are unaltered. Only if the call fails, for example because **file** could not be found or because its execute-permission bit was not set, does a return take place from the **execute** primitive; it resembles a "jump" machine instruction rather than a subroutine call.

5.4 Process synchronization

Another process control system call:

$$\textbf{processid} = \textbf{wait}(\textbf{status})$$

causes its caller to suspend execution until one of its children has completed execution. Then **wait** returns the **processid** of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available.

5.5 Termination

Lastly:

$$\textbf{exit}(\textbf{status})$$

terminates a process, destroys its image, closes its open files, and generally obliterates it. The parent is notified through the **wait** primitive, and **status** is made available to it. Processes may also

terminate as a result of various illegal actions or user-generated signals (Section VII below).

VI. THE SHELL

For most users, communication with the system is carried on with the aid of a program called the shell. The shell is a command-line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. (The shell is described fully elsewhere,[9] so this section will discuss only the theory of its operation.) In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

$$\texttt{command } \texttt{arg}_1 \ \texttt{arg}_2 \ \ldots \ \texttt{arg}_n$$

The shell splits up the command name and the arguments into separate strings. Then a file with name **command** is sought; **command** may be a path name including the "/" character to specify any file in the system. If **command** is found, it is brought into memory and executed. The arguments collected by the shell are accessible to the command. When the command is finished, the shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file **command** cannot be found, the shell generally prefixes a string such as **/bin/** to **command** and attempts again to find the file. Directory **/bin** contains commands intended to be generally used. (The sequence of directories to be searched may be changed by user request.)

6.1 Standard I/O

The discussion of I/O in Section III above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the shell, however, start off with three open files with file descriptors 0, 1, and 2. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user's terminal. Thus programs that wish to write informative information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs that wish to read messages typed by the user read this file.

The shell is able to change the standard assignments of these file descriptors from the user's terminal printer and keyboard. If one of the arguments to a command is prefixed by ">", file descriptor 1 will, for the duration of the command, refer to the file named after the ">". For example:

```
ls
```

ordinarily lists, on the typewriter, the names of the files in the current directory. The command:

```
ls >there
```

creates a file called **there** and places the listing there. Thus the argument **>there** means "place output on **there**." On the other hand:

```
ed
```

ordinarily enters the editor, which takes requests from the user via his keyboard. The command

```
ed <script
```

interprets **script** as a file of editor commands; thus **<script** means "take input from **script**."

Although the file name following "<" or ">" appears to be an argument to the command, in fact it is interpreted completely by the shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

File descriptor 2 is, like file 1, ordinarily associated with the terminal output stream. When an output-diversion request with ">" is specified, file 2 remains attached to the terminal, so that commands may produce diagnostic messages that do not silently end up in the output file.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line:

```
ls | pr −2 | opr
```

ls lists the names of the files in the current directory; its output is passed to pr, which paginates its input with dated headings. (The argument "−2" requests double-column output.) Likewise, the output from pr is input to opr; this command spools its input onto a file for off-line printing.

This procedure could have been carried out more clumsily by:

```
ls >temp1
pr −2 <temp1 >temp2
opr <temp2
```

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the ls command to accept user requests to paginate its output, to print in multi-column format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as ls to provide such a wide variety of output options.

A program such as pr which copies its standard input to its standard output (with processing) is called a *filter*. Some filters that we have found useful perform character transliteration, selection of lines according to a pattern, sorting of the input, and encryption and decryption.

6.3 Command separators; multitasking

Another feature provided by the shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons:

```
ls; ed
```

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by "&," the shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example:

```
as source >output &
```

causes source to be assembled, with diagnostic output going to output; no matter how long the assembly takes, the shell returns

immediately. When the shell does not wait for the completion of a command, the identification number of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The "&" may be used several times in a line:

```
as source >output & ls >files &
```

does both the assembly and the listing in the background. In these examples, an output file other than the terminal was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The shell also allows parentheses in the above operations. For example:

```
(date; ls) >x &
```

writes the current date and time followed by a list of the current directory onto the file x. The shell also returns immediately for another request.

6.4 The shell as a command; command files

The shell is itself a command, and may be called recursively. Suppose file **tryout** contains the lines:

```
as source
mv a.out testprog
testprog
```

The **mv** command causes the file **a.out** to be renamed **testprog**. **a.out** is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the keyboard, **source** would be assembled, the resulting program renamed **testprog**, and **testprog** executed. When the lines are in **tryout**, the command:

```
sh <tryout
```

would cause the shell **sh** to execute the commands sequentially.

The shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It also provides general conditional and looping constructions.

6.5 Implementation of the shell

The outline of the operation of the shell can now be understood.

Most of the time, the shell is waiting for the user to type a command. When the newline character ending the line is typed, the shell's **read** call returns. The shell analyzes the command line, putting the arguments in a form appropriate for **execute**. Then **fork** is called. The child process, whose code of course is still that of the shell, attempts to perform an **execute** with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the **fork**, which is the parent process, **wait**s for the child process to die. When this happens, the shell knows the command is finished, so it types its prompt and reads the keyboard to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line contains "**&**," the shell merely refrains from waiting for the process that it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the **fork** primitive, it inherits not only the memory image of its parent but also all the files currently open in its parent, including those with file descriptors 0, 1, and 2. The shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children—the command programs—inherit them automatically. When an argument with "<" or ">" is given, however, the offspring process, just before it performs **execute**, makes the standard I/O file descriptor (0 or 1, respectively) refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is **open**ed (or **create**d); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after "<" or ">" and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the shell need not know the actual names of the files that are its own standard input and output, because it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the shell never terminates. (The main loop includes the branch of the return from **fork** belonging to the parent process; that is, the branch that does a **wait**, then reads another command line.) The one thing that causes the shell to terminate is discovering an end-of-file condition on its

input file. Thus, when the shell is executed as a command with a given input file, as in:

```
sh <comfile
```

the commands in **comfile** will be executed until the end of **comfile** is reached; then the instance of the shell invoked by **sh** will terminate. Because this shell process is the child of another instance of the shell, the **wait** executed in the latter will return, and another command may then be processed.

6.6 Initialization

The instances of the shell to which users type commands are themselves children of another process. The last step in the initialization of the system is the creation of a single process and the invocation (via **execute**) of a program called **init**. The role of **init** is to create one process for each terminal channel. The various subinstances of **init** open the appropriate terminals for input and output on files 0, 1, and 2, waiting, if necessary, for carrier to be established on dial-up lines. Then a message is typed out requesting that the user log in. When the user types a name or other identification, the appropriate instance of **init** wakes up, receives the log-in line, and reads a password file. If the user's name is found, and if he is able to supply the correct password, **init** changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an **execute** of the shell. At this point, the shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of **init** (the parent of all the subinstances of itself that will later become shells) does a **wait**. If one of the child processes terminates, either because a shell found an end of file or because a user typed an incorrect name or password, this path of **init** simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another log-in message. Thus a user may log out simply by typing the end-of-file sequence to the shell.

6.7 Other programs as shell

The shell as described above is designed to allow users full access to the facilities of the system, because it will invoke the execution of any program with appropriate protection mode. Sometimes,

however, a different interface to the system is desirable, and this feature is easily arranged for.

Recall that after a user has successfully logged in by supplying a name and password, **init** ordinarily invokes the shell to interpret command lines. The user's entry in the password file may contain the name of a program to be invoked after log-in instead of the shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system might specify that the editor **ed** is to be used instead of the shell. Thus when users of the editing system log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g., chess, blackjack, 3D tic-tac-toe) available on the system illustrate a much more severely restricted environment. For each of these, an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the shell. People who log in as a player of one of these games find themselves limited to the game and unable to investigate the (presumably more interesting) offerings of the UNIX system as a whole.

VII. TRAPS

The PDP-11 hardware detects a number of program faults, such as references to non-existent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. Unless other arrangements have been made, an illegal action causes the system to terminate the process and to write its image on file **core** in the current directory. A debugger can be used to determine the state of the program at the time of the fault.

Programs that are looping, that produce unwanted output, or about which the user has second thoughts may be halted by the use of the **interrupt** signal, which is generated by typing the "delete" character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a **core** file. There is also a **quit** signal used to force an image file to be produced. Thus programs that loop unexpectedly may be halted and the remains inspected without prearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by a process. For example, the shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating-point hardware, unimplemented instructions are caught and floating-point instructions are interpreted.

VIII. PERSPECTIVE

Perhaps paradoxically, the success of the UNIX system is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This (essentially personal) effort was sufficiently successful to gain the interest of the other author and several colleagues, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, the system had proved useful enough to persuade management to invest in the PDP-11/45, and later in the PDP-11/70 and Interdata 8/32 machines, upon which it developed to its present form. Our goals throughout the effort, when articulated at all, have always been to build a comfortable relationship with the machine and to explore ideas and inventions in operating systems and other software. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations that influenced the design of UNIX are visible in retrospect.

First: because we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly designed interactive system is much more productive and satisfying to use than a "batch" system. Moreover, such a system is rather easily adaptable to noninteractive use, while the converse is not true.

Second: there have always been fairly severe size constraints on the system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint

has encouraged not only economy, but also a certain elegance of design. This may be a thinly disguised version of the "salvation through suffering" philosophy, but in our case it worked.

Third: nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Because all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large "access method" routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, less than a page long, that buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no "control blocks" with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load, with all programs, routines for dealing with each device, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process-control scheme and the command interface have proved both convenient and efficient. Because the shell operates as an ordinary, swappable user program, it consumes no "wired-down" space in the system proper, and it may be made as powerful as desired at little cost. In particular, given the framework in which the shell executes as a process that spawns other processes to perform commands, the notions of I/O redirection, background

processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The **fork** operation, essentially as we implemented it, was present in the GENIE time-sharing system.[10] On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls[11] and both the name of the shell and its general functions. The notion that the shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX.[12]

IX. STATISTICS

The following numbers are presented to suggest the scale of the Research UNIX operation. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important "applications" programs.

Overall, we have today:

125	user population
33	maximum simultaneous users
1,630	directories
28,300	files
301,700	512-byte secondary storage blocks used

There is a "background" process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant *e*, and other semi-infinite problems. Not counting this background work, we average daily:

13,500	commands
9.6	CPU hours
230	connect hours

X. ACKNOWLEDGMENTS

The contributors to UNIX are, in the traditional but here especially apposite phrase, too numerous to mention. Certainly, collective salutes are due to our colleagues in the Computing Science Research Center. R. H. Canaday contributed much to the basic design of the file system. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M. D. McIlroy, and J. F. Ossanna.

REFERENCES

1. L. P. Deutch and B. W. Lampson, "An online editor," Commun. Assn. Comp. Mach., *10* (December 1967), pp. 793-799, 803.
2. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," Commun. Assn. Comp. Mach., *18* (March 1975), pp. 151-157.
3. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," B.S.T.J., this issue, pp. 2115-2135.
4. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 164-168.
5. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," B.S.T.J., this issue, pp. 2177-2200.
6. H. Lycklama, "UNIX Time-Sharing System: UNIX on a Microprocessor," B.S.T.J., this issue, pp. 2087-2101.
7. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, 1978.
8. Aleph-null, "Computer Recreations," Software Practice and Experience, *1* (April-June 1971), pp. 201-204.
9. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," B.S.T.J., this issue, pp. 1971-1990.
10. L. P. Deutch and B. W. Lampson, *Doc. 30.10.10, Project GENIE*,. April 1965.
11. R. J. Feiertag and E. I. Organick, "The Multics input-output system," Proc. Third Symposium on Operating Systems Principles (October 18-20, 1971), pp. 35-41.
12. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," Commun. Assn. Comp. Mach., *15* (March 1972), pp. 135-143.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

UNIX Implementation

By K. THOMPSON
(Manuscript received December 5, 1977)

This paper describes in high-level terms the implementation of the resident UNIX kernel. This discussion is broken into three parts. The first part describes how the UNIX system views processes, users, and programs. The second part describes the I/O system. The last part describes the UNIX file system.*

I. INTRODUCTION

The UNIX kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression "the UNIX operating system." The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a

* UNIX is a trademark of Bell Laboratories.

great responsibility and a great power. It is a soap-box platform on "the way things should be done." Even so, if "the way" is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

II. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.

2.1 Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the

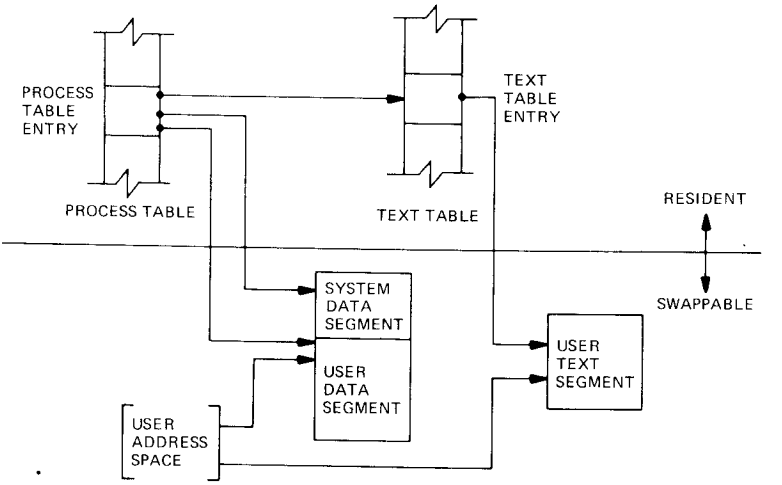

Fig. 1—Process control data structure.

current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **exec**s the second program. This is analogous to a "goto." If a program wishes to regain control after **exec**ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1]

2.2 Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary

memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

2.3 Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By

convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,[2] in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of "how much" can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.[3]

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-

of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

III. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been "structured I/O" and "unstructured I/O," respectively; while the term "block I/O" has some meaning, "character I/O" is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to

handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in most cases is created automatically by a program that reads the system's parts list.

3.1 Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as "dirty." The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On non-random devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic

tapes is "cured" by allowing only one outstanding write request per drive.

3.2 Character I/O system

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the "classical" character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means "everything other than block." I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

3.2.1 Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the presort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2 Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a

queue will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

3.2.3 Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines

and fast line printers. These devices either have their own buffers or "borrow" block I/O buffers for a while and then give them back.

IV. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further discussion of the external view of files and directories, see Ref. 4.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases. Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks

of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (*viz.*, 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

4.1 File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the

last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are **open**, **create**, **read**, **write**, **seek**, and **close**. The data structures maintained are shown in Fig. 2. In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of **forks**) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A

using the kernel as a tool. A good example of this is the command language.[5] Each user may have his own command language. Maintenance of such code is as easy as maintaining user code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.[6]

REFERENCES

1. R. E. Griswold and D. R. Hanson, "An Overview of SL5," SIGPLAN Notices, *12* (April 1977), pp. 40-50.
2. E. W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages,* ed. F. Genuys, New York: Academic Press (1968), pp. 43-112.
3. J. A. Hawley and W. B. Meyer, "MUNIX, A Multiprocessing Version of UNIX," M.S. Thesis, Naval Postgraduate School, Monterey, Cal. (1975).
4. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
5. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," B.S.T.J., this issue, pp. 1971-1990.
6. E. I. Organick, *The MULTICS System,* Cambridge, Mass.: M.I.T. Press, 1972.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

A Retrospective†

By D. M. RITCHIE
(Manuscript received January 6, 1978)

UNIX is a general-purpose, interactive, time-sharing operating system for the DEC PDP-11 and Interdata 8/32 computers. Since it became operational in 1971, it has become quite widely used. This paper discusses the strong and weak points of the UNIX system and some areas where we have expended no effort. The following areas are touched on:*

(i) The structure of files: a uniform, randomly-addressable sequence of bytes. The irrelevance of the notion of "record." The efficiency of the addressing of files.
(ii) The structure of file system devices: directories and files.
(iii) I/O devices integrated into the file system.
(iv) The user interface: fundamentals of the shell, I/O redirection, and pipes.
(v) The environment of processes: system calls, signals, and the address space.
(vi) Reliability: crashes, losses of files.
(vii) Security: protection of data from corruption and inspection; protection of the system from stoppages.
(viii) Use of a high-level language—the benefits and the costs.
(ix) What UNIX does not do: "real-time," interprocess communication, asynchronous I/O.
(x) Recommendations to system designers.

† A version of this paper was presented at the Tenth Hawaii International Conference on the System Sciences, Honolulu, January, 1977.
* UNIX is a trademark of Bell Laboratories.

UNIX* is a general-purpose, interactive time-sharing operating system primarily for the DEC PDP-11 series of computers, and recently for the Interdata 8/32. Since its development in 1971, it has become quite widely used, although publicity efforts on its behalf have been minimal, and the license under which it is made available outside the Bell System explicitly excludes maintenance. Currently, there are more than 300 Bell System installations, and an even larger number in universities, secondary schools, and commercial and government institutions. It is useful on a rather broad range of configurations, ranging from a large PDP-11/70 supporting 48 users to a single-user LSI-11 system.

I. SOME GENERAL OBSERVATIONS

In most ways, UNIX is a very conservative system. Only a handful of its ideas are genuinely new. In fact, a good case can be made that it is in essence a modern implementation of M.I.T.'s CTSS system.[1] This claim is intended as a compliment to both UNIX and CTSS. Today, more than fifteen years after CTSS was born, few of the interactive systems we know of are superior to it in ease of use; many are inferior in basic design.

UNIX was never a "project"; it was not designed to meet any specific need except that felt by its major author, Ken Thompson, and soon after its origin by the author of this paper, for a pleasant environment in which to write and use programs. Although it is rather difficult, after the fact, to try to account for its success, the following reasons seem most important.

(*i*) It is simple enough to be comprehended, yet powerful enough to do most of the things its users want.

(*ii*) The user interface is clean and relatively surprise-free. It is also terse to the point of being cryptic.

(*iii*) It runs on a machine that has become very popular in its own right.

(*iv*) Besides the operating system and its basic utilities, a good deal of interesting software is available, including a sophisticated text-processing system that handles complicated mathematical material[2] and produces output on a typesetter or a typewriter terminal, and a LALR parser-generator.[3]

* UNIX is a trademark of Bell Laboratories.

This paper discusses the strong and weak points of the system and lists some areas where no effort has been expended. Only enough design details are given to motivate the discussion; more can be found elsewhere in this issue.[4,5]

One problem in discussing the capabilities and deficiencies of UNIX is that there is no unique version of the system. It has evolved continuously both in time, as new functions are added and old problems repaired, and in space, as various organizations add features intended to meet their own needs. Four important versions of the system are in current use:

(*i*) The standard system maintained by the UNIX Support Group at Bell Laboratories for Bell System projects.

(*ii*) The "Programmer's Workbench" version,[6,7] also in wide use within Bell Laboratories, especially in areas in which text-processing and job-entry to other machines are important. Recently, PWB/UNIX has become available to outside organizations as well.

(*iii*) The "Sixth Edition" system (so called from the manual that describes it), which is the most widely used under Western Electric licenses by organizations outside the Bell System.

(*iv*) The version currently used in the Computing Science Research Center, where the UNIX system was developed, and at a few other locations at Bell Laboratories.

The proliferation of versions makes some parts of this paper hard to write, especially where details (e.g., how large can a file be?) are mentioned. Although compilation of a list of differences between versions of UNIX is a useful exercise, this is not the place for such a list, so the paper will concentrate on the properties of the system as it exists for the author, in the current research version of the system.

The existence of several variants of UNIX is, of course, a problem not only when attempting to describe the system in a paper such as this, but also to the users and administrators. The importance of this problem is not lost upon the proprietors of the various versions; indeed, vigorous effort is under way to combine the best features of the variants into a single system.

II. THE STRUCTURE OF FILES

The UNIX file system is simple in structure; nevertheless, it is more powerful and general than those often found even in

considerably larger operating systems. Every file is regarded as a featureless, randomly addressable sequence of bytes. The system conceals physical properties of the device on which the file is stored, such as the size of a disk track. The size of a file is the number of bytes it contains; the last byte is determined by the high-water mark of writes to the file. It is not necessary, nor even possible, to preallocate space for a file. The system calls to read and write each come in only one form, which specifies the local name of an open file, a buffer to or from which to perform I/O, and a byte count. I/O is normally sequential, so the first byte referred to by a read or write operation immediately follows the final byte transferred by the preceding operation. "Random access" is accomplished using a **seek** system call, which moves the system's internal read (or write) pointer for the instance of the open file to another byte that the next read or write will implicitly address. All I/O appears completely synchronous; read-ahead and write-behind are performed invisibly by the system.

This particularly simple way of viewing files was suggested by the Multics I/O system.[8]

The addressing mechanism for files must be carefully designed if it is to be efficient. Files can be large (about 10^9 bytes), are grown without pre-allocation, and are randomly accessible. The overhead per file must be small, because there can be many files (the machine on which this paper was written has about 27,000 on the disk storing most user's files); many of them are small (80 percent have ten or fewer 512-byte blocks, and 37 percent are only one block long). The details of the file-addressing mechanism are given elsewhere.[5]

No careful study has been made of the efficiency of disk I/O, but a simple experiment suggests that the efficiency is comparable to two other systems, DEC's IAS for the PDP-11, and Honeywell's GCOS TSS system running on the H6070. The experiment consisted of timing a program that copied a file that, on the PDP-11, contained 480 blocks (245,760 bytes). The file on the Honeywell had the same number of bytes (each of nine bits rather than eight), but there were 1280 bytes per block. With otherwise idle machines, the real times to accomplish the file copies were

system	*sec.*	*msec./block*
UNIX	21	21.8
IAS	19	19.8
H6070	9	23.4

The effective transfer rates on the PDP-11s are essentially identical, and the Honeywell rate is not far off when measured in blocks per second. No general statistical significance can be ascribed to this little experiment. Seek time, for example, dominates the measured times (because the disks on the PDP-11 transfer one block of data in only 0.6 millisecond once positioned), and there was no attempt to optimize the placement of the input or output files. The results do seem to suggest, however, that the very flexible scheme for representing UNIX files carries no great cost compared with at least two other systems.

The real time per block of I/O observed under the UNIX system in this test was about 22 milliseconds. Because the system overhead per block is 6 milliseconds, most of which is overlapped, it would seem that the overall transfer rate of the copy might be nearly doubled if a block size of 1024 bytes were used instead of 512. There are some good arguments against making such a change. For example, space utilization on the disk would suffer noticeably: doubling the block size would increase the space occupied by files on the author's machine by about 15 percent, a number whose importance becomes apparent when we observe that the free space is currently only 5 percent of the total available. Increasing the block size would also force a decrease in the size of the system's buffer cache and lower its hit rate, but this effect has not been reliably estimated.

Moreover, the copy program is an extreme case in that it is totally I/O bound, with no processing of the data. Most programs do at least look at the data as it goes by; thus to sum the bytes in the file mentioned above required 10 seconds of real time, 5 of which were "user time" spent looking at the bytes. To read the file and ignore it completely required 9 seconds, with negligible user time. It may be concluded that the read-ahead strategy is almost perfectly effective, and that a program that spends as little as 50 microseconds per byte processing its data will not be significantly delayed waiting for I/O (unless, of course, it is competing with other processes for use of the disk).

The basic system interface conceals physical aspects of file storage, such as blocks, tracks, and cylinders. Likewise, the concept of a record is completely absent from the operating system proper and nearly so from the standard software. (By the term "record" we mean an identifiable unit of information consisting either of a fixed number of bytes or of a count together with that number of bytes.)

A text file, for example, is stored as a sequence of characters with new-line characters to delimit lines. This form of storage is not only efficient in space when compared with fixed-length records, or even records described by character counts, but is also the most convenient form of storage for the vast majority of text-processing programs, which almost invariably deal with character streams. Most important of all, however, is the fact that there is only one representation of text files. One of the most valuable characteristics of UNIX is the degree to which separate programs interact in useful ways; this interaction would be seriously impaired if there were a variety of representations of the same information.

We recall with a certain horrified fascination a system whose Fortran compiler demanded as input a file with "variable-length" records each of which was required to be 80 bytes long. The prevalence of this sort of nonsense makes the following test of software flexibility (due to M. D. McIlroy) interesting to try when meeting new systems. It consists of writing a Fortran (or PL/I, or other language) program that copies itself to another file, then running the program, and finally attempting to compile the resulting output. Most systems eventually pass, but often only after an expert has been called in to mutter incantations that convert the data file generated by the Fortran program to the format expected by the Fortran compiler. In sum, we would consider it a grave imposition to require our users or ourselves, when mentioning a file, to specify the form in which it is stored.

For the reasons discussed above, UNIX software does not use the traditional notion of "record" in relation to files, particularly those containing textual information. But certainly there are applications in which the notion has use. A program or self-contained set of programs that generates intermediate files is entitled to use any form of data representation it considers useful. A program that maintains a large data base in which it must frequently look up entries may very well find it convenient to store the entries sequentially, in fixed-size units, sorted by index number. With some changes in the requirements or usual access style, other file organizations become more appropriate. It is straightforward to implement any number of schemes within the UNIX file system precisely because of the uniform, structureless nature of the underlying files; the standard software, however, does not include mechanisms to do it. As an example of what is possible, INGRES[9] is a relational data base manager running under UNIX that supports five different file organizations.

III. THE STRUCTURE OF THE FILE SYSTEM

On each file system device such as a disk, the accessing information for files is arranged in an array starting at a known place. A file may thus be identified by its device and its index within the device. The internal name of a file is, however, never needed by users or their programs. There is a hierarchically arranged directory structure in which each directory contains a list of names (character strings) and the associated file index, which refers implicitly to the same device as does the directory. Because directories are themselves files, the naming structure is potentially an arbitrary directed graph. Administrative rules restrict it to have the form of a tree, except that nondirectory files may have several names (entries in various directories).

A file is named by a sequence of directories separated by "/" leading towards a leaf of the tree. The path specified by a name starting with "/" originates at the root; without an initial "/" the path starts at the current directory. Thus the simple name **x** indicates the entry **x** in the current directory; **/usr/dmr/x** searches the root for directory **usr**, searches it for directory **dmr**, and finally specifies **x** in **dmr**.

When the system is initialized, only one file system device is known (the *root device*); its name is built into the system. More storage is attached by mounting other devices, each of which contains its own directory structure. When a device is mounted, its root is attached to a leaf of the already accessible hierarchy. For example, suppose a device containing a subhierarchy is mounted on the file **/usr**. From then on, the original contents of **/usr** are hidden from view, and in names of the form **/usr/...** the ... specifies a path starting at the root of the newly mounted device.

This file system design is inexpensive to implement, is general enough to satisfy most demands, and has a number of virtues: for example, device self-consistency checks are straightforward. It does have a few peculiarities. For example, instantaneously enforced space quotas, either for users or for directories, are relatively difficult to implement (it has been done at one university site). Perhaps more serious, duplicate names for the same file (**links**) while trivial to provide on a single device, do not work across devices; that is, a directory entry cannot point to a file on another device. Another limitation of the design is that an arbitrary subset of members of a given directory cannot be stored on another device. It is common for the totality of user files to be too voluminous for a

given device. It is then impossible for the directories of all users to be members of the same directory, say /usr. Instead they must be split into groups, say /usr1 and /usr2; this is somewhat inconvenient, especially when space on one device runs out so that some users must be moved. The data movement can be done expeditiously, but the change in file names from /usr1/... to /usr2/... is annoying both to those people who must learn the new name and to programs that happen to have such names built into them.

Earlier variants of this file system design stored disk block addresses as 16-bit quantities, which limited the size of a file-system volume to 65,536 blocks. This did not mean that the rest of a larger physical device was wasted, because there could be several logical devices per drive, but the limitation did aggravate the difficulty just mentioned. Recent versions of the system can handle devices with up to about 16 million blocks.

IV. INPUT/OUTPUT DEVICES

The UNIX system goes to some pains to efface differences between ordinary disk files and I/O devices such as terminals, tape drives, and line printers. An entry appears in the file system hierarchy for each supported device, so that the structure of device names is the same as that of file names. The same read and write system calls apply to devices and to disk files. Moreover, the same protection mechanisms apply to devices as to files.

Besides the traditionally available devices, names exist for disk devices regarded as physical units outside the file system, and for absolutely addressed memory. The most important device in practice is the user's terminal. Because the terminal channel is treated in the same way as any file (for example, the same I/O calls apply), it is easy to redirect the input and output of commands from the terminal to another file, as explained in the next section. It is also easy to provide inter-user communication.

Some differences are inevitable. For example, the system ordinarily treats terminal input in units of lines, because character-erase and line-delete processing cannot be completed until a full line is typed. Thus if a program attempts to read some large number of bytes from a terminal, it waits until a full line is typed, and then receives a notification that some smaller number of bytes has actually been read. All programs must be prepared for this eventuality in any case, because a read operation from any disk file will return fewer bytes than requested when the end of the file is encountered.

Ordinarily, therefore, reads from the terminal are fully compatible with reads from a disk file. A subtle problem can occur if a program reads several bytes, and on the basis of a line of text found therein calls another program to process the remainder of the input. Such a program works successfully when the input source is a terminal, because the input is returned a line at a time, but when the source is an ordinary file the first program may have consumed input intended for the second. At the moment the simplest solution is for the first program to read one character at a time. A more general solution, not implemented, would allow a mode of reading wherein at most one line at a time was returned, no matter what the input source.*

V. THE USER INTERFACE

The command interpreter, called the "shell," is the most important communication channel between the system and its users. The shell is not part of the operating system, and enjoys no special privileges. A part of the entry for each user in the password file read by the login procedure contains the name of the program that is to be run initially, and for most users that program is the shell. This arrangement is by now commonplace in well-designed systems, but is by no means universal. Among its advantages are the ability to swap the shell even though the kernel is not swappable, so that the size of the shell is not of great concern. It is also easy to replace the shell with another program, either to test a new version or to provide a non-standard interface.

The full language accepted by the shell is moderately complicated, because it performs a number of functions; it is discussed in more detail elsewhere in this issue.[10] Nevertheless, the treatment of individual commands is quite simple and regular: a command is a sequence of words separated by white space (spaces and tabs). The first word is the name of the command, where a command is any executable file. A full name, with "/" characters, may be used to specify the file unambiguously; otherwise, an agreed-upon sequence of directories is searched. The only distinction enjoyed by a system-provided command is that it appears in a directory in the search path of most users. (A very few commands are built into the shell.) The other words making up a command line fall into three types:

*This suggestion may seem in conflict with our earlier disdain of "records." Not really, because it would only affect the way in which information is read, not the way it is stored. The same bytes would be obtained in either case.

buffering in each pipe. Finally, although an acceptable (if complicated) notation has been proposed that creates only deadlock-free graphs, the need has never been felt keenly enough to impel anyone to implement it.

Other aspects of UNIX, not closely tied to any particular program, are also valuable in providing a pleasant user interface. One thing that seems trivial, yet makes a surprising difference once one is used to it, is full-duplex terminal I/O together with read-ahead. Even though programs generally communicate with the user in terms of lines, rather than single characters, full-duplex terminal I/O means that the user can type at any time, even if the system is typing back, without fear of losing or garbling characters. With read-ahead, one need not wait for a response to every line. A good typist entering a document becomes incredibly frustrated at having to pause before starting each new line; for anyone who knows what he wants to say any slowness in response becomes psychologically magnified if the information must be entered bit by bit instead of at full speed.

Both input and output of UNIX programs tend to be very terse. This can be disconcerting, especially to the beginner. The editor, for example, has essentially only one diagnostic, namely "?", which means "you have done something wrong." Once one knows the editor, the error or difficulty is usually obvious, and the terseness is appreciated after a period of acclimation, but certainly people can be confused at first. However, even if some fuller diagnostics might be appreciated on occasion, there is much noise that we are happy to be rid of. The command interpreter does not remark loudly that each program finished normally, or announce how much space or time it took; the former fact is whispered by an unobtrusive prompt, and anyone who wishes to know the latter may ask explicitly.

Likewise, commands seldom prompt for missing arguments; instead, if the argument is not optional, they give at most a one-line summary of their usage and terminate. We know of some systems that seem so proud of their ability to interact that they force interaction on the user whether it is wanted or not. Prompting for missing arguments is an issue of taste that can be discussed in calm tones; insistence on asking questions may cause raised voices.

Although the terseness of typical UNIX programs is, to some extent, a matter of taste, it is also connected with the way programs tend to be combined. A simple example should make the situation clear. The command **who** writes out one line for each user logged into the system, giving a name, a terminal name, and the time of login. The command **wc** (for "word count") writes out the number

of lines, the number of words, and the number of characters in its input. Thus

```
who | wc
```

tells in the line-count field how many users are logged in. If **who** produced extraneous verbiage, the count would be off. Worse, if **wc** insisted on determining from its input whether lines, words, or characters were wanted, it could not be used in this pipeline. Certainly, not every command that generates a table should omit headings; nevertheless, we have good reasons to interpret the phrase "extraneous verbiage" rather liberally.

VI. THE ENVIRONMENT OF A PROCESS

The virtual address space of a process is divided into three regions: a read-only, shared-program text region; a writable data area that may grow at one end by explicit request; and a stack that grows automatically as information is pushed onto it by subroutine calls. The address space contains no "control blocks."

New processes are created by the **fork** operation, which creates a child process whose code and data are copied from the parent. The child inherits the open files of the parent, and executes asynchronously with it unless the parent explicitly waits for termination of the child. The **fork** mechanism is essential to the basic operation of the system, because each command executed by the shell runs in its own process. This scheme makes a number of services extremely easy to provide. I/O redirection, in particular, is a basically simple operation; it is performed entirely in the subprocess that executes the command, and thus no memory in the parent command interpreter is required to rescind the change in standard input and output. Background processes likewise require no new mechanism; the shell merely refrains from waiting for the completion of a command specified to be asynchronous. Finally, recursive use of the shell to interpret a sequence of commands stored in a file is in no way a special operation.

Communication by processes with the outside world is restricted to a few paths. Explicit system calls, mostly to do I/O, are the most common. A new program receives a set of character-string arguments from its invoker, and returns a byte of status information when it terminates. It may be sent "signals," which ordinarily force

the software were written in a rather open environment, so the continuous, careful effort required to maintain a fully secure system has not always been expended; as a result, there are several security problems.

The weakest area is in protecting against crashing, or at least crippling, the operation of the system. Most versions lack checks for overconsumption of certain resources, such as file space, total number of files, and number of processes (which are limited on a per-user basis in more recent versions). Running out of these things does not cause a crash, but will make the system unusable for a period. When resource exhaustion occurs, it is generally evident what happened and who was responsible, so malicious actions are detectable, but the real problem is the accidental program bug.

The theoretical aspects of the situation are brighter in the area of information protection. Each file is marked with its owner and the "group" of users to which the owner belongs. Files also have a set of nine protection bits divided into three sets of three bits specifying permission to read, to write, or execute as a program. The three sets indicate the permissions applicable to the owner of the file, to members of the owner's group, and to all others.

For directories, the meanings of the access bits are modified: "read" means the ability to read the directory as a file, that is, to discover all the names it contains; "execute" means the ability to search a directory for a given name when it appears as part of a qualified name; "write" means the ability to create and delete files in that directory, and is unrelated to writing of files in the directory.

This classification is not fine enough to account for the needs of all installations, but is usually adequate. In fact, most installations do not use groups at all (all users are in the same group), and even those that do would be happy to have more possible user IDs and fewer group-IDs. (Older versions of the system had only 256 of each; the current system has 65536, however, which should be enough.)

One particular user (the "super-user") is able to access all files without regard to permissions. This user is also the only one permitted to exercise privileged system entries. It is recognized that the very existence of the notion of a super-user is a theoretical, and often practical, blemish on any protection scheme.

An unusual feature of the protection system is the "set-user-ID" bit. When this bit is on for a file, and the file is executed as a program, the user number used in file permission checking is not that of the person running the program, but that of the owner of the file.

In practice, the bit is used to mark the programs that perform the privileged system functions mentioned above (such as creation of directories, changing the owner of a file, and so forth).

In theory, the protection scheme is adequate to maintain security, but, in practice, breakdowns can easily occur. Most often these come from incorrect protection modes on files. Our software tends to create files that are accessible, even writable, by everyone. This is not an accident, but a reflection of the open environment in which we operate. Nevertheless, people in more hostile situations must adjust modes frequently; it is easy to forget, and in any case there are brief periods when the modes are wrong. It would be better if software created files in a default mode specifiable by each user. The system administrators must be even more careful than the users to apply proper protection. For example, it is easy to write a user program that interprets the contents of a physical disk drive as a file system volume. Unless the special file referring to the disk is protected, the files on it can be accessed in spite of their protection modes. If a set-user-ID file is writable, another user can copy his own program onto it.

It is also possible to take advantage of bugs in privileged set-user-ID programs. For example, the program that sends mail to other users might be given the ability to send to directories that are otherwise protected. If so, this program must be carefully written in order to avoid being fooled into mailing other people's private files to its invoker.

There are thus a number of practical difficulties in maintaining a fully secure system. Nevertheless, the operating system itself seems capable of maintaining data security. The word "seems" must be used because the system has not been formally verified, yet no security-relevant bugs are known (except the ability to run it out of resources, which was mentioned above). In some ways, in fact, UNIX is inherently safer than many other systems. For example, I/O is always done on open files, which are named by an object local to a process. Permissions are checked when the file is opened. The I/O calls themselves have as argument only the (local) name of the open file, and the specification of the user's buffer; physical I/O occurs to a system buffer, and the data are copied in or out of the user's address space by a single piece of code in the system. Thus, there is no need for complicated, bug-prone verification of device commands and channel programs supplied by the user. Likewise, the absence of user "data control blocks" or other control blocks from the user's address space means that the interface between user processes and

arranged file system. It is very useful for maintaining directories containing related files, it is efficient because the amount of searching for files is bounded, and it is easy to implement.

(*ii*) The notion of "record" seems to be an obsolete remnant of the days of the 80-column card. A file should consist of a sequence of bytes.

(*iii*) The greatest care should be taken to ensure that there is only one format for files. This is essential for making programs work smoothly together.

(*iv*) Systems should be written in a high-level language that encourages portability. Manufacturers who build more than one line of machines and also build more than one operating system and set of utilities are wasting money.

XII. ACKNOWLEDGMENT

Much, even most, of the design and implementation of UNIX is the work of Ken Thompson. My use of the term "we" in this paper is intended to include him; I hope his views have not been misrepresented.

REFERENCES

1. P. A. Crisman, Ed., *The Compatible Time-Sharing System,* Cambridge, Mass.: M.I.T. Press, 1965.
2. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," B.S.T.J., this issue, pp. 2115-2135.
3. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories (July 1975).
4. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
5. K. Thompson, "UNIX Time-Sharing System: UNIX Implementation," B.S.T.J., this issue, pp. 1931-1946.
6. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 164-168.
7. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," B.S.T.J., this issue, pp. 2177-2200.
8. R. J. Feiertag and E. I. Organick, "The Multics input-output system," Proc. Third Symposium on Operating Systems Principles (October 18-20, 1971), pp. 35-41.
9. M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES," ACM Trans. on Database Systems, *1* (September 1976), pp. 189-222.
10. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," B.S.T.J., this issue, pp. 1971-1990.
11. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Englewood Cliffs, N.J.: Prentice-Hall, 1978.

12. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," B.S.T.J., this issue, pp. 1991-2019.
13. S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of C Programs and the UNIX System," B.S.T.J., this issue, pp. 2021-2048.
14. H. Lycklama and D. L. Bayer, "UNIX Time-Sharing System: The MERT Operating System," B.S.T.J., this issue, pp. 2049-2086.
15. G. L. Chesson, "The Network UNIX System," Operating Systems Review, 9 (1975), pp. 60-66. Also in Proc. 5th Symp. on Operating Systems Principles.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

The UNIX Shell

By S. R. BOURNE
(Manuscript received January 30, 1978)

The UNIX shell is a command programming language that provides an interface to the UNIX operating system. It contains several mechanisms found in algorithmic languages such as control-flow primitives, variables, and parameter passing. Constructs such as* **while**, **if**, **for**, *and* **case** *are available. Two-way communication is possible between the shell and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands and may be used to determine the flow of control, and the standard output from a command may be used as input to the shell. The shell can modify the environment in which commands run. Input and output can be redirected and processes that communicate through "pipes" can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user.*

I. INTRODUCTION

The UNIX shell† is both a programming language and a command language. As a programming language, it contains control-flow primitives and string-valued variables. As a command language, it provides a user interface to the process-related facilities of the UNIX operating system. The design of the shell is based in part on the

* UNIX is a trademark of Bell Laboratories.
† This term (shell) seems to have first appeared in the MULTICS system (Ref. 1). It is, however, not universal; other terms include *command interpreter, command language.*

original UNIX shell[2] and the PWB/UNIX shell,[3,4] some features having been taken from both. Similarities also exist with the command interpreters of the Cambridge Multiple Access System[5] and of CTSS.[6] The language described here differs from its predecessors in that the control-flow notations are more powerful and are understood by the shell itself. However, the notation for simple commands and for parameter passing and substitution is similar in all these languages.

The shell executes commands that are read either from a terminal or from a file. The design of the shell must therefore take into account both interactive and noninteractive use. Except in some minor respects, the behavior of the shell is independent of its input source.

II. NOTATION

Simple commands are written as sequences of "words" separated by blanks. The first word is the name of the command to be executed. Any remaining words are passed as arguments to the invoked command. For example, the command

```
ls -l
```

prints a list of the file names in the current directory. The argument -l tells ls to print the date of last use, the size, and status information for each file.

Commands are similar to procedure calls in languages such as Algol 68 or PL/I. The notation is different in two respects. First, although the arguments are arbitrary strings, in most cases they need not be enclosed in quotes. Second, there are no parentheses enclosing the list of arguments nor commas separating them. Command languages tend not to have the extensive expression syntax found in algorithmic languages. Their primary purpose is to issue commands; it is therefore important that the notation be free from superfluous characters.

To execute a command, the shell normally creates a new process and waits for it to finish. Both these operations are primitives available in the UNIX operating system. A command may be run without waiting for it to finish using the postfix operator &. For example,

```
print file &
```

calls the print command with argument file and runs it in the

background. The & is a metacharacter interpreted by the shell and is not passed as an argument to **print**.

Associated with each process, UNIX maintains a set of file descriptors numbered 0,1,... that are used in all input-output transactions between processes and the operating system. File descriptor 0 is termed the standard input and file descriptor 1 the standard output. Most commands produce their output on the standard output that is initially (following **login**) connected to a terminal. This output may be redirected for the duration of a command, as in

```
ls -l >file
```

The notation **>file** is interpreted by the shell and is not passed as an argument to **ls**. If the file does not exist, the shell creates it; otherwise, the contents of the file are replaced with the output from the command. To append to a file, the notation

```
ls -l >>file
```

is provided. Similarly, the standard input may be taken from a file by writing, for example,

```
wc <file
```

wc prints the number of characters, words, and lines on the standard input.

The standard output of one command may be connected to the standard input of another by writing the "pipe" operator, indicated by **|**, as in

```
ls -l | wc
```

Two commands connected in this way constitute a "pipeline," and the overall effect is the same as

```
ls -l >file
wc <file
```

except that no file is used. Instead, the two processes are connected by a pipe that is created by an operating system call. Pipes are unidirectional; synchronization is achieved by halting **wc** when there is nothing to read and halting **ls** when the pipe is full. This matter is dealt with by UNIX, not the shell.

A *filter* is a command that reads its input, transforms it in some way, and prints the result as output. One such filter, **grep**, selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those file names from the current directory that contain the string **old**.

A pipeline may consist of more than two commands, the input of each being connected to the output of its predecessor. For example,

```
ls | grep old | wc
```

When a command finishes execution it returns an *exit status* (return code). Conventionally, a zero exit status means that the command succeeded; nonzero means failure. This Boolean value may be tested using the **if** and **while** constructs provided by the shell.

The general form of the conditional branch is

```
if command-list
then command-list
else command-list
fi
```

The **else** part is optional. A *command-list* is a sequence of commands separated by semicolons or newlines and is evaluated from left to right. The value tested by **if** is that of the last simple-command in the *command-list* following **if**. Since this construction is bracketed by **if** and **fi**, it may be used unambiguously in any position that a simple command may be used. This is true of all the control-flow constructions in the shell. Furthermore, in the case of **if** there is no dangling **else** ambiguity. Apart from considerations of language design, this is important for interactive use. An Algol 60 style **if then else**, where the **else** part is optional, requires look-ahead to see whether the **else** part is present. In this case, the shell would be unable to determine that the **if** construct was ended until the next command was read.

The McCarthy "andf" and "orf" operators are also provided for testing the success of a command and are written **&&** and **||** respectively.

```
command1 && command2          (1)
```

executes **command**$_2$ only if **command**$_1$ succeeds. It is equivalent to

```
if command1
then command2
fi
```

Conversely,

$$\textbf{command}_1 \;||\; \textbf{command}_2 \qquad (2)$$

executes **command**$_2$ only if **command**$_1$ fails. The value returned by these constructions is the value of the last **command** executed. Thus (1) returns **true** iff both **command**$_1$ and **command**$_2$ succeed, whereas (2) returns **true** iff either **command**$_1$ or **command**$_2$ succeeds.

The **while** loop has a form similar to **if**.

while *command-list*$_1$
do *command-list*$_2$
done

command-list$_1$ is executed and its value tested each time around the loop. This provides a notation for a break in the middle of a loop, as in

```
while a; b
do c
done
```

First **a** is executed, then **b**. If **b** returns **false**, then the loop exits; otherwise, **c** is executed and the loop resumes at **a**. Although this deals with many loop breaks, **break** and **continue** are also available. Both take an optional integer argument specifying how many levels of loop to break from or at which level to continue, the default being one.

if and **while** test the value returned by a command. The **case** and **for** constructs provide for data-driven branching and looping. The **case** construct is a multi-way branch that has the general form

case *word* **in**
    pattern) *command-list* ;;
    ...
esac

The shell attempts to match *word* with each *pattern*, in the order in which the patterns appear. If a match is found, the associated *command-list* is executed and execution of the **case** is complete. Patterns are specified using the following metacharacters.

* Matches any string including the null string.

? Matches any single character.

[...] Matches any of the enclosed characters. A pair of characters separated by a minus matches any character lexically between the pair.

For example, ***.c** will match any string ending with **.c**. Alternatives are separated by **|**, as in

```
case ... in
        x|y) ...
```

which, for single characters, is equivalent to

```
case ... in
        [xy]) ...
```

There is no special notation for the default case, since it may be written as

```
case ... in
        ...
        *) ...
esac
```

Since it is difficult to determine the equivalence of patterns, no check is made to ensure that only one pattern matches the **case** word. This could lead to obscure bugs, although in practice it appears not to present a problem.

The **for** loop has the general form

for *name* **in** $word_1$ $word_2$ ...
do *command-list*
done

and executes the *command-list* once for each *word* following **in**. Each time around the loop the shell variable (q.v.) *name* is set to the next *word*.

III. SHELL PROCEDURES

The shell may be used to read and execute commands contained in a file. For example,

sh file $\mathbf{arg}_1$ $\mathbf{arg}_2$ ...

calls the shell to read commands from **file**. Such a file is called a "shell procedure." Arguments supplied with the call are referred to within the shell procedure using the positional parameters **$1**,

$2, ... For example, if the file **wg** contains

```
who | grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

UNIX files have three independent attributes, *read, write,* and *execute.* If the file **wg** is executable, then

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably.

A frequent use of shell procedures is to loop through the arguments (**$1, $2,...**) executing commands once for each argument. An example of such a procedure is **tel** that searches the file **/usr/lib/telnos** containing lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of **tel** is

```
for i
do grep $i </usr/lib/telnos; done
```

The default **in** list for a **for** loop is the positional parameters. The command

```
tel fred bert
```

prints those lines in **/usr/lib/telnos** that contain the string **fred** followed by those lines that contain **bert.**

Shell procedures can be used to tailor the command environment to the taste and needs of an individual or group. Since procedures are text files requiring no compilation, they are easy to create and maintain. Debugging is also assisted by the ability to try out parts of a procedure at a terminal. To further assist debugging, the shell

provides two tracing mechanisms. If a procedure is invoked with the −v flag, as in

```
sh −v proc
```

then the shell will print the lines of **proc** as they are read. This is useful when checking procedures for syntactic errors, particularly in conjunction with the −n flag which suppresses command execution. An execution trace is specified by the −x flag and causes each command to be printed as it is executed. The −x flag is more useful than −v when errors in the flow of control are suspected.

During the execution of a shell procedure, the standard input and output are left unchanged. (In earlier versions of the UNIX shell the text of the procedure itself was the standard input.) Thus shell procedures can be used naturally as filters. However, commands sometimes require in-line data to be available to them. A special input redirection notation "<<" is used to achieve this effect. For example, the UNIX editor takes its commands from the standard input. At a terminal,

```
ed file
```

will call the editor and then read editing requests from the terminal. Within a shell procedure this would be written

```
ed file <<!
editing requests
!
```

The lines between <<! and ! are called a *here* document; they are read by the shell and made available as the standard input. The string ! is arbitrary, the document being terminated by a line that consists of the string following <<. There are a number of advantages to making *here* documents explicitly visible. First, the number of lines read from the shell procedure is under the control of the procedure writer, enabling a procedure to be understood without having to know what commands such as **ed** do. Further, since the shell is the first to see such input, parameter substitution can, optionally, be applied to the text of the document.

IV. SHELL VARIABLES

The shell provides string-valued variables that may be used both within shell programs and, interactively, as abbreviations for

frequently used strings. Variable names begin with a letter and consist of letters, digits, and underscores.

Shell variables may be given values when a shell procedure is invoked. An argument to a shell procedure of the form *name=value* causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking shell is not affected. Such *names* are sometimes called keyword parameters.

Keyword parameters may also be exported from a procedure by saying, for example,

```
export user box
```

Modification of such variables within the called procedure does not affect the values in the calling procedure. (It is generally true of a UNIX process that it may not modify the environment of its caller without explicit request on the part of that caller. Files and shared file descriptors are the exceptions to this rule.)

A name whose value is intended to remain constant throughout a procedure may be declared **readonly**. The form of this command is the same as that of the **export** command,

```
readonly name ...
```

Subsequent attempts to set **readonly** variables are illegal.

Within a shell procedure, shell variables are set by writing, for example,

```
user=fred
```

The value of a variable may be substituted by preceding its name with $; for example,

```
echo $user
```

will echo **fred**. (**echo** is a standard UNIX command that prints its arguments, separated by blanks.) The general notation for parameter (or variable) substitution is

```
${name}
```

and is used, for example, when the parameter name is followed by a letter or digit. If a shell parameter is not set, then the null string is substituted for it. Alternatively, a default string may be given, as in

```
echo ${d-.}
```

which will echo the value of **d** if it is set and "." otherwise. Substitutions may be nested, so that, for example,

```
echo ${d-$1}
```

will echo the value of **d** if it is set and the value (if any) of **$1** otherwise. A variable may be assigned a default value using the notation

```
${d=.}
```

which substitutes the same string as

```
${d-.}
```

except that, if **d** were not previously set, then it will be set to the string ".". (The notation **${...=...}** is not available for positional parameters.)

In cases when a parameter is required to be set, the notation

```
${d?message}
```

will substitute the value of the variable **d** if it has one, otherwise *message* is printed by the shell and execution of the shell procedure is abandoned. If *message* is absent then a standard message is printed. A shell procedure that requires some parameters to be set might start as follows.

```
: ${user?} ${acct?} ${bin?}
...
```

A colon (:) is a command built in to the shell that does nothing once its arguments have been evaluated. In this example, if any of the variables **user**, **acct** or **bin** are not set, then the shell will abandon execution of the procedure.

The following variables have a special meaning to the shell.

- **$?** The exit status (return code) of the last command executed as a decimal string.
- **$#** The number of positional parameters as a decimal string.
- **$$** The UNIX process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is typically used to generate unique temporary file names (UNIX has no genuine temporary files).
- **$!** The process number of the last process initiated in the background.
- **$-** The current shell flags.

The following variables are used, but not set, by the shell.

Typically, these variables are set in a *profile* which is executed when a user logs on to UNIX.

$MAIL When used interactively, the shell looks at the file specified by this variable before it issues a prompt. If this file has been modified since it was last examined, the shell prints the message **you have mail** and then prompts for the next command.

$HOME The default argument (*home* directory) for the **cd** command. The current directory is used to resolve file name references that do not begin with a /, and is changed using the **cd** command.

$PATH A list of directories that contain commands (the *search path*). Each time a command is executed by the shell, a list of directories is searched for an executable file. If **$PATH** is not set, then the current directory, **/bin**, and **/usr/bin** are searched by default. Otherwise **$PATH** consists of directory names separated by :. For example,

```
PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first :), **/usr/fred/bin**, **/bin** and **/usr/bin**, are to be searched, in that order. In this way, individual users can have their own "private" commands accessible independently of the current directory. If the command name contains a /, then this directory search mechanism is not used; a single attempt is made to find the command.

V. COMMAND SUBSTITUTION

The standard output from a command enclosed in grave accents (`...`) can be substituted in a similar way to parameters. For example, the command **pwd** prints on its standard output the name of the current directory. If the current directory is **/usr/fred/bin** then

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents is the command to be

executed and is replaced with the output from that command. This mechanism allows string-processing commands to be used within shell procedures. The shell itself does not provide any built-in string processing other than concatenation and pattern matching. Command substitution occurs in all contexts where parameter substitution occurs and the treatment of the resulting text is the same in both cases.

VI. FILE NAME GENERATION

The shell provides a mechanism for generating a list of file names that match a pattern. The specification of patterns is the same as that used by the **case** construct. For example,

```
ls −l *.c
```

generates, as arguments to **ls**, all file names in the current directory that end in **.c**.

```
[a−z]*
```

matches all names in the current directory beginning with one of the letters **a** through **z**.

```
/usr/srb/test/?
```

matches all file names in the directory **/usr/srb/test** consisting of a single character. If no file name is found that matches the pattern, then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/srb/*/core
```

finds and prints the names of all **core** files in subdirectories of **/usr/srb**. This last feature can be expensive, requiring a scan of all subdirectories of **/usr/srb**.

There is one exception to the general rules given for patterns. The character "." at the start of a file name must be explicitly matched.

```
echo *
```

will therefore echo all file names in the current directory not beginning with ".".

```
echo .*
```

will echo all those file names that begin with ".". This avoids inadvertent matching of the names "." and ".." which, conventionally, mean "the current directory" and "the parent directory" respectively.

VII. EVALUATION AND QUOTING

The shell is a macro processor that provides parameter substitution, command substitution, and file name generation for the arguments to commands. This section discusses the order in which substitutions occur and the effects of the various quoting mechanisms.

Commands are initially parsed according to the grammar given in Appendix A. Before a command is executed, the following evaluations occur.

Parameter substitution, e.g., **$user**.
Command substitution, e.g., **`pwd`**.

The shell does not rescan substituted strings. For example, if the value of the variable X is the string *$x*, then

```
echo $X
```

will echo *$x*.

After these substitutions have occurred, the resulting characters are broken into words (*blank interpretation*); the null string is not regarded as a word unless it is quoted. For example,

```
echo ''
```

will pass on the null string as the first argument to **echo**, whereas

```
echo $null
```

will call **echo** with no arguments if the variable **null** is not set or set to the null string.

Each word is then scanned for the file pattern characters *, ?, and [...], and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

Metacharacters such as < > * ? | (Appendix B has a complete list) have a special meaning to the shell. Any character preceded by a \ is *quoted* and loses its special meaning, if any. The \ is elided so that

```
echo \?\\
```

will echo ?\. To allow long strings to be continued over more than one line, the sequence **newline** is ignored.

\ is convenient for quoting single characters. When more than one character needs quoting, the above mechanism is clumsy and error-prone. A string of characters may be quoted by enclosing (part of) the string between single quotes, as in

```
echo '*'
```

The quoted string may *not* contain a single quote.

A third quoting mechanism using double quotes prevents interpretation of some but not all metacharacters. Within double quotes, parameter and command substitution occurs, but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using \.

$	parameter substitution
`	command substitution
"	ends the quoted string
\	quotes the special characters $ ` " \

For example,

```
echo "$x"
```

will pass the value of the variable x to **echo**, whereas

```
echo '$x'
```

will pass the string *$x* to **echo**.

In cases where more than one evaluation of a string is required, the built-in command **eval** may be used. **eval** reads its arguments (which have therefore been evaluated once) and executes the resulting command(s). For example, if the variable X has the value *$x*, and if x has the value *pqr* then

```
eval echo $X
```

will echo the string *pqr*.

VIII. ERROR AND FAULT HANDLING

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a

terminal. Execution of a command may fail for any of the following reasons.

(*i*) Input output redirection may fail, for example, if a file does not exist or cannot be created. In this case, the command is not executed.
(*ii*) The command itself does not exist or is not executable.
(*iii*) The command runs and terminates abnormally, for example, with a "memory fault."
(*iv*) The command terminates normally but returns a nonzero exit status.

In all of these cases, the shell will go on to execute the next command. Except for the last case, an error message will be printed by the shell.

All remaining errors cause the shell to exit from a command procedure. An interactive shell will return to read another command from the terminal. Such errors include the following.

(*i*) Syntax errors; e.g., **if ... then ... done**.
(*ii*) A signal such as terminal interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
(*iii*) Failure of any of the built-in commands such as **cd**.

The shell flag **−e** causes the shell to terminate if any error is detected.

Shell procedures normally terminate when an interrupt is received from the terminal. Such an interrupt is communicated to a UNIX process as a signal. If some cleaning-up is required, such as removing temporary files, the built-in command **trap** is used. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for terminal interrupt (signal 2) and, if this interrupt is received, will execute the commands

```
rm /tmp/ps$$; exit
```

exit is another built-in command that terminates execution of a shell procedure. The **exit** is required in this example; otherwise, after the trap has been taken, the shell would resume executing the procedure at the place where it was interrupted.

UNIX signals can be handled by a process in one of three ways. They can be ignored, in which case the signal is never sent to the

process; they can be caught, in which case the process must decide what to do; lastly, they can be left to cause termination of the process without it having to take any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking the procedure in the background, then **trap** commands (and the signal) are ignored.

A shell procedure may, itself, elect to ignore signals by specifying the null string as the argument to **trap**. A trap may be reset by saying, for example,

```
trap 2
```

which resets the trap for signal 2 to its default value (which is to exit).

The following procedure **scan** is an example of the use of **trap** without an exit in the trap command. **scan** takes each directory in the current directory, prompts with its name, and then executes the command typed at the terminal. Interrupts are ignored while executing the requested commands but cause termination when **scan** is waiting for input.

```
d=`pwd`
for i in *
do    if test -d $d/$i
      then  cd $d/$i
            while echo "$i:"
                  trap exit 2
                  read x
            do trap : 2; eval $x; done
      fi
done
```

The command

```
read x
```

is built in to the shell and reads the next line from the standard input and assigns it to the variable **x**. The command

```
test -d arg
```

returns **true** if **arg** is a directory and **false** otherwise.

IX. COMMAND EXECUTION

To execute a command, the shell first creates a new process using

the system call **fork**. The execution environment for the command includes input, output, and the states of signals, and is established in the created process before the command is executed. The built-in command **exec** is used in the rare cases when no **fork** is required.

The environment for a command run in the background, such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file **/dev/null**. This prevents two processes (the shell and the command), that are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case.

```
ed file &
```

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the **quit** and **interrupt** signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate.

X. ACKNOWLEDGMENTS

I would like to thank Dennis Ritchie and John Mashey for many discussions during the design of the shell. I am also grateful to the members of the Computing Science Research Center for their comments on drafts of this document.

APPENDIX A

Grammar

item:	*word*
	input-output
simple-command:	*item*
	simple-command item
command:	*simple-command*
	(*command-list*)
	{ *command-list* }

	for *name* **do** *command-list* **done**
	for *name* **in** *word* ... **do** *command-list* **done**
	while *command-list* **do** *command-list* **done**
	until *command-list* **do** *command-list* **done**
	case *word* **in** *case-part* ... **esac**
	if *command-list* **then** *command-list else-part* **fi**
pipeline:	*command*
	pipeline **|** *command*
andor:	*pipeline*
	andor **&&** *pipeline*
	andor **||** *pipeline*
command-list:	*andor*
	command-list ;
	command-list **&**
	command-list ; *andor*
	command-list **&** *andor*
input-output:	> *word*
	>> *word*
	< *word*
	<< *word*
case-part:	*pattern*) *command-list* ;;
pattern:	*word*
	pattern **|** *word*
else-part:	**elif** *command-list* **then** *command-list else-part*
	else *command-list*
	empty
empty:	
word:	a sequence of non-blank characters
name:	a sequence of letters, digits or underscores starting with a letter
digit:	0 1 2 3 4 5 6 7 8 9

APPENDIX B

Metacharacters and Reserved Words

(*i*) Syntactic

|	pipe symbol
&&	"andf" symbol
||	"orf" symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping
<	input redirection
<<	input from a *here* document
>	output creation
>>	output append

(*ii*) Patterns

*	matches any character(s) including none
?	matches any single character
[...]	matches any of the enclosed characters

(*iii*) Substitution

${...}	substitution of shell variables
`...`	substitution of command output

(*iv*) Quoting

\	quotes the next character
'...'	quotes the enclosed characters except for '
"..."	quotes the enclosed characters except for $ ` \ "

(*v*) Reserved words

```
if then else elif fi
case in esac
for while until do done
```

REFERENCES

1. E. I. Organick, *The MULTICS System,* Cambridge, Mass.: M.I.T. Press, 1972.
2. K. Thompson, "The UNIX Command Language," in *Structured Programming—Infotech State of the Art Report,* Nicholson House, Maidenhead, Berkshire, England: Infotech International Ltd. (March 1975), pp. 375-384.
3. J. R. Mashey, "Using a Command Language as a High-Level Programming Language," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 169-176.
4. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 164-168.
5. D. F. Hartley (Ed.), *The Cambridge Multiple Access System — Users Reference Manual,* Cambridge, England: University Mathematical Laboratory, 1968.
6. P. A. Crisman, Ed., *The Compatible Time-Sharing System,* Cambridge, Mass.: M.I.T. Press, 1965.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

The C Programming Language

By D. M. RITCHIE, S. C. JOHNSON, M. E. LESK,
and B. W. KERNIGHAN
(Manuscript received December 5, 1977)

C is a general-purpose programming language that has proven useful for a wide variety of applications. It is the primary language of the UNIX system, and is also available in several other environments. This paper provides an overview of the syntax and semantics of C and a discussion of its strengths and weaknesses.*

C is a general-purpose programming language featuring economy of expression, modern control flow and data structure capabilities, and a rich set of operators and data types.

C is not a "very high-level" language nor a big one and is not specialized to any particular area of application. Its generality and an absence of restrictions make it more convenient and effective for many tasks than supposedly more powerful languages. C has been used for a wide variety of programs, including the UNIX operating system, the C compiler itself, and essentially all UNIX applications software. The language is sufficiently expressive and efficient to have completely displaced assembly language programming on UNIX.

C was originally written for the PDP-11 under UNIX, but the language is not tied to any particular hardware or operating system. C compilers run on a wide variety of machines, including the Honeywell 6000, the IBM System/370, and the Interdata 8/32.

* UNIX is a trademark of Bell Laboratories.

I. THE LINGUISTIC HISTORY OF C

The C language in use today[1] is the product of several years of evolution. Many of its most important ideas stem from the considerably older, but still quite vital, language BCPL[2] developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B,[3] which was written by Ken Thompson in 1970 for the first UNIX system on the PDP-11.

Although neither B nor C could really be considered dialects of BCPL, both share several characteristic features with it:

(*i*) All are able to express the fundamental flow-control constructions required for well-structured programs: statement grouping, decision-making (**if**), looping (**while**) with the termination test either at the top or the bottom of the loop, and branching out to a sequence of possible cases (**switch**). It is interesting that BCPL provided these constructions in 1967, well before the current vogue for "structured programming."

(*ii*) All three languages include the concept of "pointer" and provide the ability to do address arithmetic.

(*iii*) In all three languages, the arguments to functions are passed by copying the value of the argument, and it is impossible for the function to change the actual argument. When it is desired to achieve "call by reference," a pointer may be passed explicitly, and the function may change the object to which the pointer points. Any function is allowed to be recursive, and its local variables are typically "automatic" or specific to each invocation.

(*iv*) All three languages are rather low-level, in that they deal with the same sorts of objects that most computers do. BCPL and B restrict their attention almost completely to machine words, while C widens its horizons somewhat to characters and (possibly multi-word) integers and floating-point numbers. None deals directly with composite objects such as character strings, sets, lists, or arrays considered as a whole. The languages themselves do not define any storage allocation facility beside static definition and the stack discipline provided by the local variables of functions; likewise, I/O is not part of any of these languages. All these higher mechanisms must be provided by explicitly called routines from libraries.

B and BCPL differ mainly in their syntax, and many differences stemmed from the very small size of the first B compiler (fewer

than 4K 18-bit words on the PDP-7). Several constructions in BCPL encourage a compiler to maintain a representation of the entire program in memory. In BCPL, for example,

```
valof $(
        . . .
        resultis expression
        . . .
$)
```

is syntactically an expression. It provides a way of packaging a block of many statements into a sort of unnamed internal procedure yielding a single result (delivered by the **resultis** statement). The **valof** construction can occur in the middle of any expression, and can be arbitrarily large. The B language avoided the difficulties caused by this and some other constructions by rigorously simplifying (and in some cases adjusting to personal taste) the syntax of BCPL.

In spite of many syntactic changes, B remained very close to BCPL semantically. The most characteristic feature of both languages is their nearly identical treatment of addresses (pointers). They support a model of the storage of the machine consisting of a sequence of equal-sized cells, into which values can be placed; in typical implementations, these cells will be machine words. Each identifier in a program corresponds to a cell, and a cell may contain a variety of values. Most often the value is an integer, or perhaps a representation of a character. All the cells, however, are numbered; the address of a cell is just the integer giving its ordinal position. BCPL has a unary operator **lv** (in some versions, and also in B and C, shortened to **&**) that, when applied to a name, yields the address of the cell corresponding to the name. The inverse operator **rv** (later *) yields the value in the cell pointed to its argument. Thus the statement

```
px = &x;
```

of B assigns to **px** the number that can be interpreted as the address of **x**; the statements

```
y = *px + 2;
*px = 5;
```

first use the value in the cell pointed to by **px** (which is the same cell as **x**) and then assign 5 to this cell.

Arrays in BCPL and B are intimately tied up with pointers. An array declaration, which might in BCPL be written

```
let Array = vec 10
```

and in B

```
auto Array[10];
```

creates a single cell named Array and initializes it with the address of the first of a sequence of 10 unnamed cells containing the array itself. Since the quantity stored in Array is just the address of the cell of the first element of the array, the expression

```
Array + i
```

is the address of the ith element, counting from zero. Likewise, applying the indirection operator,

```
* (Array + i)
```

refers to the value of the ith member of the array. This operation is so frequent that special syntax was invented to express it:

```
Array[i]
```

Thus, despite its asymmetric appearance, subscripting is a commutative operation; the above example could equally well be written

```
i[Array]
```

In BCPL and B there is only one type of object, the machine word, so when the same language operator is applied to two operands, the calculation actually carried out must always be the same. Thus, for example, if one wishes to provide the ability to do floating-point arithmetic, the "+" operator notation cannot be used, since it implies an integer addition. Instead (in a version of BCPL for the GE 635), a "." was placed in front of each operator that had floating-point operands. As may be appreciated, this was a frequent source of errors.

The machine model implied by the definitions of BCPL and B is simple and self-consistent. It is, however, inadequate for many purposes, and on many machines it causes inefficiencies when implemented. The problems became evident to us after B began to be used heavily on the first PDP-11 version of UNIX. The first followed from the fact that the PDP-11, like a number of machines (including, for example, the IBM System/370), is byte addressed; a machine address refers to any of several bytes (characters) in a word, not the word alone. Most obviously, the word orientation of B cut us off from any convenient ability to access individual bytes. Equally

important was the fact that before any address could be used, it had to be shifted left by one place. The reason for this is simple: there are two bytes per PDP-11 word. On the one hand, the language guaranteed that if 1 was added to an address quantity, it would point to the next word; on the other, the machine architecture required that word addresses be even and equal to the byte number of the first byte in the word. Since, finally, there was no way to distinguish cells containing ordinary integers from those containing pointers, the only solution visible was to represent pointers as word numbers and then, at the point of use, convert to the byte representation by multiplication by 2.

Yet another problem was introduced by the desire to provide for floating-point arithmetic. The PDP-11 supports two floating-point formats, one of which requires two words, the other four. In neither case was it satisfactory to use the trick used on the GE 635 (operators like ".+") because there was no way to represent the requirement for a single data item occupying four or eight bytes. This problem did not arise on the 635 because integers and single-precision floating-point both require only one word.

Thus the problems evidenced by B led us to design a new language that (after a brief period under the name NB) was dubbed C. The major advance provided by C is its typing structure, which completely solved the difficulties mentioned above. Each declaration in a C program specifies (sometimes implicitly) a *type*, which determines how much storage the object requires and how it is to be interpreted. The original fundamental types provided were single character (byte), integer, single-precision floating-point, and double-precision floating-point. (Others discussed below were added later.) Thus in the program

```
double a, b;
  . . .
a = b + 3;
```

the compiler is able to determine from the declarations of a and b the fact that they require four words of storage each, that the "+" means a double-precision floating add, and that "3" must be converted to floating.

Of course, the idea of typing variables is in no way original with C; in fact, it was the general rule among the most widely used and influential languages, including Algol, Fortran, and PL/I. Nevertheless, the introduction of types marked an important change in our own thinking. The typeless nature of BCPL and B had seemed to

promise a great simplification in the implementation, understanding, and use of these languages. By the time that C was created (circa 1972), advocates of languages like Algol 68 and Pascal recommended a strongly enforced type structure on psychological grounds; but even disregarding their arguments, the typeless nature of BCPL and B seemed inappropriate, for purely technological reasons, to the available hardware.

II. THE TYPE STRUCTURE OF C

The introduction of types in C, although a major departure from the tradition of BCPL and B, was done in such a way that many of the characteristic usages of the earlier languages survived. To some extent, this continuity was an attempt to preserve as much as possible of the considerable corpus of existing software written in B, but even more important, especially in retrospect, was the desire to minimize the intellectual distance between the past and the future ways of expression.

2.1 Pointers, arrays and address arithmetic

One clear example of the similarity of C to the earlier languages is its treatment of pointers and arrays. In C an array of 10 integers might be declared

```
int Array[10];
```

which is identical to the corresponding declaration in B. (Arrays begin at zero; the elements of Array are Array[0], ..., Array[9].) As discussed above, the B implementation caused a cell named Array to be allocated and initialized with a pointer to 10 otherwise unnamed cells to hold the array. In C, the effect is a bit different; 10 integers are allocated, and the first is associated with the name Array. But C also includes a general rule that, whenever the name of an array appears in an expression, it is converted to a pointer to the first member of the array. Strictly speaking, we should say, for this example, it is converted to an *integer pointer* since all C pointers are associated with a particular type to which they point. In most usages, the actual effects of the slightly different meanings of Array are indistinguishable. Thus in the C expression

```
Array + i
```

the identifier Array is converted to a pointer to the first element of

the array; i is scaled (if required) before it is added to the pointer. For a byte-addressed machine, the scale factor is the number of bytes in an integer; for a word-addressed machine the scale factor is unity. In any event, the result is a pointer to the ith member of the array. Likewise identical in effect to the interpretation of B,

```
* (Array + i)
```

is the ith member itself, and

```
Array[i]
```

is another notation for the same thing. In all these cases, of course, should **Array** be an array of, or pointer to, some objects other than integers, the scale factor is adjusted appropriately. The pointer arithmetic, as written, is independent of the type of object to which the pointer points and indeed of the internal representation of the pointer.

2.2 Derived types

As mentioned above, the basic types in C were originally **int**, which represents an integer in the basic size provided by the machine architecture; **char**, which represents a single byte; **float**, a single-precision floating-point number; and **double**, double-precision floating-point. Over the years, **long**, **short**, and **unsigned** integers have been added. In current C implementations, **long** is at least 32 bits; **short** is usually 16 bits; and **int** remains the "natural" size for the machine at hand. **Unsigned** integers exist mainly to squeeze an extra bit out of the machine, since the sign bit need not be represented.

In addition to these basic types, C provides a conceptually infinite hierarchy of derived types, which are formed by composition of the basic types with pointers, arrays, structures, unions, and functions. Examples of pointer and array declarations have already been exhibited; another is

```
double *vecp, vector[100];
```

which declares a pointer **vecp** to double-precision floating numbers, and an array **vector** of the same kind of objects. The size of an array, when specified, must always be a constant.

A *structure* is an aggregate of one or more objects, usually of various types, which can be treated as a unit. C structures are essentially the same as records in languages like Pascal, and semantically,

though not syntactically, like PL/I and Cobol structures. Thus,

```
struct tag {
        int     i;
        float   f;
        char    c[3];
};
```

defines a template, called **tag**, for a structure containing three *members*: an integer **i**, a floating point number **f**, and a three-character array **c**. The declaration

```
struct tag x, y[10], *p;
```

declares a structure **x** of this type, an array **y** of 10 such structures, and a pointer **p** to this kind of structure. The hierarchical nature of derived types is clearly evident here: **y** is an array of structures whose members include an array of characters. References to individual members of structures use the . operator:

```
x.i
x.f
y[i].c[0]
(*p).c[1]
```

Parentheses in the last line are necessary because the . binds more tightly than *. It turns out that pointers to structures are so common that special syntax is called for to express structure access through a pointer.

```
p->c[1]
p->i
```

This soon becomes more natural than the equivalent

```
(*p).c[1]
(*p).i
```

A **union** is capable of holding, at different times, objects of different types, with the compiler keeping track of size and alignment requirements. Unions provide a way to manipulate different kinds of data in a single part of storage, without embedding machine-dependent information (like the relative sizes of **int** and **float**) in a program. For example, the union **u**, declared

```
union {
        int     i;
        float   f;
} u;
```

can hold either an int (written u.i) or a float (written u.f). Regardless of the machine it is compiled on, it will be large enough to hold either one of these quantities. A union is syntactically identical to a structure; it may be considered as a structure in which all the members begin at the same offset. Unions in C are more analogous to PL/I's CELL than to the unions of Algol 68 or the variant records of Pascal, because it is the responsibility of the programmer to avoid referring to a union that does not currently contain an object of the implied type.

A *function* is a subprogram that returns an object of a given type:

```
unsigned unsf();
```

declares a function that returns unsigned. The type of a function ignores the number and types of its arguments, although in general the call and the definition must agree.

2.3 Type composition

The syntax of declarations borrows from that of expressions. The key idea is that a declaration, say

```
int     ... ;
```

contains a part "..." that, if it appeared in an expression, would be of type int. The constructions seen so far, for example,

```
int     *iptr;
int     ifunc();
int     iarr[10];
```

exhibit this approach, but more complicated declarations are common. For example,

```
int     *funcptr();
int     (*ptrfunc)();
```

declare respectively a function that returns a pointer to an integer, and a pointer to a function that returns an integer. The extra parentheses in the second are needed to make the * apply directly to ptrfunc, since the implicit function-call operator () binds more

tightly than *. Functions are not variables, so arrays or structures of functions are not permitted. However, a pointer to a function, like ptrfunc, may be stored, copied, passed as an argument, returned by a function, and so on, just as any other pointer.

Arrays of pointers are frequently used instead of multidimensional arrays. The usage of a and b when declared

```
int a[10][10];
int *b[10];
```

may be similar, in that a[5][5] and b[5][5] are both legal references to a single int, but a is a true array: all 100 storage cells have been allocated, and the conventional rectangular subscript calculation is done. For b, however, the declaration has only allocated 10 pointers; each must be set to point to an array of integers. Assuming each does point to a 10-element array, then there will be 100 storage cells set aside, plus the 10 cells for the pointers. Thus the array of pointers uses slightly more space and may require an extra initialization step, but has two advantages: it trades an indirection for a subscript multiplication, and it permits the rows of the array to be of different lengths. (That is, each element of b need not point to a 10-element vector; some may point to 2 elements, some to 20). Particularly with strings whose length is not known in advance, an array of pointers is often used instead of a multidimensional array. Every C main program gets access to its invoking command line in this form, for example.

The idea of specifying types by appropriating some of the syntax of expressions seems to be original with C, and for the simpler cases, it works well. Occasionally some rather ornate types are needed, and the declaration may be a bit hard to interpret. For example, a pointer to an array of pointers to functions, each returning an int, would be written

```
int (*(*funnyarray)[])();
```

which is certainly opaque, although understandable enough if read from the inside out. In an expression, funnyarray might appear as

```
i = (*(*funnyarray)[j])(k);
```

The corresponding Algol 68 declaration is

```
ref [] ref proc int funnyarray
```

which reads from left to right in correspondence with the informal

description of the type if **ref** is taken to be the equivalent of C's "pointer to." The Algol may be clearer, but both are hard to grasp.

III. STATEMENTS AND CONTROL FLOW

Control flow in C differs from other languages primarily in details of syntax. As in PL/I, semicolons are used to terminate statements, not to separate them. Most statements are just expressions followed by a semicolon; since assignments are expressions, there is no need for a special assignment statement.

Statements are grouped with braces { and }, rather than with words like **begin-end** or **do-od**, because the more concise form seems much easier to read and is certainly easier to type. A sequence of statements enclosed in { } is syntactically a single statement.

The **if-else** statement has the form

```
if (expression)
        statement
else
        statement
```

The *expression* is evaluated; if it is "true" (that is, if *expression* has a non-zero value), the first statement is done. If it is "false" (*expression* is zero) and if there is an **else** part, the second *statement* is executed instead. The **else** part is optional; if it is omitted in a sequence of nested if's, the resulting ambiguity is resolved in the usual way by associating the **else** with the nearest previous **else**-less if.

The **switch** statement provides a multi-way branch depending on the value of an integer expression:

```
switch (expression) {
        case const:
                code
        case const:
                code
        ...
        default:
                code
}
```

The *expression* is evaluated and compared against the various **cases**, which are labeled with distinct integer constant values. If any case

matches, execution begins at that point. If no case matches but there is a **default** statement, execution begins there; otherwise, no part of the **switch** is executed.

The **cases** are just labels, and so control may flow through one case to the next. Although this permits multiple labels on cases, it also means that in general most cases must be terminated with an explicit exit from the **switch** (the **break** statement below).

The **switch** construction is part of C's legacy from BCPL; it is so useful and so easy to provide that the lack of a corresponding facility of acceptable generality in languages ranging from Fortran through Algol 68, and even to Pascal (which does not provide for a **default**), must be considered a real failure of imagination in language designers.

C provides three kinds of loops. The **while** is simply

```
while (expression)
        statement
```

The *expression* is evaluated; if it is true (non-zero), the *statement* is executed, and then the process repeats. When *expression* becomes false (zero), execution terminates.

The **do** statement is a test-at-the-bottom loop:

```
do
        statement
while (expression);
```

statement is performed once, then *expression* is evaluated. If it is true, the loop is repeated; otherwise it is terminated.

The **for** loop is reminiscent of similarly named loops in other languages, but rather more general. The **for** statement

```
for (expr1; expr2; expr3)
        statement
```

is equivalent to

```
expr1;
while (expr2) {
        statement
        expr3;
}
```

Grammatically, the three components of a **for** loop are expressions. Any of the three parts can be omitted, although the semicolons must remain. If *expr1* or *expr3* is left out, it is simply dropped from

the expansion. If the test, *expr2*, is not present, it is taken as permanently true, so

```
for (;;) {
    ...
}
```

is an "infinite" loop, to be broken by other means, for example by **break**, below.

The **for** statement keeps the loop control components together and visible at the top of the loop, as in the idiomatic

```
for (i = 0; i < N; i = i+1)
```

which processes the first N elements of an array, the analogue of the Fortran or PL/I DO loop. The **for** is more general, however. The test is re-evaluated on each pass through the loop, and there is no restriction on changing the variables involved in any of the expressions in the **for** statement. The controlling variable i retains its value regardless of how the loop terminates. And since the components of a **for** are arbitrary expressions, **for** loops are not restricted to arithmetic progressions. For example, the classic walk along a linked list is

```
for (p = top; p != NULL; p = p->next)
    ...
```

There are two statements for controlling loops. The **break** statement, as mentioned, causes an immediate exit from the immediately enclosing **while**, **for**, **do** or **switch**. The **continue** statement causes the next iteration of the immediately enclosing loop to begin. **break** and **continue** are asymmetric, since **continue** does not apply to **switch**.

Finally, C provides the oft-maligned **goto** statement. Empirically, **goto**'s are not much used, at least on our system. The operating system itself, for example, contains 98 in some 8300 lines. The PDP-11 C compiler, in 9660 lines, has 147. Essentially all of these implement some form of branch to the top or bottom of a loop, or to error recovery code.

IV. OPERATORS AND EXPRESSIONS

C has been characterized as having a relatively rich set of operators. Some of these are quite conventional. For example, the basic

binary arithmetic operators are +, −, * and /. To these, C adds the modulus operator %; m%n is the remainder when m is divided by n.

Besides the basic logical or bitwise operators & (bitwise AND), and | (bitwise OR), there are also the binary operators ^ (bitwise exclusive OR), >> (right shift), and << (left shift), and the unary operator ~ (ones complement). These operators apply to all integers; C provides no special bit-string type.

The relational operators are the usual >, >=, <, <=, == (equality test), and != (inequality test). They have the value 1 if the stated relation is true, 0 if not.

The unary pointer operators * (for indirection) and & (for taking the address) were described in Section I. When y is such as to make the expressions &*y or &*y legal, either is just equal to y. Note that & and * are used as both binary and unary operators (with different meanings).

The simplest assignment is written =, and is used conventionally: the value of the expression on the right is stored in the object whose address is on the left. In addition, most binary operators can be combined with assignment by writing

$$a \; op= \; b$$

which has the effect of

$$a = a \; op \; b$$

except that a is only evaluated once. For example,

$$x \; += \; 3$$

is the same as

$$x = x + 3$$

if x is just a variable, but

$$p[i+j+1] \; += \; 3$$

adds 3 to the element selected from the array p, calculating the subscript only once, and, more importantly, requiring it to be written out only once. Compound assignment operators also seem to correspond well to the way we think; "add 3 to x" is said, if not written, much more commonly than "assign x+3 to x."

Assignment expressions have a value, just like other expressions, and may be used in larger expressions. For example, the multiple assignment

```
i = j = k = 0;
```

is a byproduct of this fact, not a special case. Another very common instance is the nesting of an assignment in the condition part of an if or a loop, as in

```
while ((c = getchar()) != EOF) ...
```

which fetches a character with the function **getchar**, assigns it to **c**, then tests whether the result is an end of file marker. (Parentheses are needed because the precedence of the assignment = is lower than that of the relational !=.)

C provides two novel operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand; the decrement operator -- subtracts 1. Thus the statement

```
++i;
```

increments i. The unusual aspect is that ++ and -- may be used either as prefix operators (before the variable, as in ++i), or postfix (after the variable: i++). In both cases, the effect is to increment i. But the expression ++i increments i *before* using its value, while i++ increments i *after* its value has been used. If i is 5, then

```
x = i++;
```

sets x to 5, but

```
x = ++i;
```

sets x to 6. In both cases, i becomes 6.

For example,

```
stack[i++] = ... ;
```

pushes a value on a stack stored in an array **stack** indexed by i, while

```
... = stack[--i];
```

retrieves the value and pops the stack. Of course, when the quantity incremented or decremented is a pointer, appropriate scaling is done, just as if the "1" were added explicitly:

```
*stackp++ = ... ;
... = *--stackp;
```

are analogous to the previous example, this time using a stack pointer instead of an index.

Tests may be combined with the logical connectives && (AND), || (OR), and ! (truth value negation). The && and || operators guarantee left-to-right evaluation, with termination as soon as the truth value is known. For example, in the test

```
if (i <= N && array[i] > 0) ...
```

if i is greater than N, then array[i] (presumably at that point an out-of-bounds reference) will not be accessed. This predictable behavior is especially convenient, and much preferable to the explicitly random order of evaluation promised by most other languages. Most C programs rely heavily on the properties of && and ||.

Finally, the *conditional expression,* written with the ternary operator ? :, provides an analogue of if-else in expressions. In the expression

```
e1 ? e2 : e3
```

the expression *e1* is evaluated first. If it is non-zero (true), then the expression *e2* is evaluated, and that is the value of the conditional expression. Otherwise, *e3* is evaluated, and that is the value. Only one of *e2* and *e3* is evaluated. Thus to set z to the maximum of a and b,

```
z = (a > b) ? a : b;    /* z = max(a, b) */
```

We have already discussed how integers are scaled appropriately in pointer arithmetic. C does a number of other automatic conversions between data types, more freely than Pascal, for example, but without the wild abandon of PL/I. In all contexts, char variables and constants are promoted to int. This is particularly handy in code like

```
n = c - '0';
```

which assigns to n the integer value of the character stored in c, by subtracting the value of the character '0'. Generally, in fact, the basic types fall into only two classes, integral and floating-point; char variables, and the various lengths of int's, are taken to be representations of the same kind of thing. They occupy different amounts of storage but are essentially compatible. Boolean values as such do not exist; relational or truth-value expressions have value 1 if true, and 0 if false.

Variables of type int are converted to floating-point when

combined with **floats** or **doubles** and in fact all floating arithmetic is carried out in double precision, so **floats** are widened to **double** in expressions.

Conversions that involve "narrowing" an expression (for example, when a longer value is assigned to a shorter) are also well behaved. Floating point values are converted to integer by truncation; integers convert to shorter integers or characters by dropping high-order bits.

When a conversion is desired, but is not implicit in the context, it is possible to force a conversion by an explicit operator called a *cast.* The expression

(type) expression

is a new expression whose type is that specified in *type.* For example, the **sin** routine expects an argument of type **double**; in the statement

```
x = sin((double) n);
```

the value of **n** is converted to **double** before being passed to **sin**.

V. THE STRUCTURE OF C PROGRAMS

Complete programs consist of one or more files containing function and data declarations. Thus, syntactically, a program is made up of a sequence of declarations; executable code appears only inside functions. Conventionally, the run-time system arranges to call a function named **main** to start execution.

The language distinguishes the notions of *declaration* and *definition.* A declaration merely announces the properties of a variable (like its type); a definition declares a variable and also allocates storage for it or, in the case of a function, supplies the code.

5.1 Functions

The notion of *function* in C includes the subroutines and functions of Fortran and the procedures of most other languages. A function call is written

name(arglist)

where the parentheses are required even if the argument list is empty. All functions may be used recursively.

Arguments are passed by value, so the called function cannot in

any way affect the actual argument with which it was called. This permits the called program to use its formal arguments as conveniently initialized local variables. Call by value also eliminates the class of errors, familiar to Fortran programmers, in which a constant is passed to a subroutine that tries to alter the corresponding argument. An array name as an actual argument, however, is converted to a pointer to the first array element (as it always is), so the effect is as if arrays were called by reference; given the pointer, the called function can work its will on the individual elements of the array. When a function must return a value through its argument list, an explicit pointer may be passed, and the function references the ultimate target through this pointer. For example, the function swap(pa, pb) interchanges two integers pointed to by its arguments:

```
swap(px, py)        /* flip int's pointed to by px and py */
int *px, *py;
{
        int temp;

        temp = *px;
        *px = *py;
        *py = temp;
}
```

This also demonstrates the form of a function definition: the name is followed by an argument list; the arguments are declared, and the body of the function is a block, or compound statement, enclosed in braces. Declarations of local variables may follow the opening brace.

A function returns a value by

return *expression*;

The *expression* is automatically coerced to the type that the function returns. By default, functions are assumed to return int; if this is not the case, the function must be declared both in the calling routine and when it is defined. For example, a function definition is

```
double sqrt(x)      /* returns square root of x */
double x;
{
        ...
}
```

In the caller, the declaration is

```
double y, sqrt();

y = sqrt(y);
```

A function argument may be any of the basic types or a pointer, but not an array, structure, union, or function. The same is true of the value returned by a function. (The most recent versions of the language, still not standard everywhere, permit structures and unions as arguments and values of functions and allow them to be assigned.)

5.2 Data

Data declared at the top level (that is, outside the body of any function definition) are static in lifetime, and exist throughout the execution of the program. Variables declared within a function body are by default *automatic*: they come into existence when the function is entered and vanish when it is exited. Automatic variables may be declared to be **register** variables; when possible they will be placed in machine registers, which may result in smaller, faster code. The **register** declaration is only considered a hint to the compiler; no hardware register names are mentioned, and the hint may be ignored if the compiler wishes.

Static variables exist throughout the execution of a program, and retain their values across function calls. Static variables may be local to a function or (if defined at the top level) common to several functions.

External variables have the same lifetime as static, but they are also accessible to programs from other source files. That is, all references to an identically named external variable are references to the same thing.

The "storage class" of a variable can be explicitly announced in its declaration:

```
static int x;
extern double y[10];
```

More often the defaults for the context are sufficient. Inside a function, the default is **auto** (for automatic). Outside a function, at the top level, the default is **extern**. Since automatic and register variables are specific to a particular call of a particular function, they cannot be declared at the top level. Neither top-level variables nor

functions explicitly declared **static** are visible to functions outside the file in which they appear.

5.3 Scope

Declarations may appear either at the top level or at the head of a block (compound statement). Declarations in an inner block temporarily override those of identically named variables outside. The scope of a declaration persists until the end of its block, or until the end of the file, if it was at the top level.

Since function definitions may be given only at the top level (that is, they may not be nested), there are no internal procedures. They have been forbidden not for any philosophical reason, but only to simplify the implementation. It has turned out that the ability to make certain functions and data invisible to programs in other files (by explicitly declaring them **static**) is sufficient to provide one of their most important uses, namely hiding their names from other functions. (However, it is not possible for one function to access the internal variables of another, as internal procedures could do.) Similarly, the ability to conceal functions and data in one file from access by another satisfies some of the most crucial requirements of modular programming (as in languages like Alphard, CLU, and Euclid), even though it does not satisfy them all.

VI. C PREPROCESSOR

It is well recognized that "magic numbers" in a program are a sign of bad programming. Most languages, therefore, provide a way to define symbolic names for constants, so that the value of a magic number need be specified in only one place, and the rest of the code can refer to the value by some mnemonic name. In C such a mechanism is available, but it is not part of the syntax of the language; instead, symbolic naming is provided by a macro preprocessor automatically invoked as part of every C compilation. For example, given the definitions

```
#define    PI    3.14159
#define    E     2.71284
```

the preprocessor replaces all occurrences of a defined name by the corresponding defining string. (Upper-case names are normally chosen to emphasize that these are not variables.) Thus, when the programmer recognizes that he has written an incorrect value for *e*,

only the definition line has to be changed to

```
#define    E    2.71828
```

instead of each instance of the constant in the program.

Providing this service by a macro processor instead of by syntax has some significant advantages. The replacement text is not restricted to being numbers; any string of characters is permitted. Furthermore, the token being replaced need not be a variable, although it must have the form of a name. For example, one can define

```
#define    forever        for (;;)
```

and then write infinite loops as

```
forever {
    ...
}
```

The macro processor also permits macros to have arguments; this capability is heavily used by some I/O packages.

A second service of the C preprocessor is library file inclusion: a source line of the form

```
#include    "name"
```

causes the contents of the file **name** to be interpolated into the source at that point. (**includes** may be nested.) This feature is much used, especially in larger programs, for making sure that all the source files of the program are supplied with identical **#defines**, global data declarations, and the like.

VII. ENVIRONMENTAL CONSIDERATIONS

By intent, the C language confines itself to facilities that can be mapped relatively efficiently and directly into machine instructions. For example, writing matrix operations that look exactly like scalar operations is possible in some programming languages and occasionally misleads programmers into believing that matrix operations are as cheap as scalar operations. More important, restricting the domain of the C compiler to those areas where it knows how to do a relatively effective job provides the freedom to design subroutine libraries for the remaining tasks without constraining them to fit into some language specification. When the compiler cannot implement some facility without heavy costs in nonportability, complexity, or

efficiency, there are many benefits to leaving out such a facility: it simplifies the language and the compiler, frequently without inconveniencing the user (who often rejects a high-cost built-in operation and does it himself anyway).

At present, C is restricted to simple operations on simple data types. As a result, although the C area of operation is comparatively clean and pleasant, the user must know something about the polluting effects of the environment to get most jobs done. A program can always access the raw system calls on each system if very close interaction with the operating system is needed, but standard library routines have been implemented in each C environment that try to encourage portability while retaining speed and flexibility. The basic areas covered by the standard library at present are storage allocation, string handling, and I/O. Additional libraries and utilities are available for such areas as graphics, coroutine sequencing, execution time monitoring, and parsing.

The only automatic storage management service provided by C itself is the stack discipline for automatic variables. Two subroutines exist for more flexible storage handling. The function **calloc(n, s)** returns a pointer to a properly aligned storage block that will hold **n** items each of which is **s** bytes long. Normally **s** is obtained from the **sizeof** pseudo-function, a compile-time function that yields the size in bytes of a variable or data type. To return a block obtained from **calloc** to the free storage pool, **cfree(p)** may be called, where **p** is a value returned by a previous call to **calloc**.

Another set of routines deals with string handling. There is no "string" data type, but an array of characters, with a convention that the end of a string is indicated by a null byte, can be used for the same purpose. The most commonly used string routines perform the functions of copying one string to another, comparing two strings, and computing a string length. More sophisticated string operations can often be performed using the I/O routines, which are described next.

Most of the routines in the standard library deal with input and output. Most C programmers use stream I/O, although there is no reason why record I/O could not be used with the language. There are three default streams: the standard input, the standard output, and the error output. The most elementary routines for dealing with these streams are **getchar()** which reads a character from the standard input, and **putchar(c)**, which writes the character **c** on the standard output. In the environments in which C programs run, it is generally possibly to redirect these streams to files or other

programs; the program itself does not change and is unaware of the redirection.

The most common output function is **printf(format, data1, data2, ...)**, which performs data conversion for formatted output. The string **format** is copied to the standard output, except that when a conversion specification introduced by a % character is found in **format** it is replaced by the value of the next **data** argument, converted according to the specification. For example,

```
printf("n = %d, x = %f", n, x);
```

prints n as a decimal integer and x as a floating point number, as in

```
n = 17, x = 12.34
```

A similar function **scanf** performs formatted input conversion.

All the routines mentioned have versions that operate on streams other than the standard input or output, and **printf** and **scanf** variants may also process a string, to allow for in-memory format conversion. Other routines in the I/O library transmit whole lines between memory and files, and check for error or end-of-file status.

Many other routines and utilities are used with C, somewhat more on UNIX than on other systems. As an example, it is possible to compile and load a C program so that when the program is run, data are collected on the number of times each function is called and how long it executes. This profile pinpoints the parts of a program that dominate the run-time.

VIII. EXPERIENCE WITH C

C compilers exist for the most widely used machines at Bell Laboratories (the IBM S/370, Honeywell 6000, PDP-11) and perhaps 10 others. Several hundred programmers within Bell Laboratories and many outside use C as their primary programming language.

8.1 Favorable experiences

C has completely displaced assembly language in UNIX programs. All applications code, the C compiler itself, and the operating system (except for about 1000 lines of initial bootstrap, etc.) are written in C. Although compilers or interpreters are available under UNIX for Fortran, Pascal, Algol 68, Snobol, APL, and other

languages, most programmers make little use of them. Since C is a relatively low-level language, it is adequately efficient to prevent people from resorting to assembler, and yet sufficienctly terse and expressive that its users prefer it to PL/I or other very large languages.

A language that doesn't have everything is actually easier to program in than some that do. The limitations of C often imply shorter manuals and easier training and adaptation. Language design, especially when done by a committee, often tends toward including all doubtful features, since there is no quick answer to the advocate who insists that the new feature will be useful to some and can be ignored by others. But this results in long manuals and hierarchies of "experts" who know progressively larger subsets of the language. In practice, if a feature is not used often enough to be familiar and does not complete some structure of syntax or semantics, it should probably be left out. Otherwise, the manual and compiler get bulky, the users get surprises, and it becomes harder and harder to maintain and use the language. It is also desirable to avoid language features that cannot be compiled efficiently; programmers like to feel that the cost of a statement is comparable to the difficulty in writing it. C has thus avoided implementing operations in the language that would have to be performed by subroutine call. As compiler technology improves, some extensions (e.g., structure assignment) are being made to C, but always with the same principles in mind.

One direction for possible expansion of the language has been explicitly avoided. Although C is much used for writing operating systems and associated software, there are no facilities for multiprogramming, parallel operations, synchronization, or process control. We believe that making these operations primitives of the language is inappropriate, mostly because language design is hard enough in itself without incorporating into it the design of operating systems. Language facilities of this sort tend to make strong assumptions about the underlying operating system that may match very poorly what it actually does.

8.2 Unfavorable experiences

The design and implementation of C can (or could) be criticized on a number of points. Here we discuss some of the more vulnerable aspects of the language.

8.2.1 Language level

Some users complain that C is an insufficiently high-level language; for example, they want string data types and operations, or variable-size multi-dimensional arrays, or generic functions. Sometimes a suggested extension merely involves lifting some restriction. For example, allowing variable-size arrays would actually simplify the language specification, since it would only involve allowing general expressions in place of constants in certain contexts.

Many other extensions are plausible; since the low level of C was praised in the previous section as an advantage of the language, most will not be further discussed. One is worthy of mention, however. The C language provides no facility for I/O, leaving this job to library routines. The following fragment illustrates one difficulty with this approach:

```
printf("%d\n", x);
```

The problem arises because on machines on which **int** is not the same as **long**, x may not be **long**; if it were, the program must be written

```
printf("%D\n", x);
```

so as to tell **printf** the length of x. Thus, changing the type of x involves changing not only its declaration, but also other parts of the program. If I/O were built into the language, the association between the type of an expression and the format in which it is printed could be reconciled by the compiler.

8.2.2 Type safety

C has traditionally been permissive in checking whether an expression is used in a context appropriate to its type. A complete list of examples would be long, but two of the most important should illustrate sufficiently. The types of formal arguments of functions are in general not known, and in any case are not checked by the compiler against the actual arguments at each call. Thus in the statement

```
s = sin(1);
```

the fact that the **sin** routine takes a floating-point argument is not noticed until the erroneous result is detected by the programmer.

In the structure reference

```
p->memb
```

p is simply assumed to point to a structure of which **memb** is a member; p might even be an integer and not a pointer at all.

Much of the explanation, if not justification, for such laxity is the typeless nature of C's predecessor languages. Fortunately, a justification need no longer be attempted, since a program is now available that detects all common type mismatches. This utility, called **lint** because it picks bits of fluff from programs, examines a set of files and complains about a great many dubious constructions, ranging from unused or uninitialized variables through the type errors mentioned. Programs that pass unscathed through **lint** enjoy about as complete freedom from type errors as do Algol 68 programs, with a few exceptions: unions are not checked dynamically, and explicit escapes are available that in effect turn off checking.

Some languages, such as Pascal and Euclid, allow the writer to specify that the value of a given variable may assume only a given subrange of the integers. This facility is often connected with the usage of arrays, in that any array index must be a variable or expression whose type specifies a subset of the set given by the bounds of the array. This approach is not without theoretical difficulties, as suggested by Habermann.[4] In itself it does not solve the problems of variables assuming unexpected values or of accessing outside array bounds; such things must (in general) be detected dynamically. Still, the extra information provided by specifying the permissible range for each variable provides valuable information for the compiler and any verifier program. C has no corresponding facility.

One of the characteristic features of C is its rather complete integration of the notion of pointer and of address arithmetic. Some writers, notably Hoare,[5] have argued against the very notion of pointer. We feel, however, that the facilities offered by pointers are too valuable to give up lightly.

8.2.3 Syntax peculiarities

Some people are annoyed by the terseness of expression that is one of the characteristics of the language. We view C's short operators and general lack of noise as a benefit. For example, the use of braces { } for grouping instead of **begin** and **end** seems appropriate in view of the frequency of the operation. The use of braces even fits well into ordinary mathematical notation.

Terseness can lead to code that is hard to read, however. For example,

```
*++*argv
```

where **argv** has been declared **char** ***argv** (pointer into an array of character pointers) means: select the character pointer pointed at by **argv** (***argv**), increment it by one (**++*argv**), then fetch the character that *that* pointer points at (***++*argv**). This is concise and efficient but reminiscent of APL.

An example of a minor problem is the comment convention, which is PL/I's /* ... */. Comments do not nest, so an effort to "comment out" a section of code will fail if that section contains a comment. And a number of us can testify that it is surprisingly hard to recognize when an "end comment" delimiter has been botched, so that the comment silently continues until the next comment is reached, deleting a line or two of code. It would be more convenient if a single unique character were reserved to introduce a comment, and if comments always terminated at an end of line.

8.2.4 Semantic peculiarities

There are some occasionally surprising operator precedences. For example,

```
a >> 4 + 5
```

shifts right by 9. Perhaps worse,

```
(x & MASK) == 0
```

must be parenthesized to associate the proper way. Users learn quickly to parenthesize such doubtful cases; and when feasible **lint** warns of suspicious expressions (including both of these).

We have already mentioned the fact that the **case** actions in a switch flow through unless explicitly broken. In practice, users write so many **switch** statements that they become familiar with this behavior and some even prefer it.

Some problems arise from machine differences that are reflected, perhaps unnecessarily, into the semantics of C. For example, the PDP-11 does sign extension on byte fetches, so that a character (viewed arithmetically) can have a value ranging from −128 to +127, rather than 0 to +255. Although the reference manual makes it quite clear that the precise range of a **char** variable is machine dependent, programmers occasionally succumb to the

temptation of using the full range that their local machine can represent, forgetting that their programs may not work on another machine. The fundamental problem, of course, is that C permits small numbers, as well as genuine characters, to be stored in char variables. This might not be necessary if, for example, the notion of subranges (mentioned above) were introduced into the language.

8.2.5 Miscellaneous

C was developed and is generally used in a highly responsive interactive environment, and accordingly the compiler provides few of the services usually associated with batch compilers. For example, it prepares no listing of the source program, no cross reference table, and no indication of the nature of the generated code. Such facilities are available, but they are separate programs, not parts of the compiler. Programmers used to batch environments may find it hard to live without giant listings; we would find it hard to use them.

IX. CONCLUSIONS AND FUTURE DIRECTIONS

C has continued to develop in recent years, mostly by upwardly compatible extensions, occasionally by restrictions against manifestly nonportable or illegal programs that happened to be compiled into something useful. The most recent major changes were motivated by the extension of C to other machines, and the resulting emphasis on portability. The advent of union and of casts reflects a desire to be more precise about types when moving to other machines is in prospect. These changes have had relatively little effect on programmers who remained entirely on the UNIX system. Of more importance was a new library, which changed the use of a "portable" library from an option into an effective standard, while simultaneously increasing the efficiency of the library so that users would not object.

It is more difficult, of course, to speculate about the future. C is now encountering more and more foreign environments, and this is producing many demands for C to adapt itself to the hardware, and particularly to the operating systems, of other machines. Bit fields, for example, are a response to a request to describe externally imposed data layouts. Similarly, the procedures for external storage allocation and referencing have been made tighter to conform to requirements on other systems. Portability of the basic language

seems well handled, but interactions with operating systems grow ever more complex. These lead to requests for more sophisticated data descriptions and initializations, and even for assembler windows. Further changes of this sort are likely.

What is not likely is a fundamental change in the level of the language. Realistically, the very acceptance of C has compelled changes to be made only most cautiously and compatibly. Should the pressure for improvements become too strong for the language to accommodate, C would probably have to be left as is, and a totally new language developed. We leave it to the reader to speculate on whether it should be called D or P.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Englewood Cliffs, N.J.: Prentice-Hall, 1978.
2. M. Richards, "BCPL: A Tool for Compiler Writing and Systems Programming," Proc. AFIPS SJCC, *34* (1969), pp. 557-566.
3. S. C. Johnson and B. W. Kernighan, "The Programming Language B," Comp. Sci. Tech. Rep. No. 8, Bell Laboratories (January 1973).
4. A. N. Habermann, "Critical Comments on the Programming Language PASCAL," Acta Informatica, *3* (1973), pp. 47-58.
5. C. A. R. Hoare, "Data Reliability," ACM SIGPLAN Notices, *10* (June 1975), pp. 528-533.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

Portability of C Programs and the UNIX System

By S. C. JOHNSON and D. M. RITCHIE
(Manuscript received December 5, 1977)

Computer programs are portable to the extent that they can be moved to new computing environments with much less effort than it would take to rewrite them. In the limit, a program is perfectly portable if it can be moved at will with no change whatsoever. Recent C language extensions have made it easier to write portable programs. Some tools have also been developed that aid in the detection of nonportable constructions. With these tools many programs have been moved from the PDP-11 on which they were developed to other machines. In particular, the UNIX operating system and most of its software have been transported to the Interdata 8/32. The source-language representation of most of the code involved is identical in all environments.*

I. INTRODUCTION

A program is portable to the extent that it can be easily moved to a new computing environment with much less effort than would be required to write it afresh. It may not be immediately obvious that lack of portability is, or needs to be, a problem. Of course, practically no assembly-language programs are portable. The fact is, however, that most programs, even in high-level languages, depend explicitly or implicitly on assumptions about such machine-

* UNIX is a trademark of Bell Laboratories.

dependent features as word and character sizes, character set, file system structure and organization, peripheral device handling, and many others. Moreover, few computer languages are understood by more than a handful of kinds of machines, and those that are (for example, Fortran and Cobol) tend to be rather limited in their scope, and, despite strong standards efforts, still differ considerably from one machine to another.

The economic advantages of portability are very great. In many segments of the computer industry, the dominant cost is development and maintenance of software. Any large organization, certainly including the Bell System, will have a variety of computers and will want to run the same program at many locations. If the program must be rewritten for each machine and maintained for each, software costs must increase. Moreover, the most effective hardware for a given job is not constant as time passes. If a nonportable program remains tied to obsolete hardware to avoid the expense of moving it, the costs are equally real even if less obvious. Finally, there can be considerable benefit in using machines from several manufacturers simply to avoid being utterly dependent on a single supplier.

Most large computer systems spend most of their time executing application programs; circuit design and analysis, network routing, simulation, data base applications, and text processing are particularly important at Bell Laboratories. For years, application programs have been written in high-level languages, but the programs that provide the basic software environment of computers (for example, operating systems, compilers, text editors, etc.) are still usually coded in assembly language. When the costs of hardware were large relative to the costs of software, there was perhaps some justification for this approach; perhaps an equally important reason was the lack of appropriate, adequately supported languages. Today hardware is relatively cheap, software is expensive, and any number of languages are capable of expressing well the algorithms required for basic system software. It is a mystery why the vast majority of computer manufacturers continue to generate so much assembly-language software.

The benefits of writing software in a well-designed language far exceed the costs. Aside from potential portability, these benefits include much smaller development and maintenance costs. It is true that a penalty must be paid for using a high-level language, particularly in memory space occupied. The cost in time can usually be controlled: experience shows that the time-critical part of most

programs is only a few percent of the total code. Careful design allows this part to be efficient, while the remainder of the program is unimportant.

Thus, we take the position that essentially all programs should be written in a language well above the level of machine instructions. While many of the arguments for this position are independent of portability, portability is itself a very important goal; we will try to show how it can be achieved almost as a by-product of the use of a suitable language.

We have recently moved the UNIX system kernel, together with much of its software, from its original host machine (DEC PDP-11) to a very different machine (Interdata 8/32). Almost all the programs involved are written in the C language,[1,2] and almost all are identical on the two systems. This paper discusses some of the problems encountered, and how they were solved by changing the language itself and by developing tools to detect and resolve nonportable constructions. The major lessons we have learned, and that we hope to teach, are that portable programs are good programs for more reasons than that they are portable, and that making programs portable costs some intellectual effort but need not degrade their performance.

II. HISTORY

The Computing Science Research Center at Bell Laboratories has been interested in the problems and technologies of program portability for over a decade. Altran[3] is a substantial (25,000 lines) computer algebra system, written in Fortran, which was developed with portability as one of its primary goals. Altran has been moved to many incompatible computer systems; the effort involved for each move is quite moderate. Out of the Altran effort grew a tool, the PFORT verifier,[4] that checks Fortran programs for adherence to a strict set of programming conventions. Most importantly, it detects (where possible) whether the program conforms to the ANSI standard for Fortran,[5] but because many compilers fail to accept even standard-conforming programs, it also remarks upon several constructions that are legal but nevertheless nonportable. Successful passage of a program through PFORT is an important step in assuring that it is portable. More recently, members of the Computer Science Research Center and the Computing Technology Center jointly created the PORT library of mathematical software.[6] Implementation of PORT required research not merely into the language issues, but

also into deeper questions of the model of floating point computations on the various target machines.

In parallel with this work, the development at Bell Laboratories of Snobol4[7] marks one of the first attempts at making a significant compiler portable. Snobol4 was successfully moved to a large number of machines, and, while the implementation was sometimes inefficient, the techniques made the language widely available and stimulated additional work leading to more efficient implementations.

III. PORTABILITY OF C PROGRAMS — INITIAL EXPERIENCES

C was developed for the PDP-11 on the UNIX system in 1972. Portability was not an explicit goal in its design, even though limitations in the underlying machine model assumed by the predecessors of C made us well aware that not all machines were the same.[2] Less than a year later, C was also running on the Honeywell 6000 system at Murray Hill. Shortly thereafter, it was made available on the IBM 370 series machines as well. The compiler for the Honeywell was a new product,[8] but the IBM compiler was adapted from the PDP-11 version, as were compilers for several other machines.

As soon as C compilers were available on other machines, a number of programs, some of them quite substantial, were moved from UNIX to the new environments. In general, we were quite pleased with the ease with which programs could be transferred between machines. Still, a number of problem areas were evident. To begin with, the C language was growing and developing as experience suggested new and desirable features. It proved to be quite painful to keep the various C compilers compatible; the Honeywell version was entirely distinct from the PDP-11 version, and the IBM version had been adapted, with many changes, from a by-then obsolete version of the PDP-11 compiler. Most seriously, the operating system interface caused far more trouble for portability than the actual hardware or language differences themselves. Many of the UNIX primitives were impossible to imitate on other operating systems; moreover, some conventions on these other operating systems (for example, strange file formats and record-oriented I/O) were difficult to deal with while retaining compatibility with UNIX. Conversely, the I/O library commonly used sometimes made UNIX conventions excessively visible—for example, the number 518 often found its way into user programs as the size, in bytes, of a particularly efficient I/O buffer structure.

Additional problems in the compilers arose from the decision to use the local assemblers, loaders, and library editors on the host operating systems. Surprisingly often, they were unable to handle the code most naturally produced by the C compilers. For example, the semantics of possibly initialized external variables in C was quite consciously designed to be implementable in a way identical to Fortran's COMMON blocks to guarantee its portability. It was an unpleasant surprise to discover that the Honeywell assembler would allow at most 61 such blocks (and hence external variables) and that the IBM link-editor preferred to start external variables on even 4096-byte boundaries. Software limitations in the target systems complicated the compilers and, in one case, the problems with external variables just mentioned, forced changes in the C language itself.

IV. THE UNIX PORTABILITY PROJECT

The realization that the operating systems of the target machines were as great an obstacle to portability as their hardware architecture led us to a seemingly radical suggestion: to evade that part of the problem altogether by moving the operating system itself.

Transportation of an operating system and its software between non-trivially different machines is rare, but not unprecedented.[9-13] Our own situation was a bit different in that we already had a moderately large, complete, and mature system in wide use at many installations. We could not (or at any rate did not want to) start afresh and redesign the language, the operating system interfaces, and the software. It seemed, though, that despite some problems in each we had a good base to build on.

Our project had three major goals:

(*i*) To write a compiler for C that could be changed without grave difficulty to generate code for a variety of machines.

(*ii*) To refine and extend the C language to make most C programs portable to a wide variety of machines, mechanically identifying non-portable constructions where possible.

(*iii*) To revise or recode a substantial portion of UNIX in portable C, detecting and isolating machine dependencies, and demonstrate its portability by moving it to another machine.

By pursuing each goal, we hoped to attain a corresponding benefit:

(*i*) A C compiler adaptable to other machines (independently of UNIX), that puts into practice some recent developments in the theory of code generation.

(*ii*) Improved understanding of the proper design of languages that, like C, operate on a level close to that of real machines but that can be made largely machine-independent.

(*iii*) A relatively complete and usable implementation of UNIX on at least one other machine, with the hope that subsequent implementations would be fairly straightforward.

We selected the Interdata 8/32 computer to serve as the initial target for the system portability research. It is a 32-bit computer whose design resembles that of the IBM System/360 and /370 series machines, although its addressing structure is rather different; in particular, it is possible to address any byte in virtual memory without use of a base register. For the portability research, of course, its major feature is that it is *not* a PDP-11. In the longer term, we expect to find it especially useful for solving problems, often drawn from numerical analysis, that cannot be handled on the PDP-11 because of its limited address space.

Two portability projects besides those referred to above are particularly interesting. In the period 1976-1977, T. L. Lyon and his associates at Princeton adapted the UNIX kernel to run in a virtual-machine partition under VM/370 on an IBM System/370.[14] Enough software was also moved to demonstrate the feasibility of the effort, though no attempt was made to produce a complete, working system. In the midst of our own work on the Interdata 8/32, we learned that a UNIX portability project, for the similar Interdata 7/32, was under way at the University of Wollongong in Australia.[15] Since everything we know of this effort was discovered in discussion with its major participant, Richard Miller,[16] we will remark only that the transportation route chosen was markedly different from ours. In particular, an Interdata C compiler was adapted from the PDP-11 compiler, and was moved as soon as possible to the Interdata, where it ran under the manufacturer's operating system. Then the UNIX kernel was moved in pieces, first running with dummy device drivers as a task under the Interdata system, and only at the later stages independently. This approach, the success of which must be scored as a real *tour de force*, was made necessary by the 100 kilometers separating the PDP-11 in Sydney from the Interdata in Wollongong.

4.1 Project chronology

Work began in the early months of 1977 on the compiler, assembler, and loader for the Interdata machine. Soon after its delivery at

the end of April 1977, we were ready to check out the compiler. At about the same time, the operating system was being scrutinized for nonportable constructions. During May, the Interdata-specific code in the kernel was written, and by June, it was working well enough to begin moving large amounts of software; T. L. Lyon aided us greatly by tackling the bulk of this work. By August, the system was unmistakably UNIX, and it was clear that, as a research project, the portability effort had succeeded, although there were still programs to be moved and bugs to be stamped out. From late summer until October 1977, work proceeded more slowly, owing to a combination of hardware difficulties and other claims on our time; by the spring of 1978 the portability work as such was complete. The remainder of this paper discusses how success was achieved.

V. SOME NON-GOALS

It was and is clear that the portability achievable cannot approach that of Altran, for example, which can be brought up with a fortnight of effort by someone skilled in local conditions but ignorant of Altran itself. In principle, all one needs to implement Altran is a computer with a standard Fortran compiler and a copy of the Altran system tape; to get it running involves only defining of some constants characterizing the machine and writing a few primitive operations in assembly language.

In view of the intrinsic difficulties of our own project, we did not feel constrained to insist that the system be so easily portable. For example, the C compiler is not bootstrapped by means of a simple interpreter for an intermediate language; instead, an acceptably efficient code generator must be written. The compiler is indeed designed carefully so as to make changes easy, but for each new machine it inevitably demands considerable skill even to decide on data representations and run-time conventions, let alone the code sequences to be produced. Likewise, in the operating system, there are many difficult and inevitably machine-dependent issues, including especially the treatment of interrupts and faults, memory management, and device handling. Thus, although we took some care to isolate the machine-dependent portions of the operating system into a set of primitive routines, implementation of these primitives involves deep knowledge of the most recondite aspects of the target machine.

Moreover, we could not attempt to make the portable UNIX system compatible with software, file formats, or inadequate character

sets already existing on the machine to which it is moved; to promise to do so would impossibly complicate the project and, in fact, might destroy the usefulness of the result. If UNIX is to be installed on a machine, its way of doing business must be accepted as the right way; afterwards, perhaps, other software can be made to work.

VI. THE PORTABLE C COMPILER

The original C compiler for the PDP-11 was not designed to be easy to adapt for other machines. Although successful compilers for the IBM System/370 and other machines were based on it, much of the modification effort in each case, particularly in the early stages, was concerned with ridding it of assumptions about the PDP-11. Even before the idea of moving UNIX occurred to us, it was clear that C was successful enough to warrant production of compilers for an increasing variety of machines. Therefore, one of the authors (SCJ) undertook to produce a new compiler intended from the start to be easily modified. This new compiler is now in use on the IBM System/370 under both OS and TSS, the Honeywell 6000, the Interdata 8/32, the SEL86, the Data General Nova and Eclipse, the DEC VAX-11/780, and a Bell System processor. Versions are in progress for the Intel 8086 microprocessor and other machines.

The degree of portability achieved by this compiler is satisfying. In the Interdata 8/32 version, there are roughly 8,000 lines of source code. The first pass, which does syntax and lexical analysis and symbol table management, builds expression trees, and generates a bit of machine-dependent code such as subroutine prologues and epilogues, consists of 4,600 lines of code, of which 600 are machine-dependent. In the second pass, which does the bulk of the code generation, 1,000 out of 3,400 lines are machine-dependent. Thus, out of a total of 8,000 lines, 1,600, or 20 percent, are machine-dependent; the remaining 80 percent are shared with the Honeywell, IBM, and other compilers. As the Interdata compiler becomes more carefully tuned, the machine-dependent figures will rise somewhat; for the IBM, the machine-dependent fraction is 22 percent; for the Honeywell, 25 percent.

These figures both overstate and understate the true difficulty of moving the compiler. They represent the size of those source files that contain machine-dependent code; only a half or a third of the lines in many machine-dependent functions actually differ from machine to machine, because most of the routines involved remain similar in structure. As an example, routines to output branches,

align location counters, and produce function prologues and epilogues have a clear machine-dependent component, but nevertheless are logically very similar for all the compilers. On the other hand, as we discuss below, the hardest part of moving the compiler is not reflected in the number of lines changed, but is instead concerned with understanding the code generation issues, the C language, and the target machine well enough to make the modifications effectively.

The new compiler is not only easily adapted to a new machine, it has other virtues as well. Chief among these is that all versions share so much code that maintenance of all versions simultaneously involves much less work than would maintaining each individually. For example, if a bug is discovered in the machine-independent portion, the repair can be made to all versions almost mechanically. Even if the language itself is changed, it is often the case that most of the job of installing the change is machine-independent and usable for all versions. This has allowed the compilers for all machines to remain compatible with a minimum of effort.

The interface between the two passes of the portable C compiler consists of an intermediate file containing mostly representations of expression trees together with character representations of stereotyped code for subroutine prologues and epilogues. Thus a different first pass can be substituted provided it conforms to the interface specifications. This possibility allowed S. I. Feldman to write a first pass that accepts the Fortran 77 language instead of C. At the moment, the Fortran front-end has two versions (which differ by about as much as do the corresponding first passes for C) that feed the code generators for the PDP-11 and the Interdata machines. Thus we apparently have not only the first, but the first two implementations of Fortran 77.

6.1 Design of the portable compiler

Most machine-dependent portions of a C compiler fall into three categories.

(*i*) Storage allocation.
(*ii*) Rather stereotyped code sequences for subroutine entry points and exits, switches, labels, and the like.
(*iii*) Code generation for expressions.

For the most part, storage allocation issues are easily parameterized in terms of the number of bits required for objects of the

various types and their alignment requirements. Some issues, like addressability on the IBM 360 and 370 series, cause annoyance, but generally there are few problems in this area.

The calling sequence is very important to the efficiency of the result and takes considerable knowledge and imagination to design properly. However, once designed, the calling sequence code and the related issue of stack frame layout are easy to cope with in the compiler.

Generating optimal code for arithmetic expressions, even on idealized machines, can be shown theoretically to be a nearly intractable problem. For the machines we are given in real life, the problem is even harder. Thus, all compilers have to compromise a bit with optimality and engage in heuristic algorithms to some extent, in order to get acceptably efficient code generated in a reasonable amount of time.

The design of the code generator was influenced by a number of goals, which in turn were influenced by recent theoretical work in code generation. It was recognized that there was a premium in being able to get the compiler up and working quickly; it was also felt, however, that this was in many ways less important than being able to evolve and tune the compiler into a high-quality product as time went on. Particularly with operating system code, a "quick and dirty" implementation is simply unacceptable. It was also recognized that the compiler was likely to be applied to machines not well understood by the compiler writer that might have inadequate or nonexistent debugging facilities. Therefore, one goal of the compiler was to permit it to be largely self-checking. Rather than produce incorrect code, we felt it far preferable for the compiler to detect its own inadequacies and reject the input program.

This goal was largely met. The compiler for the Interdata 8/32 was working within a couple of weeks after the machine arrived; subsequently, several months went by with very little time lost due to compiler bugs. The bug level has remained low, even as the compiler has begun to be more carefully tuned; many of the bugs have resulted from human error (e.g., misreading the machine manual) rather than actual compiler failure.

Several techniques contribute considerably to the general reliability of the compiler. First, a conscious attempt was made to separate information about the machine (e.g., facts such as "there is an add instruction that adds a constant to a register and sets the condition code") from the strategy, often heuristic, that makes use of these facts (e.g., if an addition is to be done, first compute the left-hand

operand into a register). Thus, as the compiler evolves, more effort can be put into improving the heuristics and the recognition of important special cases, while the underlying knowledge about the machine operations need not be altered. This approach also improves portability, since the heuristic programs often remain largely unchanged among similar machines, while only the detailed knowledge about the format of the instructions (encoded in a table) changes.

During compilation of expressions, a model of the state of the compilation process, including the tree representing the expression being compiled and the status of the machine registers, is maintained by the compiler. As instructions are emitted, the expression tree is simplified. For example, the expression a = b+c might first be transformed into a = *register*+b as a load instruction for a is generated, then into a = *register* when an add is produced. The possible transformations constitute the "facts" about the machine; the order in which they are applied correspond to the heuristics. When the input expression has been completely transformed into nothing, the expression is compiled. Thus, a good portion of the initial design of a new version of the compiler is concerned with making the model within the compiler agree with the actual machine by building a table of machine operations and their effects on the model. When this is done correctly, one has a great deal of confidence that the compiler will produce correct code, if it produces any at all.

Another useful technique is to partition the code generation job into pieces that interact only through well-defined paths. One module worries about breaking up large expressions into manageable pieces, and allocating temporary storage locations when needed. Another module worries about register allocation. Finally, a third module takes each "manageable" piece and the register allocation information, and generates the code. The division between these pieces is strict; if the third module discovers that an expression is "unmanageable," or a needed register is busy, it rejects the compilation. The division enforces a discipline on the compiler which, while not really restricting its power, allows for fairly rapid debugging of the compiler output.

The most serious drawback of the entire approach is the difficulty of proving any form of "completeness" property for the compiler—of demonstrating that the compiler will in fact successfully generate code for all legal C programs. Thus, for example, a needed transformation might simply be missing, so that there might be no

way to further simplify some expression. Alternatively, some sequence of transformations might result in a loop, so that the same expression keeps reappearing in a chain of transformations. The compiler detects these situations by realizing that too many passes are being made over the expression tree, and the input is rejected. Unfortunately, detection of these possibilities is difficult to do in advance because of the use of heuristics in the compiler algorithms. Currently, the best way of ensuring that the compiler is acceptably complete is by extensive testing.

6.2 Testing the compiler

We ordered the Interdata 8/32 without any software at all, so we first created a very crude environment that allowed stand-alone programs to be run; all interrupts, memory mapping, etc., were turned off. The compiler, assembler, and loader ran on the PDP-11, and the resulting executable files were transferred to the Interdata for testing. Primitive routines permitted individual characters to be written on the console. In this environment, the basic stack management of the compiler was debugged, in some cases by single-stepping the machine. This was a painful but short period.

After the function call mechanism was working, other short tests established the basic sanity of simple conditionals, assignments, and computations. At this point, the stand-alone environment could be enriched to permit input from the console and more informative output such as numbers and character strings, so ordinary C programs could be run. We solicited such programs, but found few that did not depend on the file system or other operating system features. Some of the most useful programs at this stage were simple games that pitted the computer against a human; they frequently did a large amount of computing, often with quite complicated logic, and yet restricted themselves to simple input and output. A number of compiler bugs were found and fixed by running games. After these tests, the compiler ceased to be an explicit object of testing, and became instead a tool by which we could move and test the operating system.

Some of the most subtle problems with compiler testing come in the maintenance phase of the compiler, when it has been tested, declared to work, and installed. At this stage, there may be some interest in improving the code quality as well as fixing the occasional bug. An important tool here is regression testing; a collection of test programs are saved, together with the previous compiler output.

Before a new compiler is installed, the new compiler is fed these test programs, the new output is compared with the saved output, and differences are noted. If no differences are seen, and a compiler bug has been fixed or improvement made, the testing process is incomplete, and one or more test programs are added. If differences are detected, they are carefully examined. The basic problem is that frequently, in attempting to fix a bug, the most obvious repair can give rise to other bugs, frequently breaking code that used to work. These other bugs can go undetected for some time, and are very painful both to the users and the compiler writer. Thus, regression tests attempt to guard against introducing new bugs while fixing old ones.

The portable compiler is sufficiently self-checked that many potential compiler bugs were detected before the compiler was installed by the simple expedient of turning the compiler loose on a large amount (tens of thousands of lines) of C source code. Many constructions turned up there that were undreamed of by the compiler writer, and often mishandled by the compiler.

It is worth mentioning that this kind of testing is easily carried out by means of the standard commands and features in the UNIX system. In particular, C source programs are easily identified by their names, and the UNIX shell provides features for applying command sequences automatically to each of a list of files in turn. Moreover, powerful utilities exist to compare two similar text files and produce a minimal list of differences. Finally, the compiler produces assembly code that is an ordinary text file readable by all of the usual utilities. Taken together, these features make it very simple to invent test drivers. For example, it takes only a half-dozen lines of input to request a list of differences between the outputs of two versions of the compiler applied to tens (or hundreds) of source files. Perhaps even more important, there is little or no output when the compilers compare exactly. On many systems, the "job control language" required to do this would be so unpleasant as to insure that it would not be done. Even if it were, the resulting hundreds of pages of output could make it very difficult to see the places where the compiler needed attention.

The design of the portable C compiler is discussed more thoroughly in Ref. 17.

VII. LANGUAGE AND COMPILER ISSUES

We were favorably impressed, even in the early stages, by the

general ease with which C programs could be moved to other machines. Some problems we did encounter were related to weaknesses in the C language itself, so we undertook to make a few extensions.

C had no way of accounting in a machine-independent way for the overlaying of data. Most frequently, this need comes up in large tables that contain some parts having variable structure. As an invented example, a compiler's table of constants appearing in a source program might have a flag indicating the type of each constant followed by the constant's value, which is either integer or floating. The C language as it existed allowed sufficient cheating to express the fact that the possible integer and floating value might be overlaid (both would not exist at once), but it could not be expressed portably because of the inability to express the relative sizes of integers and floating-point data in a machine-independent way. Therefore, the **union** declaration was added; it permits such a construction to be expressed in a natural and portable manner. Declaring a union of an integer and a floating point number reserves enough storage to hold either, and forces such alignment properties as may be required to make this storage useful as both an integer and a floating point number. This storage may be explicitly used as either integer or floating point by accessing it with the appropriate descriptor tag.

Another addition was the **typedef** facility, which in effect allows the types of objects to be easily parameterized. **typedef** is used quite heavily in the operating system kernel, where the types of a number of different kinds of objects, for example, disk addresses, file offsets, device numbers, and times of day, are specified only once in a header file and assigned to a specific name; this name is then used throughout. Unlike some languages, C does not permit definition of new operations on these new types; the intent was increased parameterization rather than true extensibility.

Although the C language did benefit from these extensions, the portability of the average C program is improved more by restricting the language than by extending it. Because it descended from typeless languages, C has traditionally been rather permissive in allowing dubious mixtures of various types; the most flagrant violations of good practice involved the confusion of pointers and integers. Some programs explicitly used character pointers to simulate unsigned integers; on the PDP-11 the two have the same arithmetic properties. Type **unsigned** was introduced into the language to eliminate the need for this subterfuge.

More often, type errors occurred unconsciously. For example, a function whose only use of an argument is to pass it to a subfunction might allow the argument to be taken to be an integer by default. If the top-level actual argument is a pointer, the usage is harmless on many machines, but not type-correct and not, in general, portable.

Violations of strict typing rules existed in many, perhaps most, of the programs making up the entire stock of UNIX system software. Yet these programs, representing many tens of thousands of lines of source code, all worked correctly on the PDP-11 and in fact would work on many other machines, because the assumptions they made were generally, though not universally, satisfied. It was not feasible simply to declare all the suspect constructions illegal. Instead, a separate program was written to detect as many dubious coding practices as possible. This program, called **lint**, picks bits of fluff from programs in much the same way as the PFORT verifier mentioned above. C programs acceptable to **lint** are guaranteed to be free from most common type errors; **lint** also checks syntax and detects some logical errors, such as uninitialized variables, unused variables, and unreachable code.

There are definite advantages in separating program-checking from compilation. First, **lint** was easy to produce, because it is based on the portable compiler and thus shares the machine-independent code of the first pass with the other versions of the compiler. More important, the compilers, large programs anyway, are not burdened with a great deal of checking code which does not necessarily apply to the machine for which they are running. A good example of extra capability feasible in **lint** but probably not in the compilers themselves is checking for inter-program consistency. The C compilers all permit separate compilation of programs in several files, followed by linking together of the results. **lint** (uniquely) checks consistency of declarations of external variables, functions, and function arguments among a set of files and libraries.

Finally, **lint** itself is a portable program, identical on all machines. Although care was taken to make it easy to propagate changes in the machine-independent parts of the compilers with a minimum of fuss, it has proved useful for the sometimes complicated logic of **lint** to be totally decoupled from the compilers. **lint** cannot possibly affect their ability to produce code; if a bug in **lint** turns up, its output can be ignored and work can continue simply by ignoring the spurious complaints. This kind of separation of function is characteristic of UNIX programs in general. The compiler's one important

job is to generate code; it is left to other programs to print listings, generate cross-reference tables, and enforce style rules.

VIII. THE PORTABILITY OF THE UNIX KERNEL

The UNIX operating system kernel, or briefly the operating system, is the permanently resident program that provides the basic software environment for all other programs running on the machine. It implements the "system calls" by which user's programs interact with the file system and request other services, and arranges for several programs to share the machine without interference. The structure of the UNIX operating system kernel is discussed elsewhere in this issue.[18,19]

To many people, an operating system may seem the very model of a nonportable program, but in fact a major portion of UNIX and other well-written operating systems consists of machine-independent algorithms: how to create, read, write, and delete files, how to decide who to run and who to swap, and so forth. If the operating system is viewed as a large C program, then it is reasonable to hope to apply the same techniques and tools to it that we apply to move more modest programs.

The UNIX kernel can be roughly divided into three sections according to their degree of portability.

8.1 Assembly-language primitives

At the lowest level, and least portable, is a set of basic hardware interface routines. These are written in assembly language, and consist of about 800 lines of code on the Interdata 8/32. Some of them are callable directly from the rest of the system, and provide services such as enabling and disabling interrupts, invoking the basic I/O operations, changing the memory map so as to switch execution from one process to another, and transmitting information between a user process's address space and that of the system. Most of them are machine-independent in specification, although not implementation. Other assembly-language routines are not called explicitly but instead intercept interrupts, traps, and system calls and turn them into C-style calls on the routines in the rest of the operating system.

Each time UNIX is moved to a new machine, the assembly-language portion of the system must be rewritten. Not only is the assembly code itself machine-specific, but the particular features

provided for memory mapping, protection, and interrupt handling and masking differ greatly from machine to machine. In moving from the PDP-11 to the Interdata 8/32, a huge preponderance of the bugs occurred in this section. One reason for this is certainly the usual sorts of difficulties found in assembly-language programming: we wrote loops that did not loop or looped forever, garbled critical constants, and wrote plausible-looking but utterly incorrect address constructions. Lack of familiarity with the machine led us to incorrect assumptions about how the hardware worked, and to inefficient use of available status information when things went wrong.

Finally, the most basic routines for multi-programming, those that pass control from one process to another, turned out (after causing months of nagging problems) to be incorrectly specified and actually unimplementable correctly on the Interdata, because they depended improperly on details of the register-saving mechanism of the calling sequence generated by the compiler. These primitives had to be redesigned; they are of special interest not only because of the problems they caused, but because they represent the only part of the system that had to be significantly changed, as distinct from expressed properly, to achieve portability.

8.2 Device drivers

The second section of the kernel consists of device drivers, the programs that provide the interrupt handling, I/O command processing, and error recovery for the various peripheral devices connected to the machine. On the Interdata 8/32 the total size of drivers for the disk, magnetic tape, console typewriter, and remote typewriters is about 1100 lines of code, all in C. These programs are, of course, machine-dependent, since the devices are.

The drivers caused far fewer problems than did the assembly-language programs. Of course, they already had working models on the PDP-11, and we had faced the need to write new drivers several times in the past (there are half a dozen disk drivers for various kinds of hardware attached to the PDP-11). In adapting to the Interdata, the interface to the rest of the system survived unchanged, and the drivers themselves shared their general structure, and even much code, with their PDP-11 counterparts. The problems that occurred seem more related to the general difficulty of dealing with the particular devices than in expressing what had to be done.

8.3 The remainder of the system

The third and remaining section of the kernel is the largest. It is all written in C, and for the Interdata 8/32 contains about 7,000 lines of code. This is the operating system proper, and clearly represents the bulk of the code. We hoped that it would be largely portable, and as it turned out our hopes were justified. A certain amount of work had to be done to achieve portability. Most of it was concerned with making sure that everything was declared properly, so as to satisfy **lint**, and with replacing constants by parameters. For example, macros were written to perform various unit conversions previously written out explicitly: byte counts to memory segmentation units and to disk blocks, etc. The important data types used within the system were identified and specified using **typedef**: disk offsets, absolute times, internal device names, and the like. This effort was carried out by K. Thompson.

Of the 7,000 lines in this portion of the operating system, only about 350 are different in the Interdata and PDP-11 versions; that is, they are 95 percent identical. Most of the differences are traceable to one of three areas.

(*i*) On the PDP-11, the subroutine call stack grows towards smaller addresses, while on the Interdata it grows upwards. This leads to different code when increasing the size of a user stack, and especially when creating the argument list for an inter-program transfer (**exec** system call) because the arguments are placed on the stack.

(*ii*) The details of the memory management hardware on the two machines are different, although they share the same general scheme.

(*iii*) The routine that handles processor traps (memory faults, etc.) and system calls is rather different in detail on the two machines because the set of faults is not identical, and because the method of argument transmission in system calls differs as well.

We are extremely gratified by the ease with which this portion of the system was transferred. Only a few problems showed up in the code that was not changed; most were in the new code written specifically for the Interdata. In other words, what we thought would be portable did in fact move without trouble.

Not everything went perfectly smoothly, of course. Our first set of major problems involved the mechanics of transferring test

systems and other programs from the PDP-11 to the Interdata 8/32 and debugging the result. Better communications between the machines would have helped considerably. For a period, installing a new Interdata system meant creating an 800 BPI tape on the sixth-floor PDP-11, carrying the tape to another PDP-11 on the first floor to generate a 1600 BPI version, and finally lugging the result to the fifth-floor Interdata. For debugging, we would have been much aided by a hardware interface between the PDP-11 and the front panel of the Interdata to allow remote rebooting. This class of problems is basically our own fault, in that we traded the momentary ease of not having to write communications software or build hardware for the continuing annoyance of carrying tapes and hands-on debugging.

Another class of problems seems impossible to avoid, since it stems from the basic differences in the representation of information on the two machines. In the machines at issue, only one difference is important: the PDP-11 addresses the two bytes in a 16-bit word with the first byte as the least significant 8 bits, while on the Interdata the first byte in a 16-bit half-word is the most significant 8 bits. Since all the interfaces between the two machines are byte-serial, the effect is best described by saying that when a true character stream is transmitted between them, all is well; but if integers are sent, the bytes in each half-word must be swapped. Notice that this problem does not involve portability in the sense in which it has been used throughout this paper; very few C programs are sensitive to the order in which bytes are stored on the machine on which they are running. Instead it complicates "portability" in its root meaning wherein files are carried from one machine to the other. Thus, for example, during the initial creation of the Interdata system we were obliged to create, on the PDP-11, an image of a file system disk volume that would be copied to tape and thence to the Interdata disk, where it would serve as an actual file system for the latter machine. It required a certain amount of cleverness to declare the data structures appropriately and to decide which bytes to swap.

The ordering of bytes in a word on the PDP-11 is somewhat unusual, but the problem it poses is quite representative of the difficulties of transferring encoded information from machine to machine. Another example is the difference in representation of floating-point numbers between the PDP-11 and the Interdata. The assembler for the Interdata, when it runs on the PDP-11, must invoke a routine to convert the "natural" PDP-11 notation to the foreign notation, but of course this conversion must not be done

when the assembler is run on the Interdata itself. This makes the assembler *necessarily* non-portable, in the sense that it must execute different code sequences on the two machines. However, it can have a single source representation by taking advantage of conditional compilation depending on where it will run.

This kind of problem can get much worse: how are we to move UNIX to a target machine with a 36-bit word length, whose machine word cannot even be represented by **long** integers on the PDP-11? Nevertheless, it is worth emphasizing that the problem is really vicious only during the initial bootstrapping phase; all the software should run properly if only it can be moved once!

IX. TRANSPORTATION OF THE SOFTWARE

Most UNIX code is in neither the operating system itself nor the compiler, but in the many user-level utilities implementing various commands and in subroutine libraries. The sheer bulk of the programs involved (about 50,000 lines of source) meant that the amount of work in transportation might be considerable, but our early experience, together with the small average size of each individual program, convinced us that it would be manageable. This proved to be the case.

Even before the advent of the Interdata machine, it was realized, as mentioned above, that many programs depended to an undesirable degree not only on UNIX I/O conventions but on details of particularly favorable buffering strategies for the PDP-11. A package of routines, called the "portable I/O library," was written by M. E. Lesk[20] and implemented on the Honeywell and IBM machines as well as the PDP-11 in a generally successful effort to overcome the deficiencies of earlier packages. This library too proved to have some difficulties, not in portability, but in time efficiency and space required. Therefore a new package of routines, dubbed the "standard I/O library," was prepared. Similar in spirit to the portable library, it is somewhat smaller and much faster. Thus, part of the effort in moving programs to the Interdata machine was devoted to making programs use the new standard I/O library. In the simplest cases, the effort involved was nil, since the fundamental character I/O functions have the same names in all libraries.

Next, each program had to be examined for visible lack of portability. Of course, **lint** was a valuable tool here. Programs were also scrutinized by eye to detect dubious constructions. Often these involved constants. For example, on the 16-bit PDP-11 the

expression

```
x & 0177770
```

masks off all but the last three bits of x, since 0177770 is an octal constant. This is almost certainly better expressed

```
x & ~07
```

(where ~ is the ones-complement operator) because the latter expression actually does yield the last three bits of x independently of the word length of the machine. Better yet, the constant should be a parameter with a meaningful name.

UNIX software has a number of conventional data structures, ranging from objects returned or accepted by the operating system kernel (such as status information for a named file) to the structure of the header of an executable file. Programs often had a private copy of the declaration for each such structure they used, and often the declaration was nonportable. For example, an encoded file mode might be declared **int** on the 16-bit PDP-11, but on the 32-bit Interdata machine, it should be specified as **short**, which is unambiguously 16 bits. Therefore, another major task in making the software portable was to collect declarations of all structures common to several routines, to put the declarations in a standard place, and to use the **include** facility of the C preprocessor to insert them in the source program. The compiler for the PDP-11 and the cross-compiler for the Interdata 8/32 were adjusted to search a different standard directory to find the canned declarations appropriate to each.

Finally, an effort was made to seek out frequently occurring patches of code and replace them by standard subroutines, or create new subroutines where appropriate. It turned out, for example, that several programs had built-in subroutines to find the printable user name corresponding to a numerical user ID. Although in each case the subroutine as written was acceptably portable to other machines, the function it performed was not portable in time across changes in the format of the file describing the name-number correspondence; encapsulating the translation function insulated the program against possible changes in a data base.

X. THE MACHINE MODEL FOR C

One of the hardest parts of designing a language in which to write portable programs is deciding which properties are guaranteed to

remain invariant. Likewise, in trying to develop a portable operating system, it is very hard to decide just what properties of the underlying machine can be depended on. The design questions in each case are many in number; moreover, the answer to each individual question may involve tradeoffs that are difficult to evaluate in advance. Here we try to show the nature of these tradeoffs and what sort of compromises are required.

Designing a language in which every program is portable is actually quite simple: specify precisely the meaning of every legal program, as well as what programs are legal. Then the portability problem does not exist: by definition, if a correct program fails on some machine, the language has not been implemented properly. Unfortunately, a language like C that is intended to be used for system programming is not very adaptable to such a Procrustean approach, mainly because reasonable efficiency is required. Any well-defined language can be implemented precisely on any general-purpose computer, but the implementation may not be usable in practice if it implies use of an interpreter rather than machine instructions. Thus, with both language and operating system design, one must strike a balance between convenient and powerful features and the ease of implementing them efficiently on a variety of machines. At any point, some machine may be found on which some feature is very expensive to provide, and a decision must be made whether to modify the feature, and thus compromise the portability of programs that use it, or to insist that the meaning is immutable and must be preserved. In the latter case portability is also compromised since the cost of using the feature may be so high that no one can afford the programs that use it, or the people attempting to implement the feature on the new machine give up in despair.

Thus a language definition implies a model of the machine on which programs in the language will run. If a real machine conforms well to the model, then an implementation on that machine is likely to be efficient and easily written; if not, the implementation will be painful to provide and costly to use. Here we shall consider the major features of the abstract C machine that have turned out to be most relevant so far.

10.1 Integers

Probably the most frequent operations are on integers consisting of various numbers of bits. Variables declared **short** are at least 16 bits in length; those declared **long** are at least 32 bits. Most are

declared **int**, and must be at least as precise as **short** integers, but may be **long** if accessing them as such is more efficient. It is interesting that the word length, which is one of the machine differences that springs first to mind, has caused rather little trouble. A small amount of code (mostly concerned with output conversion) assumes a twos complement representation.

10.2 Unsigned integers

Unsigned integers corresponding to **short** and **int** must be provided. The most relevant properties of unsigned integers appear when they are compared or serve as numerators in division and remaindering. Unsigned arithmetic may be somewhat expensive to implement on some machines, particularly if the number representation is sign-magnitude or ones complement. No use is made of unsigned **long** integers.

10.3 Characters

A representation of characters (bytes) must be provided with at least 8 bits per byte. It is irrelevant whether bytes are signed, as in the PDP-11, or not, as in all other known machines. It is moderately important that an integer of any kind be divisible evenly into bytes. Most programs make no explicit use of this fact, but the I/O system uses it heavily. (This tends to rule out one plausible representation of characters on the DEC PDP-10, which is able to access five 7-bit characters in a 36-bit word with one bit left over. Fortunately, that machine can access four 9-bit characters equally well.) Almost all programs are independent of the order in which the bytes making up an integer are stored, but see the discussion above on this issue.

A fair number of programs assume that the character set is ASCII. Usually the dependence is relatively minor, as when a character is tested for being a lower case letter by asking if it is between *a* and *z* (which is not a correct test in EBCDIC). Here the test could be easily replaced by a call to a standard macro. Other programs that use characters to index a table would be much more difficult to render insensitive to the character set. ASCII is, after all, a U. S. national standard; we are inclined to make it a UNIX standard as well, while not ruling out C compilers for other systems based on other character sets (in fact the current IBM System/370 compiler uses EBCDIC).

10.4 Pointers

Pointers to objects of the various basic types are used very heavily. Frequent operations on pointers include assignment, comparison, addition and subtraction of an integer, and dereferencing to yield the object to which the pointer points. It was frequently assumed in earlier UNIX code that pointers and integers had a similar representation (for example, that they occupied the same space). Now this assumption is no longer made in the programs that have been moved. Nevertheless, the representation of pointers remains very important, particularly in regard to character pointers, which are used freely. A word-addressed machine that lacks any natural representation of a character pointer may suffer serious inefficiency for some programs.

10.5 Functions and the calling sequence

UNIX programs tend to be built out of many small, frequently called functions. It is not unusual to find a program that spends 20 percent of its time in the function prologue and epilogue sequence, nor one in which 20 percent of the code is concerned with preparing function argument lists. On the PDP-11/70 the calling sequence is relatively efficient (it costs about 20 microseconds to call and return from a function) so it is clear that a less efficient calling sequence will be quite expensive. Any function in C may be recursive (without special declaration) and most possess several "automatic" variables local to each invocation. These characteristics suggest strongly that a stack must be used to store the automatic variables, caller's return point, and saved registers local to each function; in turn, the attractiveness of an implementation will depend heavily on the ease with which a stack can be maintained. Machines with too few index or base registers may not be able to support the language well.

Efficiency is important in designing a calling sequence; moreover, decisions made here tend to have wide implications. For example, some machines have a preferred direction of growth for the stack. On the PDP-11, the stack is practically forced to grow towards smaller addresses; on the Interdata the stack prefers (somewhat more weakly) to grow upwards. Differences in the direction of stack growth leads to differences in the operating system, as has already been mentioned.

XI. THE MACHINE MODEL OF UNIX

The definition of C suggests that some machines are more suitable for C implementations than others; likewise, the design of the UNIX kernel fits in well with some machine architectures and poorly with others. Once again, the requirements are not absolute, but a serious enough mismatch may make an implementation unattractive. Because the system is written in C, of course, a (perhaps necessarily) slow or bulky implementation of the language will lead to a slow or bulky operating system, so the remarks in the previous section apply. But other aspects of machine design are especially relevant to the operating system.

11.1 Mapping and the user program

As discussed in other papers,[18,21] the system provides user programs with an address space consisting of up to three logical segments containing the program text, an extensible data region, and a stack. Since the stack and the data are both allowed to grow at one edge, it is desirable (especially where the virtual address space is limited) that one grow in the negative direction, towards the other, so as to optimize the use of the address space. A few programs still assume that the data space grows in the positive direction (so that an array at its end can grow contiguously), although we have tried to minimize this usage. If the virtual address space is large, there is little loss in allowing both the data and stack areas to grow upwards.

The PDP-11 and the Interdata provide examples of what can be done. On the former machine, the data area begins at the end of the program text and grows upwards, while the stack begins at the end of the virtual address space and grows downwards; this is, happily, the natural direction of growth for the stack. On the Interdata the data space begins after the program and grows upwards; the stack begins at a fixed location and also grows upwards. The layout provides for a stack of at most 128K bytes and a data area of 852K bytes less the program size, as compared to the total data and stack space of 64K bytes possible on the PDP-11.

It is hard to characterize precisely what is required of a memory mapping scheme except by discussing, as we do here, the uses to which it is put. In general, paging or segmentation schemes seem to offer sufficient generality to make implementation simple; a single base and limit register (or even dual registers, if it is desired to

write-protect the program text) are marginal, because of the difficulty of providing independently growable data and stack areas.

11.2 Mapping and the kernel

When a process is running in the UNIX kernel, a fixed region of the kernel's address space contains data specific to that process, including its kernel stack. Switching processes essentially involves changing the address map so that the same fixed range of virtual addresses refers to the data area and stack of the new process. This implies, of course, that the kernel runs in mapped mode, so that mapping should not be tied to operating in user mode. It also means that if the machine has but a single set of mapping specification registers, these registers will have to be reloaded on each system call and certain interrupts, for example from the clock. This causes no logical problems but may affect efficiency.

11.3 Other considerations

Many other aspects of machine design are relevant to implementation of the operating system but are probably less important, because on most machines they are likely to cause no difficulty. Still, it is worthwhile to attempt a list.

(*i*) The machine must have a clock capable of generating interrupts at a rate not far from 50 or 60 Hz. The interrupts are used to schedule internal events such as delays for mechanical motion on typewriters. As written, the system uses clock interrupts to maintain absolute time, so the interrupt rate should be accurate in the long run. However, changes to consult a separate time-of-day clock would be minimal.

(*ii*) All disk devices should be able to handle the same, relatively small, block sizes. The current system usually reads and writes 512-byte blocks. This number is easy to change, but if it is made much larger, the efficacy of the system's cache scheme will degrade seriously unless a large amount of memory is devoted to buffers.

XII. WHAT HAS BEEN ACCOMPLISHED?

In about six months, we have been able to move the UNIX operating system and much of its software from its original host, the PDP-11, to another, rather different machine, the Interdata 8/32. The

standard of portability achieved is fairly high for such an ambitious project: the operating system (outside of device drivers and assembly language primitives) is about 95 percent unchanged between the two systems; inherently machine-dependent software such as the compiler, assembler, loader, and debugger are 75 to 80 percent unchanged; other user-level software (amounting to about 20,000 lines so far) is identical, with few exceptions, on the two machines.

It is true that moving a program from one machine to another does not guarantee that it can be moved to a third. There are many issues in portability about which we worried in a theoretical way without having to face them in fact. It would be interesting, for example, to tackle a machine in which pointers were a different size from integers, or in which character pointers were fundamentally different in structure from integer pointers, or with a different character set. There are probably even issues in portability that we failed to consider at all. Nevertheless, moving UNIX to a third new machine, or a fourth, will be easier than it was to the second. The operating system and the software have been carefully parameterized, and this will not have to be done again. We have also learned a great deal about the critical issues (the "hard parts").

There are deeper limitations to the generality of what we have done. Consider the use of memory mapping: if the hardware cannot support the model assumed by the code as it is written, the code must be changed. This may not be difficult, but it does represent a loss of portability. Correspondingly, the system as written does not take advantage of extra capability beyond its model, so it does not support (for example) demand paging. Again, this would require new code. More generally, algorithms do not always scale well; the optimal methods of sorting files of ten, a thousand, and a million elements do not much resemble one another. Likewise, some of the design of the system as it exists may have to be reworked to take full advantage of machines much more powerful (along many possible dimensions) than those for which it was designed. This seems to be an inherent limit to portability; it can only be handled by making the system easy to change, rather than easily portable unchanged. Although we believe UNIX possesses both virtues, only the latter is the subject of this paper.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Englewood Cliffs, N.J.: Prentice-Hall, 1978.

2. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," B.S.T.J., this issue, pp. 1991-2019.
3. W. S. Brown, *ALTRAN User's Manual,* 4th ed., Murray Hill, N.J.,: Bell Laboratories, 1977.
4. B. G. Ryder, "The PFORT Verifier," Software — Practice and Experience, *4* (October-December 1974), pp. 359-377.
5. *American National Standard FORTRAN,* New York, N.Y.: American National Standards Institute, 1966. (ANS X3.9)
6. P. A. Fox, A. D. Hall, and N. L. Schryer, "The PORT Mathematical Subroutine Library," ACM Trans. Math. Soft. (1978), to appear.
7. R. E. Griswold, J. Poage, and I. Polonsky, *The SNOBOL4 Programming Language,* Englewood Cliffs, N. J.: Prentice-Hall, 1971.
8. A. Snyder, *A Portable Compiler for the Language C,* Cambridge, Mass.: Master's Thesis, M.I.T., 1974.
9. D. Morris, G. R. Frank, and C. J. Theaker, "Machine-Independent Operating Systems," in *Information Processing 77,* North-Holland (1977), pp. 819-825.
10. J. E. Stoy and C. Strachey, "OS6—An experimental operating system for a small computer. Part 1: General principles and structure," Comp. J., *15* (May 1972), pp. 117-124.
11. J. E. Stoy and C. Strachey, "OS6—An experimental operating system for a small computer. Part 2: Input/output and filing system." Comp. J., *15* (August 1972), pp. 195-203.
12. D. Thalmann and B. Levrat, "SPIP, a Way of Writing Portable Operating Systems," Proc. ACM Computing Symposium (1977), pp. 452-459.
13. L. S. Melen, *A Portable Real-Time Executive, Thoth,* Waterloo, Ontario, Canada: Master's Thesis, Dept. of Computer Science, University of Waterloo, October 1976.
14. T. L. Lyon, private communication
15. R. Miller, "UNIX — A Portable Operating System?" Australian Universities Computing Science Seminar (February, 1978).
16. R. Miller, private communication.
17. S. C. Johnson, "A Portable Compiler: Theory and Practice," Proc. 5th ACM Symp. on Principles of Programming Languages (January 1978).
18. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
19. D. M. Ritchie, "UNIX Time-Sharing System: A Retrospective," B.S.T.J., this issue, pp. 1947-1969. Also in Proc. Hawaii International Conference on Systems Science, Honolulu, Hawaii, Jan. 1977.
20. D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, "The C Programming Language," Comp. Sci. Tech. Rep. No. 31, Bell Laboratories (October 1975).
21. K. Thompson, "UNIX Time-Sharing System: UNIX Implementation," B.S.T.J., this issue, pp. 1931-1946.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

The MERT Operating System

By H. LYCKLAMA and D. L. BAYER
(Manuscript received December 5, 1977)

The MERT operating system supports multiple operating system environments. Messages provide the major means of inter-process communication. Shared memory is used where tighter coupling between processes is desired. The file system was designed with real-time response being a major concern. The system has been implemented on the DEC PDP-11/45 and PDP-11/70 computers and supports the UNIX time-sharing system, as well as some real-time processes.*

The system is structured in four layers. The lowest layer, the kernel, provides basic services such as inter-process communication, process dispatching, and trap and interrupt handling. The second layer comprises privileged processes, such as I/O device handlers, the file manager, memory manager, and system scheduler. At the third layer are the supervisor processes which provide the programming environments for application programs of the fourth layer.

To provide an environment favorable to applications with real-time response requirements, the MERT system permits processes to control scheduling parameters. These include scheduling priority and memory residency. A rich set of inter-process communication mechanisms including messages, events (software interrupts), shared memory, inter-process traps, process ports, and files, allow applications to be implemented as several independent, cooperating processes.

Some uses of the MERT operating system are discussed. A

* UNIX is a trademark of Bell Laboratories.

retrospective view of the MERT system is also offered. This includes a critical evaluation of some of the design decisions and a discussion of design improvements which could have been made to improve overall efficiency.

I. INTRODUCTION

As operating systems become more sophisticated and complex, providing more and more services for the user, they become increasingly difficult to modify and maintain. Fixing a "bug" in some part of the system may very likely introduce another "bug" in a seemingly unrelated section of code. Changing a data structure is likely to have major impact on the total system. It has thus become increasingly apparent over the past years that adhering to the principles of structured modularity[1,2] is the correct approach to building an operating system. The objective of the MERT system has been to extend the concept of a process into the operating system, factoring the traditional operating system functions into a small kernel surrounded by a set of independent cooperating processes. Communication between these processes is accomplished primarily through messages. Messages define the interface between processes and reduce the number of ways a bug can be propagated through the system.

The MERT kernel establishes an extended instruction set via system primitives vis-à-vis the virtual machine approach of CP 67. Operating systems are implemented on top of the MERT kernel and define the services available to user programs. Communication and synchronization primitives and shared memory permit varying degrees of cooperation between independent operating systems. An operating system functionally equivalent to the UNIX* time-sharing system has been implemented to exercise the MERT executive and provide tools for developing and maintaining other operating system environments. An immediate benefit of this approach is that operating system environments tailored to the needs of specific classes of real-time projects can be implemented without interfering with other systems having different objectives.

One of the basic design goals of the system was to build modular and independent processes having data structures and tables which are known only to the particular process. Fixing a "bug" or making major internal changes in one process does not affect the other

* UNIX is a trademark of Bell Laboratories.

processes with which it communicates. The work described here builds on previous operating system designs described by Dijkstra[1] and Brinch Hansen.[2] The primary differences between this system and previous work lies in the rich set of inter-process communication techniques and the extension of the concept of independent modular processes, protected from other processes in the system, to the basic I/O and real-time processes. It can be shown that messages are not an adequate communication path for some real-time problems.[3] Controlled access to shared memory and software-generated interrupts are often required to maintain the integrity of a real-time system. The communication primitives were selected in an attempt to balance the need for protection with the need for real-time response. The primitives include event flags, message buffers, inter-process system traps, process ports and shared segments.

One of the major influences on the design of the MERT system came from the requirements of various application systems at Bell Laboratories. They made use of imbedded minicomputers to provide support for development of application programs and for controlling their specific application. Many of these projects had requirements for real-time response to various external events. Real-time can be classified into two categories. One flavor of real time requires the collection of large amounts of data. This necessitates the implementation of large and contiguous files and asynchronous I/O. The second flavor of real time demands quick response to hardware-generated interrupts. This necessitates the implementation of processes locked in memory. Yet another requirement for some applications was the need to define a more controlled environment with better control over a program's virtual address space layout than that provided in a general-purpose time-sharing environment.

This paper gives a detailed description of the system design including the kernel and a definition and description of processes and of segments. A detailed discussion of the communication primitives follows. The structure of the file system is then discussed, along with how the file manager and time-sharing processes make use of the communication primitives.

A major portion of this paper deals with a critical retrospective on the MERT system. This includes a discussion of features of the MERT system which have been used by various projects within the Bell System. Some trade-offs are given that have been made for efficiency reasons, thereby sacrificing some protection. Some

operational statistics are also included here. The pros and cons of certain features of the MERT operating system are discussed in detail. The portability of the operating system as well as user software is currently a topic of great interest. The prospects of the portability of the MERT system are described. Finally, we discuss some features of the MERT system which could have been implemented differently for the sake of efficiency.

II. HARDWARE REQUIREMENTS

The MERT system currently runs on the DEC PDP-11/45 and PDP-11/70 computers.[4] These computers provide an eight-level static priority interrupt structure with priority levels numbered from 0 (lowest) to 7 (highest). Associated with the interrupt structure is the programmed interrupt register which permits the processor to generate interrupts at priorities of one through seven. The programmed interrupt serves as the basic mechanism for driving the system.

The PDP-11 computer is a 16-bit word machine with a direct address space of 32K words. The memory management unit on the PDP-11/45 and PDP-11/70 computers provides a separate set of address mapping and access control registers for each of the processor modes: kernel, supervisor, and user. Furthermore, each virtual address space can provide separate maps for instruction references (called I-space) and data references (D-space). The MERT system makes use of all three processor modes (kernel, supervisor, and user) and both the instruction and data address spaces provided by these machines.

III. SYSTEM DESIGN

Processes are arranged in four levels of protection (see Fig. 1). The lowest level of the operating system structure, called the kernel, allocates the basic computer resources. These resources consist of memory, segments, the CPU, and interrupts. All process dispatching, including interrupt processing, is handled by the kernel dispatcher. The kernel is the most highly privileged system component and therefore must be the most reliable.

The second level of software consists of kernel-mode processes which comprise the various I/O device drivers. Each process at this level has access to a limited number of I-space base registers in the kernel mode, providing a firewall between it and sensitive system

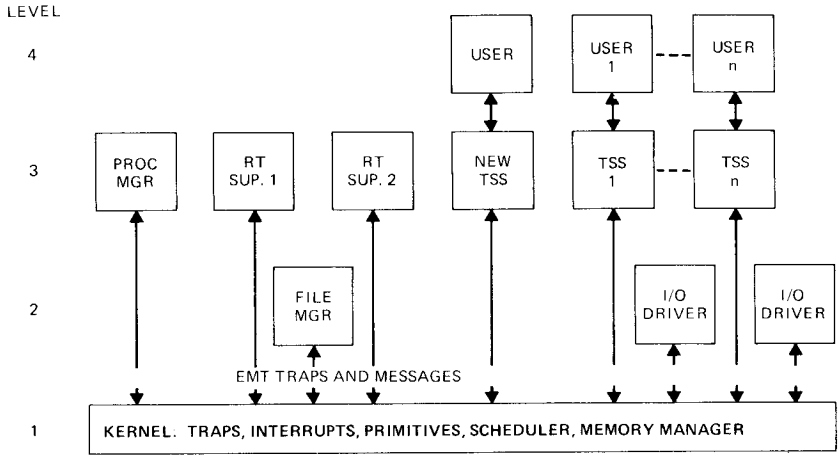

Fig. 1—System structure.

data accessible only using D-space mode. Within this level processes are linked onto one of five priority lists. These lists correspond to the processor priority required while the process is executing. Three kernel processes must exist for the system to function:

(*i*) The file manager is required since all processes are derived from files.

(*ii*) The swap process is required to move segments between secondary storage and main memory.

(*iii*) The root process is required to carry out data transfers between the file manager and the disk.

Since the same device usually contains both the swap area and root file system, one process usually serves for both (*ii*) and (*iii*).

At the third software level are the various operating system supervisors which run in supervisor mode. These processes provide the environments which the user sees and the interface to the basic kernel services. All processes at this level execute at a processor priority of either one or zero. A software priority is maintained for the supervisor by the scheduler process. Two supervisor processes are always present: the process manager which creates all new processes* and produces post-mortem dumps of processes which terminate abnormally, and the time-sharing supervisor.

* The time-sharing supervisor can create a new process consisting of an exact copy of itself.

At the fourth level are the various user procedures which execute in user mode under control of the supervisory environments. The primitives available to the user are provided by the supervisory environments which process the user system calls. Actually, the user procedure is merely an extension of the supervisor process. This is the highest level of protection provided by the computer hardware.

IV. DEFINITIONS

4.1 Segments

A logical segment is a piece of contiguous memory, 32 to 32K 16-bit words long, which can grow in increments of 32 words. Associated with each segment are an internal segment identifier (ID) and an optional global name. The segment identifier is allocated to the segment when it is created and is used for all references to the segment. The global name uniquely defines the initial contents of the segment. A segment is created on demand and disappears when all processes which are linked to it are removed. The contents of a segment may be initialized by copying all or part of a file into the segment. Access to the segment can be controlled by the creator (parent) as follows:

(*i*) The segment can be private — that is, available only to the creator.

(*ii*) The segment can be shared by the creator and some or all of its descendants (children). This is accomplished by passing the segment ID to a child.

(*iii*) The segment can be given a name which is available to all processes in the system. The name is a unique 32-bit number which corresponds to the actual location on secondary storage of the initial segment data. Processes without a parent-child relationship can request the name from the file system and then attempt to create a segment with that name. If the segment exists, the segment ID is returned and the segment user count is incremented. Otherwise, the segment is created and the process initializes it.

4.2 Processes

A process consists of a collection of related logical segments executed by the processor. Processes are divided into two classes,

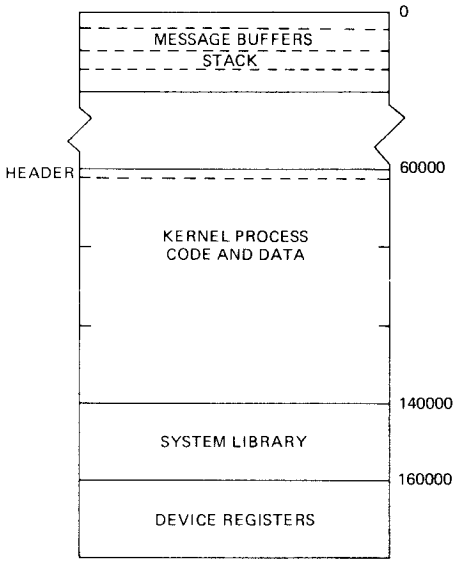

Fig. 2—Virtual address space of a typical kernel process.

kernel processes and supervisor processes, according to the mode of the processor while executing the segments of the process.

4.2.1 Kernel processes

Kernel processes are driven by software and hardware interrupts, execute at processor hardware priority 2 to 7, are locked in memory, and are capable of executing all privileged instructions. Kernel processes are used to control peripheral devices and handle functions with stringent real-time response requirements.

The virtual address space of each kernel process begins with a short header which defines the virtual address space and various entry points (see Fig. 2). Up to 12K words (base registers 3 - 5) of instruction space and 12K words of data space are available. All kernel processes share a common stack and can read and write the I/O device registers.

To reduce duplication of common subprograms used by independent kernel processes and to provide common data areas between independent cooperating kernel and supervisor processes, three mechanisms for sharing segments are available.

The first type of shared segment, called the system library, is available to all kernel processes. The routines included in this

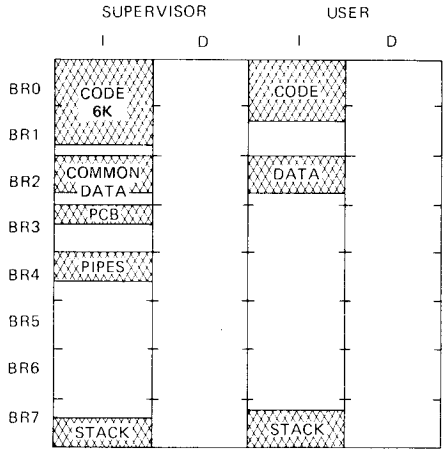

Fig. 3—UNIX™ process virtual address space.

library are determined by the system administrator at system generation time. The system library begins at virtual address 140000(8) (base register 6) and is present whether or not it is used by any kernel processes.

The second type of shared segment, called a public library, is assigned to base register 4 or 5 of the process instruction space. References to routines in the library are satisfied when the process is formed, but the body of the segment is loaded into memory only when the first process which accesses it is loaded.

A third sharing mechanism allows a parent to pass the ID of a segment that is included in the address space of a kernel process when it is created. This form of sharing is useful when a hierarchy of cooperating processes is invoked to accomplish a task.

4.2.2 Supervisor processes

All processes which execute in supervisor mode and user mode are called supervisor processes. These processes run at processor priority 0 or 1 and are scheduled by the kernel scheduler process. The segments of a supervisor may be kept in memory, providing response on the order of several milliseconds, or supervisor segments may be swappable, providing a response time of hundreds of milliseconds.

The virtual address space of a supervisor process consists of 32K words of instruction space and 32K words of data space in both supervisor and user modes (see Fig. 3). Of these 128K, at least part

of each of three base registers (a total of 12K) must be used for access to:

(*i*) The process control block (PCB), a segment typically 160 words long, which describes the entire virtual address space of the process to the kernel and provides space to save the state of the process during a context switch. The PCB also includes a list of capabilities which define the range of abilities of the process.
(*ii*) The process supervisor stack and data segment.
(*iii*) The read-only code segment of the supervisor.

The rest of the virtual address space is controlled by the supervisor. The primitives available to supervisor processes include the ability to control the virtual address space (both supervisor and user) which can be accessed by the process.

4.3 Capabilities

Associated with each supervisor process is a list of keys, each of which allows access to one object. The capability key must be passed as an argument in all service requests on objects. Each key is a concatenation of the process ID of the creator of the object and a bit pattern, defined by the creator, which describes allowed operations on the object. The capability list (C-list) for each supervisor process resides in the PCB and is maintained by the kernel through **add** and **delete** capability messages to the memory manager. A special variation of the **send** message primitive copies the capability from the PCB into the body of a message, preventing corruption of the capability mechanism.

Capabilities are used by the file manager to control access to files. The capability for a file is granted upon opening the file. A read or write request is validated by decoding the capability into a 14-bit object descriptor (file descriptor) and a 2-bit permission field. The capability is removed from the process C-list when the file is closed.

V. THE KERNEL

The concept of an operating system nucleus or kernel has been used in several systems. Each system has included a different set of logical functions.[5,6] The MERT kernel is to be distinguished from a *security* kernel. A *security* kernel provides the basis of a secure operating system environment.

The basic kernel provides a set of services available to all processes, kernel and supervisor, and maintains the system process tables and segment tables. Included as part of the kernel are two special system processes, the memory manager and the scheduler. These are distinguished from other kernel processes in that they are bound into the basic kernel address space and do not require the set-up of a base register when control is turned over to one of these processes.

5.1 Kernel modules

The kernel consists of a process dispatcher, a trap handler, and routines (procedures) which implement the system primitives. Approximately 4K words of code are dedicated to these modules.

The *process dispatcher* is responsible for saving the current state and setting up and dispatching to all kernel processes. It can be invoked by an interrupt from the programmed interrupt register, an interrupt from an external device, or an inter-process system trap from a supervisor process (an **emt** trap).

The *trap handler* fields all traps and faults and, in most cases, transfers control to a trap handling routine in the process which caused the trap or fault.

The *kernel primitives* can be grouped into eight logical categories. These categories can be subdivided into those which are available to all processes and others which are available only to supervisor processes. The primitives which are available to all processes are:

(*i*) Interprocess communication and synchronization primitives. These include sending and receiving messages and events, waking up processes which are sleeping on a bit pattern, and setting the sleep pattern.
(*ii*) Attaching to and detaching from interrupts.
(*iii*) Setting a timer to cause a time-out event.
(*iv*) Manipulation of segments for the purposes of input/output. This includes locking and unlocking segments and marking segments altered.
(*v*) Setting and getting the time of day.

The primitives available only to supervisor processes are:

(*vi*) Primitives which alter the attributes of the segments of a process. These primitives include creating new segments,

returning segments to the system, adding and deleting segments from the process address space, and altering the access permissions.

(*vii*) Altering scheduler-related parameters by roadblocking, changing the scheduling priority, or making the segments of the process nonswap or swappable.

(*viii*) Miscellaneous services such as reading the console switches.

5.2 Kernel system processes

Closely associated with the kernel are the memory management and scheduler processes. These two processes are special in that they reside in the kernel address space permanently. In all other respects, they follow the discipline established for kernel processes.

5.2.1 Memory manager

The memory manager is a special system process. It communicates with the rest of the system via messages and is capable of handling four types of requests:

(*i*) Setting the segments of a process into the active state, making space by swapping or shifting other segments if necessary.

(*ii*) Loading and locking a segment contiguous with other locked segments to reduce memory fragmentation.

(*iii*) Deactivating the segments of a process.

(*iv*) Adding and deleting capabilities from the capability list in a supervisor process PCB.

5.2.2 Scheduler

The scheduler is the second special system process and is responsible for scheduling all supervisor processes. The main responsibility of the scheduler is to select the next process to be executed. The actual loading of the process is accomplished by the memory manager.

5.3 Dispatcher mechanism

The system maintains seven process lists, one for each processor priority at which software interrupts can be triggered using the

programmed interrupt register. All kernel processes are linked into one of the six lists for processor priorities 2 through 7; all supervisor processes are linked to the processor priority 1 list. The occurrence of a software interrupt at priorities 2 through 7 causes the process dispatcher to search the corresponding process list and dispatch to all processes which have one or more event flags set. The entire list is searched for each software interrupt.

5.4 Scheduling policy

All software interrupts at processor priority 1, which are not for the currently active process, cause the dispatcher to send a wakeup event to the scheduler process. The scheduler uses a byte in the system process tables to maintain the scheduling priority of each process. This byte is manipulated by the scheduler as follows:

(*i*) Incremented when a process receives an event.
(*ii*) Increased by 10 when awakened by a kernel process.
(*iii*) Decremented when the process yields control due to a roadblock system call.
(*iv*) Lowered according to an exponential function each successive time the process uses its entire time slice (becomes compute bound).

The process list is searched for the highest priority process which is ready to run, and if this process has higher priority than the current process, the new process will preempt the current process.

To minimize thrashing and swapping, the scheduler uses a "will receive an event soon" flag which is set by the process when it roadblocks. This flag is typically set when a process roadblocks awaiting completion of I/O which is expected to finish in a short time relative to the length of the time slice. The scheduler will keep the process in memory for the remainder of its time slice. When memory becomes full and all processes which require loading are of sufficiently low priority, the scheduler stops making load requests until one of the processes being held runs out of its time slice.

VI. INTER-PROCESS COMMUNICATION

A structured system requires a well-defined set of communication primitives to permit inter-process communication and synchronization. The MERT system makes use of the following communication primitives to achieve this end:

(*i*) Event flags.
(*ii*) Message buffers.
(*iii*) **emt** traps.
(*iv*) Shared memory.
(*v*) Files.
(*vi*) Process ports.

Each of these is discussed in further detail here.

6.1 Event flags

Event flags are an efficient means of communication between processes for the transfer of small quantities of data. Of the 16 possible event flags per process, eight are predefined by the system for the following events: wakeup, timeout, message arrival, hangup, interrupt, quit, abort, and initialization. The other eight event flags are definable by the processes using the event flags as a means of communication. Events are sent by means of the kernel primitive:

event(procid, event)

When control is passed to the process at its event entry point, the event flags are in its address space.

6.2 Message buffers

The use of message buffers for inter-process communication was introduced in the design of the RC4000 operating system.[2] The SUE project[7] also used a message sending facility and the related device called a mailbox to achieve process synchronization. We introduce here a set of message buffer primitives which provide an efficient means of inter-process communication and synchronization.

A kernel pool of message buffers is provided, each of which may be up to a multiple of 7 times 16 words in size. Each message consists of a six-word header and the data being sent to the receiving process. The format of the message is specified in Fig. 4. The primitives available to a process consist of:

alocmsg (*nwords*)
queuem (*message*)
queuemn (*message*)
dequeuem (*process*)
dqtype (*process*)
messink (*message*)

To open a communication channel between two processes P1 and P2, P1 must allocate a message buffer using **alocmsg**, fill in the appropriate data in the message header and data areas and then send the message to process P2 using **queuem**. Efficiency is achieved by allowing P1 to send multiple messages before waiting for an acknowledgment (answer). The acknowledgment to these messages is returned in the same buffer by means of the **messink** primitive. The message buffer address space is freed up automatically if the message is an acknowledgment to an acknowledgment. Buffer space may also be freed explicitly by means of the **freemsg** primitive. When no answer is expected back from a process, the **queuemn** primitive is used.

Synchronization is achieved by putting the messages on P2's message input queue using the **link** word in the message header and sending P2 a message event flag. This will immediately invoke the scheduling of process P2 if it runs at a higher priority than P1. Process P1 is responsible for filling in the *from process* number, the *to process* number, the *type* and the *identifier* fields in the message header. The *type* field specifies which routine P2 must execute to process the message. A type of "−1" is reserved for acknowledgment messages to the original sender of the message. The status of the processed message is returned in the *status* field of the message header, a non-zero value indicating an error. The status of −1 is

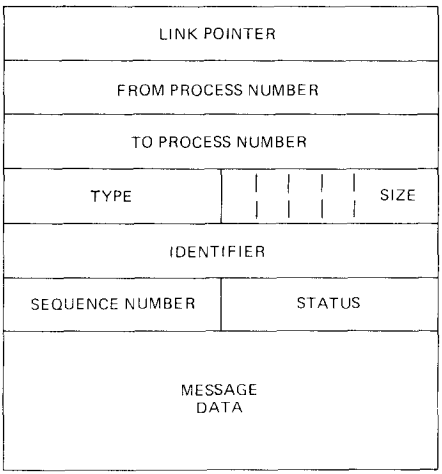

Fig. 4—Message format.

reserved for use by the system to indicate that process P2 does not exist or was terminated abnormally while processing the message. The *sequence number* field is used solely for debugging purposes. The *identifier* field may be planted by P1 to be used to identify and verify acknowledgment messages. This word is not modified by the system.

Process P2 achieves synchronization by waiting for a message. In general, a process may receive any message type from any process by means of the **dequeuem** primitive. However, P2 may request a message type by means of **dqtype** in order to process messages in a certain sequence for internal process management. In each case, the kernel primitive will return a success/fail condition. In the case of a fail return, P2 has the option of roadblocking to wait for a message event or of doing further processing and looking for an input message at a later time.

6.3 emt traps

The emulator trap (**emt**) instruction is used not only to implement the system primitives, but also to provide a mechanism by which a supervisor and kernel process can pass information. The supervisor process passes the process number of the kernel process with which it would like to communicate to the kernel. The kernel then dispatches to the kernel process through its **emt** entry point, passing the process number of the calling supervisor process and a pointer to an argument list. The kernel process will typically access data in the supervisor process address space by setting part of its virtual address space to overlap that of the supervisor. This method of communication is used mainly to pass characters from a time-sharing user to the kernel process which controls communications equipment.

6.4 Shared memory

Supervisor processes may share memory by means of named as well as unnamed segments. Segments may be shared on a supervisor as well as a user level. In both cases, pure code is shared as named segments. In the case of a time-sharing supervisor (described in Section VIII), a segment is shared for I/O buffers and file descriptors. A shared segment is also used to implement the concept of a *pipe*,[8] which is an inter-process channel used to communicate streams of data between related processes. At the user

level, related processes may share a segment for the efficient communication of a large quantity of data. For related processes, a parent process may set up a sharable segment in its address space and restrict the access permissions of all child processes to provide a means of protecting shared data. Facilities are also provided for sharing segments between unrelated supervisors and between kernel and supervisor processes.

6.5 Files

The file system has a hierarchical structure equivalent to the UNIX file system[8] and as such has certain protection keys (see Section VII). Most files have general read/write permissions and the contents are sharable between processes.

In some cases, the access permissions of the file may themselves serve as a means of communication. If a file is created with read/write permissions for the owner only, another process may not access this file. This is a means of making that file name unavailable to a second process.

6.6 Process ports

Knowing the identity of another process gives a process the ability to communicate with it. The identity of certain key processes must be known to all other processes at system startup time to enable communication. These globally known processes include the scheduler, the memory manager, the process manager, the file manager, and the swap device driver process. These comprise a sufficient set of known processes to start up new processes which may then communicate with the original set.

Device driver processes are created dynamically in the system. They are in fact created, loaded, and locked in memory upon opening a "device" file (see Section VII). The identity of the device driver process is returned by the process manager to the file manager which in turn may return the identity to the process which requested the opening of the "device" file. These processes are referred to as "external" processes by Brinch Hansen.[2]

The above process-communication primitives do not satisfy the requirements of communication between unrelated processes. For this reason the concept of process ports has been introduced. A process port is a globally known "device" (name) to which a process may attach itself in order to communicate with "unknown"

processes. A process may connect itself to a port, disconnect itself from a port, or obtain the identity of a process connected to a specific port. Once a process identifies itself globally by connecting itself to a port, other processes may communicate with it by sending messages to it through the port. The port thus serves as a two-way communication channel. It is a means of communication for processes which are not descendants of each other.

VII. FILE SYSTEM

The multi-environment as well as the real-time aspects of the MERT system requires that the file system structure be capable of handling many different types of requests. Time-sharing applications require that files be both dynamically allocatable and dynamically growable. Real-time applications require that files be large, and possibly contiguous; dynamic allocation and growth are usually not required.

For data base management systems, files may be very large, and it is often advantageous that files be stored in one contiguous area of secondary storage. Such large files are efficiently described by a file-map entry which consists of starting block number and number of consecutive blocks (a two-word extent). A further benefit of this allocation scheme is that file accesses require only one access to secondary storage. Another commonly used scheme, using indexed pointers to blocks of a file in a file-map entry, may require more than one access to secondary storage to read or write a block of a file. However, this latter organization is usually quite suitable for time-sharing applications. The disadvantage of using two-word extents in the file-map entry to describe a dynamic time-sharing file is that this may lead to secondary storage fragmentation. In practice, the efficient management of the in-core free extents reduces storage fragmentation significantly.

Three kinds of files are discernible to the user: ordinary disk files, directories, and special files. The directory structure is identical to the UNIX file system directory structure. Directories provide the mapping between the names of files and the files themselves and impose a hierarchical naming convention on the files. A directory entry contains only the name of the file and a file identifier which is essentially a pointer to the file-map entry (i-node) for that file. A file may have more than one link to it, thus enabling the sharing of files.

Special files in the MERT system are associated with each I/O

device. The opening of a special file causes the file manager to send a message to the process manager to create and load the appropriate device driver process and lock it in memory. Subsequent reads and writes to the file are translated into read/write messages to the corresponding I/O driver process by the file manager process.

In the case of ordinary files, the contents of a file are whatever the user puts in it. The file system process imposes no structure on the contents of the file.

The MERT file system distinguishes between contiguous files and other ordinary files. Contiguous files are described by one extent and the file blocks are not freed until the last link to the file is removed. Ordinary files may grow dynamically using up to 27 extents to describe their secondary storage allocation. To minimize fragmentation of the file system, a growing file is allocated 40 blocks at a time. Unused blocks are freed when the file is closed.

The list of free blocks of secondary storage is kept in memory as a list of the 64 largest extents of contiguous free blocks. Blocks for files are allocated and freed from this list using an algorithm which minimizes file system fragmentation. When freeing blocks, the blocks are merged into an existing entry in the free list if possible, or placed in an unused entry in the free list. Failing these, an entry in the free list which contains a smaller number of free blocks is replaced.

The entries which are being freed or allocated are also added to an update list in memory. These update entries are used to update a bit map which resides on secondary storage. If the in-core free list should become exhausted, the bit map is consulted to re-create the 64 largest entries of contiguous free blocks. The nature of the file system and the techniques used to reduce file system fragmentation ensure that this is a very rare occurrence.

Very active file systems consisting of many small time-sharing files may be compacted periodically by a utility program to minimize file system fragmentation still further. File system storage fragmentation actually only becomes a problem when a file is unable to grow dynamically having used up all 27 extents in its file map entry. Normal time-sharing files do not approach this condition.

Communication with the file system process is achieved entirely by means of messages. The file manager can handle 25 different types of messages. The file manager is a kernel process using both I and D space. It is structured as a task manager controlling a number of parallel cooperating tasks which operate on a common data base and which are not individually preemptible. Each task acts

on behalf of one incoming message and has a private data area as well as a common data area. The parallel nature of the file manager ensures efficient handling of the file system messages. The mode of communication, message buffers, also guarantees that other processes need not know the details of the structure of the file system. Changes in the file system structure are easily implemented without affecting other process structures.

VIII. A TIME-SHARING SUPERVISOR

The first supervisor process developed for the MERT system was a time-sharing supervisor logically equivalent to the UNIX time-sharing system.[8] The UNIX supervisor process was implemented using messages to communicate with the file system manager. This makes the UNIX supervisor completely independent of the file system structure. Changes and additions can then be made to the file system process as well as the file system structure on secondary storage without affecting the operation of the UNIX supervisor.

The structure of the system requires that there be an independent UNIX process for each user who "logs in." In fact, a UNIX process is started up when a "carrier-on" transition is detected on a line which is capable of starting up a user.

For efficiency purposes, the code of the UNIX supervisor is shared among all processes running in the UNIX system environment. Each supervisor has a private data segment for maintaining the process stack and hence the state of the process. For purposes of communication, one large data segment is shared among all UNIX processes. This data segment contains a set of shared buffers used for system side buffering and a set of shared file descriptors which define the files that are currently open.

The sharing of this common data segment does introduce the problem of critical regions, i.e., regions during which common resources are allocated and freed. The real-time nature of the system means that a process could be preempted even while running in a critical region. To ensure that this does not occur, it is necessary to inhibit preemption during a critical region and then permit preemption again upon exiting from the critical region. This also guarantees that the delivery of an event at a higher hardware priority will not cause a critical region to be re-entered. Note that a semaphore implemented at the supervisor level cannot prevent such reentry unless events are inhibited during the setting of the semaphore.

The UNIX supervisor makes use of all the communication primitives discussed previously. Messages are used to communicate with the file system process. Events and shared memory are used to communicate with other UNIX processes. Communication with character device driver processes is by means of **emt** traps. Files are used to share information among processes. Process ports are used in the implementation of an error logger process to collect error messages from the various I/O device driver processes.

The entire code for the UNIX supervisor process (excluding the file system, drivers, etc.) consists of 8K words. This includes all the standard UNIX system routines as well as the many extra system routines which have been added to the MERT/UNIX supervisor. The extra system routines make use of the unique features available under MERT. These include the ability to:

(*i*) Create a new environment.
(*ii*) Send and receive messages.
(*iii*) Send and receive events.
(*iv*) Set up shared segments.
(*v*) Invoke new file system primitives such as allocate contiguous files.
(*vi*) Set up and communicate with process ports.
(*vii*) Initiate physical and asynchronous I/O.

All memory management and process scheduling functions are performed by the kernel.

IX. REAL-TIME ASPECTS

Several features of the MERT architecture make it a sound base on which to build real-time operating systems. The kernel provides the primitives needed to construct a system of cooperating, independent processes, each of which is designed to handle one aspect of the larger real-time problem. The processes can be arranged in levels of decreasing privilege depending on the response requirements. Kernel processes are capable of responding to interrupts within 100 microseconds, nonswap supervisor processes can respond within a few milliseconds, and swap processes can respond in hundreds of milliseconds. Shared segments can be used to pass data between the levels and to insure that the most up-to-date data are always available. This is sufficient to solve the data integrity problem discussed by Sorenson.[3]

The system provides a low-resolution interval timer which can be

used to generate events at any multiple of 1/60th of a second up to 65535. This is used to stimulate processes which update data bases at regular intervals or time I/O devices. Since the timer event is an interrupt, supervisor processes can use it to subdivide a time slice to do internal scheduling.

The preemptive priority scheduler and the control over which processes are swappable allow the system designer to specify the order in which tasks are processed. Since the file manager is an independent process driven by messages, all processes can communicate directly with it, providing a limited amount of device independence. The ability to store a file on a contiguous area of secondary storage is aimed at minimizing access time. Finally, the availability of a sophisticated time-sharing system in the same machine as the real-time operating system provides powerful tools which can be exploited in designing the man-machine interface to the real-time processes.

X. PROCESS DEBUGGING

One of the most useful features of the system is the ability to carry on system development while users are logged in. New I/O drivers have been debugged and experiments with new versions of the time-sharing supervisor have been performed without adversely affecting the user community.

Three aspects of the system make this possible:

(*i*) Processes can be loaded dynamically.
(*ii*) Snapshot dumps of the process can be made using the time-sharing supervisor.
(*iii*) Processes are gracefully removed from the system and a core dump produced on the occurrence of a "break point trap."

As an example, we recently interfaced a PDP-11/20 to our system using an inter-processor DMA (direct memory access) link. During the debugging of the software, the two machines would often get out of phase leading to a breakdown in the communication channel. When this occurred, a dump of the process handling the PDP-11/45 end of the link was produced, a core image of the PDP-11/20 was transmitted to the PDP-11/45, and the two images were analyzed using a symbolic debugger running under the time-sharing supervisor. When the problem was fixed, a new version of the kernel-mode link process was created, loaded, and tested. Turnaround time in this mode of operation is measured in seconds or minutes.

XI. MERT-BASED PROJECTS

A number of PDP-11-based minicomputer systems have taken advantage of the MERT system features to meet their system specifications. The features which various projects have found useful include:

Contiguous files.
Asynchronous input/output.
Interprocess communication facilities.
Large virtual address space.
Public libraries.
Real-time processes.
Dynamic debugging features.

Most projects have had experience with or were using the UNIX time-sharing system. Thus the path of least resistance dictated the use of the MERT/UNIX system calls which were added to the original UNIX system calls to take advantage of the MERT system features. The next step was to write a special-purpose supervisor process to give the programmer more control in an environment better suited to the application than the UNIX time-sharing system environment. Almost all projects used the dynamic debugging features of the MERT system to test out new supervisor and new kernel processes.

To take advantage of all of the system calls which were added to the MERT/UNIX supervisor, a modified command interpreter, i.e. an extended shell, was written.[9] The user of this shell is able to make use of all of the MERT system interprocess communication facilities without having to know the details of the arguments required. A number of interesting new supervisor processes were written to run on the MERT system. One of the user environments emulated was the RSX-11 system, a DEC PDP-11 operating system. This required the design of an interface to the MERT file manager process. The new supervisor process provided the same interface to the user as that seen by the RSX-11 user on a dedicated machine. This offered the user access to all language subsystems and utilities provided by RSX-11 itself, most notably the Fortran IV compiler. Another supervisor process written was one which provided an interface to a user on a remote machine (SEL86) to the MERT file system. Here the supervisor process communicates with the MERT file manager process by means of messages much as the MERT/UNIX supervisor does. A special kernel device driver process controls the hardware channels between the SEL86 and the PDP-11/45 computers. The UNIX programming environment in the MERT system is used both for

PDP-11 programming and for preparing files and programs to be used on the SEL86 machine.

XII. PROTECTION/PERFORMANCE TRADE-OFFS

We summarize here the results of our experience with the MERT system as designers, implementers, and users. Some of the features added or subtracted from the MERT system have been the result of feedback from various users. We pay particular attention to various aspects of the system design concerning trade-offs made between efficiency and protection. The advantages of the system architecture as well as its disadvantages are discussed.

Each major design decision is discussed with respect to performance versus protection. By protection, we mean protection against inadvertent bugs and the resulting corruption, not protection against security breaches. In general, for the sake of a more efficient system, protection has been sacrificed when it was believed that this extra protection would degrade system performance significantly. In most cases, the system is used in dedicated applications where some protection could be sacrificed. Maximum protection is provided mainly by separating the various functions into layers, putting each function at the highest possible level, according to the access privileges required. All processes were written in the high-level language, C.[10] This forced some structure in the processes. C controls access to the stack pointer and program counter and automatically saves the general-purpose registers in a subroutine call. This provides some protection which is helpful in confining the access of a program or process.

12.1 Hardware

The hardware of the PDP-11 computers permits a distinction to be made between kernel processes and supervisor processes. Kernel processes have direct access to the kernel-mode address space and may use all privileged instructions. For efficiency reasons, one base register always points to the complete I/O page. This is 4K words of the address space of the PDP-11 computer which is devoted to device addresses. It is not possible to limit access to only the device registers required for a particular device driver. The virtual address space is limited to 16-bit addressing. This presents a limitation to some large processes.

12.2 Kernel

The number of base registers provided by the PDP-11 segmentation unit is a restriction in the kernel. The use of I and D space separation is necessitated to provide a reasonable number (16) of segments. Some degree of protection is provided for the sensitive kernel system tables by the address space separation, since the kernel drivers do not use I/D space separation in general. Such kernel processes do not have access to sensitive system data in kernel D space.

12.3 Kernel process

Most kernel-mode processes use only kernel I space. This prohibits access to system segment tables and to kernel code procedures. However, access to message buffers, dispatcher control tables, and the I/O page is permitted. A kernel process is the most privileged of all processes which the user can load into a running system. The stack used by a kernel process is the same as that used by kernel procedures.

To provide complete security in the kernel would require that each process use its own stack area and that access to all base registers other than those required by the process be turned off. The time to set up a kernel process would become prohibitive. Since control is most often given to a kernel process by means of an interrupt, the interrupt overhead would become intolerable, making it more difficult to guarantee real-time response.

In actual practice, the corruption of the kernel by kernel processes has occurred very infrequently and then only when debugging a new kernel process. Fatal errors were seldom caused by the modification of data outside the process's virtual address range. Most errors were timing-dependent errors which would not have been detected even with better protection mechanisms. Hence we conclude that the degree of protection provided for kernel processes in dedicated systems is sufficient without degrading system performance. The only extra overhead for dispatching to a kernel process is that of saving and restoring some base registers and saving the current stack pointer.

12.4 Supervisor process

Supervisor processes do not have direct access to the segments of

other processes, kernel or supervisor. Therefore, it is possible to restrict the impact of these processes on the rest of the system by means of careful checking in the kernel procedures. All communication with other processes must go through the kernel. Of course, one pays a price for this protection since all supervisor base registers must have the appropriate access permissions set when a supervisor process is scheduled. Message traffic overhead is higher than for kernel processes.

For protection reasons, capabilities were added to the system. This adds extra overhead for each message to the file manager, since each capability must be validated by the file manager. An alternate implementation of capabilities which reduces overhead at the cost of some protection is discussed in a later section.

12.5 Message buffers

System message buffers are maintained in kernel address space. These buffers are corruptible by a kernel process. The only way to protect against corruption completely would be to make a kernel emt call to copy the message from the process's virtual address space to the kernel buffer pool. For efficiency reasons, this was not done.

For a supervisor process, the copying of a message from the supervisor's address space to the kernel message buffer pool area is necessary. This increases message traffic overhead for supervisor to kernel or supervisor to supervisor transfers. The overhead for sending and receiving a message between kernel processes amounts to 300 microseconds. whereas for supervisor processes the overhead is of the order of 800 microseconds (on a PDP-11/45 computer without cache memory).

12.6 File manager process

The file manager process is implemented as a kernel-mode process with I and D space separated to obtain enough virtual address space. In the early implementation stage of the MERT system, the file manager was a supervisor process, but the heavy traffic to the file manager process induced many context changes and contributed significantly to system overhead. Implementation of the file manager process as a kernel-mode process improved system throughput by an average of about 25 percent. Again, this was a protection/efficiency trade-off. Protection is sacrificed since the file

manager process has access to all system code and data. In practice, it has not proven to be difficult to limit the access of the file manager to its intended virtual address space. Making the file manager a separate process has made it easy to implement independent processes which communicate with the file manager. The file manager is the only process with knowledge of the detailed structure of the file system. To prevent corruption of the file system, all incoming messages must be carefully validated. This includes careful checking of each capability specified in the message. This is a source of some system overhead which would not exist if the file system were tightly coupled with a supervisor process. However, this separation of function has proven very helpful in implementing new supervisors.

12.7 Process manager

The process manager is implemented as a swappable supervisor process. Its primary function is to create and start up new processes and handle their termination. An example is the loading of the kernel driver process for the magnetic tape drive. This is an infrequent occurrence, and thus the time penalty to bring in the process manager is tolerable. Other more frequent creations and deletions of processes associated with the UNIX system forking of processes is handled by the system scheduler process. In the early stages of implementation of the MERT system, the creation and deletion of all processes required the intervention of the process manager. This required the loading of the process manager in each case and added significantly to the overhead of creating and deleting processes.

12.8 Response comparisons

The fact that a "UNIX-like" environment was implemented as one environment under the MERT kernel gives us a unique opportunity to compare the overall response of a system running as a general-purpose development system to that of a system running a dedicated UNIX time-sharing system on the same hardware. This gives us a means of determining what system overhead is introduced by using messages as a basic means of inter-process communication. Application programs which take advantage of the UNIX file system structure give better response in a dedicated UNIX time-sharing system, whereas those which take advantage of the MERT file system

structure give a better response under the MERT system. Compute-bound tasks respond in the same time under both systems. It is only where there is substantial system interaction that the structure of the MERT system introduces extra system overhead, which is not present in a dedicated UNIX system. Comparisons of the amount of time spent in the kernel and supervisor modes using synthetic jobs indicate that the MERT system requires from 5 to 50 percent more system time for the more heavily used system calls. This translates to an increase of 5 to 10 percent in elapsed time for the completion of a job stream consisting of compilation, assembly, and link-edit. We believe that this overhead is a small price to pay to achieve a well-structured operating system with the ability to support customized applications. The structure of the system provides a basis for doing further operating system research.

XIII. DESIGN DECISIONS IN RETROSPECT

A number of design decisions were made in the MERT system which had no major impact on efficiency or protection. However, many of these impacted the interface presented to the user of the system. The pros and cons of these decisions are discussed here.

13.1 File system

The first file system for the MERT system was designed for real-time applications. For that, it is well-suited. For those applications which require the collection of data at a high rate, the use of contiguous files and asynchronous I/O proved quite adequate. However, the number of applications which required contiguous files was not overwhelming. For those applications which used the MERT system as a development system as well, the allocation of files by extents is not optimal, although adequate. The number of files which exhausted their 27 extents was small indeed. Also the need for compaction of file systems due to fragmentation was not as great as might have been expected and seems not to have posed any problems. The root file system very rarely needs to be compacted due to the nature of file system activity on it.

The file manager process uses multi-tasking to increase its throughput. This has added another degree of parallelism to the system, but on the other hand has also been the source of many hard-to-find timing problems.

The use of 16-bit block numbers is a shortcoming in the file system with the advent of larger and larger disks. However, this has been rectified in a new 32-bit file system which has features that make it more suitable for small time-sharing files and yet allows the allocation of large contiguous files. Compaction of this file system is not required.

13.2 Error logging

A special port process to collect error messages has proven to be very useful for tracking down problems with the peripheral devices. Sending messages rather than printing diagnostics out at the control terminal minimizes impact on real-time response. One drawback of this means of reporting errors is that the user is not told of the occurrence of an error immediately at his terminal unless the error is unrecoverable. He must examine the error logger file for actual error indications.

13.3 Process ports

Process ports were implemented as a means of enabling communication among unrelated processes. This has proven to be an easy-to-use mechanism for functions such as the error logger. Other uses have been made of it, such as a centralized data base manager. The nature of the implementation of ports requires that the port numbers be assigned by some convention agreed upon by users of ports. Probably a better implementation of ports would have been to use named ports, i.e., to refer to ports by name rather than by number. The number then is not dependent on any user-assigned scheme.

13.4 Shared memory

Shared memory allows the access to a common piece of memory by more than one process. The use of named segments to implement sharing enables two or more processes to pass a large amount of data between them without actually copying any of the data. The PDP-11 memory management unit and the 16-bit virtual address space are limitations imposed on shared memory. Only up to 16 segments may be in a process' address space at any one time. Sometimes it would be desirable to limit access to less than a total

logical segment. The implementation chosen in the MERT system does not allow this.

13.5 Public libraries

Public libraries are used in the MERT system at all levels: kernel, supervisor, and user. The use of public libraries at the kernel level has allowed device drivers to share a common set of routines. At the user level, many programs have made use of public libraries to make a substantial savings in total memory requirements. The initial implementation of public libraries required that when a public library was reformed, all programs which referenced it had to be link-edited again to make the appropriate connection to subroutine entry points in the public library. The current implementation makes use of transfer vectors at the beginning of the public library through which subroutine transfers are performed. Thus, if no new entry points are added when a public library is formed again, the link-edit of all programs which use it becomes unnecessary. This has proven to be very helpful for maintaining a set of user programs which share public libraries. It has proven to be convenient also for making minor changes to the system library when new subroutines are not added. This makes the re-forming of all device drivers unnecessary each time a minor change is made to a system library.

13.6 Real-time capabilities

The real-time capabilities of the MERT system are determined in part by the mode of the process running, i.e., kernel or supervisor. Control is given to a kernel mode process by an interrupt or an event. Time-out events may be used effectively to guarantee repetitive scheduling of a process. The response of a kernel process is limited by the occurrence of high priority interrupts, and therefore can only be guaranteed for the highest priority process. A supervisor process' scheduling priority can be made high by making it a nonswap process and giving it a high software priority. A response of the order of a few milliseconds can then be obtained. The scheduler uses preemption to achieve this. One aspect missing from the scheduler is deadline scheduling. Thus, it cannot be guaranteed that a task will finish by a certain time. The requirement for preemption has added another degree of complexity to the scheduler and of necessity adds overhead in dispatching to a process. Preemption has also complicated the handling of critical regions. It is

necessary to raise the hardware priority around a critical region. This is difficult to do in a supervisor, since it requires making a kernel emt call, adding to response time. Shifting of segments in memory also adds to the response time which can be guaranteed.

13.7 Debugging features

Overall, the debugging features provided by the MERT system have proven to be adequate. The kernel debugger has proven useful in looking at the history of events in the kernel and examining the detailed state of the system both after a crash and while the system is running. In retrospect, it would have been helpful to have some more tools in this area to examine structures according to named elements rather than by offsets.

The dynamic loading, dumping, and then debugging of processes, both kernel and supervisor, on a running system have been helpful in achieving fast debugging turnaround. While post-mortem debugging is useful, interactive debugging would eliminate the need to introduce traces and local event logging to supervisor and kernel processes as debugging aids. One danger of planting break-point traps at arbitrary points in the UNIX supervisor has been that of planting them in a critical region in which a resource is allocated. The resource may not be freed up properly and other processes may hang waiting for the resource to be freed up.

13.8 Memory manager

The memory manager is a separate kernel process and handles incoming requests as messages in a fairly sequential manner. One thing it does do in parallel, however, is the loading of the next process to be run while the current one is running. In certain cases, the memory manager can act as a bottleneck in the system throughput. This can also have serious impact on real-time response in a heavily loaded system.

13.9 Scheduler

The scheduler in the MERT system is another separate kernel process. One improvement which could be made in this area is to separate mechanism from policy. The fact that the scheduler and memory manager are separate processes has system-wide impact in that the scheduler cannot always tell which process is the best one to

run based on which one has the most segments in memory. The memory manager does not tend to throw out segments based on which process owns it but rather on usage statistics.

13.10 Messages

Messages have proven to be an effective means of communication between processes. At the lowest level, they have been helpful in separating functions into processes and of making these processes modular and independent. It has made things like error logging easy to implement. Communication with the file manager process by means of messages has removed the dependency of supervisor processes on file system structures. In fact, a number of different file managers have been written to run using the identical "UNIX-like" supervisor. The UNIX file manager was brought up to run in place of the original MERT file manager without any impact on the supervisor processes. Messages at a higher level have not always been easy to deal with. It is difficult to prevent a number of user processes from swamping the kernel message buffer pool and thereby impacting system response.

The MERT system implementation of messages solves the problem of many processes sending to one process quite effectively. However, the reverse problem of one process sending to many processes (i.e., many servers) is not handled efficiently at all.

13.11 Firewalls

Having separate processes for separate functions has modularized the design of the system. It has eased the writing of new processes but required them to obey a new set of rules. To ensure that processes obey these rules requires an amount of checking which would not be necessary if processes were merged in one address space. This has been especially true of the file manager where corruption of data is very crucial, as it can very quickly spread as a cancer in the system.

XIV. PORTABILITY

Recently a great deal of interest has been expressed in porting complete operating systems and associated user programs to hardware configurations other than the DEC 16-bit PDP-11 computer.

We discuss here some of the hardware characteristics on which the MERT system depends and the impact of these on the software.

14.1 Hardware considerations

At the time that we designed the MERT operating system (circa 1973), the DEC PDP-11/45 processor with a memory management unit allowing the addressing of up to 124K words of memory was a new system. Moreover, the memory management unit was rather sophisticated for minicomputers at that time, since it supported three address modes: kernel, supervisor, and user. It also supported two address spaces per mode, instruction and data. This enables a mode to address up to 64K words in its address space. Two address modes are generally sufficient for operating systems which provide one environment to the user. To support multi-environments, three modes are required (or at least are desirable), one of which provides the various environments to the user. We decided to make use of this feature. The separation of instruction and data address space provides more address space for a process. It also provides a greater number of segments per user and allows some degree of protection. This was used in the kernel where a large number of separate pieces of code and data need to be referenced concurrently. The protection provided is made use of in kernel processes which need very few base registers and do not need access to very much data; in fact, the less the better. Thus a kernel process is not allowed to run with instruction and data space separated so as to protect sensitive system tables.

The third unique feature of the PDP-11/45 computer is that it has a programmable interrupt register (PIR). This enables the system to trigger a software interrupt at one of seven hardware priority levels. The interrupt goes off when the processor starts to run at less than the specified priority. This is used heavily in the MERT system scheduler process and by kernel system routines which trigger various events to occur at specified hardware priorities. It is not sufficient to depend on the line clock for a preemptive scheduler to guarantee real-time response.

We have identified here three unique features of the PDP-11/45 processor (and the PDP-11/70) which have been heavily used in the MERT system. These features are identified as unique in that a general class of minicomputers does not have all of these features, although some may have one or more. They are also identified as unique in that the UNIX operating system has not made critical use

of them. Therefore, the portability of the UNIX system is not impacted by them. For the portability of the MERT system, three or more address modes, a large number of segments (at least eight) per address mode, and a programmed interrupt register are highly desirable.

14.2 Software considerations

Currently, most of the MERT system is written in C. This includes all device driver processes, the scheduler, the file manager, the process manager, and the UNIX supervisor. Most of the basic kernel, including the memory manager process, is written in PDP-11 assembly language. This portion is of course not portable to other machines. Recently, a portable C compiler has been written for various machines, both 16-bit and 32-bit machines, the two classes of minicomputers which are of general interest for portability purposes. These include the PDP-11 and the Interdata 8/32 machines.

The UNIX system has been ported to the Interdata 8/32 machine; this includes all user programs as well as the operating system itself.[11] Thus, if the portability of the MERT system to the Interdata 32-bit machine were to be considered, all user programs have already been ported. The main pieces of software which have to be written in portable format include all device drivers, the scheduler, the process manager, the file manager and the UNIX supervisor. Of these, only the device drivers have machine dependencies and need substantial rewriting. The file manager, being a kernel process, has some machine-dependent code. The bulk of the software which must be rewritten is in the kernel itself, being substantially written in PDP-11 assembly language. Also, all library interface routines must be rewritten. Many of the calling sequences for library routines have to be reworked, since arguments are passed specifically as 16-bit integers. Some sizes, especially of segments, are specified in terms of 16-bit words. For portability reasons, all sizes must be treated in terms of 8-bit bytes.

XV. REFLECTIONS

In designing any system, one must make a number of crucial decisions and abide by them in order to come up with a complete and workable system. We have made a number of these, some of which have been enumerated and discussed in the above sections. Upon reflecting on the results and getting feedback from users of the

MERT system, we have come up with a number of design decisions which could probably have been made differently from what was actually done. Users have pushed the system in directions which we never considered, finding holes in the design and also some bugs which were never exercised in the original MERT system.

15.1 Capabilities

Capabilities were implemented in the system as a result of the experience of one user in writing a new supervisor process which sent messages to the file manager. There were two major deficiencies. The first had to do with protection. Under the old design (without capabilities), it was possible to ignore the protection bits. Upon reading/writing a file, no check was made of the protection bits. As long as a file was open, any action could be taken on the file, reading or writing; this included directories. With the addition of capabilities, when a file is opened, the capability is put in the user's PCB. The capability includes the entry number in the file manager tables, protection bits, and a usage count. The capability is put in a message to the file manager by the kernel when a request is made to read/write a file. These three quantities are checked by the file manager. A capability must be satisfactorily validated before an access can be made to a file. This provides the degree of protection desired.

The second deficiency of the file manager had to do with the maintenance of an up-to-date set of open file tables. If a process is abnormally terminated, i.e., terminated by the scheduler without being given a chance to clean up, the process may not have been able to close all its files. This would typically occur when a breakpoint trap was planted in an experimental version of the UNIX supervisor. The fact that no table is maintained in a central place with a list of all files open by each process caused file tables to get out of synchronization. Capabilities provide such a central table to the process manager and the memory manager. Thus when an abnormal termination is triggered on a process, the memory manager can access the process PCB and take down the capabilities one by one, going through the capability list in the PCB, sending close messages to the file manager. This provides a clean technique for maintaining open counts on files in the file manager tables.

In retrospect, the implementation of capabilities in the MERT system was probably carried to an extreme, i.e. not in keeping with the other protection/efficiency trade-offs made. The trade-off was made

in favor of protection rather than efficiency, in this case. The current implementation of capabilities is expensive in that extra messages are generated in opening and closing files. For instance, in closing a file, a **close** message is sent to the file manager; this in turn generates a message to the capability manager (i.e., the memory manager) to take down the capability from the PCB of the process which sent the **close** message. The asynchronous message is necessary since the memory manager process must bring the PCB into memory to take down the capability if the PCB is not already in memory.

A more efficient means of achieving the same result would be to maintain this list of capabilities in the supervisor address space with general read/write permissions with a pointer to the capability list maintained in the PCB. It would then be the supervisor's responsibility to fill in the capability when sending a message to the file manager and to take down the capability when closing the file. This requires no extra message traffic overhead as compared to the original implementation without capabilities. Upon abnormal termination, the memory manager could still go through the capability list to take down all capabilities by sending **close** file messages to the file manager. Protection is still achieved by the encoded capability. Efficiency is maintained by eliminating extra messages to the memory manager. This proposed implementation also has the added benefit that it can be implemented for kernel processes in the same manner, i.e., using a pointer to a capability list in the kernel process header.

15.2 Global system buffers

In the current implementation of the MERT system, each process maintains its own set of system buffers. The file manager provides its own set of buffers, used entirely for file mapping functions (e.g., superblocks for mounted file systems, i-nodes, and directories). The UNIX supervisor provides its own set of buffers for use by all UNIX processes. These buffers are used almost exclusively for the contents of files. However, it is possible for a file to be the image of a complete file system, in which case a buffer may actually contain the contents of a directory or i-node. This means there may be more than one copy of a given disk block in memory simultaneously. Because of the orthogonal nature of the uses of buffers in the UNIX system and the file manager, this duplication hardly ever

occurs and does not pose a serious problem. Within the UNIX system itself, all buffers are shared in a common data segment.

However, if one wishes to implement other supervisors and these supervisor processes share a common file system with the UNIX supervisor, it becomes quite possible that more than one copy of some disk blocks exists in memory. This presents a problem for concurrent updates.

An alternate method of implementation of buffers would have been to make use of named segments to map buffers into a globally accessible buffer pool. The allocation and deallocation of buffers would then become a kernel function, and this would guarantee that each disk block would have a unique copy in memory. If the MERT system had allowed protection on sections of a segment, then system buffers could have been implemented as one big buffer segment broken up into protectable 512-byte sections. The system overhead in this implementation probably would have been no greater than the current implementation. Each time a buffer is allocated and released, a kernel **emt** call would be necessary. However, even the present implementation requires two short-duration **emt** calls to prevent process preemption during a critical region in the UNIX supervisor both during the allocation and releasing of a buffer.

15.3 Diagnostics

One of the shortcomings of the MERT system has been the lack of system diagnostics printed out at the control console reporting system troubles. The UNIX system provides diagnostic print-outs at the control console upon detection of system inconsistencies or the exhaustion of crucial system resources such as file table entries, i-node table entries, or disk free blocks. Device errors are also reported at the control console. In the MERT system, device errors are permanently recorded on an error logger file. One reason for not providing diagnostic print-out at the control console is that the print-out impacts real-time response.

The lack of diagnostic messages has been particularly noticeable in the file system manager and in the basic kernel when resources are exhausted. Providing diagnostic messages in the system requires the use of some address space in each process making use of diagnostic messages; this would require duplication of the basic printing routines in the kernel, the file manager, and any other process which wished to report diagnostics or the inclusion of the printing routines in the system library. A possible solution would have been to make

use of the MERT message facilities to send diagnostic data to a central process connected to a port to print out all diagnostics both on the control console and into a file for later analysis. Using this technique, it would also be possible to send diagnostic messages directly to the user's terminal which caused the diagnostic condition to occur. The diagnostic logger process would be analogous to the error logger process.

15.4 Scheduler process

The current MERT scheduler is a separate kernel system process which implements both the mechanism and the policy of system-wide scheduling. It would be more flexible to implement only the mechanism in the kernel process and let the policy be separated from this mechanism in other user-written processes.

XVI. ACKNOWLEDGMENTS

Some of the concepts incorporated in the kernel were developed in a previous design and implementation of an operating system kernel by C. S. Roberts and one of the authors (H. Lycklama). The authors are pleased to acknowledge many fruitful discussions with Mr. Roberts during the design stage of the current operating system.

We are grateful to the first users of the MERT system for their initial support and encouragement. These include: S. L. Arnold, W. A. Burnette, L. L. Hamilton, J. E. Laur, J. J. Molinelli, R. W. Peterson, M. A. Pilla, and T. F. Tabloski. They made suggestions for additions and improvements, many of which were incorporated into the MERT system and make the MERT system what it is today.

The current MERT system has benefited substantially from the documentation and debugging efforts of E. A. Loikits, G. W. R. Luderer, and T. M. Raleigh.

REFERENCES

1. E. W. Dijkstra, "The Structure of the THE Multiprogramming System," Commun. Assn. Comp. Mach., *11* (May 1968), p. 341.
2. P. Brinch Hansen, "The Nucleus of a Multiprogramming System," Commun. Assn. Comp. Mach., *13* (April 1970), p. 238.
3. P. G. Sorenson, "Interprocess Communication in Real-Time Systems," Proc. Fourth ACM Symp. on Operating System Principles (October 1973), pp. 1-7.
4. Digital Equipment Corporation, *PDP-11/45 Processor Handbook.* 1971.
5. W. A. Wulf, "HYDRA — A Kernel Protection System," Proc. AFIPS NCC, *43* (1974), pp. 998-999.

6. D. J. Frailey, "DSOS — A Skeletal, Real-Time, Minicomputer Operating System," Software — Practice and Experience, *5* (1975), pp. 5-18.
7. K. C. Sevcik, J. W. Atwood, M. S. Grushcow, R. C. Hold, J. J. Horning, and D. Tsichritzis, "Project SUE as a Learning Experience," Proc. AFIPS FJCC, *41, Pt. 1* (1972), pp. 331-339.
8. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
9. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," B.S.T.J., this issue, pp. 1971-1990.
10. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," B.S.T.J., this issue, pp. 1991-2019.
11. S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of C Programs and the UNIX System," B.S.T.J., this issue, pp. 2021-2048.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

UNIX on a Microprocessor

By H. LYCKLAMA
(Manuscript received December 5, 1977)

The decrease in the cost of computer hardware, brought about by the advent of the microprocessor and inexpensive solid-state memory, has brought the personal computer system to reality. Although the cost of software development shows no sign of decreasing soon, the fact that a large amount of software has been developed for the UNIX time-sharing system in the high-level language, C, makes much of this software portable to another processor with rather limited hardware in comparison. A single-user UNIX system has been developed for the DEC LSI-11 microprocessor using 20K words of primary memory and floppy disks for secondary storage. By preserving the user-system interface of the UNIX system, it is possible to run almost all of the standard UNIX languages and subsystems on this single-user version of the UNIX system.*

A background process as well as foreground processes may be run. The file system is "UNIX-like," but has provisions for dealing with contiguous files. Subroutines have been written to interface to the file system on the floppy disks. Asynchronous read/write routines are also available to the user.

The LSI-UNIX system (LSX) has appeal as a stand-alone system for dedicated applications, as well as many potential uses as an intelligent terminal system.

I. INTRODUCTION

The UNIX operating system[1] has enjoyed wide acceptance as a powerful, general-purpose time-sharing system. It supports a large

* UNIX is a trademark of Bell Laboratories.

variety of languages and subsystems. It runs on the Digital Equipment Corporation PDP-11/40, 11/45, and 11/70 computers, all 16-bit word machines that have a memory management unit which makes multiprogramming easy to support. The UNIX system is written in the system programming language, C,[2] in which most user programs and subsystems are written. Other languages and subsystems supported include Basic, Fortran, Snobol, TMG, and Yacc (a compiler-compiler). The file system is a general hierarchical structure supporting device independence.

With the advent of the DEC LSI-11 microprocessor[3] it has become desirable to transport to this machine as much as possible of the software developed for the UNIX system. One of the biggest problems faced is the lack of a memory management unit, which limits the total address space of both system and user to 28K words. The challenge, then, is to reduce the 20K-word, original UNIX operating system to 8K words and yet maintain a useful operating system. This limits the number of device drivers as well as the system functions that can be supported. The secondary storage used is floppy disks (diskettes). The operating system was written in the C language and provides most of the capabilities of the standard UNIX operating system. The system occupies 8K words in the lower part of memory, leaving up to 20K words for a user program. This configuration permits most of the UNIX user programs to run on the LSI-11 microcomputer. The operating system (LSX) allows a background process as well as foreground processes.

The fact that a minimum system can be configured for about $6000 makes the LSX system an attractive stand-alone system for dedicated applications such as control of special hardware. The system also has appeal as an intelligent terminal and for applications that require a secure and private data base. In fact, this is a personal computer system with almost all the functions of the standard UNIX time-sharing system.

This paper describes some of the objectives of the LSX system as well as some of its more important features. Its capabilities are compared with the powerful UNIX time-sharing system which runs on the PDP-11/40, 11/45, and 11/70 computers,[4] where appropriate. A summary and some thoughts on future directions are also presented.

II. WHY UNIX ON A MICROPROCESSOR?

Why develop a microprocessor-based UNIX system? The

increasing trend to microprocessors and the proliferation of intelligent terminals make it desirable to harness the UNIX software into an inexpensive microcomputer and give the user a personal computer system. A number of factors must be considered in doing this:

(*i*) Cost of hardware.
(*ii*) Cost of software.
(*iii*) UNIX software base.
(*iv*) Size of system.

The hardware costs of a computer system have come down dramatically over the last few years (even over the past few months). This trend is likely to continue in the foreseeable future. Microprocessors on a chip are a reality. The cost of primary memory (e.g., dynamic MOS memory) is decreasing rapidly as 4-kb chips are being replaced by 16-kb chips. A large amount of expertise exists in PDP-11 hardware interfacing. The similarity of the Q-bus of the LSI-11 microcomputer to the UNIBUS of other members of the PDP-11 family of computers makes this expertise available.

Software development costs continue to increase, since the development of new software is so labor-intensive, making it difficult to estimate the cost of writing a particular software application program. Until automatic program writing techniques become better understood and used, this trend is not likely to be turned around any time soon. Thus it becomes imperative to take advantage of as much software that has already been written as possible, including the tremendous amount of software that has already been written to run under the UNIX operating system. The operating system developed for the LSI-11 microcomputer supports most of the UNIX user programs which run under UNIX time-sharing, even though LSX is a single-user system. Thus most of the software for the system is already available, minimizing the cost of software development.

With the advent of some powerful microprocessors, the sizes of some computer systems have shrunk correspondingly. Small secondary storage units (floppy disks) are also becoming increasingly popular. In particular, DEC is marketing the LSI-11 microcomputer, which is a 16-bit word machine with an instruction set compatible with the PDP-11 family of computers. It is conceivable that in the next five years or so the power of a minicomputer system will be available in a microcomputer. It will become possible to allow a user to have a dedicated microcomputer rather than a part of a

minicomputer time-sharing system. LSX is a step in this direction. It will give the user a cost-effective interactive and powerful computer system with a known response time to given requests, since the machine is not time-shared. A dedicated, one-user system can be made available to guarantee "instantaneous" response to requests of a user. There are no unpredictable time-sharing delays to deal with. The system has applications in areas where security is important. A user can gain access to the system only in the room in which the system resides. It is thus possible to limit access to a user's data.

Local text-editing and text-processing features are now available. Other features can be added easily. Interfaces to special I/O equipment on the Q-bus for dedicated experiments can be added. The user then has direct access to this equipment. Using floppy disks as secondary storage gives the user a rather small data base. A link to a larger machine can provide access to a larger data base. Interfaces such as the DLV11 (serial interface) and the DRV-11 (parallel interface) can provide access to other computers.

One of the main benefits of using the UNIX software base is that the C compiler is available for writing application programs in the structured high-level language, C. The use of the shell command interpreter[5] is also a great asset. A general hierarchical file system is available.

The LSX system has two main areas of application:

(*i*) Control of dedicated experiments.
(*ii*) Intelligent terminals.

As a dedicated experiment controller, one can interface special I/O equipment to the LSI-11 Q-bus and both support and control the experiment with the same LSX system. The applications as an intelligent terminal are many-fold:

(*i*) Development system.
(*ii*) General text-processing applications.
(*iii*) Form editor.
(*iv*) Two-dimensional cursor-controlled text editor.

III. HARDWARE CONSIDERATIONS

The hardware required to build a useful LSX system is minimal. The absolute minimum pieces required are:

(*i*) LSI-11 microcomputer (with 4K memory).

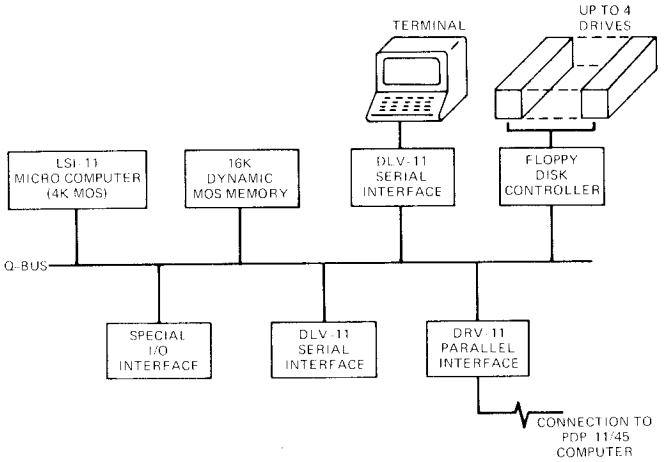

Fig. 1—LSI-11 configuration.

(*ii*) 16K memory (e.g., dynamic MOS).
(*iii*) EIS chip (extended instruction set).
(*iv*) Floppy disk controller with one drive.
(*v*) DLV-11 serial interface.
(*vi*) Terminal with serial interface.
(*vii*) Power supply.
(*viii*) Cabinet.

A more flexible and powerful system is shown in Fig. 1. An actual total system is shown in Fig. 2.

The instruction set of the LSI-11 microcomputer is compatible

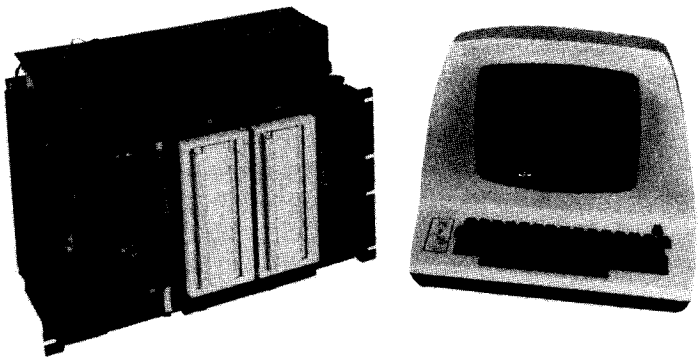

Fig. 2—LSI-11 system.

Table I

Controller	DEC	BTL	AED
Sector size (bytes)	128	512	512
Sectors per track	26	8	16
Number of tracks	77	77	77
Total capacity (bytes)	256256	315392	630784
DMA capability (y/n)	no	yes	yes
Max. transfer rate	6656	24576	49152

with that of the members of the PDP-11 family of computers with the exception of 10 instructions. The missing instructions are provided by means of the EIS chip. These special instructions may be generated by high-level compilers, and it is advantageous not to have to emulate these instructions on the microprocessor. The instructions include the multiply, divide, and multiple shift instructions.

A floppy disk controller with up to 4 drives is shown in Fig. 1. At present, only a few controllers for floppy disks interface to the LSI-11 Q-bus. The typical rotation time of the floppy disks is 360 rpm, i.e., six times per second. All floppy disks have 77 tracks; however, the number of sectors and the size of sectors is variable. The comparative data for the various floppy diskettes are shown in Table I. The maximum transfer rate is quoted in bytes per second. The outside vendor (AED Systems*) supplies dual-density drives for an increase in storage capacity. The DEC drives are IBM-compatible and have less storage capacity. We have chosen to build our own floppy disk controller for some special Bell System requirements.[6] The advantages of DMA (direct memory access) capabilities are obvious in regard to ease of programming and transfer rate. If IBM format compatibility is important, the throughput and capacity of the system are somewhat diminished.

At least one serial interface card is required to provide a terminal for the user of the system. Provided the terminal uses the standard RS232C interface, most terminals are suitable. For quick editing capabilities, CRT terminals are appropriate. For hard copy, either the common TTY33 or other terminals which run at higher baud rates may be more suitable.

The choice of memory depends on the importance of system size and whether power-fail capabilities are important. Core memory is, of course, nonvolatile, but it takes more logic boards and more

* Advanced Electronics Design, Inc., 440 Potrero Ave., Sunnyvale, California, 94086.

space and is therefore more expensive than dynamic MOS memory. Dynamic MOS memory does not take as much space, is less expensive, and takes less power, but its contents are volatile in case of power dips. Memory boards up to 16K words in size are available* for the LSI-11 microprocessor at a very reasonable price. The memory costs are likely to continue decreasing in the foreseeable future.

Another serial or parallel interface is often useful for connection to a larger machine with a large data base and a complete program development and support system. It is, of course, necessary to use such a connection to bootstrap up a system on the LSI-11 microcomputer. The central machine in this case is used to store all source for the LSX system and to compile the binary object programs required.

The system hardware is flexible enough so that, if necessary, a bus extender may be used to interface special devices to the Q-bus. This provides the ability to add special-purpose hardware that can now be controlled by the LSX system. In Section XI we describe a TV raster scan terminal that was built for editing and graphics applications. Other systems have interfaced special signal-processing equipment to the Q-bus. As DEC provides more of the interfaces to standard I/O peripherals, the applications will no doubt expand.

IV. LSX FILE SYSTEM

The hierarchical file structure of the UNIX system is maintained. The system distinguishes between ordinary files, directories, and special files. Device independence is inherent in the system. Mounted file systems are also supported. Each file system contains its own i-list of i-nodes which contain the file maps. Each i-node contains the size, number of links and the block numbers in the file. Space on disk is divided into 512-byte blocks. In contrast with the UNIX file system, two types of ordinary files are allowed. The "UNIX-type" file i-node contains the block numbers that make up a file. If the file is larger than eight blocks, the numbers in the i-node are pointers to the blocks which contain the block numbers. This requires two accesses to the disk for random file access. LSX recognizes another type of file, the contiguous file, in which the i-node contains a starting block number and the number of consecutive blocks in the file. This requires only one disk access for a random

* Monolithic Memory Systems, Inc.

access to a file, which is important for slow access devices such as floppy disks. Two special commands are provided for dealing with contiguous files; one for allocating space for a file and a second one for moving a file into a contiguous area. The layout of the disk is also crucial for optimum response to commands. By locating directories and i-nodes close to each other, file access is measurably improved over a random distribution on disk.

There is no read/write protection on files. File protection is strictly the user's responsibility. The user is essentially given super-user permissions. Only execute and directory protection is given on files. Group IDs are not implemented. File system space is limited to the capacity of the diskette in use (616 blocks for the Bell Laboratories controller).

V. LSX SYSTEM FEATURES

The LSX operating system is written in the C language, and, as such, bears a strong resemblance to the multi-user UNIX system developed for the PDP-11/40, 11/45, and 11/70 computers. The total system occupies 8K words of memory and has room for only six system buffers. Because the C compiler itself requires up to 12K words of user address space, it is possible to run the C compiler using only 20K words of total memory. It is possible to increase the system size if more capabilities are required in the operating system since the total memory space available to the system and user is actually 28K words. More system buffers could be provided in the system. If the system is kept to 8K words, a 20K-word user program could be run. However, this requires more swap space, which is at a premium.

The system is a single-user system with only one process running at any one time. A process is defined as the execution of an image contained in a file. However, a process may fork up to two levels deep, giving rise to a total of three active foreground processes. The last process forked will run to completion first. More foreground processes can be run, but this requires more swap space on the diskette used for this purpose.

The command interpreter, the shell, is identical to that used in the UNIX system. The file name given as a command is sought in the current directory. If not found, /bin/ is prepended and the /bin directory searched. The /bin directory contains all of the commands generally used. Standard input, output, and diagnostic files are

supported. Redirection of standard I/O is possible. Shell "scripts" are also executable by the command interpreter.

"Pipes" are not supported in the system, but pseudo-pipes are supported in the command shell. Pipes provide an interprocess communication channel in the UNIX time-sharing system. These pseudo-pipes are accomplished by expanding the shell syntax | to > ._pf; < ._pf. In other words, a temporary file is used to store the intermediate data passed between the commands. Providing that sufficient disk space exists, the pipe implementation is transparent to the user.

During initialization, the system automatically mounts a user file system on a second diskette if it is desired. The **mount** and **unmount** commands are not available to the user. Thus, a reboot of the system is necessary to mount a new user diskette. The system diskette is normally configured with swap space and temporary file space. User programs and files may reside on the system diskette if a user diskette is not mounted.

The size of memory available and the lack of memory protection (i.e., memory segmentation unit) have put some restrictions on the capabilities of the LSX operating system. However these are not severe in the single-user environment in which the system is run. Profiling is not provided in the system. Timing information only becomes available if a clock interrupt is provided on the LSI-11 event line at 60 times per second. Only one character device driver is allowed at present, as well as only one block device driver. No physical I/O is provided for. There is also no read-ahead on file I/O. Only six system buffers are provided, and the buffering algorithm is much simpler than in the UNIX system. Interactive debugging is not possible, but the planting of break-point traps and post-mortem debugging of a core image is possible. All user programs must be relocated to begin execution at 8K in memory. This required modifications to the UNIX link edit (**ld**) and debugger (**db**) programs. Most other differences between the LSX and the UNIX systems are not perceived by the user.

VI. BACKGROUND PROCESS

It is possible to run a background process on LSX while running a number of foreground processes to get some concurrency out of the system. The background process is run only while the current foreground process is in an input wait state. Two new system calls were added to LSX, **bground** and **kill**, to enable the user to run and

remove a background process. Only one background process is allowed to run and it is not allowed to **fork** another "child" process; however, it may execute another program. The background process either may be compute-bound or may perform some I/O functions, such as outputting to a hard-copy terminal. When the background process is compute-bound, it may take up to 2 seconds to respond to a foreground user's interactive command.

VII. STAND-ALONE ROUTINES

Under LSX, it is possible to run a dedicated program (<20K words) in real time using all the conveniences of the UNIX system calls to communicate with the file system. For programs that require more than 20K words of memory or that require more flexibility than provided by the LSX system, a set of subroutines provide the user with a UNIX-compatible interface to the file system without using the LSX system calls. A user is given more control over the program. Disk I/O issued by the user is buffered using the read-ahead and write-behind features of the standard UNIX system. A much greater number of system buffers are provided than is possible in the LSX system. Eight standard file system interface routines are provided. The arguments required for each routine and the calling sequence are identical to those required by the UNIX system C-interface routines. These include: **read**, **write**, **open**, **close**, **creat**, **sync**, **unlink**, and **seek**. Three unique routines: **saread**, **sawrite**, and **statio** are provided to enable the user to do asynchronous I/O directly into buffers in the user's area rather than into system buffers. These additional routines allow a user to start multiple I/O operations to and from multiple files concurrently, do some computation, and then wait for completion of a particular outstanding I/O transfer at some later time. To provide real-time response in applications that require it, contiguous files may be created by means of an **salloc** routine. The size of the file is specified in blocks. Once created, the file may be grown by means of the **sextend** routine. A **load** program under LSX enables the user to load a stand-alone program that must start execution at location 0 in memory.

VIII. A PROGRAM DEVELOPMENT SYSTEM

One system disk has been configured to contain a fairly complete program development system. The development programs include:

the editor,
the assembler,
the C compiler,
the link editor,
the debugger,
the command interpreter,
and the dump program,

as well as a number of libraries that contain frequently used routines for use by the link editor. It is thus possible to compile, run, and debug application programs completely on-line without access to a larger machine. In a typical application, the contents of the system disk remain quite stable, whereas all user programs are maintained on a permanently mounted user diskette. It is possible to run minimal systems with only one diskette. Although, because of the lack of protection, it is possible to crash the system, in practice, the use of the high-level language C minimizes the number of fatal bugs that actually occur, since the stack frame and program counter are quite well controlled.

In our particular installation, it is often convenient to use the Satellite Processor System[7] to aid in the running and debugging of new user programs. This is possible since programs running in the LSI-11 satellite microcomputer behave as if they are running on the central machine with access to its file system. This emulates the environment on LSX quite closely. Thus a program may be compiled on a central machine supporting the C compiler, run on the LSI-11 microcomputer, and debugged. When the program has been completely debugged, it is possible to load the program onto the floppy file system using the stand-alone routines described previously and the satellite processor system. This program may then be run under LSX.

IX. TEXT PROCESSING SYSTEM

Another area of application for the LSX system is as a personal computer system for text processing. Files may be prepared using the editor and run off using the UNIX **nroff** command with a hard-copy device. This system disk includes programs such as:

ed	editor
cat	output ASCII files

pr	print ASCII files
od	octal dump files
roff	formatter
nroff	formatter
neqn	mathematical equation formatter

The file transfer program referred to in the previous section enables one to transfer files to or from a machine with a larger data base. Users' files may be maintained on their personal mounted diskettes. If a hard-copy device is attached to the computer as well as to the user's interactive terminal, hard-copy output can be obtained using a background process while another file is edited in the foreground.

X. SUPPORT OF AN LSX SYSTEM

The limited secondary storage capacity available to LSX on floppy disks prevents the mounting of all the system source and user program source code simultaneously. Thus one must be selective as to which programs are mounted at any one time. If a great deal of program development is desirable on LSX, it is often desirable to have a connection to a host machine on which the source code for the application programs can be maintained and compiled. Two means are available to do this. One is to use the Satellite Processor System[7] and the stand-alone routines described in a previous section as a connection program. This enables one to transfer files (including complete file systems) between the host machine and the satellite processor. The SPS must exist on the host machine and the satellite processor must not be too far distant from the host machine.

A second means of providing support for LSX software is to use a serial line connection such as the DLV-11 between the host machine and the LSI-11 processor. The connection may be either dedicated or dial-up. It requires just five programs, three on the LSX system and two on the host processor. The three programs on LSX include a program to set up the connection to the host machine, i.e., login as a user to the host machine, a program to transfer files from the host to LSX, and a third program to transfer files from LSX to the host. On the host machine, the programs include one to transfer a file from the host to the LSX system and vice versa. Complete file systems as well as individual files may be transferred. Checksums are included to ensure error-free transmission.

XI. LSX SYSTEM USES

The LSX system has been put to a number of innovative uses at Bell Laboratories. These include projects that use it as a research tool, for exploratory development in intelligent terminals, and for software support for dedicated applications. LSX is well-suited for the control of an intelligent terminal. As an example, some dual-ported memory has been interfaced to the LSI-11 Q-bus. One port allows direct reading and writing of this memory by the LSI-11 CPU. The other port is used by a microcontroller to display characters on a TV raster scan screen. This enables one to change screen contents "instantaneously." The terminal is suitable for either a two-dimensional text editor or form entry applications. LSX is being used as a vehicle for investigating the future uses of programmable terminals in an office environment for word processing applications.

Other LSX installations are being used to control dedicated hardware configurations. One of the most exciting and in fact the original application for LSX was the software support system for a digital sound synthesizer system. Here the contiguous files supported by LSX are necessary for the real-time application, written as a stand-alone program consisting of a complex multiprocessing system controlling about 100 processes.[8] The system is capable of existing as a completely stand-alone system and of providing program support on itself.

XII. SUMMARY

The LSX system is currently being used for research in intelligent terminals and in stand-alone dedicated systems. Plans exist to use this system for further research in other areas of Bell Laboratories. Hard-copy features have yet to be incorporated into the system in a clean fashion. Currently, our system is connected to a larger machine using the Satellite Processor System. More general connections to larger machines or possibly to a network of machines has yet to be investigated. The LSX system also has potential uses in multiterminal or cluster control terminal systems where multitasking features are important. These application areas have only been looked at superficially and warrant further investigation.

As a development system, LSX functions quite well. The response to most programs is only a factor of four or so slower than on the conventional minicomputers, due mainly to the slow secondary storage devices used by LSX. Optimization of file storage allocation

on secondary should somewhat improve the response. For instance, the placement of directories close to the i-nodes has improved throughput significantly. The placement of the system swap area needs more investigation as to its effect on throughput.

The advent of large memory boards (64K words) will require the installation of memory mapping to take full advantage of this large address space. This will enable the running of multiple processes without the need for swapping a process out of primary memory and should also improve the response of the system and increase the number of uses to which it can be put.

There is a necessary loss of some functions in the LSX system because of the size of the memory address space available on the LSI-11 computer. However, as a single user system, most functions are still available to the user. As an intelligent terminal system, a microprocessor with all of the UNIX software available is indeed quite a desirable "intelligent" terminal.

XIII. ACKNOWLEDGMENTS

The author is indebted to H. G. Alles for designing and building both the initial PERTEC floppy disk controller and the novel TV terminal. These two pieces of hardware have provided much of the motivation for doing the LSX system in the first place and for doing research in the area of intelligent terminals in particular. Many of the application and support programs described here were written by Eugene W. Stark. John S. Thompson wrote a floppy disk driver for the AED floppy disk controller to facilitate bringing up the LSX system on these disks. The author is grateful to J. C. Swartzwelder and D. R. Weller for their efforts in putting together the first LSI-11 system. M. H. Bradley wrote the initial program to connect the LSX system to a host machine.

REFERENCES

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
2. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," B.S.T.J., this issue, pp. 1991-2019.
3. DEC LSI-11 Processor Handbook, 1976.
4. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975. Sixth Edition
5. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," B.S.T.J., this issue, pp. 1971-1990.
6. H. G. Alles, private communication.

7. H. Lycklama and C. Christensen, "UNIX Time-Sharing System: A Minicomputer Satellite Processor System," B.S.T.J., this issue, pp. 2103-2113.
8. D. L. Bayer, "Real-Time Software for Digital Music Synthesizer," Proc. Second Intl. Conf. of Computer Music, San Diego (October 1977).

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U.S.A.

UNIX Time-Sharing System:

A Minicomputer Satellite Processor System

By H. LYCKLAMA and C. CHRISTENSEN
(Manuscript received December 5, 1977)

A software support system for a network of minicomputers and microcomputers is described. A powerful time-sharing system on a central computer controls the loading, running, debugging, and dumping of programs in the satellite processors. The fundamental concept involved in supporting these satellite processors is the extension of the central processor operating system to each satellite processor. Software interfaces permit a program in the satellite processor to behave as if it were running in the central processor. Thus, the satellite processor has access to the central processor's I/O devices and file system, yet has no resident operating system. The implementation of this system was considerably simplified by the fact that all processors, central and satellite, belong to the same family of computers (DEC PDP-11 series). We describe some examples of how the SPS is used in various projects at Bell Laboratories.

I. INTRODUCTION

The satellite processor system (SPS) and the concept of a satellite processor have evolved over the years at Bell Laboratories to provide software support for the ever-increasing number of mini- and microcomputer systems being used for dedicated applications. The satellite processor concept allows the advantages of a large computing system to be extended to many attached miniprocessors, giving each satellite processor (SP) access to the central processor's (CP)

file system, software tools, and peripherals while retaining the real-time response and flexibility of a dedicated minicomputer. Since the cost of the peripherals for a minicomputer often far exceeds the cost of its CPU and memory, the CP provides a pool of peripherals for the support of many SP's. Although each SP requires a hardware link to a CP, the idea of a satellite processor is basically a software concept. It allows a user program, which might normally run in the CP using its operating system, to run in an SP with no resident operating system.

This paper describes the hardware and software required for SPS, the concepts involved in SPS, and how these concepts can be extended to provide even more powerful tools for the SP. Several examples of the use of the SPS in Bell Laboratories projects are described.

II. HARDWARE CONFIGURATION

The particular SPS hardware configuration described here consists of a DEC PDP-11/45 central computer[1] with a number of satellite processors attached using a serial I/O loop[2] as one of the communication links between the SP's and the CP (see Fig. 1). Other satellite processors are attached using DR11C, DL11, and DH11 devices (see below). Each SP is a member of the DEC PDP-11 family of computers, with its own set of special I/O peripherals and at least 4K 16-bit words of memory. A local control terminal is optional. The central computer has 112K 16-bit words of main memory and 96 megabytes of on-line storage. Eight dial-up lines and various other terminals are available for interaction with the UNIX* time-sharing system,[3] supported by the MERT operating system.[4] Magnetic tape is available as one peripheral device for off-line storage of files. Access to line printers, punched card equipment, and hard-copy graphics devices is available through the connection to the central computing facility for Bell Laboratories.

III. COMMUNICATION LINKS

A number of satellite processor systems have been installed in various hardware configurations using both the UNIX and the MERT operating systems. The devices supported as communication links include the serial I/O loop mentioned above, the DL11 asynchronous

* UNIX is a trademark of Bell Laboratories.

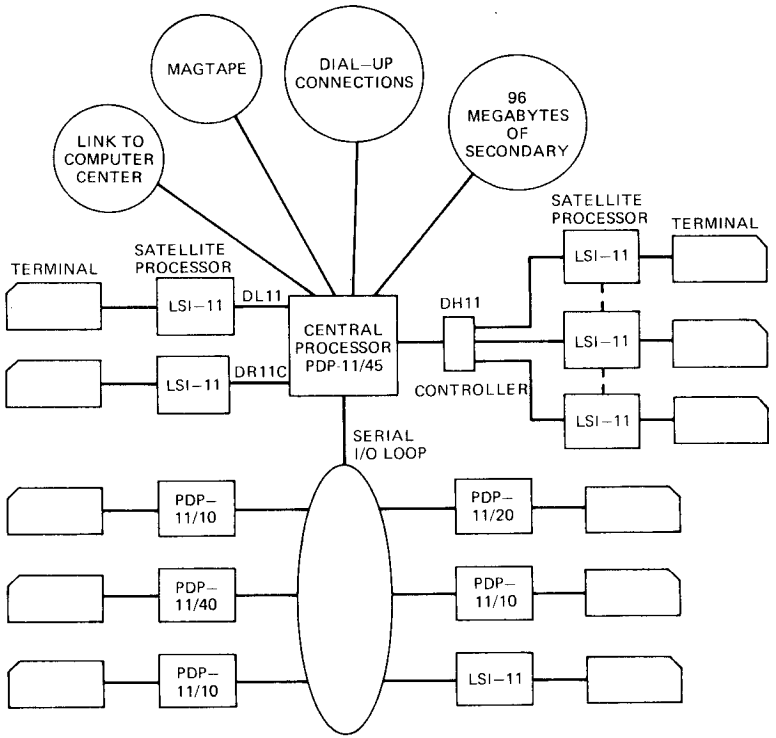

Fig. 1—Satellite processor hardware configuration.

line interface unit and the DH11 multiplexed asynchronous line interface unit. These are all essentially character-at-a-time transfer devices. The asynchronous line units may be run up to a baud rate of 9600. The most efficient communication link is the UNIBUS link device, which is a direct memory access device permitting a transfer rate of 100,000 words per second. However, the device limits the inter-processor distance to 150 feet. Another efficient link is the DR11C device, which permits word-at-a-time transfers. Its actual transfer rate is limited by software to about 10,000 words per second.

The choice of communication link is based on the distance between the SP and the CP, data transfer rate requirements, and the cost of the link. The I/O loop allows an SP to be placed at least 1000 feet from the CP and supports a data transfer rate of 3000 words per second. Thus, an SP with 16K words of memory can be loaded in 5 seconds.

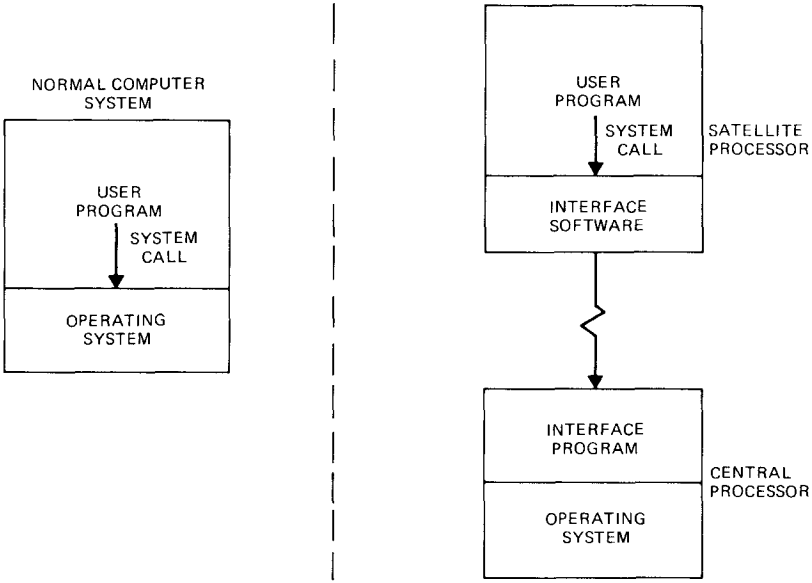

Fig. 2—Satellite processor concept.

IV. SP SOFTWARE

The satellite processor concept extends an operating system on a CP to multiple SPs. In an operating system such as the UNIX system, the interface or communication between a user program and the system is by means of the *system call*. These UNIX system calls manipulate the CP file system and other resources managed by the operating system. In the SP concept, the interface between a user program running in the SP and the operating system which is being emulated by the central processor is also the system call (see Fig. 2), except that here the extension is achieved by *trapping* the system call in the SP and passing the system call and its arguments to the CP. A process running in the CP on behalf of the SP then executes the system call and passes the results back to the SP. Control is then returned to the SP user program. Each SP executes a program locally, has access to the CP's file system and peripherals by means of the system call, and yet does not contain an operating system. This technique of partitioning a program at the UNIX system call level provides a clean, well-defined communication interface between the processors.

The local SP software required to support SPS consists of two small

functional modules, a communication package and a trap handler. The communication package transfers data between the SP and the CP on behalf of the program running in the SP. The trap handler catches processor traps (including system call traps) within the SP on behalf of the SP user program and determines whether to handle them locally or transmit the trap to the CP via the communication package.

4.1 SP communication package

The satellite processor communication package resides in the SP at the top of available memory and occupies less than 300 words. Actual size depends on the communication link used. The communication package normally resides in read-only memory. The functional requirements of the communication package include CP-SP link communication protocol, interpreting and executing CP commands, and sending trap conditions to the CP. The basic element of communication over a CP-SP link is an 8-bit byte, and messages from the CP to the SP are variable length strings of bytes containing commands and data. The SP communication package is able to distinguish commands from data by scanning for a special prefix byte. This prefix byte is followed by one of five command code bytes. Following is a list of the five commands and their arguments, which can be sent from the CP to the SP.

read memory	address	nbytes
write memory	address	nbytes
transfer	address	
return		
terminal i/o		

Each argument is two bytes (16 bits) and is sent twice, the second byte pair being the twos complement of the first to ensure error free transmission. Also, the data following the **read memory** and **write memory** commands have a checksum associated with them to guarantee proper transmission. If within the byte stream of data, a data byte corresponds to the command prefix, it is followed by an escape character to avoid treatment as a command.

This communication package is sufficient to enable the user at an SP terminal to communicate with the CP as a standard login terminal. When the SP communication package is started, it comes up in **terminal i/o** mode, passing all characters from the local SP terminal to

the CP over the communication link. In the reverse direction, all CP output is printed on the local SP terminal. The five communication commands listed above are only invoked when a program is downloaded and executed in the SP. The **read memory** and **write memory** commands are used to read and write the memory of the SP, respectively, starting at the specified address, **address** and continuing for **nbytes** bytes. The **transfer** command is used to force the SP to transfer to a specified address in the SP program, normally the beginning of the program. The **return** command is used to return control back to the SP at the address saved on the SP stack. When the CP wishes to write on or read from the local SP terminal, the SP is given the **terminal i/o** command.

4.2 SP trap handler

The second functional module which must be loaded into the SP is the trap handler. It is prepended to each program to be executed in the SP. This is the front-end package which must be link-edited with the object code produced by a UNIX compiler. The trap handler catches all SP traps and passes those that it cannot handle to the CP via the communication package. The trap handler determines the trap type (and, in the case of **system call** or SYS traps, the type of SYS trap). If the trap is an illegal instruction trap, the handler will determine if it has the capability to emulate this instruction, or whether it must be passed to the CP. If the trap is to be passed to the CP, a five-word communication area in the SP is filled with the state of the SP at the time of the trap. The communication package causes an interrupt to occur in the CP, thereby alerting the CP process running on behalf of the SP. The SP trap state is then read from the communication area and, upon processing this trap in the CP, the CP process passes argument(s) back in the communication area of the SP. Control is then returned to the SP.

The trap handler also monitors the SP program counter and local SP terminal 60 times a second using the 60-Hz clock in the satellite processor. This permits profiling a program running in the SP and controlling it from the local SP terminal. Upon detecting either a rubout character (**delete**) or a control backslash character (**quit**) from the local SP terminal, a signal is passed back to the CP, causing the SP program to abort if these signals are not handled by the SP process. At the same time a check is made to see if there have been any **delete** or **quit** signals from the CP process. If the SP has no local terminal, setting a -1 in the switch register will turn control

Table I

Instruction	PDP-11/20	PDP-11/45
mul (multiply)	830 μs	3.8 μs
div (divide)	1200	7.5
ash (shift)	660	1.5
ashc (double shift)	720	1.5
xor (exclusive or)	440	0.85
sob (sub. and branch)	400	0.85
sxt (sign extend)	400	0.85

over to the CP process. If an undebugged program in the SP halts, restarting it at location 2 will force an iot trap to the system trap handler, which in turn causes the memory of the SP to be dumped into a core file on the CP.

The trap handler consists of up to four separate submodules:

(*i*) Trap vectors, communication area, trap routines (400 words)
(*ii*) PDP-11/45 instruction emulation package (500 words)
(*iii*) Floating point instruction emulation package (1000 words)
(*iv*) Start-up routine.

Of these, the first is always required. The illegal instruction emulation packages are loaded from a library only if required. The start-up routine depends on the options specified by the user of the program to be loaded.

Estimates have been made of the execution time of the various emulation routines. The times are approximate and assume a PDP-11/20 SP, a PDP-11/45 CP, and an I/O loop connecting them.

The running times for the PDP-11/45 instructions emulated in the SP are shown in Table I. If execution time is important in a SP program, these instructions should be avoided. In C programs, these instructions are generated not only when explicit multiplies, divides, and multiple shifts are written, but also when referencing a structure in an array of structures. Using a PDP-11/35 or PDP-11/40 with a fixed point arithmetic unit as an SP would reduce the execution time for these instructions.

The average times to emulate floating point instructions in the SP are shown in Table II. For applications which require large

Table II

Instruction	PDP-11/20	PDP-11/45
add	2100 μs	4 μs
sub	2300	4
mul	3500	6
div	5600	8

quantities of CPU time running Fortran programs, it is possible to use a PDP-11/45 CPU with a floating point unit as an SP.

V. CP EMULATION OF TRAPS

During the time that the SP is executing a program, the associated CP process is roadblocked waiting for a trap signal from the SP. Upon receiving one, the CP process reads the SP trap state from the communication area, decodes the trap, and emulates it, returning results and/or errors. A check is also made to see if a signal (quit, delete, etc.) has been received.

Of the more than 40 UNIX system calls[5] emulated, about 30 are handled by simply passing the appropriate arguments from the SP to the CP process and invoking the corresponding system call in the CP. The other 10 system calls require more elaborate treatment. Their emulation is discussed in more detail here.

To emulate the **signal** system call, a table of signal registers is set aside in the CP process, one for each possible signal handled by the UNIX system. No system call is made by the CP process to handle this trap code. When a signal is received from the SP, this table is consulted to determine the appropriate action to take for the CP process. The SP program may itself catch the signals. If a signal is to cause a core dump, the entire SP memory is dumped into a CP **core** file with a header block suitable for the UNIX debugger.

The **stty** and **gtty** system calls are really not applicable to the SP process, but if one is executed, it will be applied to the CP process's control channel. The **prof** system call is emulated by transferring the four arguments to the profile buffer in the SP memory. Upon detecting nonzero entries here during each clock tick (60 times per second), the SP will collect statistics on the SP program's program counter. Upon completion of the SP program, data will be written out on the **mon.out** file. The **sbrk** system call causes the CP process to write out zeros in the SP memory to expand the **bss** area available to the program. An **exit** system call changes the communication mode between the SP and the CP back to the original terminal operation mode. It then causes the CP process to **exit**, giving the reason for the termination of the SP program.

The three most time-consuming system calls to emulate are **read**, **write**, and **exec**. The **exec** system call involves loading the executable file into the SP memory, zeroing out the data area in the SP memory, and setting up the arguments on the stack in the SP. A system **read** call involves reading from the appropriate file and then

transfering this data into the SP buffer. The system **write** call is just the reverse procedure.

The **fork**, **wait** and **pipe** system call emulations have not been written at this time and are trapped if executed in a SP. One possible means of emulating the **fork** call would be to copy an image of the parent process in one SP into another SP, permitting the piping of data between two SPs.

VI. TYPICAL SESSION

Supporting a mini-PDP-11 as an SP on a CP running the UNIX system combines all the advantages of the UNIX system programming support with the real-time response and economic advantage of a stand-alone PDP-11. In a typical SP programming session, a programmer sitting at the local SP terminal logs into the CP and uses the UNIX editor to update an SP program source file. It could be assembly language or one of the higher-level languages available on the UNIX system (C, Lil,* Fortran). Assume a C source file **prog.c**. When the edit is complete, the following commands are issued:

```
% cc -c prog.c
% ldm -me prog.o
% 11l a.out
```

cc -c compiles the C program **prog.c** in the CP and produces the object file **prog.o**. **ldm -me** combines the SP trap handler (**-m**) and instruction emulator (**e**) with the C object file **prog.o**, generating an **a.out** object file. **11l** loads the **a.out** file into the SP, and starts it with the SP terminal as the standard input and output. The programmer then observes the results of running the program or forces a core dump, and uses the UNIX debugger to examine it. If any program changes are required, the preceding steps are repeated. During this typical SP support sequence, the programmer initiates the editing, compiling, loading, running, and debugging of a program on a mini-PDP-11 without leaving its control terminal. It is the speed and convenience of this procedure along with the availability of high-level languages that make the satellite processor concept a powerful mini-PDP-11 support tool.

---

* Lil is a little implementation language for the PDP-11.

VII. USES

Some SP's may be disconnected from the CP when their software has been developed and the final product is a "stand-alone" system. Other SPs may always have a CP connection; they supply the real-time response unavailable from the CP, combined with access to the CP's software base, file system, peripherals, and connection to the computing community.

One use of the SPS system is discussed in a paper in this issue.[6] Here LSI-11 microcomputers connected to a CP by means of a DH11 device are used in a materials research laboratory, remote from the CP, to collect data, control apparatus and machinery, and analyze the results.

One of the more interesting applications of the satellite processor system is its use to support a digital sound synthesizer system. The hardware consists of an LSI-11 processor with 24K words of memory, two floppy disks, a TV raster scan terminal, and much more special digital circuitry interfaced to the LSI-11 Q-bus to provide the control of the DSSS. The heart of the software consists of a multi-tasking system designed to handle about 100 processes.[7] The basic program directs the machine's output devices such as oscillators, filters, multipliers, and a reverberation unit. The data for the program are stored and retrieved from the floppy disk. The SPS is used to download programs from the CP and produce core dumps of the LSI-11 memory back at the CP for debugging purposes. The CP is also used for program development.

VIII. SUMMARY

The advantages of the SPS system are the use of higher level languages, ease of program development and maintenance, use of debugging tools, interactive turn-around, use of a common pool of peripherals, access to files on the CP secondary storage, and connection to central computing facilities. The SP requires a minimum amount of memory since it does not contain an operating system or other supporting software. One additional advantage is that any SP may be located in a remote laboratory location.

The ability to extend an operating system to an SP may be used for purposes other than supporting software development for the SP. A new operating system environment may be defined by rewriting the CP process which acts on behalf of the SP program. In this way a new set of "system calls" emulating another operating system may

be extended to an SP. SPs other than PDP-11s may also be supported by writing an appropriate SP communication package and CP interface package. Cross compilers would be required on the CP to support software development for these non-PDP-11 processors.

Another avenue of research which has not yet been explored with the SPS concept is that of distributed computing. With a powerful SP, e.g. PDP-11/45, a compute-bound program could run on the SP rather than on the CP itself, thereby transferring the real-time load from the CP to the SP. The CP would only be called upon to load the program initially and to satisfy certain file requests. The total computing power of the system would increase greatly without duplicating the entire computer system.

REFERENCES

1. *DEC PDP-11 Processor Handbook.* 1975.
2. D. R. Weller, "A Loop Communication System for I/O to a Small Multi-User Computer," Proc. IEEE Intl. Computer Soc. Conf., Boston (September 1971).
3. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
4. H. Lycklama and D. L. Bayer, "UNIX Time-Sharing System: The MERT Operating System," B.S.T.J., this issue, pp. 2049-2086.
5. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975, sixth edition.
6. B. C. Wonsiewicz, A. R. Storm, and J. D. Sieber, "UNIX Time-Sharing System: Microcomputer Control of Apparatus, Machinery, and Experiments," B.S.T.J., this issue, pp. 2209-2232.
7. D. L. Bayer, "Real-Time Software for Digital Music Synthesizer," Proc. Second Intl. Conf. of Computer Music, San Diego (October 1977).

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

Document Preparation

By B. W. KERNIGHAN, M. E. LESK, and J. F. OSSANNA, Jr.
(Manuscript received January 6, 1978)

The UNIX operating system provides programs for sophisticated document preparation within the framework of a general-purpose operating system. The document preparation software includes a text editor, programmable text formatters, macro-definition packages for a variety of page layout styles, special processors for mathematical expressions and for tabular material, and numerous supporting programs, such as a spelling-mistake detector. In practice, this collection of facilities has proven to be easy to learn and use, even by secretaries, typists, and other nonspecialists. Experiments have shown that preparation of complicated documents is about twice as fast as on other systems. There are many benefits to using a general-purpose operating system instead of specialized stand-alone terminals or a system dedicated to "word processing." On the UNIX system, these include an excellent software development facility and the ability to share computing and data resources among a community of users.*

I. INTRODUCTION

We use the term *document preparation* to mean the creation, modification, and display of textual material, such as manuals, reports, papers, and books. "Document preparation" seems preferable to "text processing" (which is not particularly precise), or

* UNIX is a trademark of Bell Laboratories.

"word processing" (which has acquired connotations of stand-alone specialized terminals).

Computer-aided document preparation offers some clear benefits. Text need be entered only once. Thereafter, only those portions that need to be changed require editing; the remaining material is left alone. This is a significant labor reduction for any document that must be modified or maintained over a period of time.

There are many other important benefits. Special languages can be used to facilitate the entry of complex material such as tables and mathematical expressions. The style or format of a document can be decoupled from its content; the only format-control information that need be embedded is that describing textual categories and boundaries, such as titles, section headings, paragraphs, and the like. Alternative document styles are then possible through the use of different formatting programs and different interpretations applied to the embedded format control. Furthermore, programs can examine text to detect spelling mistakes, compare versions of documents, and prepare indexes automatically. Machine-generated data can be incorporated in documents; excerpts from documents can be fed to programs without transcription.

A variety of comparatively elegant output devices has become available, supplementing the traditional typewriters, terminals, and line printers; this has led to a much increased interest in automated document preparation. Automated systems are no longer limited to straight text composed in unattractive constant-width characters, but can produce a full range of printed documents in attractive fonts and page layouts. The major example of an output device with significant capabilities is the phototypesetter, which produces very high quality printed output on photographic paper or film. Other devices include typewriter-like terminals capable of high-resolution motion, dot matrix printer-plotters, microfilm recorders, and xerographic printers.

Further advantages accrue when document preparation is done on a general-purpose computer system. One is the opportune sharing of programs and data bases among users; programs originally written for some other purpose may be useful to the document preparer. Having a broad range of users, from typists to scientists, on the same system leads to an unusual degree of cooperation in the preparation of documents.

The UNIX document preparation software includes an easy-to-learn-and-use text editor, **ed**, which is the tool for creating and modifying any kind of text, from documents to data to programs.

Two programmable text formatters, nroff and troff, provide paginated formatting and allow unusual freedom and flexibility in determining the style of documents. Augmented by various macro-definition packages, nroff and troff can be programmed to provide footnote processing, multiple-column output, column-length balancing, and automatic figure placement. An equation preprocessor, eqn, translates a simple language for describing mathematical expressions into formatter input; a table-construction preprocessor, tbl, provides an analogous facility for input of data and text that is to be arranged into tables.

We then mention other programs useful to the document preparer and summarize some comparisons between manual methods of document preparation and methods using UNIX document preparation software.

II. TEXT EDITING

The UNIX text editor ed is the basic tool for entering text and for subsequent modifications. We will not try to give a complete description of ed here; details may be found in Ref. 1. Rather, we will try to mention those attributes that are most interesting and unusual.

The editor is not specialized to any particular kind of text; it is used for programs, data, and documents alike. It is based on editing commands such as "print" and "substitute," rather than on special function keys, and provides convenient facilities for selecting the text lines to be operated on and altering their contents. Since it does not use special function keys or cursor controls, it does not require a particular kind of input device. Several alternative editors are available that make use of terminals with cursors, but these have been much less widely used; for most purposes, it is fair to say that there is only one editor.

A text editor is often the primary interface between a user and the system, and the program with which most user time is spent. Accordingly, an editor has to be easy to use, and efficient of the user's time—editing commands have to "flow off the fingertips." In accordance with this principle, ed is quite terse. Each editor command is a single letter, e.g., p for "print," and d for "delete." Most commands may be preceded by zero, one, or two "line addresses" to affect, respectively, the "current line" (i.e., the line most recently referenced), the addressed line, or the range of contiguous lines between and including the pair of addresses. There are also

shorthands for the current line and the last line of the file. Lines may be addressed by line number, but more common usage is to indicate the position of a line relative to the current or last line. Arithmetic expressions involving line numbers are also permitted:

```
-5,+5p
```

prints from five lines before the current line to five lines after, while

```
$-5,$p
```

prints the last six lines. In both cases, the current line becomes the last line printed, so that subsequent editing operations may begin from there.

Most often, the lines to be affected are specified not by line number, but by "context," that is, by naming some text pattern that occurs in them. The "line address"

```
/abc/
```

refers to the first line after the current line that contains the pattern abc. This line address standing by itself will find and print the next line that contains abc, while

```
/abc/d
```

finds it and deletes it. Context searches begin with the line immediately after the current line, and wrap around from the end of the file to the beginning if necessary. It is also possible to scan the file in the reverse direction by enclosing the pattern in question marks: ?abc? finds the previous abc.

The substitute command s can replace any pattern by any literal string of characters in any group of lines. The command

```
s/ofrmat/format/
```

changes ofrmat to format on the current line, while

```
1,$s/ofrmat/format/
```

changes it everywhere. In both searches and substitute commands, the pattern // is an abbreviation for the most recently used pattern, and & stands for the most recently matched text. Both can be used to avoid repetitive typing. The "undo" command u undoes the most recent substitution.

Text can be added before or after any line, and any group of contiguous lines may be replaced by new lines. "Cut and paste" operations are also possible—any group of lines may be either moved or

copied elsewhere. Individual lines may be split or coalesced; text within a line may be rearranged.

The editor does not work on a file directly, but on a copy. Any file may be read into the working text at any point; any contiguous lines may be written out to any file. And any UNIX command may be executed from within the editor, even another instance of the editor.

So far, we have described the basic editor features: this is all that the beginning user needs to know. The editor caters to a wide variety of users, however, and has many features for more sophisticated operations. Patterns are not restricted to literal character strings, but may include several "metacharacters" that specify character classes, repetition of characters or classes, the beginning or end of a line, and so on. For example, the pattern

```
/^[0-9]/
```

searches for the next line that begins with a digit.

Any set of editing commands may be done under control of a "global" command: the editing commands are performed starting at each line that matches a pattern specified in the global command. As the simplest example,

```
g/interesting/p
```

prints all lines that contain **interesting**.

Finally, given the UNIX software for input-output redirection, it is easy to make a "script" of editing commands in advance, then run it on a sequence of files.

The basic pattern-searching and editing capabilities of **ed** have been co-opted into other, more specialized programs as well. The program **grep** ("global regular expression print") prints all input lines that contain a specified pattern; this program is particularly useful for finding the location of an item in a set of files, or for culling items from larger inputs. The program **sed** is a variant of **ed** that performs a set of editing operations on each line of an input stream of arbitrary length.

III. TROFF AND NROFF — BASIC TEXT FORMATTERS

Once a user has entered a document into the file system, it can be formatted and printed by **troff** and **nroff**.[2] These are programmable text formatters that accommodate a wide variety of formatting tasks by providing flexible fundamental tools rather than specific features.

A *trap* mechanism provides for action when certain conditions occur. The conditions are position on the current output page, length of a diversion, and an input line count. A macro associated with a vertical page position is automatically invoked when a line of output falls on or after the trap position. For example, reaching a specified place near the bottom of the page could invoke a macro that describes the bottom margin area. Similarly, a vertical position trap may be specified for diverted output. An input line count trap causes a macro to be invoked after reading a specified number of input text lines.

A variety of parameters are available to the user in predefined number registers. In addition, users may define their own registers. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired. In most circumstances, numerical input may have appended scale factors representing inches, points, ems, etc. Numerical input may be provided by expressions involving a variety of arithmetic and logical operators.

A mechanism is provided for conditionally accepting a group of lines as input. The conditions that may be tested are the value of a numerical expression, the equality of two strings, and the truth of certain built-in conditions.

Certain of the parameters that control text processing constitute an *environment,* which may be switched by the user. It is convenient, for example, to process footnotes in a separate environment from the main text. Environment parameters include line length, line spacing, indent, character size, and the like. In addition, any collected but not yet output lines or words are a part of the environment. Parameters that are global and not switched with the environment include, for example, page length, page position, and macro definitions.

It is not possible to give any substantial examples of **troff** macro definitions, but we will sketch a few to indicate the general style of use.

The simplest example is to provide pagination—an extra space at the top and bottom of each page. Two macros are usually defined—a *header* macro containing the top-of-page text and spacings, and a *footer* macro containing the bottom-of-page text and spacings. A trap must be placed at vertical position zero to cause

the header macro to be invoked and a second trap must be placed at the desired distance from the bottom for the footer. Simple macros merely providing space for the margins could be defined as follows.

```
.de hd          \" begin header definition
'sp 1i          \" space 1 inch
..              \" end of header definition
.de fo          \" footer
'bp             \" space to beginning of next page
..              \" end of footer definition
.wh 0 hd        \" set trap to invoke hd when at top of page
.wh -1i fo      \" set trap to invoke fo 1 inch from bottom
```

The sequence \" introduces a troff comment.

The production of multi-column pages requires somewhat more complicated macros. The basic idea is that the header macro records the vertical position of the column tops in a register and initializes a column counter. The footer macro is invoked at the bottom of each column. Normally it increments the column counter, increments the page offset by the column width plus the column separation, and generates a reverse vertical motion to the top of the next column (the place recorded by the header macro). After the last column, however, the page offset is restored and the desired bottom margin functions occur.

Footnote processing is complicated; only the general strategy will be summarized here. A pair of macros is defined that allows the user to indicate the beginning and end of the footnote text. The footnote-start macro begins a diversion that appends to a macro in which footnotes are being collected and changes to the footnote environment. The footnote-end macro terminates the diversion, resets the environment, and moves the footer trap up the page an amount equal to the size of the diverted footnote text. The footer eventually invokes and then removes the macro containing the accumulated footnotes and resets its own trap position. Footnotes that don't fit have their overflow rediverted and are treated as the beginning footnote on the next page.

The use of preprocessors to convert special input languages for equations and tables into troff input means that many documents reach troff containing large amounts of program-generated input. For example, a simple equation might produce dozens of troff input lines and require many string definitions, redefinitions, and detailed numerical computations for proper character positioning. The troff string that finally contains the equation contains many font and size

changes and local motion, and so can become very long. All of this demands substantial string storage, efficient storage allocation, larger text buffers than would otherwise be necessary, and the accommodation of large numbers of strings and number registers. Input generated by programs instead of people severely tests program robustness.

IV. MACROS—DECOUPLING CONTENT AND FORMAT

Although troff provides full control over typesetter (or typewriter) features, few users exercise this control directly. Just as programmers have learned to use problem-oriented languages rather than assembly languages, it has proven better for people who prepare documents to describe them in terms of content, rather than specifying point sizes, fonts, etc., in a typesetter-oriented way. This is done by avoiding the detailed commands of troff, and instead embedding in the text only macro commands that expand into troff commands to implement a desired format.

For example, the title of a document might be prefaced by

```
.TL
```

which would expand, for this journal, into "Helvetica Bold font, 14 point type, centered, at top of new page, preceded by copyright notice," but for other journals might be "Times Roman, left adjusted, preceded by a one-inch space," or whatever is desired. In a similar way, there would be macros for other common features of a document, such as author's name, abstract, section, paragraph, and footnote.

Macro packages have been prepared for a variety of document styles. Locally, these include formal and informal internal memoranda; technical reports for external distribution; the Association for Computing Machinery journals; some American Institute of Physics journals; and *The Bell System Technical Journal.* All these macro packages recognize standard macro names for titles, paragraphs, and other document features. Thus, the same input can be made to appear in many different forms, without changing it.

An important advantage of this system is the ease with which new users learn document preparation. It is necessary only to learn the correct way to describe document content and boundaries, not how to control the typesetter at a detailed level. A typist can easily learn the dozen or so most common macros in a few minutes, and

another dozen as needed. This entire article uses only about 30 distinct macro calls, rather more than the norm.

Although nroff is used for typewriter-like output, and troff for photocomposition, they accept exactly the same input language, and thus hide details of particular devices from users. Macro packages also provide a degree of independence: they permit a uniformity of *input*, so that input documents look the same regardless of the output format or device they eventually appear in. This means that to find the title of a document, for example, it is not necessary to know what format is being used to print it. Finally, macros also enforce a uniformity of *output*. Since each output format is defined in appearance by the macro package that generates it, all documents prepared in that format will look the same.

V. EQN—A PREPROCESSOR FOR MATHEMATICAL EXPRESSIONS

Much of the work of Bell Laboratories is described in technical reports and papers containing significant amounts of mathematics. Mathematical material is difficult to type and expensive to typeset by traditional methods. Because of positioning requirements and the multiplicity of characters, sizes, and fonts, it is not feasible for a human to typeset mathematics directly with troff commands. troff is richly endowed with the facilities needed for preparing mathematical expressions, such as arbitrary horizontal and vertical motions, line-drawing, size changing, etc., but it is not easy to use these facilities directly because of the difficulty of deciding the degree of size change and motion suitable in every circumstance. For this reason, a language for describing mathematical expressions was designed; this language is translated into troff by a program called eqn.

An important requirement is that the language should be easy to learn and use by people who don't know mathematics, computing, or typesetting. This implies that normal mathematical conventions about operator precedence, parentheses, and the like cannot be used, for otherwise the user would have to understand what was being typed. Further, there should be very few rules, keywords, special symbols, and few exceptions to the rules. Finally, standard actions should take place automatically—size and font changes should follow normal mathematical usage without user intervention.

When a document is typed, mathematical expressions are entered as part of the text, but are marked by user-settable delimiters. eqn reads this input and passes through untouched those parts that are

not mathematics. At the same time, it converts the mathematical parts into the necessary **troff** commands. Thus normal usage is a pipeline of the form

```
eqn files | troff
```

The language is defined by a Yacc[5] grammar to insure regularity and ease of change. We will not describe the **eqn** language in detail; see Refs. 6 and 7. Nonetheless, it is worth showing a few examples to give a feeling for the language. Throughout this section we write expressions exactly as they are typed by the user, except that we omit the delimiters that mark the beginning and end of each expression.

eqn is an oral (or perhaps aural) language. To produce

$$2\pi \int \sin(\omega t)\, dt$$

one writes

```
2 pi int sin ( omega t)dt
```

Each "word" in the input is looked up in a table. In this case, **pi** and **omega** are recognized as Greek letters, **int** is a special character, and **sin** is to be placed in Roman font instead of italic, following conventional practice. Parentheses and digits are also made Roman, and spacing is adjusted around characters to give a more pleasing appearance.

Subscripts, superscripts, fractions, radicals, and the like are introduced by words used as operators:

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r}$$

is produced by

```
x sup 2 over a sup 2 ~=~ sqrt {pz sup 2 + qz + r}
```

The operator **sub** produces a subscript in the same manner as **sup** produces a superscript. Braces { and } are used to group items that are to be treated as a unit, such as all the terms to go under the radical. **eqn** input is free-form, so blanks and new lines can be used freely to make the input easier to type, read, and subsequently edit. The tilde ~ is used to force extra space into the output when needed.

More complicated expressions are built from the same piece parts, and perhaps a few new ones. For example,

```
erf (z) ~=~ 2 over sqrt pi int sub 0 sup z
e sup -t sup 2 dt
```

produces

$$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

while

```
zeta (s) ~=~ sum from k=1 to inf
k sup -s ~~~ ( Re ~ s > 1)
```

is

$$\zeta(s) = \sum_{k=1}^{\infty} k^{-s} \quad (\text{Re } s > 1)$$

and

```
lim from {x -> pi /2} ( tan ~x) sup {sin~2x} ~=~ 1
```

yields

$$\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$$

In addition, there are built-up brackets, braces, etc.; matrices; diacritical marks such as dots and bars; font and size changes to override defaults; facilities for lining up equations; and macro substitution.

Because not all potential users have access to a typesetter, there is also a compatible version of **eqn** that interfaces to **nroff** for producing output on terminals capable of half-line motions and printing special characters. The quality of terminal output leaves something to be desired, but it is often adequate for proofreading and some internal uses.

The **eqn** language has proven to be easy to learn and use; at the present time, well over a hundred typists and secretaries use it at Bell Laboratories. Most are either self-taught, or have learned it as part of a course in UNIX system procedures taught by other secretaries and typists. Empirically, mathematically trained users (mathematicians, physicists, etc.) can learn enough **eqn** in a few minutes to begin useful work, for its syntax and rules are very similar to the way that mathematics is actually spoken. Persons not trained in mathematics take longer to get started, because the language is less familiar, but it is still true that an hour or two of instruction is enough to begin doing useful work.

By intent, eqn does not know very much about typesetting; in general, it lets troff do as much of the job as possible, including all character-width computations. In this way, eqn can be relatively independent of the particular character set, and even of the typesetter being used.

The basic design decision to make a separate language and program distinct from troff does have some drawbacks, because it is not easy for eqn to make a decision based on the way that troff will produce the output. The programs are very loosely coupled. Nonetheless, these drawbacks seem unimportant compared to the benefits of having a language that is easily mastered, and a program that is separate from the main typesetting program. Changes in one program generally do not affect the other; both programs are smaller than they would be if they were combined. And, of course if one doesn't use eqn, there is no cost, since troff doesn't contain any code for it.

VI. TBL—A PREPROCESSOR FOR TABLES

Tables also present typographic difficulties. The primary difficulty is deciding where columns should be placed to accommodate the range of widths of the various table entries. It is even harder to arrange for various lines or boxes to be drawn within the table in a suitable way. tbl[8] is a table construction program that is also an independent preprocessor, quite analogous to eqn.

tbl simplifies entering tabular data, which may be tedious to type or may be generated by a program, by separating the table format from its contents. Each table specification contains three parts: a set of global options affecting the whole table, such as "center" or "box"; then a set of commands describing the format of each line of the table; and finally the table data. Each specification describes the alignment of the fields on a line, so that the description

```
L R R
```

indicates a line with three fields, one left adjusted and two right adjusted. Other kinds of fields are "C" (centered) and "N" (numerical adjustment), with "S" (spanned) used to continue a field across more than one column. For example,

```
C S S
L N N
```

describes a table whose first line is a centered heading spanning

three columns; the three columns are left-adjusted, numerically adjusted, and numerically adjusted respectively. If there are more lines of data than of specifications (the normal case), the last specification applies to all remaining data lines.

A sample table in the format above might be

Position of Major Cities		
Tokyo	35°45′ N	139°46′ E
New York	40°43′ N	74°01′ W
London	51°30′ N	0°10′ W
Singapore	1°17′ N	103°51′ E

The input to produce the above table, with tab characters shown by the symbol ⓣ, is as follows:

```
.TS
center, box;
C S S
L N N.
Position of Major Cities
Tokyoⓣ35°45′ Nⓣ139°46′ E
New Yorkⓣ40°43′ Nⓣ74°01′ W
Londonⓣ51°30′ Nⓣ0°10′ W
Singaporeⓣ1°17′ Nⓣ103°51′ E
.TE
```

tbl also provides facilities for including blocks of text within a table. A block of text may contain any normal typesetting commands, and may be adjusted and filled as usual. tbl will arrange for adequate space to be left for it and will position it correctly. For example, the table on the next page uses text blocks, line and box drawing, size and font changes, and the facility for centering vertical placement of the headings (compare the heading of column 3 with that of columns 1 and 2). Note that there is no difficulty with equations in tables. In fact, there is sometimes a choice between writing a matrix with the matrix commands of eqn or making a table of equations. Typically, the typist picks whichever program is more familiar.

The tbl program writes troff code as output, just as eqn does. This code computes the width of each table entry, decides where to place the columns and lines separating them, and prints the table. tbl itself does not understand typesetting: it does not know the

Functional Systems		
Function Number	Function Type	Solution
1	LINEAR	Systems of equations all of which are linear can be solved by Gaussian elimination.
2	POLYNOMIAL	Depending on the initial guess, Newton's method $(f_{i+1}=f_i-\frac{f_i'}{f_i})$ will often converge on such systems.
3	ALGEBRAIC	The program ZONE by J. L. Blue will solve systems for which an accurate initial guess is not known.

widths of characters, and may (in the case of equations in tables) have no knowledge of the height, either. However, it writes **troff** output that computes these sizes, and adjusts the table accordingly. Thus tables can be printed on any device and in any font without additional work.

Most of the comments about using **eqn** apply to **tbl** as well: it is easy to learn and is in wide use at Bell Laboratories. Since it is a program separate from **troff**, it need not be learned, used, or paid for if no tables are present. Comparatively few users need to know all of the tools: typically, the workload in one area may be mathematical, in another area statistical and tabular, and in another only ordinary text.

VII. OTHER SUPPORTING SOFTWARE

One advantage of doing document preparation in a general-purpose computing environment instead of with a specialized word processing system is that programs not directly related to document preparation may often be used to make the job easier. In this section, we discuss some examples from our experience.

One of the most tedious tasks in document preparation is detection of spelling and typographical errors. Existing data bases originally obtained for other purposes are used by a program called **spell**, which detects potential spelling mistakes. Machine-readable dictionaries (more precisely, word lists) have been available for some time. Ours was originally used for testing hyphenation algorithms and for checking voice synthesizer programs. It was realized, however, that a rudimentary program for detecting spelling mistakes

could be made simply by comparing each word in a document with each word in the dictionary; any word in the document but not in the dictionary is a potential misspelling.

The first program for this approach was developed in a few minutes by combining existing UNIX utilities for sorting, comparing, etc. This was sufficiently promising that additional small programs were written to handle inflected forms like plurals and past participles. The resulting program was quite useful, for it provided a good match of computer capabilities to human ones. The machine can reduce a very large document to a tractable list of suspicious words that a human can rapidly scan to detect the genuine errors.

Naturally, normal output from **spell** contains not only legitimate errors, but a fair amount of technical jargon and some proper names. The next step is to use that output to refine the dictionary. In fact, we have carried this step to its logical conclusion, by creating a brand new dictionary that contains only words culled from documents. This new dictionary is about one-third the size of the original, and produces rather better results.

One of the more interesting peripheral devices supported by the UNIX system is an inexpensive voice synthesizer.[9] The program **speak**[10] uses this synthesizer to pronounce arbitrary text. Speaking text has proven especially handy for proofreading tedious data like lists of numbers: the machine speaks the numbers, while a person reads a list in parallel.

Another example of a borrowed program is **diff**,[11] which compares two inputs and prepares a list of all the places in which they differ. Normally, **diff** is used for comparing two versions of a program, as a check on the changes that have been made. But of course it can also be used on two versions of a document as well. In fact, the **diff** output can be captured and used to produce a set of **troff** commands that will print the new version with marginal bars indicating the places where the document has been changed.

We have already mentioned two major preprocessors for **troff** and **nroff**, for mathematics and tables. The same approach, of writing a separate program instead of cluttering up an existing one, has been applied to *post*processors as well. Typically, these postprocessors are concerned with matching **troff** or **nroff** output with the characteristics of some different output device. One example is a processor called **col** that converts **nroff** output containing reverse motions (e.g., multi-column output) into page images suitable for printing on devices incapable of reverse motion. Another example is a program that converts **troff** output intended for a phototypesetter into a form

suitable for display on the screen of a Tektronix 4014 terminal (or analogous graphic devices). This permits a view of the formatted document without actually printing it; this is especially convenient for checking page layout.

One final area worth mentioning concerns the problem of training new users. Since there seems to be no substitute for hands-on experience, a program called **learn** was written to walk new users through sets of lessons.[12] Lesson scripts are available for fundamentals of UNIX file handling commands, the editor **ed**, and **eqn**, as well as for topics not related to document preparation. **learn** has been heavily used in the courses taught by secretaries and typists for their colleagues.

VIII. EXPERIENCE

UNIX document preparation software has now been used for several years within Bell Laboratories, with many secretaries and typists in technical organizations routinely preparing technical memoranda and papers. Several books[13-19] printed with this software have been published directly from camera-ready copy. Technical articles have been prepared in camera-ready form for periodicals ranging from the *Journal of the ACM* to *Science*.

The longest-running use of the UNIX system for document preparation is in the Bell Laboratories Legal and Patent Division, where patent applications have been prepared on a UNIX system for nearly seven years. Computer program documentation has been produced for several years by clerks using UNIX facilities at the Business Information Systems Programs Area of Bell Laboratories. More recently, the "word processing" centers at Bell Laboratories have begun significant use of the UNIX system because of its ability to handle complicated material effectively.

It can be difficult to evaluate the cost-effectiveness of computer-aided versus manual documentation preparation. We took advantage of the interest of the American Physical Society in the UNIX system to make a systematic comparison of costs of their traditional typewriter composition and a UNIX document preparation system. Five manuscripts submitted to *Physical Review Letters* were typeset at Bell Laboratories, using the programs described above to handle the text, equations, tables, and special layout of the journal.

On the basis of these experiments, it appears that computerized typesetting of difficult material is substantially cheaper than typewriter composition. The primary cost of page composition is

keyboarding, and the aids provided by UNIX software to facilitate input of complex mathematical and tabular material reduce input time significantly. Typing and correcting articles on the UNIX system, with an experienced typist, was between 1.5 and 3.3 times as fast as typewriter composition. Over the trial set of manuscripts, input using the UNIX system was 2.4 times as fast. These documents were extremely complicated, with many difficult equations. Typists at *Physical Review Letters* averaged less than four pages per day; whereas our (admittedly very proficient) UNIX system typist could type a page in 30 minutes. We estimate a very substantial saving in production cost for camera-ready pages using a UNIX system instead of conventional composition or typewriting. A typical UNIX system for photocomposition of *Physical Review* style pages might produce 200 finished pages per day on a capital investment of about $200,000 and with 20 typists.

The advantage of the UNIX system is greatest when mathematics and tables abound in a document. For example, it is a great time saving that keys need never be changed because all equation input is ordinary text. The automatic page layout saves time when multiple drafts, versions, or editions of a document are needed. Further details of this comparison can be found in Ref. 20.

IX. CONCLUSIONS

It is important to note that these document preparation programs are simply application programs running on a general-purpose system. Any document preparation user can exercise any command whenever desired.

As mentioned above, a surprising number of the programming utilities are directly or indirectly useful in document preparation. For example, the program that makes cross-reference listings of computer programs is largely identical with the one that makes keyword-in-context indexes of natural language text. It is also easy to use the programming facilities to generate small utilities, such as one which checks the consistency of equation usage.

Besides applying programming utilities to text processing, we also apply document processors to programs and numerical data. Statistical data are often extracted from program output and inserted into documents. Computer programs are often printed in papers and books; because the programs are tested and typeset from the same source file, transcription errors are eliminated.

In addition to the technical advantages of having programming

and word processing on the same machine, there can be personnel advantages. The fact that secretaries and typists work on the same system as the authors allows both to share the document preparation job. A document may be typed originally by a secretary, with the author doing the corrections; in the case of an author who types rough drafts but doesn't like editing after proofreading, the reverse may occur. We have observed the full spectrum, from authors who give hand-written material to typists in the traditional manner to those who compose at the terminal and do their own typesetting. Most authors, however, seem to operate somewhere in between.

The UNIX system provides a convenient and cost-effective environment for document preparation. A first-class program development facility encourages the development of good tools. The ability to use preprocessors has enabled us to write separate languages for mathematics, tables, and several other formatting tasks. The separate programs are easier to learn than if they were all jammed into one package, and are vastly easier to maintain as well. And since all of this takes place within a general-purpose operating system, programs and data can be used as convenient, whether they are intended for document preparation or not.

REFERENCES

1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975. See ED (1).
2. J. F. Ossanna, "NROFF/TROFF User's Manual," Comp. Sci. Tech. Rep. No. 54, Bell Laboratories (April 1977).
3. J. E. Saltzer, "Runoff," in *The Compatible Time-Sharing System,* ed. P. A. Crisman, Cambridge, Mass.: M.I.T. Press (1965).
4. M. D. McIlroy, "The Roff Text Formatter," Computer Center Report MHCC-005, Bell Laboratories (October 1972).
5. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories (July 1975).
6. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," Commun. Assn. Comp. Mach., *18* (March 1975), pp. 151-157.
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," Comp. Sci. Tech. Rep. No. 17, Bell Laboratories (April 1977).
8. M. E. Lesk, "Tbl — A Program to Format Tables," Comp. Sci. Tech. Rep. No. 49, Bell Laboratories (September 1976).
9. Federal Screw Works, *Votrax ML-1 Multi-Lingual Voice System.*
10. M. D. McIlroy, "Synthetic English Speech by Rule," Comp. Sci. Tech. Rep. No. 14, Bell Laboratories (March 1974).
11. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Comp. Sci. Tech. Rep. No. 41, Bell Laboratories (June 1976).
12. B. W. Kernighan and M. E. Lesk, unpublished work (1976).
13. B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style,* New York: McGraw-Hill, 1974.
14. C. H. Sequin and M. F. Tompsett, *Charge Transfer Devices,* New York: Academic Press, 1975.
15. B. W. Kernighan and P. J. Plauger, *Software Tools,* Reading, Mass.: Addison-Wesley, 1976.

16. T. A. Dolotta et al., *Data Processing in 1980-1985: A Study of Potential Limitations to Progress,* New York: Wiley-Interscience, 1976.
17. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Reading, Mass.: Addison-Wesley, 1977.
18. Committee on Impacts of Stratospheric Change, *Halocarbons: Environmental Effects of Chlorofluoromethane Release,* Washington, D. C.: National Academy of Sciences, 1977.
19. W. H. Williams, *A Sampler on Sampling,* New York: John Wiley & Sons, 1977.
20. M. E. Lesk and B. W. Kernighan, "Computer Typesetting of Technical Journals on UNIX," Proc. AFIPS NCC, *46* (1977), pp. 879-888.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

Statistical Text Processing

By L. E. McMAHON, L. L. CHERRY, and R. MORRIS
(Manuscript received December 5, 1977)

Several studies of the statistical properties of English text have used the UNIX system and UNIX programming tools. This paper describes several of the useful UNIX facilities for statistical studies and summarizes some studies that have been made at the character level, the character-string level, and the level of English words. The descriptions give a sample of the results obtained and constitute a short introduction, by case-study, on how to use UNIX tools for studying the statistics of English.*

I. INTRODUCTION

The UNIX system is an especially friendly environment in which to do statistical studies of English text. The file system does not impose arbitrary limits on what can be done with different kinds of files and allows tools to be written to apply to files of text, files of text statistics, etc. Pipes and filters allow small steps of processing to be combined and recombined to effect very diverse purposes, almost as English words can be recombined to express very diverse thoughts. The C language, native to the UNIX system, is especially convenient for programs which manipulate characters. Finally, an accidental but important fact is that many UNIX systems are heavily used for document preparation, thus ensuring the ready availability of text for practicing techniques and sharpening tools.

This paper gives short reports on several different statistical

* UNIX is a trademark of Bell Laboratories.

projects as examples of the way UNIX tools can be used to gather statistics describing text. A section describing briefly some of the more important tools used in all the projects is followed by three sections dealing with a variety of studies. The studies are divided according to the level of atomic unit they consider: characters, character strings, and English words. The order of sections is also in almost the chronological order of when the projects were done; future work will almost surely push forward toward more and more meaningful treatment of English.

II. TOOLS FOR GATHERING STATISTICS

2.1 Word breakout

Throughout this paper, *word* means a character string. Different words are made up of different characters or characters in a different order. For example, *man* and *men* are different words; *cat* and *cat's* are different words. We have arbitrarily taken hyphens to be word delimiters, so that *single-minded* is two words: *single* and *minded*. An apostrophe occurring within an alphabetic string is part of the word; an apostrophe before or after a word is not. Digits are discarded. Upper- and lower-case characters are considered to be identical, so that *The* and *the* are the same word. All these decisions could be made differently; the authors believe that the events are rare enough that no substantive conclusions would be changed.

The program that implements the definition of word just given is **prep**. It takes a file of text in ordinary form and converts it into a file containing one word per line. Throughout the rest of this paper, "word" will mean one line of a **prep** output file.

Optionally, **prep** will split out only words on a given list, or all the words not on a given list:

only option:	**prep -o list**
ignore option:	**prep -i list**

Another option which will be referred to below is the **-d** option, which gives the sequence number of each output word in the running input text.

2.2 Sorting

Central to almost all the examples in the rest of the paper is the

sort program. sort is implemented as a filter; that is, it takes its input from the standard input, sorts it, and writes the sorted result to the standard output. The ability to send sorted output easily to a terminal, a file, or through another program is essential to make statistics-gathering convenient. The same sort program works on either letters or numbers. Among the many other features of the sort program which are used in the following are the flags:

-n: sort a leading field numerically
-r: sort in reverse order (largest first)
-u: discard duplicate lines

The sorting method used is especially well adapted to the kind of files dealt with in statistical investigations of text. Its skeleton, which decides which elements to compare, takes advantage of repetition of values in the file to be sorted. The algorithm used for in-core sorting is a version of Quicksort which has been modified to run faster when values in the input are repeated. The standard version of Quicksort requires $n \log n$ comparisons, where n is the number of input items; the UNIX version requires at most $n \log m$ comparisons, where m is the number of *distinct* input values.

2.3 Counting

Another tool of interest for many statistics-gathering processes is a program named uniq. Its fundamental action is to take a sorted file and produce an output containing exactly one instance of each *different* line in the file. (This process simply duplicates the action of sort with the -u option; it runs much more quickly if a sorted file is already available.) More often useful is its ability to count and report the number of occurrences of each of the output lines (uniq -c).

A very generally useful tool is the program wc. It simply counts the number of lines, words, and characters in a file. Throughout any investigation of text statistics, the question arises again and again: How many? Either as a command itself or as the last filter in a chain of pipes, wc is invaluable for answering these questions.

2.4 Searching and pattern-matching

A program of common use for several purposes is grep. grep searches through a file or files for the occurrence of strings of characters which match a pattern. (The patterns are essentially the same

as the editor's patterns and, indeed, the etymology of the name is from the editor command g/r.e./p where r.e. stands for regular expression.) It will print out all matching lines, or, optionally, a count of all matching lines. For example,

```
prep document | grep "^....$" | sort | uniq -c >fours
```

will find all of the four-letter words in document and create a file named fours which contains each such different word along with its frequency of occurrence.

sed, the stream editor, is a program which will not only search for patterns (like grep), but also modify the line before writing it out. So, for example, the following command (using the file fours created by the previous example) will print only the four-letter words which appear exactly once in the document (without the frequency count):

```
sed -n "s/^ *1 //p" fours
```

This ability to search for a given pattern, but to write out the selected information in a different format (e. g., without including the search key), makes sed a useful adhesive to glue together programs which make slightly different assumptions about the format of input and output files.

III. CHARACTER LEVEL

Frequency statistics of English text at the character level have proved useful in the areas of text compression and typographical error correction.

3.1 Compression

Techniques for text compression capitalize on statistical regularity of the text to be compressed, or rather its predictability. The statistical text-processing programs on UNIX have found use in the design and implementation of text-compression routines for a variety of applications.

Suppose that a file of text has been properly formatted so that it does not contain unnecessary leading zeros and trailing blanks and the like, and that it does not devote fixed-length fields to variable-length quantities. Then the most elementary observation that leads to reducing the size of the file is that the possible characters of the character set do not all occur with equal frequency in the text. Most

text uses ASCII or other 8-bit representation for its characters and typically one of these eight bits is never used at all, but one can go much further. If we take as a measure of the information content of a string of characters

$$H=\sum -p_i \log_2 p_i$$

where p_i is the probability of occurrence of the character x_i and the sum is taken over the whole character set, then it is theoretically possible to recode the text so that it requires only H bits per character for its representation. It is possible to find practical methods which come close to but do not attain the value of H. Of course, in deriving this estimate of information content, we have ignored any regularity of the text which extends over more than one character, like digram statistics or words.

It is a simple matter to compute the value of H for any file whether it is a text file or not. The value of H turns out to be very nearly equal to 4.5 for ordinary technical or non-technical English text. This leads immediately to the possibility of recoding the text from ASCII to a variable-length encoding so as to approach a compression to 56 percent of the original length.

Data files other than English text usually have quite different statistics from English text. For example, telephone service orders, parts lists, and telephone directories all have character statistics which are quite different from those of English and different from each other. In general, data files have values of H smaller than 4.5; when they contain a great deal of numerical information, the values of H are often less than 4.

Programs have been written on UNIX to count the occurrences of single letters, digrams and trigrams in text. Single-letter frequencies are kept for all 128 possible ASCII characters. For the digram and trigram statistics, only the 26 letters, the blank, and the newline characters are used, and upper-case letters are mapped to lower case.

The result of running this program on a rather large body of text is shown in Table I. The input was nine separate documents with a total of 213,553 characters and 36,237 words. The documents consisted of three of the Federalist Papers, each by a different author, an article from this journal, a technical paper, a sample from Mark Twain, and three samples of graded text on different topics.

Some interesting (but not novel) observations about the nature of English text can be made from these results. At the single-character level, some characters appear in text far more often than others. In fact, the 10 most frequent characters constitute 70.6 percent of the

Table I—English text statistics

Sample character, digram, and trigram counts for a sample of English text. The counts are truncated after the first 25 entries. 012 is the newline character. □ in the character column is a space character; in the digram and trigram columns, it is any word separation character.

count	character	cum. %	count	digram	count	trigram
33310	□	15.5	6156	e□	3661	□th
21590	e	25.7	5364	□t	3617	the
16080	t	33.2	4998	th	2504	he□
13260	a	39.4	4099	he	1416	□of
12584	o	45.3	3801	□a	1353	of□
12347	n	51.1	3748	s□	1301	□in
12200	i	56.8	3367	in	1249	and
10997	s	61.9	2780	er	1225	□an
10640	r	66.9	2757	t□	1144	nd□
7930	h	70.6	2738	d□	1088	□to
6622	l	73.7	2708	re	1027	to□
5929	d	76.5	2666	an	1025	ion
5409	c	79.0	2572	□i	1003	ed□
4524	012	81.2	2517	n□	946	ing
4508	u	83.3	2506	□o	875	ent
4152	m	85.2	2244	on	854	is□
4080	f	87.1	2047	es	851	in□
3649	p	88.8	2025	at	830	tio
3090	g	90.3	1990	en	805	□co
2851	y	91.6	1912	□s	779	re□
2654	w	92.9	1840	y□	747	□a□
2483	b	94.0	1835	ti	734	ng□
1984	,	94.9	1799	nd	709	on□
1884	v	95.8	1723	nt	702	□be
1824	.	96.7	1681	te	701	es□

text and the 20 most frequent characters make up 91.6 percent of the text. At the digram level, of the 784 possible 2-letter combinations, only 70 percent actually occur in the text. More dramatically, at the trigram level, of the 21952 possible combinations, only 4923, or 22.4 percent, occur in the text. One implication is that, instead of the 24 bits used to represent a trigram with an 8-bit character set, a scheme using 13 bits would do, a compression to 54 percent of the original length, using only the fact that less than 2^{13} different trigrams occur in the text. Noting the widely varying frequencies of the trigrams in the text, we can obtain a considerably better compression rate by using a variable-length encoding scheme.

3.2 Spelling error detection

The observation that English text largely consists of a relatively small proportion of the possible trigrams led to the development of a program typo which is used to find typographical errors in documents. A single erroneous keystroke on a typewriter, for example, changes the three trigrams in which it occurs; more often than not,

at least one of the erroneous trigrams will be otherwise extremely rare or nonexistent in English text. The same thing happens in the case of an erroneously omitted or repeated letter. Better performance is obtained when the comparison statistics are taken from the document itself rather than using some set of trigram statistics from English text in general.

typo accumulates the digram and trigram frequencies for a document and uses them to calculate an index of peculiarity for each word in the document.[1] This index reflects the likelihood that the trigrams in the word are from the same source as the trigrams in the document. Words with rare trigrams tend to have higher indexes and are at the top of the list.

On systems large enough and rich enough to keep a large English dictionary on line, the same function of finding likely candidates for spelling correction is performed by a nonstatistical program, spell, which looks up every word in the dictionary. Of course, suffixes like *-ing* and *-ed* must be recognized and properly stripped before the lookup can be done. What is more, very large dictionaries perform poorly because so many misspelled words turn out to be names of Chinese coins or obsolete Russian units of distance. Not surprisingly, the statistically based typo requires little storage and runs considerably faster. Moreover, not all systems have such resources, and typo has proven useful for authors and secretaries in proofreading. A sample of output from the typo program is included as Table II.

IV. STATISTICS OF CHARACTER STRINGS

In this section we consider statistics which take character strings as atomic units, without any reference to the string's use or function as an English word.

4.1 Word-frequency counts

A set of statistics from a text that is frequently collected (often as a base for further work) is a word-frequency count. A list is made of all the different words in the text, together with the number of times each occurs.[2] With the UNIX tools, it is quite convenient to make such a count:

```
prep text-files | sort | uniq -c
```

This command line produces a frequency count sorted in

Table II—Typo output

A portion of the output of the typo program from a 108-page technical document. A total of 30 misspelled words were found, of which 23 occurred in this portion. The misspelled words identified by the author of the document upon scanning the list have been marked by hand.

Apr 12 22:32:11 Possible typo's and spelling errors Page 1

☛	17 nd		14 flexible	☛	5 pesudonym
	17 heretofore		14 flags	☛	5 neames
☛	17 erroronously		14 conceptually	☛	5 namees
☛	16 suer	☛	14 bwaite		5 multiplied
	16 seized		14 broadly		5 interrelationship
☛	16 poiter	☛	14 amy		5 inefficient
	16 lengthy		14 adds		5 icalc
	16 inaccessible		14 accompanying		5 handler
	16 disagreement		13 overwritten		5 flag
☛	16 bwirte		13 occupying		5 exercised
	15 violating		13 lookup	☛	5 erroreous
	15 unaffected		13 flagged		5 dumped
	15 tape	☛	9 iin		5 dump
	15 swapped	☛	8 subrouutine		5 deficiency
	15 shortly		8 adjunct		5 controller
☛	15 mutiliated		7 drawbacks		5 contiguous
	15 multiprogramming	☛	6 thee		5 changing
	15 likewise	☛	6 odification		5 bottoms
	15 datum	☛	6 od	☛	5 bitis
☛	15 dapt		6 indicator		5 ascertain
	15 cumulatively		6 imminent	☛	5 accomodate
	15 consulted		6 formats		4 unnecessarily
	15 consolidation		6 cetera		4 traversing
	15 checking		5 zeros		4 tracing
☛	15 accordinng		5 virtually		4 totally
☛	14 typpical		5 ultimately		4 tops
	14 tabular		5 truncate		4 thirteen
	14 supplying		5 therewith	☛	4 tallyed
	14 subtle		5 thereafter		4 summarized
	14 shortcoming		5 spectre		4 strictly
	14 pivotal		5 rewritten		4 simultaneous
	14 invalid		5 raises		4 retrieval
	14 infrequently		5 prefix		4 quotient

alphabetical order, as in Table IIIa. To obtain the count in numerical order (largest first):

```
prep text−files | sort | uniq −c | sort −n −r
```

This is illustrated in Table IIIb.

4.2 Dictionary compression

A more complex but considerably more profitable approach to text compression is based on word frequencies. Text consists in large part of words; these words are easy to find in the text; the total number of different words in a text is several orders of magnitude

Table III—Word frequency counts

The beginning of (a) alphabetically sorted and (b) numerically sorted word frequency counts for an early draft of this paper.

(a)		(b)	
124	a	321	the
3	ability	212	of
3	about	124	a
3	above	114	in
1	abrupt	105	to
1	abstract	80	is
1	accidental	78	and
1	according	65	words
1	accordingly	63	text
1	account	50	for

less than the total possible number of arbitrary character strings of the same length. The approach can best be visualized by supposing that a file of text consists entirely of a sequence of English words. Then we can look up each word in a dictionary and replace each word in the text by the serial number of the word in the dictionary. Since a dictionary of reasonable size contains only about 2^{16} words, we have found an immediate and trivial method to recode English text so as to occupy 16 bits per word. Since the average length of a word in text, including the blank after it, is 6 characters, we have a representation that requires only about 2.7 bits per character. This implies a compression to 37 percent of original length. Some details, of course, could not be neglected in actual practice, like capitalization, punctuation, and the occurrence of names, abbreviations, and the like. It turns out, however, that these are sufficiently rare in ordinary running text that only about two or three extra bits per word are required, on the average, to handle them and it is possible to attain a representation requiring only about 3 bits per original character.

In the case of technical text, it is profitable to find the words from the text itself, and store them, each word once, in the compressed file. When this is done, the total number of different words is rather small and because of the tendency of technical authors to use a small technical vocabulary very heavily, the performance is very good. If the dictionary is stored in the file, then the compression performance depends on the number of times each word is used in the text. Suppose there is a word in the text which is m characters long and occurs n times. Then, the occurrences of that word occupy $m \times n$ characters in the original text, whereas in the compressed text, m characters are used for the one dictionary entry and $n \times k$ bits are used as a dictionary pointer each time the word occurs in the text,

where k is the logarithm (base 2) of the number of dictionary entries.

Of course, words in a text do not occur with equal frequency and it is possible, just as was done with letter statistics, to use a variable-length encoding scheme for the words. The information content of the words in a text can be found by passing the word-frequency count found in the previous section through one more filter:

```
prep file-name | sort | uniq -c | entropy
```

It turns out that, for nontechnical English text, the information content of the words is between 8 and 9 bits per word when it is estimated from the text itself. This implies that a text consisting entirely of a string of English words can generally be compressed to occupy only about 1.5 bits per original character. Needless to say, the amount of processing required to compress and expand text in such a way is usually prohibitively high.

4.3 Specialized vocabulary

A practical application of word-frequency counts arose when colleagues became interested in devising vocabulary tests for Bell System personnel to determine their familiarity with the vocabulary used in Bell System Practices (BSPs) in various areas. It is intuitively clear that the vocabulary used in Bell System Practices differs from the general English vocabulary in several details. Some words, like *the, of, an*, etc., are common in the language in general and in specialized writing; others, like *democracy, love, mother* would be found much more frequently in the language in general than in BSPs; others, like *line, circuit, TTY* would be more frequent in BSPs than in the language generally. What was desired was an automatic procedure which would identify such words without relying on intuition. The general problem proposed was to identify the specialized vocabulary of a specific field; the immediate interest was in words with moderate frequencies in BSPs dealing with a certain area, and which are much less frequent in the language as a whole. It was hoped that familiarity with such words would indicate familiarity with the field.

A word-frequency count of approximately one million words of English text was available. It was made from the text of the Brown Corpus[3] and closely resembles the published frequency count of that corpus. It differs in detail only because we used **prep**'s definition of

Table IV—Indexes (see text) of specialization

Frequency of words in a half-million words of BSPs, frequency in a million words of general English, and the words for (a) words which occur too often in BSPs relative to general English; and (b) words which appear too seldom.

(a)				(b)			
Index	BSP Frequency	English Frequency	Word	Index	BSP Frequency	English Frequency	Word
4362	2585	73	fig	-1005	218	2670	their
4150	2334	8	trunk	-1008	23	1909	what
3933	2643	233	control	-1034	584	3941	have
3541	2216	120	test	-1085	4	1961	said
3399	1950	25	circuit	-1207	132	2719	would
3277	2326	266	b	-1212	18	2252	who
3106	2059	168	equipment	-1234	14439	36472	of
3094	1881	76	frame	-1356	298	3617	they
2990	1684	7	cable	-1395	34	2652	we
2828	1785	104	unit	-1457	2	2619	him
2472	1940	315	line	-1593	1	2858	she
2445	1367	1	ess	-1658	71	3284	were
2418	1479	64	message	-1696	0	3037	her
2213	10707	10098	is	-1716	2389	10596	that
2190	1316	46	wire	-1788	287	4381	but
2133	1892	418	system	-1809	10	3285	you
2129	1443	133	list	-2096	1439	8768	it
2117	1513	178	data	-2502	177	5247	i
2018	2085	612	used	-2797	27	5132	had
1936	1118	18	lamp	-3839	27	6999	his

a word, which is slightly different from that of Kučera and Francis. This frequency count was used as representative of English as a whole.

Also available were approximately a half-million words of BSPs, from plant, station, and ESS (Electronic Switching System) maintenance areas. Frequency counts were made of these three areas separately and of the BSP text as a whole.

An index of peculiarity was defined for each word as follows: The two frequency distributions were considered as a single two-way classification (source by word), and single-degree-of-freedom χ^2 statistics were computed for each word. To distinguish words that appear too frequently in the BSPs from words that appear too seldom, a minus sign was attached to the index when the word appeared less often than might be expected in the BSPs (with reference to the English frequency count). This index has the advantage of automatically taking into account differences in size of the two frequency counts, and also de-emphasizing moderate differences in frequency of words which occur rarely in either set of texts.

Samples of the output are shown in Table IV. The indexes and frequencies are for the entire half million words of BSPs compared with English. The first word in the table, "fig," does not mean that

the BSPs discussed tropical fruit to any extent; it is a sample of the difficulties of defining words as character strings. The word is prep's version of Fig. (as in "Fig. 22"). The next several words are, as expected, nouns which refer to objects prominent in telephony. The occurrence of *is* more frequently in BSPs than would be expected seems to be a comment on the style of the writing, one with many passive constructions and predicate noun or adjective constructions—a quite abstract style.

The general method for comparing the vocabulary of specialized texts with general English text worked well for the specific problem proposed; it allowed our colleagues to choose words for their test conveniently. It also shows promise as a more generally applicable method.

4.4 Readability

The number of *tokens* (N) in a text is the number of running words; the number of *types* (T) is the number of different words. For example, the sentence

The man bit the dog.

contains five tokens, and only four types: *the*, *man*, *bit*, *dog*. For our purposes, the number of tokens in a file is taken to be the number of words in the output file of the prep command. We will call a type which occurs exactly once in a text a *hapax*, from the Greek *hapax legomenon*, meaning occurring only once. The number of hapaxes in a text is H.

Two summary statistics often calculated from word-frequency counts are the type/token ratio T/N and the hapax/type ratio H/T. The T/N ratio gives the average repetition rate for words in a text; its usefulness is limited by its erratic decrease with increasing N. The H/T ratio also varies with N, but more slowly and less erratically. We have found it to be of interest in investigations of readability.

Readability is an index which is calculated from various statistics of a text, and is intended to vary inversely with the difficulty of the text for reading. Several such indexes have been proposed; none is universally satisfactory (see, for example, Ref. 4).

In the following discussion, the text used to measure the effectiveness of proposed indexes of readability is taken from an extensive study by Bormuth.[4] He gathered passages from published works in several different fields, which were intended by the

publisher to be appropriate for reading by students at various grade levels. He carefully graded the difficulty of each text by an independent psychological criterion and calculated an index of difficulty from the results of the psychological tests. To judge the effectiveness of indexes calculated from the statistics of the texts themselves, we used two criteria: Bormuth's psychological index and the publishers' assignment of the text to grade level. (The publishers' assignment is in general not based on empirical tests, but on considerable experience and art. It correlates well with Bormuth's empirical measures; we use it simply as a check on oddities that might arise from the specific nature of Bormuth's index.)

One factor which, intuitively, makes some text more difficult to read than other is the speed with which new ideas are introduced. A passage which deals with several ideas in a few words tends to be more difficult to comprehend than a passage of the same length which spends more time developing a single idea. The computer, which of course does not understand the text, cannot measure the number of ideas in a passage. But several statistics regarding the number of different words used, and the number of times they are repeated, might plausibly be expected to vary with the number of ideas in a passage.

Of the statistics related to the breadth of vocabulary in a passage, the H/T ratio was found to correlate best with Bormuth's empirical measure of readability. Over twenty 275-word passages, the correlation is -0.79 with Bormuth's index, 0.77 with grade placement. This correlation is high for a single statistic with an empirical measure of readability, and the correlation remains whether or not the rationale given above is convincing for why there should be a correlation.

We return to readability in the last section of this paper.

4.5 Dispersion of words

A final example of statistics based on words purely as character strings concerns the dispersion of words in text. When an author writes a passage, it is plausible to believe that he has an over-all topic, which unites the passage as a whole. As he proceeds, however, he attends to first one aspect of the topic, then another. It might be expected that as he concentrates on one aspect he would use words from one part of his vocabulary, and when he shifts to another aspect, the vocabulary would also change somewhat. (See Ref. 5, pp. 22-35.) This tendency might be measured by observing

Table V—Separation between words

Mean and standard deviation for the separation (as fractions of the document) between words that occurred exactly twice in documents. *N* is the number of words on which the mean and S.D. are based. The 275 word entries are averages for 4 passages from different sources; the mixed entries are for a concatenation of the four passages; matched entries are for a continuous 1,200-word passage; expected entries are on the hypothesis of random placement of words.

	N	Mean	S. D.
275 word	21	0.26	0.20
mixed	62	0.15	0.16
matched	62	0.32	0.23
expected		0.33	0.24

the distance between repeated occurrences of words. If the tendency to change vocabulary is strong, repeated instances of the same word would be closer together than when the topic and therefore the vocabulary is uniform over an entire passage. In any case, since an English text is presumably an organized sequence of words, the dispersion of words should be less than would be expected for a random placement of the words in a passage.

To gather statistics on the dispersion of words, an option of prep will write the sequence number of each word (in the input stream) on the same line as the word. By using this option together with the -o (only) option, the position in the input text of each occurrence of each word that appears twice, three times, etc. can be written into a file. This file, sorted on the word field, provides input to a simple special-purpose program to calculate the distances between repeated occurrences of the same word. The entire process required writing only one very simple special-purpose program to find the differences between sets of numbers in a file.

Sample results to illustrate the behavior of the statistic are displayed in Table V. The observed and expected means and standard deviations for the separation of words that occurred exactly twice in a text are given as fractions of the length of the text. (The expected fractions are calculated on the hypothesis of random placement of the words in the text.) The line labeled *275 word* gives the average statistics for four passages of about 275 words each, drawn from different biology texts, each on a separate topic. The *mixed* line is statistics from the concatenation of the four texts; the *matched* line gives statistics from a text of the same length as the concatenation, but drawn from a continuous, coherent text. The *expected* line gives the expected separations on the hypothesis of random placement of the words in the text.

As can be seen in Table V, the mean dispersion behaves as

expected; it is smaller than random placement for all texts, but larger for coherent texts than for samples constructed to have abrupt changes of topic. The large standard deviation relative to the mean makes it a difficult statistic to work with, but it shows promise as a measure of uniformity of topic in a passage.

V. ENGLISH WORDS

In this section, we go a short step beyond the character-string orientation of the last section and consider different functional uses of our character strings as English words. Words will still be defined by prep as character strings separated by space or punctuation, but we attend to the way these character strings are used as English words.

5.1 Readability revisited

In the previous section, we considered the correlation of a statistic based on breadth of vocabulary with readability. Another way in which English text can become difficult to read is for an author to use long, complicated constructions which require the reader to follow tortuously through the maze of what is a single sentence, and, therefore, presumably a single thought. (The preceding sentence is offered as a rather modest example of its own referent.) English sentences become long and complicated usually by use of connective words like prepositions (of, from, to, etc.) and conjunctions (and, when, if, although, etc.). Therefore, a list was drawn up of connective words (prepositions and conjunctions), and another list of other function words (auxiliary verbs, articles, demonstratives, etc.). Using prep -o through wc, the number of connectives and the number of other function words in each of twenty graded passages[4] were counted. As expected, reading difficulty as measured both by Bormuth's psychological index and by publishers' grade placement was correlated with both indexes. Number of connectives per token was correlated -0.72 with Bormuth's index score; 0.69 with grade placement. Number of other function words per token was correlated 0.57 with Bormuth's index; -0.50 with grade placement.

The two best predictors considered, H/T ratio and density of connective words, are, alas, highly correlated with each other, so that the multiple correlation of the two predictors is only 0.80 with Bormuth's index and 0.78 with grade placement. This finding that predictors of readability are highly correlated among themselves is

ensured that a large body of text is always at hand for practicing and sharpening tools.

REFERENCES

1. R. Morris and L. L. Cherry, "Computer Detection of Typographical Errors," IEEE Trans. on Professional Communication, *PC-18* (March 1975), pp. 54-56.
2. G. U. Yule, *The Statistical Study of Literary Vocabulary,* Cambridge: The University Press, 1944.
3. H. Kucera and W. Francis, *Computational Analysis of Present-Day American English,* Providence: Brown Univ. Press, 1967.
4. J. C. Bormuth, "Development of Readability Analyses," U. S. Dept. HEW Final Report, Project 7-0052, Contract EC-3-7-070052 0325, Chicago University Press (1969).
5. F. Mosteller and D. L. Wallace, *Inference and Disputed Authorship: The Federalist,* Reading, Mass.: Addison-Wesley, 1964.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

Language Development Tools

By S. C. JOHNSON and M. E. LESK
(Manuscript received December 27, 1977)

The development of new programs on the UNIX system is facilitated by tools for language design and implementation. These are frequently program generators, compiling into C, which provide advanced algorithms in a convenient form, while not restraining the user to a preconceived set of jobs. Two of the most important such tools are Yacc, a generator of LALR(1) parsers, and Lex, a generator of regular expression recognizers using deterministic finite automata. They have been used in a wide variety of applications, including compilers, desk calculators, typesetting languages, and pattern processors.*

I. INTRODUCTION

On the UNIX system, an effort has been made to package language development aids for general use, so that all users can share the newest tools. As a result, these tools have been used to design pleasant, structured applications languages, as well as in their more traditional roles in compiler construction. The packaging is crucial, since if the underlying algorithms are not well packaged, the tools will not be used; applications programmers will rarely spend weeks learning theory in order to use a tool.

Traditionally, algorithms have been packaged as system commands (such as **sort**), subroutines (such as **sin**), or as part of the supported features of a compiler or higher level language environment

* UNIX is a trademark of Bell Laboratories.

(such as the heap allocation in Algol 68). Another way of packaging, which is particularly appropriate in the UNIX operating system, is as a *program generator*. Program generators take a specification of a task and write a program which performs that task. The programming language in which this program is generated (called the *host language*) may be high or low level, although most of ours are high level. Unlike compilers, which typically implement an entire general-purpose source language, program generators can restrict themselves to doing one job, but doing it well.

Program generators have been used for some time in business data processing, typically to implement sorting and report generation applications. Usually, the specifications used in these applications describe the entire job to be done, and the fact that a program is generated is important, but really only one feature of the implementation of the applications package. In contrast, our program generators might better be termed module generators; the intent is to provide a single module that does an important part of the total job. The host language, augmented perhaps by other generators, can provide the other features needed in the application. This approach gains many of the advantages of modularity, as well as the advantages which the advanced algorithms provide. In particular:

(*i*) Each generator handles only one job, and thus is easier to write and to keep up to date.

(*ii*) The user can select exactly the tools needed; one is not forced to accept many unwanted features in order to get the one desired.

(*iii*) The user can also select what manuals have to be read; not only is an unused tool not paid for, but it also need not be learned.

(*iv*) Portability can be enhanced, since only the host language compiler must know the object machine code.

(*v*) Since the interfaces between the tools are well-defined and the output of the tools is in human-readable form, it is easy to make independent changes to the tools and to determine the source of difficulty when a combination of tools fails to work.

Obviously, this all depends on the specific tools fitting together well, so that several can be used in a job. On the UNIX system, this is achieved in a variety of ways. One is the use of *filters,* programs that read one input stream and write one output stream. Filters are easy to fit together; they are simply connected end to end. On the UNIX system, the command line syntax makes it easy to specify a

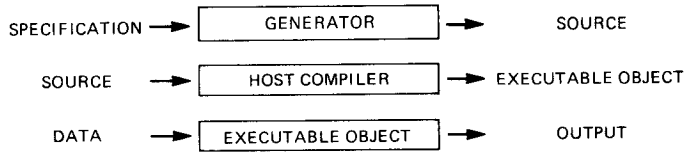

Fig. 1—Program generator.

sequence of commands, each of which uses as its input the output of the preceding command. An example appears in typesetting:

```
refer source-files | tbl | eqn | troff ...
```

where **refer** processes the references, **tbl** the tables, **eqn** the equations, and finally **troff** the text.[1] Each of the first three programs is really a program generator writing code in the host language **troff**, which in turn produces "object code" in the form of typesetter device commands.

This paper focuses on Yacc and Lex. A detailed description of the underlying theory of both programs can be found in Aho and Ullman's book,[2] while the appropriate users' manual can be consulted for further examples and details.[3,4]

Since program generators have output which is in turn input to a compiler and the compiler output is a program which in turn may have both input and output, some terminology is essential. To clarify the discussion, throughout this paper the term *specification* will be used to refer to the input of Yacc or Lex. The output program generated then becomes the *source*, which is compiled by the host language compiler. The resulting *executable object* program may then read *data* and produce *output*. To use a generator:

(*i*) The user writes a specification for the generator, containing grammar rules (for Yacc) or regular expressions (for Lex).
(*ii*) The specification is fed through the generator to produce a source code file.
(*iii*) The source code is processed by the compiler to produce an executable file.
(*iii*) The user's real data is processed by the executable file to produce the real output.

This can be diagrammed as shown in Fig. 1. Both Yacc and Lex accept both C and Ratfor[5] as host languages, although C is far more widely used.

The remainder of this paper gives more detail on the two main program generators, Yacc in Section II and Lex in Section III.

Section IV describes an example of the combined use of both generators to do a simple job, reading a date (month, day, year) and producing the day of the week on which it falls. Finally, Section V contains more general comments about program generators and host languages.

II. THE YACC PARSER GENERATOR

Yacc is a tool which turns a wide class of context-free grammars (also known as Backus-Naur Form, or BNF, descriptions) into parsers that accept the language specified by the grammar. A simple example of such a description might look like

```
date :    month day year ;

month:    "Jan" | "Feb" | "Mar" |
          "Apr" | "May" | "Jun" |
          "Jul" | "Aug" | "Sep" |
          "Oct" | "Nov" | "Dec" ;

day:      number ;

year:     "," number |     ;

number:   DIGIT |
          number DIGIT ;
```

In words, this says that a date is a month, day, and year, in that order; in the Yacc style of writing BNF, colons and semicolons are syntactic connectives that aren't really included in the actual description. The vertical bar stands for "or," so a month is **Jan** or **Feb**, and so on. Quoted strings can stand for literal appearances of the quoted characters. A day is just a number (discussed below). A year is either a comma followed by a number, or it can in fact be missing entirely. Thus, this example would allow as a date either *Jul 4, 1776*, or *Jul 4*.

The two rules for **number** say that a number is either a single digit, or a number followed by a digit. Thus, in this formulation, the number *123* is made up of the number *12*, followed by the digit *3*; the number *12* is made up of the number *1* followed by the digit *2*; and the number *1* is made up simply of the digit *1*.

Using Yacc, an action can be associated with each of the BNF

rules, to be performed upon recognizing that rule. The actions can be any arbitrary program fragments. In general, some value or meaning is associated with the components of the rule and part of the job of the action for a rule is to compute the value or meaning to be associated with the left side of the current rule. Thus, a mechanism has been provided for these program fragments to obtain the values of the components of the rule and return a value. Using the **number** example above, suppose a value has been associated with each possible DIGIT; the value of *1* is 1, etc. The rules describing the structure of numbers can be followed by associated program fragments which compute the meaning or value of the numbers. Assuming that numbers are decimal, then the value of a number which is a single digit is just the value of the digit, while the value of a number which is a number followed by a digit is 10 times the value of the number, plus the value of the digit. In order to specify the values of numbers, we can write:

```
number      : DIGIT
                        { $$ = $1; }
            | number DIGIT
                        { $$ = 10 * $1 + $2; }
            ;
```

Notice that the values of the components of the right-hand sides of the rule are described by the *pseudo-variables* **$1**, **$2**, etc. which refer to the first, second, etc. elements of the right side of the rule. A value is returned for the rule by assigning to the pseudo-variable **$$**. After writing the above actions, the other rules which use **number** will be able to access the value of the number.

Recall that the values for the digits were assumed known. In practice, BNF is rarely used to describe the complete structure of the input. Usually a previous stage, the *lexical analyzer*, is responsible for actually reading the input characters and assembling them into *tokens*, the basic input units for the BNF specification. Lex, described in the next section, is used to help build lexical analyzers; among the issues usually dealt with in the Lex specification are the assembly of alphabetic characters into names, the recognition of classes of characters (such as DIGITS), and the treatment of blanks, newlines, comments, and other similar issues. In particular, the lexical analyzer will be able to associate values to the tokens which it represents, and these values will be accessible in the BNF specification.

The programs generated by Yacc, called *parsers*, read the input

data and associate the rules and actions of the BNF to this input, or report an error if there is no correct association. If the above BNF example is given to Yacc, together with an appropriate lexical analyzer, it will produce a program that will read dates and only dates, report error if something is read that does not fit the BNF description of a date, and associate the correct actions, values, or meanings to the structures encountered during input.

Thus, parsing is like listening to prose; programmers say, "I've never parsed a thing!" but, in fact, every Fortran READ statement does parsing. Fortran FORMAT statements are simply parser specifications. BNF is very powerful, however, and, what is important in practice, many BNF specifications can be turned automatically into fast parsers with good error detection properties.

Yacc provides a number of facilities that go beyond BNF in the strict sense. For example, there is a mechanism which permits the user some control over the behavior of the parser when an error is encountered. Theoretically, one may be justified in terminating the processing when the data are discovered to be in error, but, in practice, this is unduly hostile, since it leads to the detection of only one error per run. To use such a parser to accept input interactively is totally unacceptable; one often wants to prompt the naive user if input is in error, and encourage correct input. The Yacc facilities for error recovery are used by including additional rules, in addition to those which specify the correct input. These rules may use the special token *error*. When an error is detected, the parser will attempt to recover by behaving as if it had just seen the special *error* token immediately before the token which triggered the error. It looks for the "nearest" rule (in a precise sense) for which the *error* token is legal, and resumes processing at this rule. In general, it is also necessary to skip over a part of the input in order to resume processing at an appropriate place; this can also be specified in the error rule. This mechanism, while somewhat unintuitive and not completely general, has proved to be powerful and inexpensive to implement. As an example, consider a language in which every statement ends with a semicolon. A reasonable error recovery rule might be

```
statement : error ';'
```

which, when added to the specification file, would cause the parser to advance the input to the next semicolon when an error was encountered, and then perform any action associated with this rule. One of the trickiest areas of error recovery is the semantic recovery:

how to repair partially built symbol table entries and expression trees that may be left after an error, for example. This problem is difficult and depends strongly on the particular application.

Yacc provides another very useful facility for specifying arithmetic expressions. In most programming languages, there may be a number of arithmetic operators, such as +, −, /, etc. These typically have an ordering, or *precedence*, associated with them. As an example, the expression

```
a + b * c
```

is typically taken to mean

```
a + ( b * c )
```

because the multiplication operator (*) is of higher precedence or binding power than the addition operator (+). In pure BNF, specification of precedence levels is somewhat indirect and requires a technical trick which, while easy to learn, is nevertheless unintuitive. Yacc provides the ability to write simple rules that specify the parsing of arithmetic expressions except for precedence, and then supply the precedence information about the operators separately. In addition, the left or right associativity can be specified. For example, the sequence

```
%left '+' '-'
%left '*' '/'
```

indicates that addition and subtraction are of lower precedence than multiplication and division, and that all are left associative operators. This facility has been very successful; it is not only easier for the nonspecialist to use, but actually produces faster, smaller parsers.[6,7]

Yacc provides a case history of the packaging of a piece of theory in a useful and effective way. For one thing, while BNF is very powerful it does not do everything. It is important to permit escapes from BNF, to permit real applications that can take advantage of the power of BNF, while having some relief from its restrictions. Allowing a general lexical analyzer and general C programs as actions serves this purpose in Yacc. This in turn is made possible by the packaging of the theory as a program generator; the Yacc system does not have to make available to the user all facilities for lexical analysis and actions, but can restrict itself to building fast parsers, and let these other issues be taken care of by other modules.

It is also possible to enclose the Yacc-generated parser in a larger

program. Yacc translates the user's specification into a program named **yyparse**. This program behaves like a finite automaton that recognizes the user's grammar; it is represented by a set of tables and an interpreter to process them. If the user does not supply an explicit main program, **yyparse** is invoked and it reads and parses the input sequence delivered by the lexical analyzer. If the user wishes, however, a main program can be supplied to perform any desired actions before or after calling the parser, and the parser may be invoked repeatedly. The function value returned by **yyparse** indicates whether or not a legal sentence in the specified language was recognized.

It is also possible for the user to introduce his own code at a lower level, since the Yacc parser depends on a routine **yylex** for its input and lexical analysis. This subroutine may be written with Lex (see the next section) or directly by the user. In either case, each time it is called it must return a lexical token name to the Yacc parser. It can also assign a value to the current token by assigning to the variable **yylval**. Such values are used in the same way as values assigned to the **$$** variables in parsing actions.

Thus the user's code may be placed (*i*) above the parser, in the main program; (*ii*) in the parser, as action statements on rules; and/or (*iii*) below the parser, in the lexical analyzer. All of these are in the same core load, so they may communicate through external variables as desired. This gives even the fussiest programmers enough rope to hang themselves. Note, however, that despite the presence of user code even within the parser, both the finite automaton tables and the interpreter are entirely under the control of, and generated by, Yacc, so that changes in the automaton representation need not affect the user.

In addition to generality, good packaging demands that tools be easy to use, inexpensive, and produce high quality output. Over the years, Yacc has developed increased speed and greater power, with little negative effect on the user community. The time required for Yacc to process most specifications is faster than the time required to compile the resulting C programs. The parsers are also comparable in space and time with those that may be produced by hand, but are typically very much easier to write and modify.

To summarize, Yacc provides a tool for turning a wide class of BNF descriptions into efficient parsers. It provides facilities for error recovery, specification of operator precedence, and a general action facility. It is packaged as a program generator, and requires a lexical analyzer to be supplied. The next section will discuss a

complementary tool, Lex, which builds lexical analyzers suitable for Yacc, and is also useful for many other functions.

III. THE LEX LEXICAL ANALYZER GENERATOR

Lex is similar in spirit to Yacc, and there are many similarities in its input format as well. Like Yacc, Lex input consists of rules and associated actions. Like Yacc, when a rule is recognized, the action is performed. The major differences arise from the typical input data and the model used to process them. Yacc is prepared to recognize BNF rules on input which is made up of tokens. These tokens may represent several input characters, such as names or numbers, and there may be characters in the input text that are never seen by the BNF description (such as blanks). Programs generated by Lex, on the other hand, are designed to read the input characters directly. The model implemented by Lex is more powerful than Yacc at dealing with local information — context, character classes, and repetition — but is almost totally lacking in more global structuring facilities, such as recursion. The basic model is that of the theory of regular expressions, which also underlies the UNIX text editor **ed** and a number of other UNIX programs that process text. The class of rules is chosen so that Lex can generate a program that is a deterministic finite state automaton; this means that the resulting analyzer is quite fast, even for large sets of regular expressions. The program fragments written by the user are executed in the order in which the corresponding regular expressions are matched in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial look-ahead is performed on the input, but the input stream will be backed up to the end of the final string matched, making this look-ahead invisible to the user.

For a trivial example, consider the specification for a program to delete from the input text all appearances of the word *theoretical*.

```
%%
theoretical   ;
```

This specification contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches precisely the string of characters "theoretical." No action is specified, so when these characters are seen, they are ignored. All characters which are not matched by some rule are

copied to the output, so all the rest of the text is copied. To also change *theory* to *practice*, just add another rule:

```
%%
theoretical        ;
theory             printf( "practice" );
```

The finite automaton generated for this source will scan for both rules at once, and, when a match is found, execute the desired rule action.

Lex-generated programs can handle data that may require substantial lookahead. For example, suppose there were a third rule, matching **the**, and the input data was the text string **theoretician**. The automaton generated by Lex would have to read the initial string **theoretici** before realizing that the input will not match **theoretical**. It then backs up the input, matching **the**, and leaving the input poised to read **oretician**. Such backup is more costly than the processing of simpler specifications.

As with Yacc, Lex actions may be general program fragments. Since the input is believed to be text, a character array (called **yytext**) can be used to hold the string which was matched by the rule. Actions can obtain the actual characters matched by accessing this array.

The structure of Lex output is similar to that of Yacc. A function named **yylex** is produced, which contains tables and an interpreter representing a deterministic finite automaton. By default, **yylex** is invoked from the main program, and it reads characters from the standard input. The user may provide his own main program, however. Alternatively, when Yacc is used, it automatically generates calls to **yylex** to obtain input tokens. In this case, each Lex rule which recognizes a token should have as an action

return (*token-number*)

to signal the kind of token recognized to the parser. It may also assign a value to **yylval** if desired.

The user can also change the Lex input routines, so long as it is remembered that Lex expects to be able to look ahead on and then back up the input stream. Thus, as with Yacc, user code may be above, within, and below the Lex automaton. It is even easy to have a lexical analyzer in which some tokens are recognized by the automaton and some by user-written code. This may be necessary when some input structure is not easily specified by even the large class of regular expressions supported by Lex.

The definitions of regular expressions are very similar to those in **qed**[8] and the UNIX text editor **ed**.[9] A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D*. It is also possible to use the standard C language escapes to refer to certain special characters, such as \n for newline and \t for tab. The operators may be used to:

(*i*) Specify a repetition of 0 or more, or 1 or more repetitions of a regular expression: * and +.
(*ii*) Specify that an expression is optional: ?.
(*iii*) Allow a choice of two or more patterns: |.
(*iv*) Match the beginning or the end of a line of text: ^ and $.
(*v*) Match any non-newline character: . (dot).
(*vi*) Group sub-expressions: (and).
(*vii*) Allow escaping and quoting special characters: \ and " .
(*viii*) Define classes of characters: [and].
(*ix*) Access defined patterns: { and }.
(*x*) Specify additional right context: /.

Some simple examples are

```
[0-9]
```

which recognizes the individual digits from 0 through 9,

```
[0-9]+
```

which recognizes strings of one or more digits, and

```
-?[0-9]+
```

which recognizes strings of digits optionally preceded by a minus sign. A more complicated pattern is

```
[A-Za-z][A-Za-z0-9]*
```

which matches all alphanumeric strings with a leading alphabetic

character. This is a typical expression for recognizing identifiers in computer programming languages.

Lex programs go beyond the pure theory of regular expressions in their ability to recognize patterns. As one example, Lex rules can recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by a newline). The latter operator is a special case of the / operator, which indicates trailing context. The expression

```
ab/cd
```

matches the string *ab*, but only if followed by *cd*. Left context is handled in Lex by *start conditions*. In effect, start conditions can be used to selectively enable or disable sets of rules, depending on what has come before.

Another feature of Lex is the ability to handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

(*i*) The longest match is preferred.
(*ii*) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer          keyword action ...;
[a-z]+           identifier action ...;
```

to be given in that order. If the input is *integers*, it is taken as an identifier, because **[a-z]+** matches 8 characters while **integer** matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g., *int*) will not match the expression **integer** and so the identifier interpretation is used.

Note that a Lex program normally partitions the input stream, rather than search for all possible matches of each expression. This means that each character is accounted for once and only once. Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes to be executed whatever rule was next choice after the current rule. The position of the input pointer is adjusted accordingly. In general, REJECT is

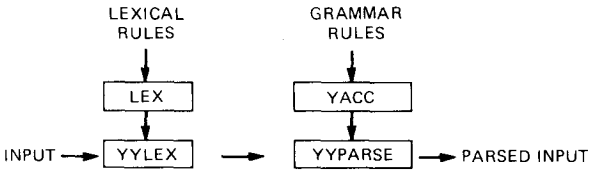

Fig. 2—Yacc and Lex cooperating.

useful whenever the purpose of a Lex program is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other.

IV. COOPERATION OF YACC AND LEX: AN EXAMPLE

This section gives an example of the cooperation of Yacc and Lex to do a simple program which, nevertheless, would be difficult to write directly in many high-level languages. Before the specific example, however, let us summarize the various mechanisms available for making Lex- and Yacc-generated programs cooperate.

Since Yacc generates parsers and Lex can be used to make lexical analyzers, it is often desirable to use them together to make the first stage of a language analyzer. In such an application, two specifications are needed: a set of lexical rules to define the input data tokens and a set of grammar rules to define how these tokens may appear in the language. The input data text is read, divided up into tokens by the lexical analyzer, and then passed to the parser and organized into the larger structures of the input language. In principle, this could be done with pipes, but usually the code produced by Lex and Yacc are compiled together to produce one program for execution. Conventionally, the Yacc program is named **yyparse** and it calls a program named **yylex** to obtain tokens; therefore, this is the name used by Lex for its output source program. The overall appearance is shown in Fig. 2.

To make this cooperation work, it is necessary for Yacc and Lex to agree on the numeric codes used to differentiate token types. These codes can be specified by the user, but ordinarily the user allows Yacc to choose these numbers, and Lex obtains the values by including a header file, written by Yacc, which contains the definitions. It is also necessary to provide a mechanism by which Yacc can obtain the values of tokens returned from Lex. These values are passed through the external variable **yylval**.

Yacc and Lex were designed to work together, and are frequently

used together. The programs using this technology include the portable C compiler and the C language preprocessor.

As a simple example, we shall specify a complete program which will allow the input of dates, such as

July 4, 1776

and it will output the days of the week on which they fall. The program will also permit dates to be input as three numbers, separated by slashes:

7 / 4 / 1776

and in European format:

4 July 1776

Moreover, the month names can be given by their common abbreviations (with an optional '.' following) or spelled in full, but nothing in between.

Conceptually, there are three parts of the program. The Yacc specification describes a list of dates, one per line, in terms of the two tokens DIGIT and MONTH, and various punctuation symbols such as comma and newline. The Lex specification recognizes MONTHs and DIGITs, deletes blanks, and passes other characters through to Yacc. Finally, the Yacc actions call a set of routines which actually carry out the day of the week computation. We will discuss each of these in turn.

```
%token DIGIT MONTH
%%
input :     /* empty file is legal */
      |     input date '\n'
      |     input error '\n'
                  { yyerrok;  /* ignore line if error */ }
      ;
date  :     MONTH day ',' year
                  { date( $1, $2, $4 ); }
      |     day MONTH year
                  { date( $2, $1, $4 ); }
      |     number '/' number '/' number
                  { date( $1, $3, $5 ); }
      ;
day   :     number
      ;
```

```
year    :       number
        ;
number  :       DIGIT
        |       number DIGIT
                        { $$ = 10 * $1 + $2; }
        ;
```

The Yacc specification file is quite simple. The first line declares the two names DIGIT and MONTH as tokens, whose meaning is to be supplied by the lexical analyzer. The %% mark separates the declarations from the rules. The input is described as either empty or some input followed by a date and a newline. Another rule specifies error recovery action in case a line is entered with an illegally formed date; the parser is to skip to the end of the line and then behave as if the error had never been seen.

Dates are legal in the three forms discussed above. In each case, the effect is to call the routine **date**, which does the work required to actually figure out the day of the week. The syntactic categories **day** and **year** are simply numbers; the routine **date** checks them to ensure that they are in the proper range. Finally, numbers are either DIGITs or a number followed by a DIGIT. In the latter case, an action is supplied to return the decimal value of the number. In the case of the first rule, the action

```
{ $$ = $1; }
```

is the implied default, and need not be specified.

Note that the Yacc specification assumes that the lexical analyzer returns values 0 through 9 for the DIGITs, and a month number from 1 to 12 for the MONTHs.

We turn now to the Lex specification.

```
%{
# include "y.tab.h"
extern int yylval;
# define MON(x) {yylval= x; return(MONTH);}
%}
%%
Jan("."|uary)?          MON(1);
Feb("."|ruary)?         MON(2);
Mar("."|ch)?            MON(3);
Apr("."|il)?            MON(4);
May                     MON(5);
```

```
Jun("."|e)?          MON(6);
Jul("."|y)?          MON(7);
Aug("."|ust)?        MON(8);
Sep("."|"t"|"t."|tember)?  MON(9);
Oct("."|ober)?       MON(10);
Nov("."|ember)?      MON(11);
Dec("."|ember)?      MON(12);
[0-9]                {    yylval = yytext[0] - '0';
                          return( DIGIT ); }
[ ]                  {  ;  /* delete blanks */ }
"\n"                 |
.                    {  return( yytext[0] );  /* return
                                single characters */ }
```

The Lex specification includes the file y.tab.h which is produced by Yacc; this defines the token names DIGIT and NUMBER, so they can be used by the Lex program. The variable yylval is defined, which is used to communicate the values of the tokens to Yacc. Finally, to make it easier to return the values of MONTHs the macro MON is defined which assigns its argument to yylval and returns MONTH.

The next portion of the Lex specification is concerned with the month names. Typically, the full month name is legal, as well as the three-letter abbreviation, with or without a following period. The action when a month name is recognized is to set yylval to the number of the month, and return the token indication MONTH; this tells Yacc that a MONTH has been seen. Similarly, the digits 0 through 9 are recognized as a character class, their value stored into yylval, and the indication DIGIT returned. The remaining rules serve to delete blanks, and to pass all other characters, including newline, to Yacc for further processing.

Finally, for completeness, we present the subroutine date which actually carries out the computation. A good fraction of the logic is concerned with leap years, in particular the rather baroque rule that a year is a leap year if it is exactly divisible by 4, and not exactly divisible by 100 unless it is also divisible by 400. Notice also that the month and day are checked to ensure that they are in range.

```
/* here are the routines that really do the work */
# include <stdio.h>
int noleap [] {
```

```
        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
        };
int leap[] {
        0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
        };
char * dayname[] {
        "Sunday",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        };
date( month, day, year ){ /* this routine does the real work */
        int *daysin;
        daysin = isleap( year ) ? leap : noleap;
        /* check the month */
        if( month < 1 || month > 12 ){
                printf( "month out of range\n" );
                return;
                }
        /* check the day of the month */
        if( day < 1 || day > daysin[month] ){
                printf( "day of month out of range\n" );
                return;
                }
        /* now, take the day of the month,
        add the days of previous months */
        while( month > 1 ) day += daysin[ -- month ];
        /* now, make day (mod 7) offset from Jan 1, 0000 */
        if( year > 0 ){
                --year;  /* make corrections for previous years */
                day += year;  /* since 365 = 1 (mod 7) */
                /* leap year correction */
                day += year/4 - year/100 + year/400;
                }
        /* Jan 1, 0000 was a Sunday, so no correction needed */
        printf( "    %s\n", dayname[day%7] );
        }
isleap( year ){
        if( year % 4 != 0 ) return( 0 );  /* not a leap year */
```

```
if( year % 100 != 0 ) return( 1 );  /* is a leap year */
if( year % 400 != 0 ) return( 0 );  /* not a leap year */
return( 1 );  /* a leap year */
}
```

Some of the Lex specification (such as the optional period after month names) might have been done in Yacc. Notice also that some of the things done in Yacc (such as the recognition of numbers) might have been done in Lex. Moreover, additional checking (such as ensuring that days of the month have only one or two digits) might have been placed into Yacc. In general, there is considerable flexibility in dividing the work between Yacc, Lex, and the action programs.

As an exercise, the reader might consider how this program might be written in his favorite programming language. Notice that the Lex program takes care of looking ahead on the input stream, and remembering characters that may delimit tokens but not be part of them. The Yacc program arranges to specify alternative forms and is clearly easy to expand. In fact, this example uses none of the precedence and little of the powerful recursive features of Yacc. Finally, languages such as Snobol in which one might reasonably do the same things as Yacc and Lex do, for this example, would be very unpleasant to write the **date** function in. Practical applications of both Yacc and Lex frequently run to hundreds of rules in the specifications.

V. CONCLUSIONS

Yacc and Lex are quite specialized tools by comparison with some "compiler-writing" systems. To us this is an advantage; it is a deliberate effort at modular design. Rather than grouping tools into enormous packages, enforcing virtually an entire way of life onto a user, we prefer a set of individually small and adaptable tools, each doing one job well. As a result, our tools are used for a wider variety of jobs than most; we have jocularly defined a successful tool as one that was used to do something undreamed of by its author (both Yacc and Lex are successful by this definition).

More seriously, a successful tool must be used. A form of Darwinism is practiced on our UNIX system; programs which are not used are removed from the system. Utilities thus compete for the available jobs and users. Lex, for example, seems to have found an ecological niche in the input phase of programs which accept

complex input languages. Originally it had been thought that it would also be employed for jobs now handled by editor scripts, but most users seem to be sticking with the various editors. Some particularly complicated rearrangements (those which involve memory), however, are done with Lex. Data validation and statistics gathering is still an open area; the editors are unsuitable, and Lex competes with C programs and a new language called **awk**,[10] with no tool having clear dominance. Yacc has a secure role as the major tool now used for the first pass of compilers. It is also used for complex input to many application programs, including Lex, the desk calculator **bc**, and the typesetting language **eqn**. Yacc is also used, with or without Lex, for some kinds of syntactic data validation.

Packaging is very important. Ideally, these tools would be available in several forms. Among the possible modes of access to an algorithm might be a subroutine library, a program generator, a command, or a full compiler. Of these, the program generator is very attractive. It does not restrict the user as much as a command or full compiler, since it is mixed with the user's own code in the host language. On the other hand, since the generator has a reasonable overview of the user's job, it can be more powerful than a subroutine library. Few operating systems today make it possible to have an algorithm available in all forms without additional work, and the program generator is a suitable compromise. The previous compiler-compiler system on UNIX was a more restrictive and inclusive system, TMG,[11] and it is now almost unused. All the users seem to prefer the greater flexibility of the program generators.

The usability and portability of the generators, however, depend on the host language(s). The host language has a difficult task: it must be a suitable target for both user and program generator. It also must be reasonably portable; otherwise, the generator output is not portable. Efficiency is important; the generator must write sufficiently good code that the users do not abandon it to write their own code directly. The host language must also not constrain the generator unduly; for example, mechanically generated **gotos** are not as dangerous as hand-written ones and should not be forbidden. As a result, the best host languages are relatively low level. Another way of seeing this is to observe that if the host language has many complex compiling algorithms in it already, there may not be much scope left for the generator. On the other hand if the language is too low level (after all, the generators typically would have little trouble writing in assembler), the users cannot use it.

What is needed is a semantically low-level but syntactically convenient (and portable) language; C seems to work well.

Giving machine-generated code to a compiler designed for human use sometimes creates problems, however. Compiler writers may limit the number of initializers that can be written to several hundred, for example, since "clearly" no reasonable user would write more than that number of array elements by hand. Unfortunately, Yacc and Lex can generate arrays of thousands of elements to describe their finite automata. Also, if the compiler lacks a facility for adjusting diagnostic line numbers, the error messages will not reflect the extra step in code generation, and the line numbers in them may be unrelated to the user's original input file. (Remember that, although the generated code is presumably error-free, there is also user-written code intermixed).

Other difficulties arise from built-in error checking and type handling. Typically, the generator output is quite reliable, as it often is based on a tight mathematical model or construction. Thus, often one may have nearly perfect confidence that array bounds do not overflow, that defaults in switches are never taken, etc. Nevertheless, compilers which provide such checks often have no way of selectively, or generally, overriding them. In the worst case, this checking can dominate the inner loop of the algorithm embodied in the generated module, removing a great deal of the attractiveness of the generated program.

We should not exaggerate the problems with host languages: in general, C has proven very suitable for the job. The concept of splitting the work between a generator and a host language is very profitable for both sides; it relieves pressure on the host language to provide many complex features, and it relieves pressure on the program generators to turn into complete general-purpose languages. It encourages modularity; and beneath all the buzzwords of "top-down design" and "structured programming," modularity is really what good programming is all about.

REFERENCES

1. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," B.S.T.J., this issue, pp. 2115-2135.
2. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Reading, Mass.: Addison-Wesley, 1977.
3. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories (July 1975).
4. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories (October 1975).

5. B. W. Kernighan, "Ratfor — A Preprocessor for a Rational Fortran," Software — Practice and Experience, *5* (1975), pp. 395-406.
6. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," Commun. Assn. Comp. Mach., *18* (August 1975), pp. 441-452.
7. A. V. Aho and S. C. Johnson, "LR Parsing," Comp. Surveys, *6* (June 1974), pp. 99-124.
8. B. W. Kernighan, D. M. Ritchie, and K. Thompson, "QED Text Editor," Comp. Sci. Tech. Rep. No. 5, Bell Laboratories (May 1972).
9. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975, section ed(I).
10. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, unpublished work (1977).
11. R. M. McClure, "TMG—a Syntax Directed Compiler," Proc. 20th ACM National Conf. (1965), pp. 262-274.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U.S.A.

UNIX Time-Sharing System:

The Programmer's Workbench

By T. A. DOLOTTA, R. C. HAIGHT, and J. R. MASHEY
(Manuscript received December 5, 1977)

Many, if not most, UNIX systems are dedicated to specific projects and serve small, cohesive groups of (usually technically oriented) users. The Programmer's Workbench UNIX system (PWB/UNIX for short) is a facility based on the UNIX system that serves as a large, general-purpose, "utility" computing service. It provides a convenient working environment and a uniform set of programming tools to a very diverse group of users. The PWB/UNIX system has several interesting characteristics:*

(i) Many of its facilities were built in close cooperation between developers and users.
(ii) It has proven itself to be sufficiently reliable so that its users, who develop production software, have abandoned punched cards, private backup tapes, etc.
(iii) It offers a large number of simple, understandable program-development tools that can be combined in a variety of ways; users "package" these tools to create their own specialized environments.
(iv) Most importantly, the above were achieved without compromising the basic elegance, simplicity, generality, and ease of use of the UNIX system.

The result has been an environment that helps large numbers of users to get their work done, that improves their productivity, that adapts quickly to their individual needs, and that provides reliable service at a relatively low cost. This paper discusses some of the problems we encountered in building the PWB/UNIX system, how we solved them, how our system is used, and some of the lessons we learned in the process.

* UNIX is a trademark of Bell Laboratories.

I. INTRODUCTION

The Programmer's Workbench UNIX* system (hereafter called PWB/UNIX for brevity) is a specialized computing facility dedicated to supporting large software-development projects. It is a production system that has been used for several years in the Business Information Systems Programs (BISP) area of Bell Laboratories and that supports there a user community of about 1,100 people. It was developed mainly as an attempt to improve the quality, reliability, flexibility, and consistency of the programming environment. The concepts behind the PWB/UNIX system emphasize several ideas:

(*i*) Program development and execution of the resulting programs are two radically different functions. Much can be gained by assigning each function to a computer best suited to it. Thus, as much of the development as possible should be done on a computer dedicated to that task, i.e., one that acts as a "development facility" and provides a superior programming environment. Production running of the developed products very often occurs on another computer, called a "target" system. For some projects, a single system may successfully fill both roles, but this is rare, because most current operating systems were designed primarily for *running* programs, with little thought having been given to the requirements of the program-development process; we did the exact opposite of this in the PWB/UNIX system.

(*ii*) Although there may be several target systems (possibly supplied by different vendors), the development facility should present a single, uniform interface to its users. Current targets for the PWB/UNIX system include IBM System/370 and UNIVAC 1100-series computers; in some sense, the PWB/UNIX system is also a target, because it is built and maintained with its own tools.

(*iii*) A development facility can be implemented on computers of moderate size, even when the target machines consist of very large systems.

Although PWB/UNIX is a special-purpose system (in the same sense that a "front-end" computer is a special-purpose system), it is specialized for use by human beings. As shown in Fig. 1, it provides the interface between program developers and their target

* UNIX is a trademark of Bell Laboratories.

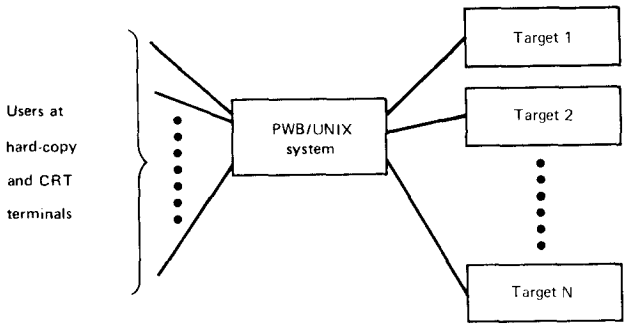

Fig. 1—PWB/UNIX™ interface with its users.

computer(s). Unlike a typical "front-end," the PWB/UNIX system supplies a separate, visible, uniform environment for program-development work.

II. CURRENT STATUS

The PWB/UNIX installation at BISP currently consists of a network of DEC PDP-11/45s and /70s running a modified version of the UNIX system.* By most measures, it is the largest known UNIX installation in the world. Table I gives a "snapshot" of it as of October 1977.

The systems are connected to each other so that each can be backed up by another, and so that files can be transmitted efficiently among systems. They are also connected by communications lines to the following target systems: two IBM 370/168s, two UNIVAC 1100-series systems, and one XDS Sigma 5. Of the card images processed by these targets, 90 to 95 percent are received from PWB/UNIX systems. Average figures for prime-shift connect time

Table I—PWB/UNIX™ hardware at BISP (10/77)

System name	CPU type	Memory (K-bytes)	Disk (M-bytes)	Dial-up ports	Login names
A	/45	256	160	15	153
B	/70	768	480	48	260
D	/70	512	320	48	361
E	/45	256	160	20	114
F	/70	768	320	48	262
G	/70	512	160	48	133
H	/70	512	320	48	139
Totals	—	3,328	1,920	275	1,422

* In order to avoid ambiguity, we use in this paper the expression "Research UNIX system" to refer to the UNIX system itself (Refs. 1 and 2).

The approach embodied in the PWB/UNIX system offers significant advantages in the presence of certain conditions, all of which existed at the original PWB/UNIX installation, thus giving us a strong motivation for adopting this approach. We discuss these conditions below.

4.1 Gain by effective specialization

The computer requirements of software *developers* often diverge quite sharply from those of the *users* of that software. This observation seems especially applicable to software-development organizations such as BISP, i.e., organizations that develop large, data-base-oriented systems. Primary needs of developers include:

(*i*) Interactive computing services that are convenient, inexpensive, and continually available during normal working hours (where often the meaning of the expression "normal working hours" is "22 hours per day, 7 days per week").
(*ii*) A file structure designed for convenient interactive use; in particular, one that never requires the user to explicitly allocate or compact disk storage, or even to be aware of these activities.
(*iii*) Good, uniform tools for the manipulation of documents, source programs, and other forms of text. In our opinion, all the tasks that make up the program-development process and that are carried out by computers are nothing more than (sometimes very arcane) forms of text processing and text manipulation.
(*iv*) A command language simple enough for everyone to use, but one that offers enough programming capability to help automate the operational procedures used to track and control project development.
(*v*) Adaptability to frequent and unpredictable changes in location, structure, and personnel of user organizations.

On the other hand, users of the end products may have any or all of the following needs:

(*i*) Hardware of the appropriate size and speed to run the end products, possibly under stringent real-time or deadline constraints.
(*ii*) File structures and access methods that can be optimized to handle large amounts of data.
(*iii*) Transaction-oriented teleprocessing facilities.

(*iv*) The use of a specific type of computer and operating system, to meet any one of a number of possible (often externally imposed) requirements.

Few systems meet all the requirements of both developers and users. As a result, it is possible to make significant gains by providing two separate kinds of facilities and optimizing each to match one of two distinct sets of requirements.

4.2 Availability of better software

Time-sharing systems that run on large computers often retain significant vestiges of batch processing. Separation of support functions onto an appropriate minicomputer may offer an easy transition to more up-to-date software. Much of the stimulus for PWB/UNIX arose from the desire to make effective use of the UNIX system, whose facilities are extremely well matched to the developers' needs discussed above.

4.3 Installations with target systems from different vendors

It is desirable to have a uniform, target-independent set of tools to ease training and to permit the transfer of personnel between projects. File structures, command languages, and communications protocols differ widely among targets. Thus, it is expensive, if not impossible, to build a single set of effective and efficient tools that can be used on all targets. Effort is better expended in building a single good development facility.

4.4 Changing environments

Changes to hardware and software occur and cause problems even in single-vendor installations. Such changes may be disastrous if they affect both development and production environments at the same time. The problem is at least partially solved by using a separate development system. As an example, in the last few years, every BISP target system has undergone several major reconfigurations in both hardware and software, and the geographic work locations of most users have changed, in some cases more than once. The availability of the PWB/UNIX system often has been able to minimize the impact of these changes on the users.

4.5 Effective testing of terminal-oriented systems

It is difficult enough to test small batch programs; effective testing of large, interactive, data-base management applications is far more difficult. It is especially difficult to perform load testing when the same computer is both generating the load and running the program being tested. It is simpler and more realistic to perform such testing with the aid of a separate computer.

V. DESIGN APPROACH

In early 1974, much thought was given to what should be the overall design approach for the PWB/UNIX system. One proposal consisted of first designing it as a completely integrated facility, then implementing it, and finally obtaining users for it. A much different, less traditional approach was actually adopted; its elements were:

(*i*) Follow the UNIX system's philosophy of building small, independent tools rather than large, interrelated ones. Follow the UNIX system's approach of minimizing the number of different file formats.

(*ii*) Get users on the system quickly, work with them closely, and let their needs and problems drive the design.

(*iii*) Build software quickly, and expect to throw much of it away, or to have to adapt it to the users' real needs, as these needs become clear. In general, emphasize the ability to adapt to change, rather than try to build perfect products that are meant to last forever.

(*iv*) Make changes to the UNIX system only after much deliberation, and only when major gains can be made. Avoid changing the UNIX system's interfaces, and isolate any such changes as much as possible. Stay close to the Research UNIX system, in order to take advantage of continuing improvements.

This approach may appear chaotic, but, in practice, it has worked better than designing supposedly perfect systems that turn out to be obsolete or unusable by the time they are implemented. Unlike many other systems, the UNIX system both permits and encourages this approach.

VI. DIFFERENCES BETWEEN RESEARCH UNIX AND PWB/UNIX

The usage and operation of the PWB/UNIX system differ somewhat

from those of most UNIX systems within Bell Laboratories. Many of the changes and additions described below derive from these crucial differences.

A good many UNIX (as opposed to PWB/UNIX) systems are run as "friendly-user" systems, and are each used by a fairly small number of people who often work closely together. A large fraction of these users have read/write permissions for most (or all) of the files on the system, have permission to add commands to the public directories, are capable of "re-booting" the operating system, and even know how to repair damaged file systems.

The PWB/UNIX system, on the other hand, is most often found in a computer-center environment. Larger numbers of users are served, and they often represent different organizations. It is undesirable for everyone to have general read/write permissions. Although groups of users may wish to have sets of commands and files whose use they share, too many people must be served to permit everyone to add commands to public directories. Few users write C programs, and even fewer are interested in file-system internals. Machines are run by operators who are not expert system programmers. Many users have to to deal with large quantities of existing source code for target computers. Many must integrate their use of the PWB/UNIX system into existing procedures and working methods.

Notwithstanding all the above problems, we continually made every attempt to retain the "friendly-user" environment wherever possible, while extending service to a large group of users characterized by a very wide spectrum of needs, work habits, and usage patterns. By and large, we succeeded in this endeavor.

VII. NEW FACILITIES

A number of major facilities had to be made available in the PWB/UNIX system to make it truly useful in the BISP environment. Initial versions of many of these additional components were written and in use during early 1974. This section describes the current form of these additions (most of which have been heavily revised with the passage of time).

7.1 Remote job entry

The PWB/UNIX Remote Job Entry (RJE) subsystem handles the problems of transmitting jobs to target systems and returning

output to the appropriate users; RJE per se consists of several components, and its use is supported by various other commands.

The **send** command is used to generate job streams for target systems; it is a form of macro-processor, providing facilities for file inclusion, keyword substitution, prompting, and character translation (e.g., ASCII to EBCDIC). It also includes a generalized interface to other UNIX commands, so that all or parts of job streams can be generated dynamically by such commands; **send** offers the users a uniform job-submission mechanism that is almost entirely target-independent.

A transmission subsystem exists to handle communications with each target. "Daemon" programs arrange for queuing jobs, submitting these jobs to the proper target, and routing output back to the user. Device drivers are included in the operating system to control the physical communications links. Some of the code in this subsystem is target-specific, but this subsystem is not visible to end users.

Several commands are used to provide status reporting. Users may inquire about the status of jobs on the target systems, and can elect to be notified in various ways (i.e., on-line or in absentia) of the occurrence of major events during the processing of their jobs.

A user may route the target's output to a remote printer or may elect to have part or all of it returned to the originating PWB/UNIX system. On return, output may be processed automatically by a user-written procedure, or may be placed in a file; it may be examined with the standard UNIX editor, or it can be scanned with a read-only editor (the "big file scanner") that can peruse larger files; RJE hides from the user the distinction between PWB/UNIX files, which are basically character-oriented, and the files of the target system, which are typically record-oriented (e.g., card images and print lines). See Ref. 5 for examples of the use of RJE.

7.2 Source code control system

The PWB/UNIX Source Code Control System (SCCS) consists of a small set of commands that can be used to give unusually powerful control over changes to *modules* of text (i.e., files of source code, documentation, data, or any other text). It records every change made to a module, can recreate a module as it existed at any point in time, controls and manages any number of

concurrently existing versions of a module, and offers various audit and administrative features.[6]

7.3 Text processing and document preparation

One of the distinguishing characteristics of the Research UNIX system is that, while it is a general-purpose time-sharing system, it also provides very good text-processing and document-preparation tools.[7] A major addition in this area provided by the PWB/UNIX system is PWB/MM, a package of formatting "macros" that make the power of the UNIX text formatters available to a wider audience; PWB/MM has, by now, become the de facto Bell Laboratories standard text-processing macro package; it is used by hundreds of clerical and technical employees. It is an easily observable fact that, regardless of the initial reasons that attract users to the PWB/UNIX system, most of them end up using it extensively for text processing. See Ref. 8 for a further discussion of this topic.

7.4 Test drivers

The PWB/UNIX system is often used as a simulator of interactive terminals to execute various kinds of tests of IBM and UNIVAC data-base management and data communications systems, and of applications implemented on these systems; it contains two test drivers that can generate repeatable tests for very complex systems; these drivers are used both to measure performance under well-controlled load and to help verify the initial and continuing correct operation of this software while it is being built and maintained. One driver simulates a *TELETYPE*® CDT cluster controller of up to four terminals, and is used to test programs running on UNIVAC 1100-series computers. The other (LEAP) simulates one or more IBM 3270 cluster controllers, each controlling up to 32 terminals. During a test, the actions of each simulated terminal are directed by a *scenario,* which is a specification of what *scripts* should be executed by that terminal. A script consists of a set of actions that a human operator might perform to accomplish some specific, functional task (e.g., update of a data-base record). A script can be invoked one or more times by one or more scenarios. High-level programming languages exist for both scripts and scenarios; these languages allow one to specify the actions of the simulated terminal-operator pairs, as well as a large

variety of test-data recording, error-detection, and error-correction actions. See Ref. 9 for more details on LEAP.

VIII. MODIFICATIONS TO THE UNIX SYSTEM

Changes that we made to the UNIX operating system and commands were made very carefully, and only after a great deal of thoughtful deliberation. Interface changes were especially avoided. Some changes were made to allow the effective use of the UNIX system in a computer-center environment. In addition, a number of changes were required to extend the effective use of the UNIX system to larger hardware configurations, to larger numbers of simultaneous users, and to larger organizations sharing the machines.

8.1 Reliability

The UNIX system has generally been very reliable. However, some problems surfaced on PWB/UNIX before showing up on other UNIX systems simply because PWB/UNIX systems supported a larger and heavier time-sharing load than most other installations based on UNIX. The continual need for more service required these systems to be run near the limits of their resources much of the time, causing, in the beginning, problems seldom seen on other UNIX systems. Many such problems arose from the lack of detection of, or reasonable remedial action for, exhaustion of resources. As a result, we made a number of minor changes to various parts of the operating system to assure such detection of resource exhaustion, especially to avoid crashes and to minimize peculiar behavior caused by exceeding the sizes of certain tables.

The first major set of reliability improvements concerned the handling of disk files. It is a fact of life that time-sharing systems are continually short of disk space; PWB/UNIX is especially prone to rapid surges in disk usage, due to the speed at which the RJE subsystem can transfer data and use disk space. Experience showed that reliable operation requires RJE to be able to suspend operations temporarily, rather than throwing away good output. The **ustat** system call was added to allow programs to discover the amount of free space remaining in a file system. Such programs could issue appropriate warnings or suspend operation, rather than attempt to write a file that would consume all

the free disk space and be, itself, truncated in the process, causing loss of precious data; the **ustat** system call is also used by the PWB/UNIX text editor to offer a warning message instead of silently truncating a file when writing it into a file system that is nearly full. In general, the relative importance of files depends on their cost in terms of human effort needed to (re)generate them. We consider information typed by people to be more valuable than that generated mechanically.

A number of operational procedures were instituted to improve file-system reliability. The main use of the PWB/UNIX system is to store and organize files, rather than to perform computations. Therefore, every weekday morning, each user file is copied to a backup disk, which is saved for a week. A weekly tape backup copy is kept for two months; bimonthly tape copies are kept "forever"—we still have the tapes from January 1974. The disk backup copies permit fast recovery from disk failure or other (rare) disasters, and also offer very fast recovery when individual user files are lost; almost always, such files are lost not because of system malfunctions, but because people inevitably make mistakes and delete files that they really wish to retain. The long-term tape backup copies, on the other hand, offer users the chance to delete files that they might want back at some time in the future, without requiring them to make "personal" copies.

A second area of improvement was motivated by the need for reliable execution of long-running procedures on machines that operate near the limits of their resources. Any UNIX system has some bound on the maximum number of processes permitted at any one time. If all processes are used, it is impossible to successfully issue the **fork** system call to create a new process. When this happens, it is difficult for useful work to get done, because most commands execute as separate processes. Such transient conditions (often lasting only a few seconds) do cause occasional, random failures that can be extremely irritating to the users (and, potentially, destroy their trust in the system). To remedy this situation, the shell was changed so that it attempts several **fork** calls, separated from one another by increasing lengths of time. Although this is not a general solution, it did have the practical effect of decreasing the probability of failure to the point that user complaints ceased. A similar remedy was applied to the command-execution failures due to the near-simultaneous attempts by several processes to execute the same pure-text program.

These efforts have yielded production systems that users are willing to trust. Although a file is occasionally lost or scrambled by the system, such an event is rare enough to be a topic for discussion, rather than a typical occurrence. Most users trust their files to the system and have thrown away their decks of cards. This is illustrated by the relative numbers of keypunches (30) and terminals (550) in BISP. Users have also come to trust the fact that their machines stay up and work. On the average, each machine is down once a week during prime shift, averaging 48 minutes of lost time, for total prime-shift availability of about 98 percent. These figures include the occasional loss of a machine for several hours at a time, i.e., for hardware problems. However, the net availability to most users has been closer to 99 percent, because most of the machines are paired and operational procedures exist so that they can be used to back each other up. This eliminates the intolerable loss of time caused by denying to an entire organization access to the PWB/UNIX system for as much as a morning or an afternoon. Such availability of service is especially critical for organizations whose daily working procedures have become intertwined with PWB/UNIX facilities, as well as for clerical users, who may have literally nothing to do if they cannot obtain access to the system.

Thus, users have come to trust the systems to run reliably and to crash very seldom. Prime-shift down-time may occur in several ways. A machine may be taken down voluntarily for a short period of time, typically to fix or rearrange hardware, or for some systems programming function. If the period is short and users are given reasonable notice, this kind of down-time does not bother users very much. Some down-time is caused by hardware problems. Fortunately, these seldom cause outright crashes; rather, they cause noticeable failures in communications activities, or produce masses of console error messages about disk failures. A system can "lock-up" because it runs out of processes, out of disk space, or out of some other resource. An alert operator can fix some problems immediately, but occasionally must take the system down and reinitialize it. The causes and effects of the "resource-exhaustion" problems are fairly well-known and generally thought to offer little reason for consternation. Finally, there is the possibility of an outright system crash caused by software bugs. As of mid-1977, the last such crash on a production machine occurred in November 1975.

8.2 Operations

At most sites, UNIX systems have traditionally been operated and administered by highly trained technical personnel; initially, our site was operated in the same way. Growth in PWB/UNIX service eventually goaded us into getting clerical help. However, the insight that we gained from initially doing the job ourselves was invaluable; it enabled us to perceive the need for, and to provide, operational procedures and software that made it possible to manage a large, production-oriented, computer-center-like service. For instance, a major operational task is "patching up" the file system after a hardware failure. In the worst cases, this work is still done by system programmers, but cases where system recovery is fairly straightforward are now handled by trained clerks. Our first attempt at writing an operator's manual dates from that time.

In the area of system administration, resource allocation and usage accounting have become more formal as the number of systems has grown. Software was developed to move entire sections of a file system (and the corresponding groups of users) from volume to volume, or from one PWB/UNIX system to another without interfering with linked files or access history. A major task in this area has been the formalization and the speeding-up of the file-system backup procedures.

By mid-1975, it was clear that we would soon run out of unique "user-ID" numbers. We resisted user pressure to re-use numbers among PWB/UNIX systems. Our original reason was to preserve our ability to back up each PWB/UNIX system with another one; in other words, the users and files from any system that is down for an extended period should be able to be moved to another, properly configured system. This was difficult enough to do without the complication of duplicated user-IDs. Such backup has indeed been carried out several times. However, the two main advantages of retaining unique user-IDs were:

(*i*) Protecting our ability to move users *permanently* from one system to another for organizational or load-balancing purposes.

(*ii*) Allowing us to develop reasonable means for communicating among several systems without compromising file security.

We return to the subject of user-IDs in Section 8.4 below.

8.3 Performance improvements

A number of changes were made to increase the ability of the PWB/UNIX system to run on larger configurations and support more simultaneous users. Although demand for service almost always outran our ability to supply it, minor tuning was eschewed in favor of finding ways to gain large payoffs with relatively low investment.

For a system such as PWB/UNIX, it is much more important to optimize the use of moving-head disks than to optimize the use of the CPU. We installed a new disk driver* that made efficient use of the RP04 (IBM 3330-style) disk drives in multi-drive configurations. The seek algorithm was rewritten to use one (sorted) list of outstanding disk-access requests per disk drive, rather than just one list for the entire system; heuristic analysis was done to determine what I/O request lead-time yields minimal rotational delay and maximal throughput under heavy load. The effect of these changes and of changes in the organization of the disk free-space lists (which are now optimized by hardware type and load expectation) have nearly tripled the effective multi-drive transfer rate. Current PWB/UNIX systems have approached the theoretical maximum disk throughput. On a heavily loaded system, three moving-head drives have the transfer capacity of a single fixed-head disk of equivalent size. The C program listing for the disk driver is only four pages long; this made it possible to experiment with it and to tune it with relative ease.

Minor changes were made in process scheduling to avoid "hot spots" and to keep response time reasonable, even on heavily loaded systems. Similarly, the scheduler and the terminal driver were also modified to help maintain a reasonable rate of output to terminals on heavily loaded systems. We have consciously chosen to give up a small amount of performance under a light load in order to gain performance under a heavy load.

Several performance changes were made in the shell. First, a change of just a few lines of code permitted the shell to use buffered "reads," eliminating about 30 percent of the CPU time used by the shell. Second, a way was found to reduce the average number of processes created in a day, also by approximately 30 percent; this is a significant saving, because the creation of a process and the activation of the corresponding program typically

* Written by L. A. Wehr.

require about 0.1 second of CPU time and also incur overhead for I/O. To accomplish this, shell accounting data were analyzed to investigate the usage of commands. Each PWB/UNIX system typically had about 30,000 command executions per day. Of these, 30 percent resulted from the execution of just a few commands, namely, the commands used to implement flow-of-control constructs in shell procedures. The overhead for invoking them typically outweighed their actual execution time. They were absorbed (without significant changes) into the shell. This reduced somewhat the CPU overhead by eliminating many **fork** calls. Much more importantly, it reduced disk I/O in several ways: swapping due to **fork**s was reduced, as was searching for commands; it also reduced the response time perceived by users executing shell procedures—the improvement was enough to make the use of these procedures much more practical. These changes allowed us to provide service to many more users without degrading the response time of our systems to an unreasonable degree.

The most important decision that we made in this entire area of reliability and performance was our conscious choice to keep our system in step with the Research UNIX system; its developers have been most helpful: they quickly repaired serious bugs, gave good advice where our needs diverged from theirs, and "bought back" the best of our changes.

8.4 User environment

During 1975, a few changes that altered the user environment were made to the operating system, the shell, and a few other commands. The main result of these changes was to more than double the size of the user population to which we could provide service without doing major harm to the convenience of the UNIX system. In particular, several problems had to be overcome to maintain the ease of sharing data and commands. This aspect of the UNIX system is popular with its users, is especially crucial for groups of users working on common projects, and distinguishes the UNIX system from many other time-sharing systems, which impose complete user-from-user isolation under the pretense of providing privacy, security, and protection.

Initially, the UNIX system had a limit of 256 distinct user-IDs;[1] this was adequate for most UNIX installations, but totally inadequate for a user population the size of ours. Various solutions were studied, and most were rejected. Duplicating user-IDs across

machines was rejected for operational reasons, as noted in Section 8.2 above. A second option considered was that of decreasing the available number of the so-called "group-IDs," or removing them entirely, and using the bits thus freed to increase the number of distinct user-IDs. Although attractive in many ways, this solution required a change in the interpretation of information stored with every single disk file (and every backup copy thereof), changes to large numbers of commands, and a fundamental departure from the Research UNIX system during a time when thought was being given to possible changes to that system's protection mechanisms. For these reasons, this solution was deemed unwise.

Our solution was a modest one that depended heavily on the characteristics of the PWB/UNIX user community, which, as mentioned above, consists mostly of groups of cooperating users, rather than of individual users working in isolation from one another. Typical behavior and opinions in these groups were:

(*i*) Users in such a group cared very little about how much protection they had from each other, as long as their files were protected from damage by users outside their group.

(*ii*) A common password was often used by members of a group, even when they owned distinct user-IDs. This was often done so that a needed file could be accessed without delay when its owner was unavailable.

(*iii*) Most users were willing to have only one or two *user-IDs* per group, but wanted to retain their own *login names* and *login directories*. We also favored such a distinction, because experience showed that the use of a single login name by more than a few users almost always produced cluttered directory structures containing useless files.

(*iv*) Users wanted to retain the convenience of inter-user communication through commands (e.g., **write** and **mail**) that automatically identified the sending person.

The Research UNIX **login** command maps a login name into a user-ID, which thereafter identifies that user. Because the mapping from login name to user-ID is many-to-one in PWB/UNIX, a given user-ID may represent many login names. It was observed that the **login** command knew the login name, but did not record it in a way that permitted consistent retrieval. The login name was added to the data recorded for each process and the **udata**

system call was added to set or retrieve this value; the **login** command was modified to record the login name and a small number of other commands (such as **write** and **mail**) were changed to obtain the login name via the **udata** system call. Finally, to improve the security of files, a few commands were changed to create files with read/write permission for their owners, but read-only for everyone else. The net effect of these changes was to greatly enlarge the size of the user community that could be served, without destroying the convenience of the UNIX system and without requiring widespread and fundamental changes.

The second problem was that of sharing commands. When a command is invoked, the shell first searches for it in the current directory, then in directory **/bin**, and finally in directory **/usr/bin**. Thus, any user may have private commands in one of his or her private directories, while **/bin** is a repository for the most frequently used public commands, and **/usr/bin** usually contains less frequently used public commands. On many systems, almost anyone can install commands in **/usr/bin**. Although this is practical for a system with twenty or so users, it is unworkable for systems with 200 or more, especially when a number of unrelated organizations share a machine. Our users wanted to create their own commands, invoked in the same way as public commands. Users in large projects often wanted several sets of such commands: project, department, group, and individual.

The solution in this case was to change the shell (and a few other commands, such as **nohup** and **time**) to search a *user-specified* list of directories, instead of the existing fixed list. In order to preserve the consistency of command searching across different programs, it was desirable to place a user-specified list where it could be accessed by any program that needed it. This was accomplished through a mechanism similar to that used for solving the previous problem. The **login** command was changed to record the name of the user's login directory in the per-process data area. Each user could record a list of directories to be searched in a file named **.path** in his or her login directory, and the shell and other commands were changed to read this file. Although a few users wished to be able to change this list more dynamically than is possible by editing the **.path** file, most users were satisfied with this facility, and, as a matter of observed fact, altered that file infrequently. In many projects, the project administrator creates an appropriate **.path** file and then makes

links to it for everyone else, thus ensuring consistency of command names within the project.

These changes were implemented in mid-1975. Their effect was an upsurge in the number of project-specific commands, written to improve project communication, to manage project data bases, to automate procedures that would otherwise have to be performed manually, and, generally, to customize the user environment provided by the PWB/UNIX system to the needs of each project. The result was a perceived increase in user productivity, because our users (who are, by and large, designers and builders of software) began spending significantly less time on housekeeping tasks, and correspondingly more time on their end products; see Ref. 5 for comments on this entire process by some early PWB/UNIX users.

8.5 Extending the use of the shell

A number of extensions were made to the shell to improve its ability to support shell programming, while leaving its user interface as unchanged as possible. These changes were made only after a great deal of trepidation, because they clearly violated the UNIX system's principle of minimizing the complexity of "central" programs, and because they represented a departure from the Research UNIX shell; these departures consisted of minor changes in syntax, but major changes in intended usage.

During 1974 and early 1975, the PWB/UNIX shell was the same as the Research UNIX shell, and its usage pattern was similar, i.e., it was mainly used to interpret commands typed at a terminal and occasionally used to interpret (fairly simple) files of commands. A good explanation of the original shell philosophy and usage may be found in Ref. 10. At that time, shell programming abilities were limited to simple handling of a sequence of arguments, and flow of control was directed by **if**, **goto**, and **exit**—separate commands whose use gave a Fortran-like appearance to shell procedures. During this period, we started experimenting with the use of the shell. We noted that anything that could be written as a shell procedure could always be written in C, but the reverse was often not true. Although C programs almost always executed faster, users preferred to write shell procedures, if at all possible, for a number of reasons:

(*i*) Shell programming has a "shallow" learning curve, because

anyone who uses the UNIX system must learn something about the shell and a few other commands; thus little additional effort is needed to write simple shell procedures.

(*ii*) Shell programming can do the job quickly and at a low cost in terms of human effort.

(*iii*) Shell programming avoids waste of effort in premature optimization of code. Shell procedures are occasionally recoded as C programs, but only after they are shown to be worth the effort of being so recoded. Many shell procedures are executed no more frequently than a few times a day; it would be very difficult to justify the effort to rewrite them in C.

(*iv*) Shell procedures are small and easy to maintain, especially because there are no object programs or libraries to manage.

Experience with shell programming led us to believe that some very modest additions would yield large gains in the kinds of procedures that could be written with the shell. Thus, in mid-1975, we made a number of changes to the shell, as well as to other commands that are used primarily in shell procedures. The shell was changed to provide 26 character-string variables and a command that sets the value of such a variable to an already existing string, or to a line read from the standard input. The **if** command was extended to allow a "nestable" **if-then-else-endif** form, and the **expr** command was created to provide evaluation of character-string and arithmetic expressions. These changes, in conjunction with those described in Section 8.4 above, resulted in a dramatic increase in the use of shell programming. For example, procedures that lessened the users' need for detailed knowledge of the target system's job control language were written for submitting RJE jobs,* groups of commands were written to manage numerous small data bases, and many manual procedures were automated. A more detailed discussion of shell usage patterns (as of June 1976) may be found in Ref. 11.

Further changes have been made since that time, mainly to complete the set of control structures (by adding the **switch** and **while** commands), and also to improve performance, as explained in Section 8.3 above.

---

* Target-system users who interact with these targets via the PWB/UNIX RJE subsystem make about 20 percent fewer errors in their job control statements than those who interact directly with the targets.

Laboratories Computing Science Research Center for creating the UNIX system itself, as well as for their advice and support.

REFERENCES

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Commun. Assn. Comp. Mach., *17* (July 1974), pp. 365-375.
2. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
3. E. L. Ivie, "The Programmer's Workbench—A Machine for Software Development," Commun. Assn. Comp. Mach., *20* (October 1977), pp. 746-753.
4. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 164-168.
5. M. H. Bianchi and J. L. Wood, "A User's Viewpoint on the Programmer's Workbench," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 193-199.
6. M. J. Rochkind, "The Source Code Control System," IEEE Trans. on Software Engineering, *SE-1* (December 1975), pp. 364-370.
7. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," B.S.T.J., this issue, pp. 2115-2135.
8. J. R. Mashey and D. W. Smith, "Documentation Tools and Techniques," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 177-181.
9. T. A. Dolotta, J. S. Licwinko, R. E. Menninger, and W. D. Roome, "The LEAP Load and Test Driver," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 182-186.
10. K. Thompson, "The UNIX Command Language," in *Structured Programming—Infotech State of the Art Report,* Nicholson House, Maidenhead, Berkshire, England: Infotech International Ltd. (March 1975), pp. 375-384.
11. J. R. Mashey, "Using a Command Language as a High-Level Programming Language," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 169-176.
12. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," B.S.T.J., this issue, pp. 1971-1990.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

The UNIX Operating System as a Base for Applications

By G. W. R. LUDERER, J. F. MARANZANO, and B. A. TAGUE
(Manuscript received March 9, 1978)

The intent of this paper is twofold: first, to comment on the general properties of the UNIX operating system as a tool for software product development and as a basis for such products; and second, to introduce the remaining papers of this issue.*

I. A BRIEF HISTORY

Bell Laboratories has employed minicomputers in laboratory work since they first became available. By the early 1970s, several hundred minicomputers were controlling experiments, supporting machine-aided design, providing remote-job-entry facilities for computation centers, and supplying peripheral support for Electronic Switching Systems laboratories. The availability of the C-language version of the UNIX system in 1973 coincided with the emergence of several new factors related to minicomputers at Bell Laboratories:

(*i*) The cost, reliability, and capacity of minicomputers—especially improvements in their peripherals—made applications possible that were previously not economical.
(*ii*) Minicomputer-based systems were being selected for installation in operating telephone companies to assist in the administration and maintenance of the telephone plant.

* UNIX is a trademark of Bell Laboratories.

(*iii*) Many such projects were started in a period of a few months, which meant that many engineers were suddenly shifted into minicomputer software development.

(*iv*) The pressures to develop and install these systems rapidly were very great.

Needless to say, the same factors applied to the laboratory uses of minicomputers, but not on the same scale. Product planning estimates suggested that over 1500 such minicomputer-based support systems would be installed in the Bell System by the early 1980s.

The developers of each of these early minicomputer-based projects had to either write their own operating system or find a suitable system elsewhere. The UNIX operating system had become available from the Bell Laboratories computing research organization, and projects were encouraged to use it for their software development. Most projects originally planned to use the UNIX system to develop their own special-purpose operating systems for deployment with their applications. However, all projects were encouraged to consider the UNIX system for deployment with their applications as well. These early projects found that the UNIX operating system provided excellent programming and documentation tools, and was significantly better than the vendor-supplied alternatives then available; the C language and the UNIX system provided a means to rapid development, test, and installation of their software products. Most of these projects found that the UNIX system, with some local modifications, could be used not only for program development, but also as the base for the product itself.

No central support of the UNIX system—i.e., counseling, training, bug fixing, etc.—was available or promised to these pioneer projects. Documentation, aside from a good user's reference manual and the C language listings, was also lacking. The first projects were handicapped by having to supply their own UNIX system support. In spite of this added load, the UNIX system still proved a better choice than the vendor-supported alternatives. Central support and improved documentation were subsequently provided in 1974.

Many requests for improvements to the UNIX system came from telephone-related development projects using it as the base for their products. These projects wanted increased real-time control capabilities in the UNIX system, as well as improved reliability. Even though the systems produced by these projects were ancillary to the telephone plant, Bell System operating telephone companies increasingly counted on them for running their business. Reliability of

these systems became an important issue, and significant support effort was devoted to improving the detection of hardware failures by the operating system. Mean-time-to-failure was generally adequate for most projects; mean-time-to-repair was the problem.

Improvements were made that allow UNIX processes to communicate and synchronize in real time more easily. However, additional real-time features were needed to control the scheduling and execution of processes. In another research organization, H. Lycklama and D. L. Bayer developed the MERT (Multi Environment Real-Time) operating system. MERT supports the UNIX program-development and text-processing tools, while providing many real-time features. Some projects found MERT to be an attractive alternative to the UNIX system for their products. This led, in 1978, to the support of two versions of UNIX: the time-sharing version (UNIX/TS) and the real-time version (UNIX/RT). UNIX/TS is based on the research version of the UNIX system with additional features found to be useful in a time-sharing environment. UNIX/RT is based on MERT and is tailored to the needs of projects with real-time requirements. The centrally supported UNIX/TS will become the basis for PWB/UNIX as described by Dolotta et al., and will provide computation-center UNIX service. Real-time projects are currently evaluating UNIX/RT, and real-time features for UNIX are still a matter for continuing investigation and study.

The UNIX and MERT operating systems have found wide acceptance both within the Bell System and without. There are currently 48 Bell Laboratories projects using the UNIX system and 18 using MERT. Some 24 of these projects are developing products for use by the Bell System operating telephone companies and have already installed over 300 UNIX system-based products, with the installation rate still accelerating. In addition, another 10 projects are using PWB/UNIX, and many operating companies are evaluating PWB/UNIX for programming and text-processing tasks within their companies. Outside the Bell System, over 300 universities and commercial institutions are using the UNIX operating system. An important by-product of university use is the growing availability of trained UNIX programmers among computer science graduates.

II. WHY UNIX AND C?

Each of the following papers suggests why the UNIX system was selected for a particular purpose, but some common themes are worth emphasizing. As was suggested above, development projects

initially selected the UNIX system as a program-development environment. The earlier paper by Dolotta et al. admirably summarizes the general case for using the UNIX system. Minicomputer-based projects, however, also have incorporated the UNIX system in their final product, and this usage raised new requirements and concerns. We shall attempt to categorize briefly these new requirements and explore their implications.

First, and most obvious, the products of many projects could be considered simply a specialized use of a dedicated time-sharing service; because the UNIX system appeared to be the most efficient available minicomputer time-sharing system, it was an obvious choice for such projects. Its flexibility, generality, and multiprogramming efficiency were just as advantageous for the applications programs as for program developers. Even the existing scheduling and priority mechanisms were adequate for some applications.

Because it was designed for the programming of the UNIX system itself, the C language could also be used to implement even the most complex subsystems. While almost all applications must communicate with special devices unique to their projects, C has an elegant mechanism for solving this problem without special language extensions. It depends upon the fact that the PDP-11 architecture presents all device data and control registers as part of the operating system's virtual address space. These registers can be accessed by assigning absolute constants to pointer variables defined and manipulated by standard C programs.

III. EXTENSIONS FOR REAL-TIME APPLICATIONS

Most projects with real-time applications found the UNIX system wanting in several areas:

(*i*) Interprocess communication.
(*ii*) Efficient handling of large files with known structure.
(*iii*) Communication with special terminals, or using special line protocols.
(*iv*) Priority scheduling of real-time processes.

The general theme is time efficiency; the standard UNIX system already provides all the required functions in some form. What was needed was the ability to "tune" and control these functions in the context of each application. The papers that follow provide three different answers to this problem:

(*i*) Modify and extend the standard UNIX system to provide the additional function or efficiency required.
(*ii*) Adopt MERT, which is a version of the UNIX system restructured to include many real-time functions.
(*iii*) Distribute the application between a central standard UNIX operating system and coupled microprocessors or minicomputers that handle the real-time activities.

Design of real-time applications is a very difficult area, and we expect to see continuing controversy over the best way to handle such applications. The paper by Cohen and Kaufeld argues eloquently why extending the UNIX system was the proper answer for the Network Operations Control System project, while the paper by Nagelberg and Pilla is equally persuasive in arguing that adoption of MERT was the right answer for the Record Base Coordination System project. The papers by Wonsiewicz et al. and Rovegno both describe successful delegation of the real-time aspects of the application to microprocessors connected to the standard UNIX configuration.

The project described by Wonsiewicz et al. is especially relevant. The designers originally selected MERT for their central machine, but in the end they did not use its real-time features. They were able to put all time-sensitive processing into LSI-11 microprocessors in the individual laboratories, and then switched from the MERT system to the UNIX system on their central machine to gain the greater robustness and reliability of UNIX (the MERT system is newer than the UNIX system, more complex, and less well-shaken-down by users).

IV. RELIABILITY AND PORTABILITY

Anything written on minicomputer applications in the telephone system would be remiss if it did not mention the issues of reliability and software portability. The introduction of commercial minicomputer hardware with its 5-year support horizons into the telephone plant that has a 40-year-support tradition raises some obvious questions in the area of reliability and field support. The Bell System investment in applications software must be preserved over long periods of time in the face of rapid evolution of the underlying hardware technology and economics.

The basic reliability of the UNIX software is very good. Note, for example, the comments in Section 8.1 of the paper by Dolotta et al. Using an operating system other than that supplied by the vendor complicates hardware maintenance by the vendor. Effort has been

expended in making the UNIX system report errors to the vendor's field engineers in the language and formats that are used by their own diagnostic software. The UNIX system is being improved in its ability to report its own hardware difficulties, but this cannot be carried very far without redesigning the hardware. None of the current UNIX-based systems in the Bell System operating telephone companies directly handle customer traffic, which significantly reduces the reliability requirements of these systems. To achieve the reliability required of Bell System electronic switching offices would necessitate a large and carefully coordinated hardware and software effort.

One of the goals in using the UNIX operating system in telephone applications is to insulate the applications code as much as possible from hardware variations that are driven by the vendors' marketing goals and convenience. By controlling the UNIX system, applications code can be largely protected from the vagaries of the underlying hardware. Such hardware independence is clearly a matter of degree, although full portability of the UNIX system across several different hardware architectures without affecting applications is the ultimate goal. Currently, the UNIX applications code moves quite easily across the PDP-11 line. Experiments are under way to move UNIX to different vendors' hardware, as described by Johnson and Ritchie earlier in this issue. It is already clear that projects must exercise care in how applications are written if they are to move easily from one vendor's architecture to another. Fortunately, much of the needed "care" is simply good C coding style, but unfortunately, precise, complete rules that guarantee portability are proving both complex and a bit slippery. However, the basic idea of using UNIX as an insulating layer continues to be the most attractive option for preserving minicomputer applications software in the face of hardware changes.

V. THE UNIX APPLICATION PAPERS

The six papers that follow describe a spectrum of applications built upon the UNIX and MERT operating systems. The first paper, by Wonsiewicz et al., describes a system handling the automation of a number of diverse instruments in a materials-science research laboratory. In the second paper, Fraser describes a UNIX system used to build an engineering design aid for fast development of customized electronic apparatus. The user works at an interactive graphics terminal with a data base of standard integrated circuit

packages, generating output ready to drive an automatic wiring machine. The system makes judicious use of special hardware and software facilities available in only two of the large Bell Laboratories computation centers, which are accessed via communication links. In the application described in the third paper, by Rovegno, the UNIX system is used both as a tool and as a model for the development of microprocessor support software. Whereas a UNIX system was initially used to generate and load microprocessor software in a time-sharing mode, many of its features were then carried over into a small, dedicated microprocessor-based support system. The fourth paper, by Pekarich, shows the use of a UNIX system in a development laboratory for electronic switching systems. It replaces the control portion of a large switching machine, illustrating the ease of interfacing to several specialized devices in the telephone plant.

Whereas these four papers deal with one-of-a-kind systems in research or development environments, the last two papers describe the UNIX system-based products that are replicated throughout the Bell System. The paper by Nagelberg and Pilla describes the MERT-based Record Base Coordination System, which coordinates the activities of several diverse data base systems. Ease of change, even in the field, is the overriding requirement here; this pertains to the interfaces as well as to the algorithms, which are all implemented in the UNIX command language (shell). The paper by Cohen and Kaufeld deals with the Network Operations Control System. It represents the top level of a hierarchy of systems that collect telephone traffic data and control the switching network. Characterized by well-defined and stable interfaces and stringent performance requirements, the design of this system exemplifies how real-time requirements can be met by modifying the UNIX operating system.

VI. SUMMARY

The UNIX system has proven to be an effective production environment for software development projects. It has proven to be an appropriate base for dedicated products as well, though it has often required modification and extension to be fully effective. The near future promises better real-time facilities and some significant portability advantages for the UNIX development community.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

Microcomputer Control of Apparatus, Machinery, and Experiments

By B. C. WONSIEWICZ, A. R. STORM, and J. D. SIEBER
(Manuscript received January 27, 1978)

Microcomputers, operating as satellite processors in a UNIX system, are at work in our laboratory collecting data, controlling apparatus and machinery, and analyzing results. The system combines the benefits of low-cost hardware and sophisticated UNIX software. Software tools have been developed that accomplish timing and synchronization; data acquisition, storage, and archiving; command signal generation; and on-line interaction with the operator. Mechanical testing, magnetic measurements, and collecting and analyzing data from low-temperature convective studies are now routine. The system configurations used and the benefits derived are discussed.*

The vision of an automated laboratory has promise: computers control equipment, collect data, and analyze and display results. The experimenter, freed from tedium, devotes more energy to creative pursuits, presumably research and development. Unfortunately, the vision has proved to be a mirage for more than one experimenter who, after a year of learning the mysteries of hardware and software, finds the control of experiments as far away as ever.

This paper describes a system for laboratory automation using the UNIX time-sharing system that has permitted experiments to be automated in hours rather than years. This is possible because the

* UNIX is a trademark of Bell Laboratories.

UNIX system makes programming easy, standardized hardware solves many interfacing problems, and a library of programming tools performs many common experimental tasks. The complex task of automating an experiment reduces to the simpler tasks of assembling hardware modules, selecting the appropriate combination of software tools, and writing the program. The experimenter need not know the details of how signals are passed from one hardware module to another.

The benefits of automation are illustrated here by examples taken from the laboratory. Among these are very precise data logging, simplified operation of complex machinery or experiments, quick display of results to the operator, easy interfacing to data analysis tools or graphics, and ease of cross-correlating among experiments.

I. APPROACHES TO AUTOMATION

A variety of approaches have been made to the problem of laboratory automation. Systems can be designed for the job at hand, or they can be designed to be multipurpose. A computer may be devoted to a single experiment, or one computer may be shared among several experiments.

1.1 Job-specific automation

One approach to automation is to tailor a system to a specific problem, either by developing dedicated hardware or by developing job-specific software. A modern digital multimeter affords a good example. Some multimeters employ specially designed digital circuitry to accomplish a myriad of operations (ac-dc voltage and current readings, resistance, autoranging or preset scales, sampling times, etc.), while others incorporate a microprocessor with specific software to accomplish the same functions.

A drawback is that the design can be too specific. Changes in the operation of the device can be made only by rewiring the circuit or by rewriting the program. Since the operation of a digital multimeter changes slowly, the job-specific development is practical. If enough instruments are sold to recover the high costs of specific design, the approach is economical.

Larger examples of job-specific design are to be found in the automation of widely used scientific apparatus such as gas chromatographs and x-ray fluorescence machines, where the high cost of software and hardware can be amortized over many units and where

the function of the apparatus changes slowly. Unfortunately, there are other examples where the automation never quite worked properly or could not be changed to meet changing needs, and still others where the high cost of developing the job-specific design was never recovered.

1.2 Multipurpose automation

In fact, many cases of automation in research and on the production line are antithetical to the conditions for job-specific development. The tasks are varied, they change rapidly, and the number of identical installations is small. Such applications are best served by a system that is versatile and easily changed.

A system can be multipurpose only if the hardware and the software are multipurpose. Flexibility in hardware at the module level is illustrated by the multimeter, which performs a variety of functions by virtue of the specific hardware and software it contains. Hardware versatility at the system level is achieved by the ease of interconnecting or substituting various modules, meters, timers, generators, switches, and the like. Recently, standards for interfacing instruments have been gaining acceptance. Many modern instruments offer the IEEE instrument bus;[1] nuclear instrumentation follows the CAMAC standard[2] which permits higher data rates and larger numbers of instruments than the IEEE bus.

One might think that because typing is easier than soldering, it should be easier to change software than to change hardware. However, the ease of changing software depends on the language at hand, the quality of the editor, the file system structure, etc. The very features of the UNIX time-sharing system that make it suitable for system development[3] also make it suitable for automating a laboratory or a production line. Most of the work in automation is in the software, and UNIX makes developing the software easy.

1.3 Multipurpose software

Software is changed more readily if it is well designed and cleanly implemented. Quite small programs can meet a large number of needs if they fit Kernighan and Plauger's description of a software tool:[4]

> it uses the machine; it solves a general problem not a special case; and it's so easy to use that people will use it rather than building their own.

Naturally, such tools will not meet every need, but they will serve for most tasks. They are usually easily modified to meet special needs.

1.4 Stand-alone systems

An experiment can either have a computer devoted exclusively to it or it can share the computer with other experiments. The stand-alone approach has several virtues; chiefly, no one else can preempt the computer at a crucial instant (real-time response) and no one else's errors can cause the system to crash (independence). Real-time response and independence must be traded off against the high cost of hardware and software. While the cost of the central processor and memory have declined significantly in recent years, the cost of terminals, graphics, and mass storage devices has remained high.

It should be remembered that most of the cost of automation is in software development. The software development tools on most stand-alone systems are primitive by UNIX standards and result in substantially higher total development costs.

1.5 Shared systems

Connecting several experiments to a well-equipped central computer shares the costs of expensive peripherals and may provide a reasonable programming environment. Simple experiments which do not require real-time response can be interfaced to a time-sharing system directly. UNIX time-sharing can be used for this purpose if the data rates are slow enough or if time delays can be tolerated. For example, an x-ray diffractometer might be interfaced as an ordinary time-sharing user since data are normally taken every five seconds or so. If there is some delay due to the load on the time-sharing system in positioning the diffractometer for the next reading, nothing but time is lost.

Some central computers do provide a system which will guarantee real-time response. The MERT system is an example of one which provides a very sophisticated real-time environment aimed at users capable of writing system level programs.[5] Nevertheless, writing programs for real-time response on a shared system is a complex task that must be done with care because a single experiment can bring down the entire system. In the past, big shared central computers have been unsuccessful in controlling many experiments at once, although recent reports indicate some success.[6] In general,

such a system is forced to rely on a small amount of very reliable code to shield the users from one another and from the system.

1.6 Satellite processors

The real-time response and independence of a stand-alone system can be combined with the lower costs and superior software of a shared system if inexpensive, minimally equipped microcomputers are connected to a large, well-equipped central processor. The microcomputer or satellite processor (SP) provides real-time response and independence when the experiment is in progress; the central processor (CP) provides data storage, an excellent programming environment, and data analysis tools. Not only is the cost of such a system significantly less than the cost of an equivalent number of stand-alone systems, but the CP also provides time-sharing services for data analysis and reduction, document preparation, and general-purpose computing.[7,8]

II. SYSTEM DESCRIPTION

In our laboratory, we have automated several experiments with the distributed processing system shown schematically in Fig. 1. The CP provides UNIX time-sharing service to 16 incoming phone lines and supports up to 16 SPs for experimental control or as ordinary time-sharing users.

2.1 The central computer

A Digital Equipment Corporation (DEC) PDP-11/45 with 124K of core storage, cache, 80M bytes of secondary disk storage, and a tape drive serves as the central computer. Graphics are provided by a high quality terminal (Tektronix 4014) and a hard-copy printer plotter (Versatec 1200A). Data can be transmitted to other computer centers with an automatic calling unit and a 2000-baud synchronous phone link.

The central facility is similar to an ordinary UNIX installation, offering a wide range of time-sharing services. Like other UNIX systems, many jobs are program development or document preparation. An unusually large number of jobs are numerical analysis run in Fortran or Ratfor[9] or graphical displays using the UNIX **graph** command. Fortran is used because of the availability of many

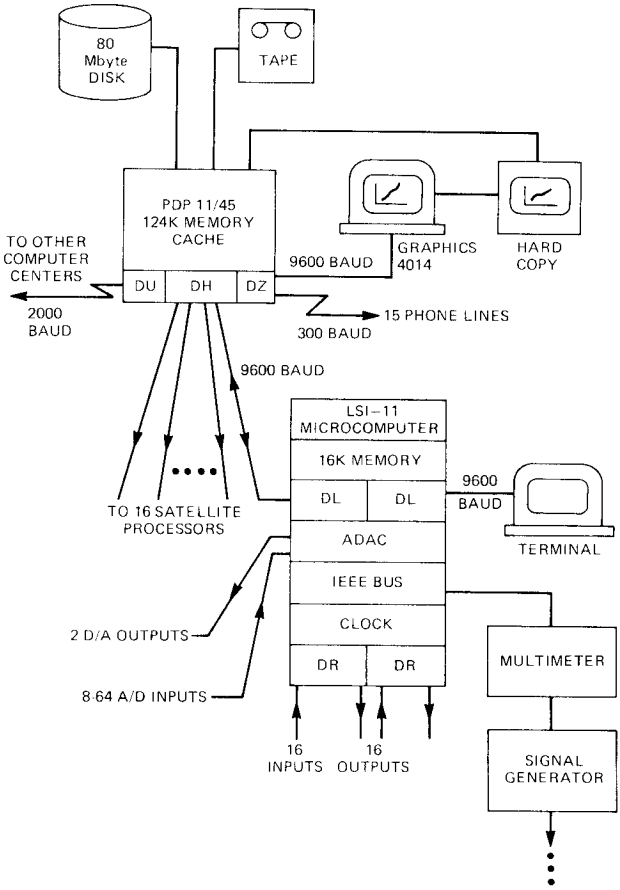

Fig. 1—Satellite processor system used for control of experiments.

mathematical subroutines, for example, the PORT library,[10] or because of inertia on the part of experimenters.

2.2 The satellite processor

Eight SPs are presently connected to this central facility, each consisting typically of an LSI-11 microcomputer with extended arithmetic, a 9600-baud CRT terminal, two serial line interfaces (one for the CP and one for the terminal), a PROM/ROM board containing a communications package, and from 8 to 28K of semiconductor

memory. Because memory is inexpensive, most users choose more than minimum storage.

2.3 The CP-SP interface

The SPs are linked to the CP by means of serial line interfaces (see DL and DH in Fig. 1), operating over two twisted pairs of wires, often 300m long.

The SP does not have the resources either in memory or secondary storage to run the UNIX system directly. For a UNIX system that will run on a microprocessor equipped with a floppy disk, see Ref. 11. However, with the cooperation of the CP, the SP can emulate the UNIX environment during execution of a program, using the Satellite Processing System (SPS).[12]

SPS prepares a program for execution in the SP, transmits it, and monitors it during execution. If the program in the SP executes a system call, it is sent to the CP for execution. In our experience, sending all system calls to the CP proved burdensome, so we modified SPS so that the SP would handle system calls itself if possible, referring only those system calls to the CP which the CP alone could handle. For example, reading and writing the local terminal is best handled by the SP itself, whereas reading or writing a remote file can only be done through the CP. Certain other system calls, fork, for example, are simply not appropriate in the present distributed processing framework and are presently treated as errors. When no program is executing in the SP, the local terminal behaves exactly as if it were connected directly to the CP under UNIX timesharing. Further revisions to the CP-SP communication scheme are under way that should permit the SP to run a long-term program acquiring data while the local terminal is used for ordinary timesharing.

Although SPS was designed to accommodate a variety of computers, cost considerations have led us to use LSI-11 microcomputers exclusively. If future needs dictate a large computer, the capacity is there, although the future may well bring bigger and faster satellite machines.

2.4 Interface to the experiment

A surprising variety of experiments can be automated with the interfaces shown schematically in Fig. 1.

(*i*) The CRT terminal operating at 9600 baud provides interaction with the operator at a speed high enough to permit messages as explicit as needed without slowing the experiment down.

(*ii*) Voltage signals are read or generated by means of an ADAC (Analog-to-Digitial Digital-to-Analog Converter). Connections are made through a standard multipin connector. From 8 to 64 lines can be used on input and two lines on output. The usual range of voltage is 10 volts with a precision of 5 mV. The programmable gain option allows the precision to be increased to 0.5 mV over a correspondingly smaller voltage range.

(*iii*) Signals that are binary in nature can be interfaced with the LSI DRV-11 parallel interface, which provides 16 input and 16 output lines. On output, pulses can be generated to step motors, set relays, or trigger devices that respond to TTL signals. On input, switch closure, TTL signals, and interrupts can be monitored. The DR also provides a way to interface to numeric displays or inputs. If the number of binary inputs is large, several DRs can be employed.

(*iv*) Sophisticated instruments, such as multimeters, frequency synthesizers, transient recorders, and the like, can be interfaced through the IEEE instrument bus.[1] More than a hundred instruments are available with the IEEE bus, and the numbers have been increasing rapidly.

(*v*) Timing during the experiment can be accomplished with the internal 60-Hz line time clock of the LSI-11 or by a more precise programmable clock, the KW-11.

III. INTERFACE TO THE EXPERIMENTER

Our goal was to create an interface between the user and the experiment which used the machine to do the dirty work, met a variety of needs, and was so easy to use that people wouldn't try to reinvent it—in short, to develop what Kernighan and Plauger describe as tools, as opposed to special-purpose programs.[4]

Each interface described above is handled with one or more tools summarized in Table I. Each is a function or series of functions written in the C programming language[13,14] which can be called from a C program. The C functions can also be called directly from programs compiled with the FORTRAN 77 compiler which is now operational on UNIX.[15] A tutorial discussion of the use of these

Table I—Tools for experimental control

```
Interaction with the experimenter
      rsvp(); number();
Data acquisition
      getvolt(); setgain(); zero(); parin(); parint();
Experimental control
      outchan(); putvolt(); ramp(); sine(); parout();
Timing and synchronization
      time(); delay(); setsync(); sync();
Dynamic data storage
      bput(); bget(); bfree();
Sending data to the central computer
      bsend(); bname();
```

tools is available[16] which has served as the text for a 16-hour course in computer automation.

3.1 Interaction with the terminal

The programmer can use two tools to ask a question of the operator and read the response: **number** which returns a floating point number and **rsvp** which matches the response to a list of expected replies. For example,

```
velocity = number("ram velocity", "mm/sec");
```

will print the message

```
ram velocity, mm/sec?
```

on the terminal, analyze the response, and return a floating-point number. If the input is unrecognizable, **number** repeats the message. If the reply is "1 ft/min," a conversion will be made to mm/sec, the units specified by the optional second argument. If the units are unknown, e.g., furlongs/fortnight, an error message will be printed and **number** will try again. It is possible for the user to supply an alternate table of conversion factors for **number**.

A second tool is provided to ask the experimenter a question and analyze the reply. The simplest use of **rsvp** is to use the terminal to get a cue, as in:

```
rsvp("Hit RETURN to start the test.");
```

which prints the message on the terminal and waits for the carriage return to be typed. **rsvp** will analyze responses, as in:

```
reply = rsvp("stress or strain?", "stress", "strain");
```

which prints the first argument on the terminal as a prompt,

analyzes the response, and sets reply to 1 if **stress** was typed, 2 if **strain** was typed, and 0 if anything else was typed.

Because **number** and **rsvp** are reasonably intelligent and cautious about accepting illegal input, they can be used to create other programs which are similarly gifted. For example, many experiments require a knowledge of the area of the specimen under study. A simple function can be written in a dozen or so lines which will calculate the area from information supplied at the terminal for either a circular or a rectangular specimen. The user can supply dimensions in units ranging from yards to microns.[16]

Because **rsvp** and **number** use the standard I/O routines, they run on the 11/45 as well as the LSI-11. Developing them was also easy, because the hard part was done by UNIX library routines like **atof**.

3.2 Interfacing with analog signals

Once the leads from the experiment are connected to the ADAC, the voltage on the **n**th channel can be read by calling

```
getvolt(n);
```

The gain on channel **n** can be set to 10X by calling

```
setgain(n, 10);
```

If this call is issued and the ADAC lacks programmable gain, or if the desired gain is not available, **setgain** will print an error message. In some experiments, it is convenient to take an arbitrary reference voltage as a zero reference. Calling

```
zero(n);
```

will take the current voltage reading on channel **n** as the zero reference for all subsequent readings on channel **n**. All data returned by **getvolt** will be in consistent internal units, so that the gain or zero can be set independently by different routines.

Voltages can be generated with the function:

```
putvolt(v);
```

If there is more than one D→A, the function **outchan** can be called to specify the channel for output. The following code causes zero volts to appear on the two output channels.

```
outchan(0);
putvolt(0);
outchan(1);
putvolt(0);
```

putvolt can be used in combination with the timing tools to create a variety of signal generators or command signals. If the response to the voltage command is measured with **getvolt** and fed back to the command, closed loop control is possible. Examples of such control are given in a later section.

3.3 Parallel interfacing

The 16 output lines of the parallel interface can be written simultaneously by writing a 16-bit word with the call:

```
parout(n, word);
```

where n refers to the number of the DR interface board. The binary representation of the word will determine which lines are on or off. Since the C language provides a number of operators for bit manipulation, the common operations of setting, clearing, or inverting the ith bit can be accomplished easily.[14]

Similarly, the state of the 16 input lines on the *n*th DR can be read simultaneously with the function:

```
parin(n);
```

which returns a 16-bit integer. The C bit operators are then used to mask off the bits which correspond to the signals of interest.

The interrupt inputs on the DR can be used to signal the processor as described in a later section.

3.4 The IEEE instrument bus

The IEEE bus interface tools are still under development with the goal of writing high-level commands that will interface to many electronic instruments. At present, the instruments can be controlled by the standard technique of writing or reading an ASCII stream on the bus with reads and writes or the high level routines **scanf** or **printf**. A command analogous to **stty** called **buscmd()** sets the state of the bus driver.

3.5 Timing

Events during the experiment can be timed by means of the internal line time clock in the LSI or by a higher precision programmable clock. In either case, the tools for timing remain the same; the precision simply changes. The simplest use of the clock is to measure elapsed time with the variable **time** which is incremented each clock tick. Events can be synchronized with a clock by the function **sync()**. A function **setsync(t)** is provided which sets the period of the sync signal to *t* ticks of the clock. Subsequent calls to **sync()** will not return until a sync signal is generated. In this way, data can be obtained at regular intervals, or pulses of a specified frequency can be generated. A third function **delay(n)** causes the program to wait for **n** sync signals. The accuracy of the timing functions is determined by the accuracy of the clock and ultimately by the speed of the LSI-11, which seems to limit practical timing to frequencies of less than about 10 kHz. In the future, faster processors may extend this limit.

3.6 Data storage and transmission

Many applications of computers to experiments are primarily data acquisition and recording. The tools for storing data on the SP and transmitting it to remote files on the CP are important. Therefore, considerable effort has been spent on devising a set of commands for storing data in buffers on the SP and transmitting the buffers to named files on the CP.

A group of buffers are provided in which data can be stored without regard to type (**int**, **long**, **float**, or **double**). The buffering routines keep track of the type automatically. This is useful in storing data for an xy graph; *n* pairs of integers which form the data for the graph could be stored in the same buffer as an initial pair of double precision scaling factors which convert the integer data into useful units. The entire buffer can be transmitted to the CP as a unit. The buffers are dynamically allocated; that is, depending on how the data are written, one buffer could consume all the buffer space, or all the buffers could share the space. The command **bfree(n)** releases the **n**th buffer whose space is then made available for data storage by any of the other buffers. The command **bsend(n)** sends the contents of buffer **n** to the CP. The function **bname()** is used to specify the name of the file.

The buffering commands can be used in combination to form a sophisticated data logging system. For example, **bsend** can be called periodically during data logging to update a file on the CP. The transmission will be incremental; that is, only the data added to the buffer since the last **bsend** will be transmitted. For applications where more data are taken than will fit in the SP, large files can be built on the CP by doing a **bsend** when the buffer fills, followed by a **bfree**. Once the files are saved on the CP, the UNIX shell and programs can be used to manipulate or forward the data to yet another computer facility for analysis on a higher speed computer.

3.7 Interrupt and signal handling

In the control of real-time experiments, it is sometimes necessary for the experiment to interrupt the computer, as for example when data collection is complete or an event of some urgency has occurred. The UNIX system's signal and interrupt mechanisms are rudimentary, as such events are rare in the fully buffered I/O environment. The single-level interrupt structure of the LSI-11 further complicates the problem of handling interrupts generated by the experiment. Our solution has been to implement a simple control primitive for doing all this, the EXECUTE command. EXECUTE(routine, priority) allows the user to specify a routine to be run when the software priority level of the program drops below the given level. Because things normally considered atomic in nature (floating-point arithmetic and system calls) may be extremely slow when simulated on the LSI-11, it is advantageous to make them interruptable at a high priority rather than atomic. To guarantee no side effects, the interrupting routines must take care not to execute non-reentrant code.

The user would not normally use EXECUTE directly but would use parint(n, routine, priority) which handles interrupts coming in on the *n*th DR interface. parint() can be used to turn off all interrupts or name a routine to be EXECUTEd at the specified priority when an interrupt occurs. For example, in the case of finding peaks in a diffraction spectrum, it is advantageous to compute while data are being collected by the counters, and be interrupted when the count is completed. Such an interrupt could be executed at a relatively low priority. If, on the other hand, an interrupt is received which indicates that a disaster has occurred, it should be handled at the highest priority.

3.8 Notes on implementation

We have tried to design a software interface to experiments which follows the UNIX unified approach to input-output (I/O), that is, I/O should take place without regard to the type of device being read or written. For example, the local terminal, remote files, or the IEEE bus can all be read or written with standard I/O.

Using standard I/O provides additional benefits. Routines for the IEEE bus were developed on the 11/45 and run on the LSI-11 without change. Development was much easier on the 11/45 where pipes and a bus simulator were used to test the program before running on the LSI-11, without the added set of problems caused by running on the unprotected LSI-11. The IEEE bus interface could easily be used as a model for a UNIX device driver if the need arises.

I/O is handled in the modified SPS framework by keeping a list of device descriptors that represent local devices. Any system call for a local device is handled without CP intervention; all others are passed to the CP. The local terminal driver includes fully buffered input and output, echoing, erase and kill processing, and interrupt and quit handling.

A different approach is necessary for the devices on which a single word or byte represents the entire transmission (DR or the ADAC). These devices can be faster than the LSI-11 itself, so we have kept the interface as low level as possible, thereby incurring less overhead than would be the case if the standard I/O system were used. We have tried to make the I/O as independent of the device as possible, so that the user will not need to understand the detailed operation of each device and so that similar devices can be interchanged without changing the user programs.

Each SP has a unique combination of hardware which requires a unique library of software tools to make it work. We are able to compile such a library by specifying options to the C preprocessor at compile time, so that the libraries can be prepared without changing the programs or including redundant information.

IV. EXAMPLES

The measure of the system and the tools is to be found in their application to actual experiments. The following examples are experiments which are now running, using the system and tools described above.

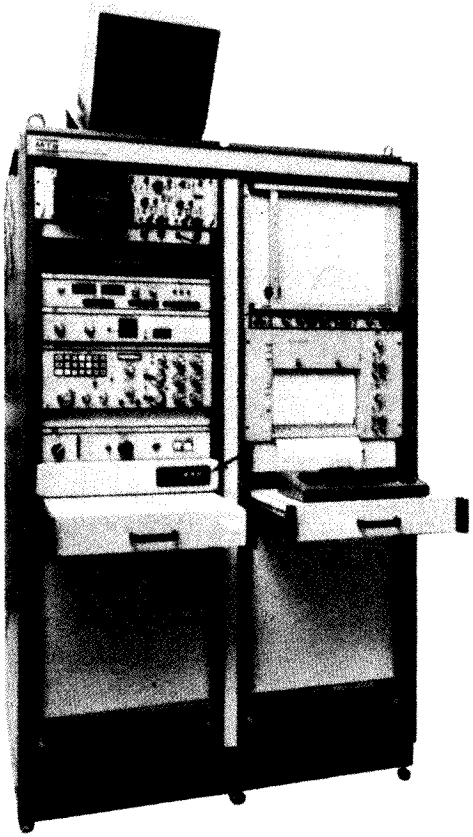

Fig. 2—Servo-hydraulic control panel. The satellite processor is the small module indicated by the arrow.

4.1 Mechanical testing

One function of computer automation is to simplify the operation of a machine and shield the inexperienced user from its intricacy. The complexity of the control module of a modern servo-hydraulic machine is apparent in Fig. 2. The microcomputer, which controls the machine, is the small module indicated by the arrow. As a simple example of the complexity of running an experiment without computer assistance, the units of the stress strain curve being displayed on the *xy* recorder in Fig. 2 are determined by the settings of four potentiometers, the sensitivity of two transducers, and the dimensions of the sample. Under computer control, the results are

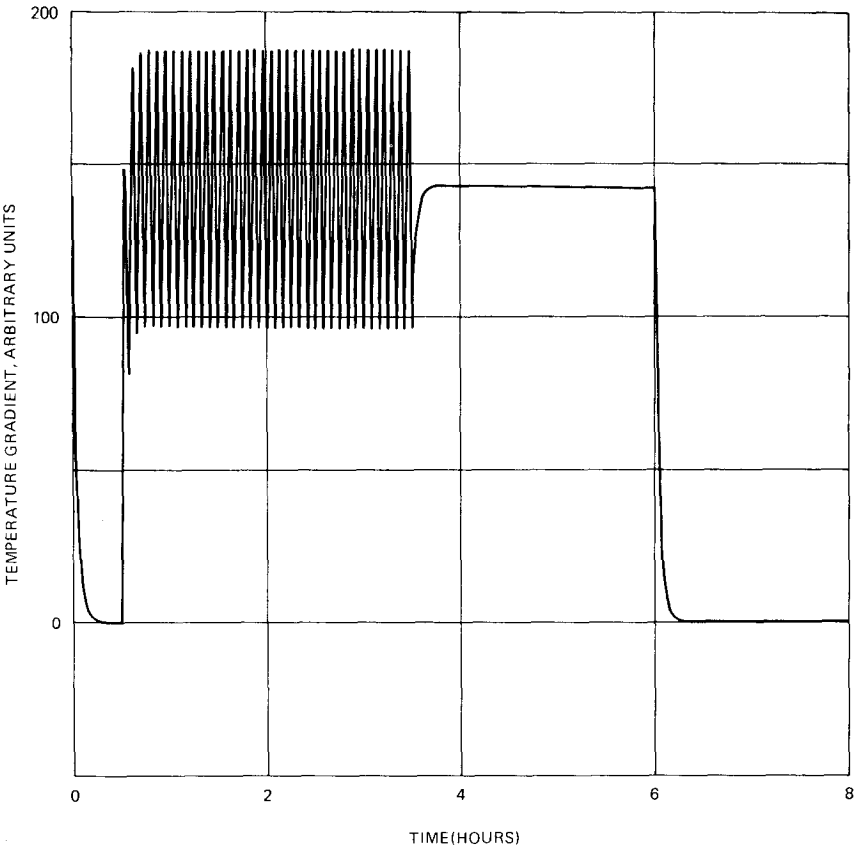

Fig. 5—Temperature gradient across a cell as a function of time. The gradient is produced by applying a power input to one plate of a cell filled with liquid He at 2.2 K. The power is varied sinusoidally at first, then is held constant at the rms power level of the sine, and then is shut off. See Fig. 6 for a magnified view of how the mean temperature approaches steady state and Fig. 7 for the power spectrum of the response.

the same time the response (the temperature gradient across the cell) is recorded with **getvolt** and stored with **bput**. The resulting gradient is displayed in Fig. 5. After a time the sinusoidal input is replaced by a constant power input of the same mean power, and the gradient is again monitored. The power is then shut off for an interval and the process is repeated with a slightly higher power density.

Figure 6 shows how the mean temperature gradient approaches steady state for the sinusoidal input and the constant input. It

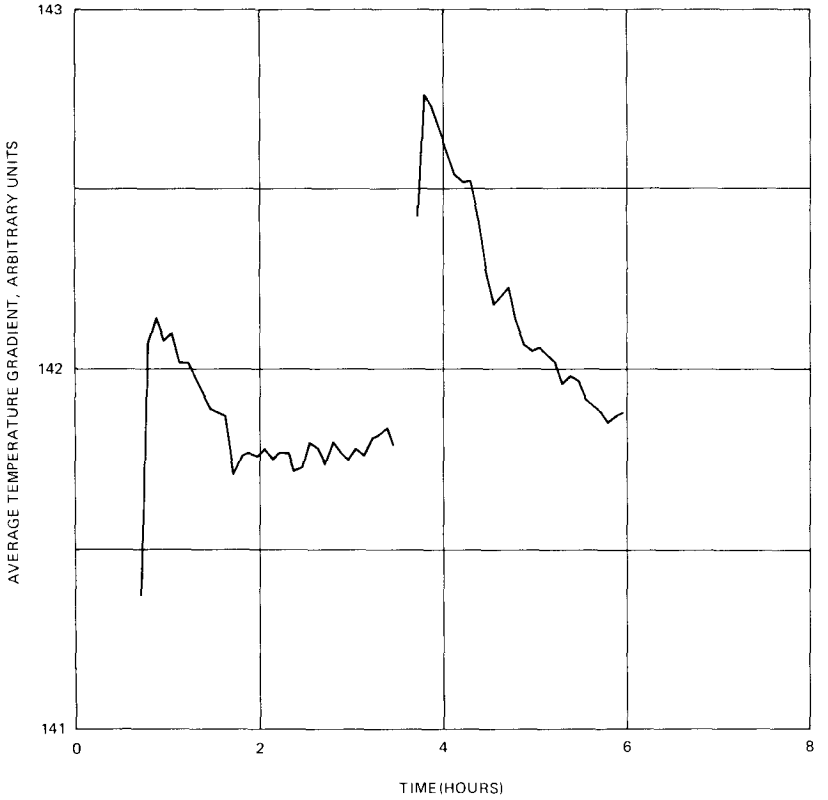

Fig. 6—Mean temperature as a function of time. The data from Fig. 5 have been averaged and magnified to show that the approach to steady-state conditions depends on whether the input is oscillating or constant.

shows that the stability of the system depends on whether the input is oscillating or steady. Still using data obtained with the LSI-11, Ahlers applies fast Fourier transforms to reveal the details of the instability (Fig. 7). The ability to write programs, control experiments, analyze data with sophisticated mathematical library functions, display it graphically, and write it up for publication on a single system is a significant advantage.

Other applications under way or in progress include x-ray diffraction, pole figure determination, scanning calorimetry, phonon echo spectroscopy, thin-film reliability studies, and semiconductor capacitance spectroscopy.

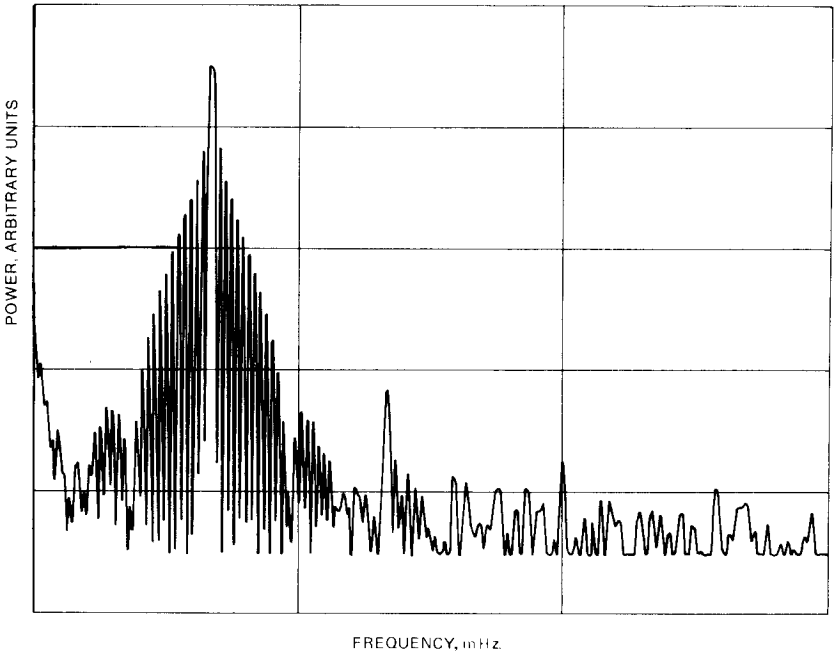

Fig. 7—Power spectrum of the temperature gradient vs. time function. The response to the sinusoidal power input is Fourier-transformed, and the corresponding power spectrum is displayed. Note the peaks at the second and third harmonic, which were not present in the power input.

V. CONCLUDING REMARKS

We have in operation a system for controlling laboratory experiments which is powerful, easy to use, and reasonably general. It combines the isolation and real-time response of a stand-alone system with the shared cost and better hardware/software facilites of a large time-shared system. It is powerful because the UNIX operating system and the C language provide facilities for file manipulation and the direct control of devices. It is easy to use because tools have been written which shield the novice from many of the interfacing details. It is general because the tools were written with general, rather than specific, applications in mind. Where great speed or specialization is necessary, the tools form a model that can easily be modified to meet the needs.

For the future, we expect bigger and faster satellite microprocessors which will add further to the attractiveness of the satellite processing scheme. As microprocessors become incorporated in test

equipment, we see a trend toward more intelligent instruments which, on command from the SP, can execute fast, complicated procedures without the SP's intervention. The test instrument should be intelligent about performing its essential functions and should provide an interface (eg. the IEEE bus) to a more general-purpose machine which controls other tests, coordinates instruments, and analyzes the results. Trends toward building full-blown software systems including file systems into large test equipment seem counterproductive.

As to the future of the software discussed here, we plan to revise the CP-SP communications interface to take advantage of new UNIX features to improve reliability, versatility, and speed. Further work remains to be done on a set of higher level tools for interfacing with instruments on the IEEE bus.

VI. ACKNOWLEDGMENTS

The authors wish to thank R. A. Laudise, T. D. Schlabach, and J. H. Wernick for support and encouragement during this project; H. Lycklama, C. Christensen, and D. L. Bayer for discussions and aid; and P. D. Lazay, P. L. Key, and G. Ahlers for numerous contributions and a critical reading of the manuscript.

REFERENCES

1. "Standard Digital Interface for Programmable Instrumentation and Related System Components," IEEE Std. 488-1975, IEEE, New York. See also D. C. Loughry, "Digital Bus Simplifies Instrument System Communication," EDN Magazine (Sept. 1972).
2. CAMAC (Computer Automated Measurement and Control) Standard, Washington, D.C.: U. S. Government Printing Office, documents TID-25875, TID-25876, TID-25877, and TID-26488. See also John Bond, "Computer-Controlled Instruments Challange Designer's Ingenuity," EDN Magazine (March 1974).
3. S. C. Johnson and M. E. Lesk, "UNIX Time-Sharing System: Language Development Tools," B.S.T.J., this issue, pp. 2155-2175.
4. B. W. Kernighan and P. J. Plauger, *Software Tools,* Reading, Mass.: Addison-Wesley, 1976.
5. H. Lycklama and D. L. Bayer, "UNIX Time-Sharing System: The MERT Operating System," B.S.T.J., this issue, pp. 2049-2086.
6. J. V. V. Kasper and L. H. Levine, "Alternatives for Laboratory Automation," Research/Development Magazine, *28* (March 1977), p. 40.
7. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
8. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," B.S.T.J., this issue, pp. 2115-2135.
9. B. W. Kernighan, "Ratfor — A Preprocessor for a Rational Fortran," Software — Practice and Experience, *5* (1975), pp. 395-406.
10. P. A. Fox, A. D. Hall, and N. L. Schryer, "The PORT Mathematical Subroutine Library," ACM Trans. Math. Soft. (1978), to appear.

need to rapidly obtain small numbers of prototype circuits. Ideally, the designer would like to be able to draw and edit circuit schematics in machine-readable form so that checking, physical layout, and prototype production are available at the touch of a button. Drafting standards and (frequently) physical design efficiency are second in priority to a fast turnaround. Automatic documentation and facilities for making changes to earlier designs are also required.

To a significant degree, this ideal has been achieved for those engineers who have access to a PDP-11 that runs the UNIX operating system. The following four command programs are used:

draw	to edit circuit drawings
wcheck	to perform consistency checks on a circuit
place	to place components on a circuit board
wrap	to generate data for wire wrapping

All programs are written in the C language and the interactive graphics terminal is a Tektronix 4014. Hard-copy output on paper or microfilm is obtained through a Honeywell 6070. The Honeywell 6070 also produces the paper tape which drives a Gardner-Denver 14YN semi-automatic wire-wrap machine. Punched cards suitable for use with a Gardner-Denver 14F automatic wire-wrap machine are obtained by executing a program on an IBM 370. In each case, use of the service machine is automatic and is possible through the communications lines linking the PDP-11 to the Honeywell 6070 and linking the Honeywell 6070 to the IBM 370.

The circuit design aid programs are intended for use with circuits composed primarily of dual in-line packages assembled on wire-wrapped boards. The intent is to provide the nonspecialist circuit designer with a tool suitable for laboratory use. It is assumed that the user's prime objective is to obtain an assembled circuit board and maintain consistent documentation with a minimum of fuss. In exchange, the user must be prepared to adopt some predefined conventions and a rather restricted drawing style.

Logic diagrams are first prepared using **draw**. One file of graphic information is constructed for each page of the drawing. When the logic drawing is complete, the graphics files are processed to yield a wire-list file. The user must now prepare a definitions file in which the circuit board and circuit components are described. Library files of definitions exist to simplify the task. The definitions file is a text file prepared using the UNIX text editor. Next, the wire-list file and the definitions file are fed to **wcheck**. That program performs a number of consistency checks and generates a cross-reference

listing. Any errors detected at this stage must be corrected in the logic diagram, and the process is then repeated. When a correct circuit description has been obtained, the definition and wire-list files are fed to **place**. That is an interactive program for choosing a circuit board layout. A drawing of the board is generated automatically. In addition, placement information is automatically written back into the logic drawings for future reference. Finally, the definitions file and a wire-list containing the placement data are fed to **wrap**, which generates the instructions necessary for wiring the circuit board. For machine-wrapped boards, **wrap** produces the appropriate paper tape or punched cards. Hard-copy printouts of the logic diagrams, the circuit board, and the cross-reference listings provide permanent documentation of the final product.

II. TERMINOLOGY AND CONVENTIONS

A circuit description contains two parts. A schematic drawing shows the interconnection of components, while a text file describes the physical properties of the components. Drawings prepared by **draw** are held in *drawing files* with one file per page of a drawing. These files have names that end in the page number and **.g**. The definitions file is prepared as ASCII text and is in Circuit Description Language (CDL). It is CDL that the **draw** program generates when asked to produce a wire list from the circuit drawing.

The terminology used in the circuit design aid programs is described below. It will be seen that this terminology is oriented toward integrated circuits and wire-wrap boards. However, with a little imagination, the user will find that a wide range of "non-standard" situations can be handled quite effectively.

signal	The information that passes over a wire and is carried by a net of wires.
special signal	A signal that is distributed about the circuit board by printed circuitry. Ground and VCC often fall into this category.
chip	A circuit component such as a NAND gate.
package	The physical unit which houses one or more chips. A dual in-line package is an example.
pin	The point where a signal wire connects to a chip.
board	The physical unit on which packages are mounted.

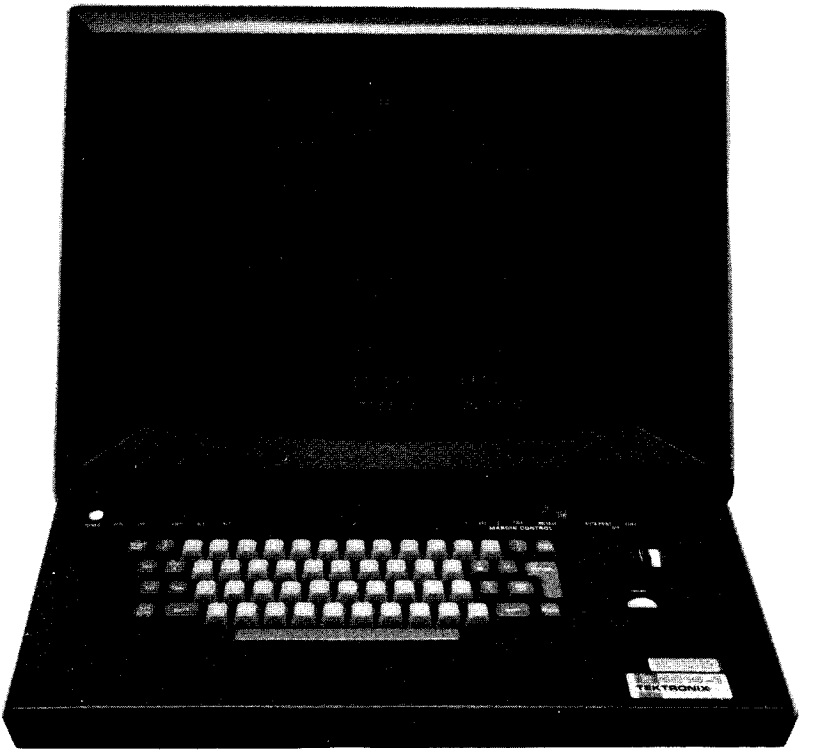

Fig. 2—The Tektronix 4014 terminal displaying a circuit.

a starts a new line segment which joins onto an existing line. Thereafter, striking the space bar extends the line to the current cross-hair position. Line segments drawn in this way are either vertical or horizontal. Diagonal lines are drawn by using **z** instead of **e**.

It may seem illogical, but **draw** treats pins as part of a wire and not part of a chip. A wire joins together a number of pins, and for each pin one can specify a name and a number. Chips are drawn separately and the association between chips and pins is made when the wire list is constructed. For that association to be made, the pin must lie within, or be very close to, a rectangle just large enough to contain the chip.

Chips are usually represented by closed geometric forms such as rectangles. To draw a chip it is only necessary to specify its shape and use the cross-hairs to give its position. A rectangle is drawn if no shape is specified. Two text strings can optionally be associated with a chip: a chip name and a type name. In some cases the shape

adequately identifies chip type and then no explicit type name need be given. Both names are treated by **draw** as text to be written, usually, within the drawing of the chip. An option, selected when defining a new shape, allows one to specify where such text should be displayed.

To define a new shape, the user first enters its name. A blank screen is then presented on which the shape may be drawn at four times the final scale. In this mode line end points must lie on 1/40-inch grid points (not the usual 1/10-inch grid) so that reasonable approximations to curved lines can be made using straight line segments.

By now it should be apparent that **draw** has little built-in intelligence. For example, it does not have a repertoire of heuristics designed to automatically route wires around chips or to minimize crossovers. We judge that such heuristics are cumbersome to program and rarely satisfactory in practice. Layout of the drawing is entirely in the user's hands. However, a number of small conveniences are provided to make the task moderately easy. For example, chips are drawn by positioning the cross-hairs on the left-hand end of the required centerline. Most chip shapes are symmetric about the centerline, and the desired centerline position is frequently known more accurately than any other point. Another convenience is a command that draws a grid centered on the current cross-hair position with grid points spaced at intervals specified by the user. In this way, one can quickly and accurately place a series of equally spaced wires or chips.

Editing facilities are equally unsophisticated. One can delete specific wires, pins, chips, and other drawing features or one can draw a rectangle around a group of components and delete them all or move the rectangle with components inside. Rectangular sections of the drawing can also be copied either to other parts of the drawing or onto a file for inclusion in another drawing. The awkward edits can be most quickly handled by redrawing some of the wires. To facilitate this with minimum risk of error, the following technique is used. First, recall that each wire joins together a certain collection of pins. The pins are part of the wire so far as **draw** is concerned. Therefore, there is a mechanism by which the user can remove all the lines of a wire without removing the pins. The pins remain and are displayed as asterisks. A new series of lines can now be drawn joining the asterisks together. In this way the possibility of omitting a pin from the wire is minimized and, at the same time,

the pin display provides an effective visual aid towards rapid selection of a new wire route.

Manipulation of the Tektronix 4014 thumb wheels is an acquired skill. **draw** simplifies the task by not requiring high accuracy when positioning the cursor. For example, when extending a horizontal line only the vertical cross-hair need be accurately positioned. When pointing at a wire, pin, or chip one need only be close to it, not precisely over it.

It has been found that the speed with which one can use a graphics terminal diminishes as the need to move one's eyes between keyboard and screen increases. Ideally one should be able to watch the screen and not the keyboard. However, it takes two hands to operate a typewriter keyboard and one more to control the cross-hairs. Since **draw** requires that the user type in signal and shape names, some hand and eye movements are unavoidable. The compromise that we have adopted requires a hand movement when a name or number is typed but not when drawing takes place. In particular, while drawing a wire one can keep the left hand in the standard position for typing, while the right hand operates the thumb-wheels. The thumb of the right hand also is used to strike the carriage return key. Each drawing action is effected by striking a single key which is in the region controlled by the left hand. Striking the carriage return terminates a drawing sequence.

The command

```
draw    therm1.g
```

is used to initiate editing of the drawing held in the file **therm1.g**. Hard copy output of that drawing is produced on the Calcomp plotter by

```
draw   -c   therm1.g
```

A wire list is generated by

```
draw   -w   therm1.g
```

and it is placed in a file called **therm1.w**. The wire list is an ASCII file in Circuit Description Language.

IV. CIRCUIT DESCRIPTION LANGUAGE

The Circuit Description Language was first designed as a convenient notation for the manual preparation of wire lists. As the design aid programs were elaborated, their needs for information

increased and the Circuit Description language grew. It is apparent that further extensions to the language will be required as new capabilities are added to the software package. Consequently, the Circuit Description language has an open-ended structure.

A circuit description consists of command lines and signal description lines. The former have a period as the leftmost character, followed by a one- or two-letter command. For example, the following is a description of a 16-pin, dual in-line package.

```
.k    DIP16 16 80 30
.kp   1 05/00  -  8 75/00
.kp   9 75/30  -  16 05/30
```

The first line gives the name of the package type, the number of pins, and the dimensions (in hundredths of an inch) of the package. The remaining two lines give the positions of the pins relative to one corner of the package.

The description of a circuit is in four parts:

(*i*) Package Descriptions. A description of each package type.
(*ii*) Board Description. A description of the board, the sockets and connectors mounted on it, and any special signals for which there is printed circuitry on the wire-wrap board.
(*iii*) Chip Type Descriptions. A description of each chip type.
(*iv*) Wire List. A list of chips and the signals wired to them.

The commands used in each of these sections of a circuit description are described briefly below.

(*i*) **package type** For each package type one must give its physical size and the number of pins attached to it. For each pin one must give the pin number and its coordinates. A shorthand notation for describing the positions of equally spaced pins is shown in the example above.
(*ii*) **board** The board name, number of sockets, and number of I/O connectors must always be provided. Additionally, one can specify the relationship between the board coordinate system as used in the circuit description and the coordinate system used by a wiring machine.
(*iii*) **socket** Sockets differ either in size or in the range of package types that can be plugged into them. For each socket type, one must list the package types that are acceptable and board coordinates at which these socket types occur. For example, the following two lines specify that there are 15

next must always be parallel to one or the other of the board edges. Diagonal wiring is not permitted. No constraints are imposed on the number of wires that can be funneled through any section of the board. The minimum wire route problem is similar to the traveling salesman problem and the algorithms used by **place** are based upon the work of Lin.[1]

The package placement routines are based upon the work of D. G. Schweikert.[2] Two routines are involved: One makes an initial placement of packages not already placed anywhere on the circuit board. The other takes those packages already assigned positions on the board and attempts to improve matters by moving them around. Both operate within the following constraints: Packages cover a rectangular area of the board, and no two packages can be so placed that the area covered by one overlaps the area covered by another. Sockets can secure to the board only those packages listed in the socket definition. No socket can hold more than one package, but it may take several sockets to handle a large package. When a package is placed in a socket, the origin of the package coordinate system is made to coincide with the origin of the socket coordinate system (i.e., the lower left-hand corners of the two rectangles are aligned). If the package is larger than the socket, all other sockets falling partly or entirely inside the package rectangle are "occupied" by that package and cannot be used for another package. A connector whose position on the board is always fixed also occupies a rectangular area of the board. A package cannot be placed so that it overlaps the connector. These constraints are considerably more elaborate than those used in other placement programs of which we are aware. They are made necessary by the growing diversity in package sizes and socket configurations.

place starts with the package placement indicated on the logic drawings. It uses the Tektronix 4014 to display the current board layout. Sockets and packages are represented as rectangles with socket numbers and chip names written within them (Fig. 3). Using the cross-hairs, one can point to a chip or socket and, by typing a command letter, cause some change either in the display or in the current placement. For example, one can move a package from one socket to another, place an unplaced package, or remove a package from the board. Other commands invoke the automatic placement routines.

Each package can be in one of three states: It can be unplaced or its placement may be either soft or hard. A soft placement is one that can be altered by an automatic placement routine while a hard

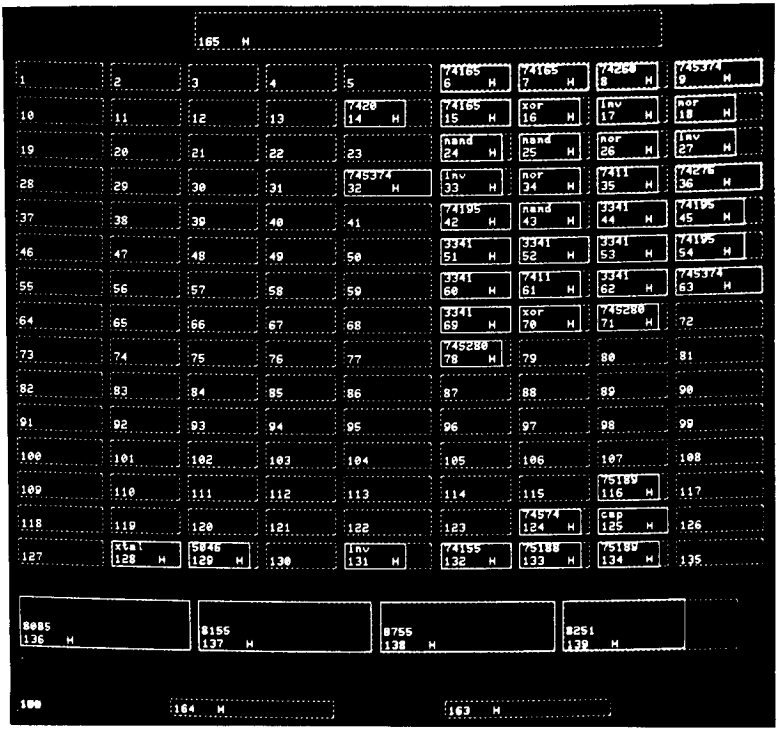

Fig. 3—Display of circuit board layout.

placement can be changed only by the user. An unoccupied socket can be in two states: It can be available for placement or it can be blocked. Packages cannot be placed in blocked sockets. Sockets are blocked and unblocked by user command.

One quite successful method of obtaining a placement is to use the automatic placement and improvement routines on selected groups of packages or sections of the board. Between such optimizations the user inspects the current placement, makes key moves, and changes the soft/hard and blocked/unblocked states of sockets and packages in preparation for the next use of the automatic routines. To assist with this process one can display an airline map of the optimized wire route for wires connected to selected packages. One can also obtain summary information such as the length of wire required by the current placement. Another helpful display is the original logic drawing with the current placement displayed on the drawing for each chip. When a satisfactory placement has been obtained, the drawing files can be permanently updated in this way.

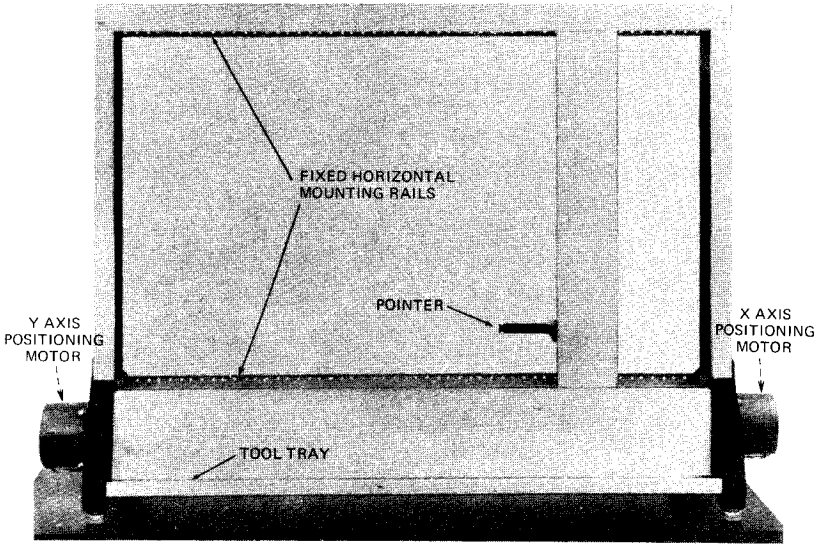

Fig. 4—Standard Logic semi-automatic wire-wrap machine.

A hard copy printout of the board layout can also be obtained from **place**.

VII. WRAP: TO WIRE-WRAP A BOARD

Three different methods of wiring a circuit board are currently supported: use of hand wiring, use of a Gardner-Denver 14YN semi-automatic wiring machine, and use of a fully automatic Gardner-Denver 14F wiring machine. Interfaces to two on-line machines are under development. One, a Standard Logic WT-600 wiring machine, is an inexpensive semi-automatic device for making wire-wrapped boards (Fig. 4). The other, a custom-made device, is used to assemble circuits using a more recently developed wiring technique called "Quick-Connect."[3] Each requires a different style of output from **wrap**. The format and information content of a hand-wiring list was designed to suit a professional wireperson. The semi-automatic machine reads paper tape and the more complex automatic machine is driven by punched cards. The last of these has been programmed by C. Wild[4] and **wrap** simply transmits the wire net data as input to C. Wild's program which runs on an IBM 370.

The wire list is produced by building a model of the circuit board in which there is a data structure for each socket, chip, and pin. Special signal nets are then broken up into many small nets by routing wires to the nearest special signal pin. On some circuit boards the special signal pins are part of a socket in the position most commonly required by integrated circuit chips. Wires are not required if the socket is used to hold a chip with standard format. However, the special signal pin must be unsoldered if another type of chip occupies the socket. This is one of the special cases that **wrap** handles by modifying the circuit board model before generating the final wire list.

VIII. IMPLEMENTATION

The design aid programs are written in the C language and, for speed and portability, avoid use of floating-point arithmetic. The only assembly code program is a subroutine that uses double-length, fixed-point arithmetic to scale graphic vectors.

Standard operating system features are used. In particular, the graphic terminal is connected as a normal character device that is operated in RAW mode (no operating system interpretation of the character stream).

Twenty separate programs communicate with one another using temporary files. Standard programs such as **sort** and the shell are also used in this way. The four command programs **draw**, **wcheck**, **place** and **wrap** are steering programs that do the necessary "plumbing" to connect together a number of subprocesses. The main work load is performed by the subprocesses. In this way the relatively complex tasks required for circuit design and analysis are broken into manageable pieces.

Memory space requirements are further reduced by using as simple a data structure as possible. For example, the graphic editor represents each chip and wire as a named list of points and shapes. No attempt is made to store cross-reference information in the data base if it can be computed directly. Thus the programs use memory space efficiently and CPU cycles less so.

The largest programs are the drawing editor and the placement program. The drawing editor requires 13,000 words of main memory plus enough space to hold one page of a logic drawing. Typical drawings occupy less than 3,000 words. The placement program requires about 11,000 words of main memory plus space to store a description of the physical characteristics of the circuit. The

latter is approximately 3 words × the number of dual in-line package pins plus 25 words × the number of packages (about 7,300 words for 100 16-pin packages).

Communication with the computing center machines is controlled by a single program whose arguments include the job control language required for each particular purpose. Shell sequences contain the job control instructions for standard tasks using the Honeywell 6070. Different installations of the UNIX operating system require different job control instructions and use different communications facilities. By concentrating the communications functions in one program, the task of installing the design aid programs on different installations is simplified.

IX. CONCLUSION

The objective in preparing these design aid programs included the hope that one day it would be possible to go automatically from schematic circuit drawing to finished circuit board. We have come close to achieving that objective for wire-wrapped boards. However, there is a need to be able to make printed circuit boards from the same circuit drawings. Some steps in this direction were taken in the form of an experimental translator which converts between Circuit Description Language and Logic Simulation Language as used by the LTX system.[5]

X. ACKNOWLEDGMENTS

The design aid programs were a joint project by G. G. Riddle and the author. We were assisted by J. Condon and J. Vollaro, and received valuable help in chosing algorithms from S. Lin and D. G. Schweikert. C. Wild assisted with the interface to the Gardner-Denver machines, and D. Russ contributed to the development of an efficient hand-wiring list. The on-line wire-wrap machine interface was developed by E. W. Stark under the supervision of J. Condon.

REFERENCES

1. S. Lin, "Computer Solutions of the Traveling Salesman Problem," B.S.T.J., *44* (December 1965), pp. 2245-2269.
2. D. G. Schweikert, "A 2-Dimensional Placement Algorithm for the Layout of Electrical Circuits," *Proc. Design Automation Conf.* (1976), pp. 408-416. IEEE Cat. No. 76-CH1098-3C.

3. "New Breadboard Design Cuts Wiring and Testing Time," Bell Labs Record, *53* (April 1975), p. 213.
4. C. Wild, "WIREWRAP—A system of programs for placement, routing and wire-wrapping," unpublished work.
5. G. Persky, D. N. Deutsch, and D. G. Schweikert, "LTX—A Minicomputer Based System for Automated LSI Layout," J. Design Automation and Fault Tolerant Computing, *1* (May 1977), pp. 217-255.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

A Support Environment for MAC-8 Systems

By H. D. ROVEGNO
(Manuscript received January 27, 1978)

An integrated software system based on the UNIX system has been developed for support of the Bell Laboratories 8-bit microprocessor, MAC-8. This paper discusses the UNIX influence on the MAC-8 project, the MAC-8 architecture, the software development and hardware prototyping system, and MAC-8 designer education.*

I. INTRODUCTION

Today's microprocessors perform functions similar to the equipment racks of yesterday. Microprocessor devices are causing a dramatic shift in the economics of computer-controlled systems: product costs and schedules are influenced more by the system architecture and support environment than by the cost or speed of the microprocessor itself. In recognition of this phenomenon, Bell Laboratories has recently introduced a complete set of development support tools based on the UNIX system for its 8-bit microprocessor, MAC-8.[1,2] This paper presents an overview of the MAC-8 architecture and development system.[3]

Development of a microprocessor-based application consists of two activities:

(*i*) Design, construction, and test of the application's hardware.
(*ii*) Design, construction, and test of the application's software.

* UNIX is a trademark of Bell Laboratories.

The development support system described here assists the application designers in both areas. For the hardware designers, a prototyping system that permits emulation as well as stand-alone monitoring of the application's hardware is provided. For the software designers, a high-level language (C), flexible linker-loader, and source-oriented symbolic debugging are supplied. The combination of these tools provides the application designer with a complete and integrated set of tools for system design.

II. WHY UNIX?

At the outset of the MAC-8 development, it was recognized that use of an embedded microprocessor would *increase* the complexity and the scope of applications rather than simply lowering their costs. The tools of the future were going to be programming, documentation, and simulation tools. Considered in this light, a UNIX[4,5] support environment was a natural choice. UNIX possessed many desirable attributes of a "host" environment by providing sophisticated tools for program development and documentation in a cost-effective and highly interactive system. There was already widespread use of the UNIX system not only as a development vehicle in the Business Information Systems Program,[6] but also as part of many "embedded" applications.

III. WHY C?

While the choices of host system and programming language(s) are conceptually independent, there is obvious merit in the consistency of languages and systems. The C language[7] was an obvious choice because it offers high-level programming features, yet also allows enough control of hardware resources to be used in the development of operating systems.

IV. MAC-8

The MAC-8 is a low-cost, single-chip, bus-structured, CMOS microprocessor, whose architecture (Fig. 1) was influenced by the C language. Its major features are:

(*i*) MAC-8 chip, packaged in a 40-pin DIP (dual in-line package), which measures 220×230 mils and uses over 7000 transistors. The chip combines the low power dissipation of CMOS with

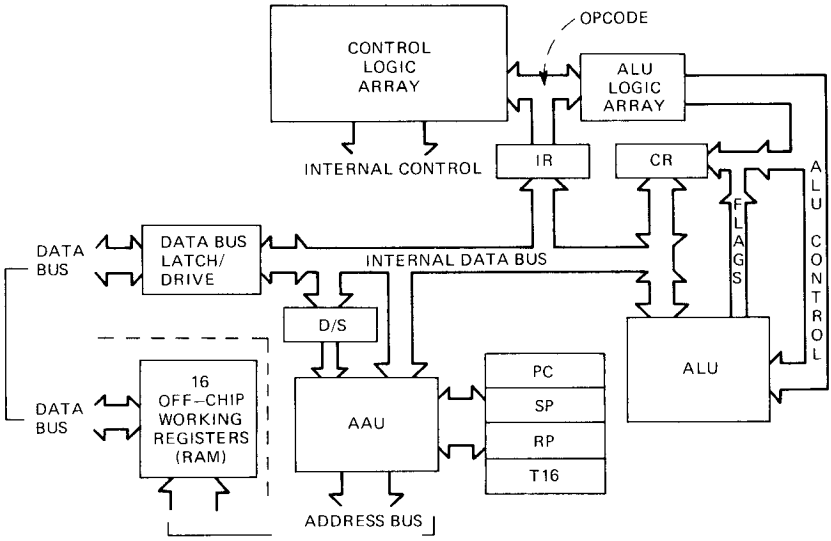

Fig. 1—MAC-8 block diagram.

the gate density of pseudo-NMOS (NMOS with a shared p-channel load transistor).

(*ii*) 16 registers in RAM (random access memory) that are pointed to by the register pointer (a MAC-8 on-chip register). Because of this, the full set of registers can be set aside and a new set "created" by executing only two MAC-8 instructions, which is particularly useful to compiler function-call protocol.

(*iii*) 65K bytes addressable memory space with DMA (direct memory access) capability.

(*iv*) Flexible addressing modes: register, indirect, direct, auto-increment, immediate, and indexed.

(*v*) Communication-oriented CPU (central processing unit) with a wide variety of 8- and 16-bit monadic and dyadic instructions, including arithmetic, logical, and bit-manipulation instructions.

(*vi*) Flexible branch, trap, and interrupt handling.

(*vii*) Processor status brought out to pins, which permits monitoring of CPU activity.

(*viii*) Internal or external clock.

Figure 1 is a block diagram of control. The major blocks are:

(*i*) *Control Logic Array* directs the CPU circuitry through the various states necessary to execute an instruction.

(*ii*) *ALU,* or Arithmetic Logic Unit, performs arithmetic and logical operations.

(*iii*) *ALU Logic Array* controls the operation of the ALU, managing the condition register (CR) flags.

(*iv*) *AAU,* or Address Arithmetic Unit, computes the address in parallel with the ALU operations.

(*v*) *Programmable registers* include the program counter (PC), stack pointer (SP), condition register (CR), and register pointer (RP).

(*vi*) *Internal registers* include instruction register (IR), destination/source register (D/S), and temporary storage register (T16).

V. DEVELOPMENT ENVIRONMENT

The MAC-8 development system (Fig. 2) is an integrated set of software tools, including a C compiler, a structured assembler, a flexible linking loader, a source-oriented simulator, and a source-oriented debugger. All the tools except the debugger reside on

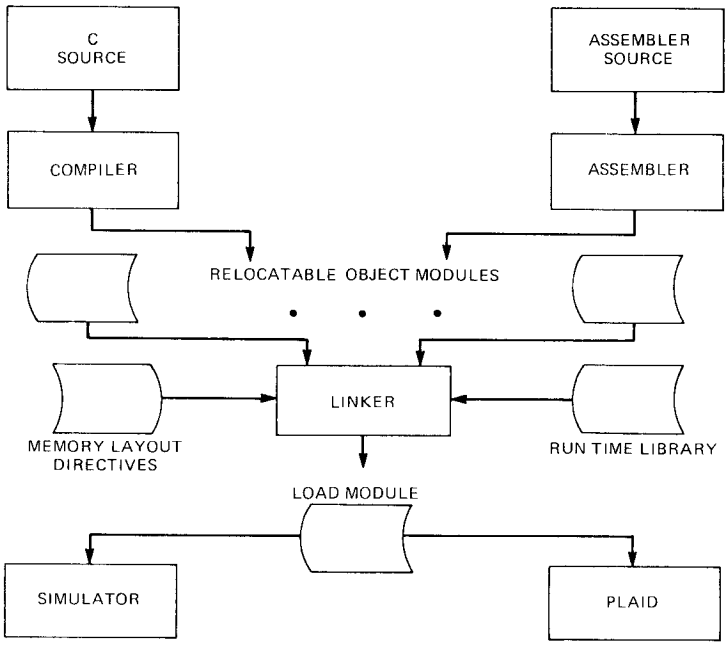

Fig. 2—MAC-8 development system.

UNIX; the debugger resides on a hardware prototyping system called PLAID (Program Logic Aid).

The following sections present a brief discussion of each of those tools.

There is a consistent user interface to all the tools that includes C syntax input language, UNIX file-oriented inter-tool communication, and names analogous to those of the corresponding UNIX tools, e.g., m8cc and m8as.

5.1 MAC-8 C compiler

The MAC-8 C compiler permits the construction of readable and modular programs, due to its structured programming constructs, high-level data aggregates, and a powerful set of operators. C is a good language for microprocessors[8] because it gives control over machine resources by use of primitive data types such as register, character, and pointer variables, and "machine-level" operators such as indirection, "address of," and post/pre- increment/decrement.

5.2 MAC-8 assembler

The MAC-8 assembler is a conventional assembler in that it permits the use of all hardware instructions; it differs from conventional assemblers in the following ways:

(*i*) The language has C-like syntax as illustrated in Fig. 3. For

```
#define NBYTES 100
char  array[NBYTES];
/*
 * Calculates sum of array elements
 */
sum()
{
        b0 = &array;
        a1 = 0;
        for (a2 = 0; a2 < NBYTES; ++a2) {
                a1 =+ b0;
                ++b0;
        }
}
```

Fig. 3—MAC-8 assembler example.

example, a move instruction looks like a C-like assignment statement. Data layout is accomplished by C-like declarations.

(*ii*) The language has structured programming constructs (e.g., if-else, for, do, while, switch) that permit one to write readable, well-structured code at the assembly level. Each construct usually generates more than one machine instruction.

The reserved words in the language identify the MAC-8 registers and also include many of the reserved words of the C language. The **#define** and **#include**, as well as the other features of the C preprocessor, are supported by the assembler.

5.3 MAC-8 loader

The diverse nature of microprocessor applications with their different types of memories and, often, noncontiguous address spaces requires a flexible loader. Besides performing the normal functions such as relocation and resolution of external references, the MAC-8 loader has the following features:

(*i*) Definition of a unit of relocation (section).

```
lowmem    { f4.o(.text) }

.text     { .=0x100 }
.data     { .=0x5000 }
.bss      { .=0x8000 }

f1.o
f2.o

.bss      {
          . = ( . + 7 ) & 0xfff8
          _RPORG =.
          f3.o(.bss)
          _SPORG = .-1
}

highmem   {
          .=0xa000
          f3.o(.data)
}
```

Fig. 4—Input specification.

(*ii*) Assignment of values to unresolved symbols.
(*iii*) Control of the location counter within sections.

These additional features are specified by an input language that has C-like syntax. For example, the input specification of Fig. 4 will take the relocatable object files (output of the compiler or of the assembler) of Fig. 5a and create the absolute binary output files depicted in Fig. 5b. Fig. 5a consists of four files, the first three containing three sections each, namely .text, .data, and .bss, and the last just .text. _RPORG and _SPORG are unresolved symbols that will

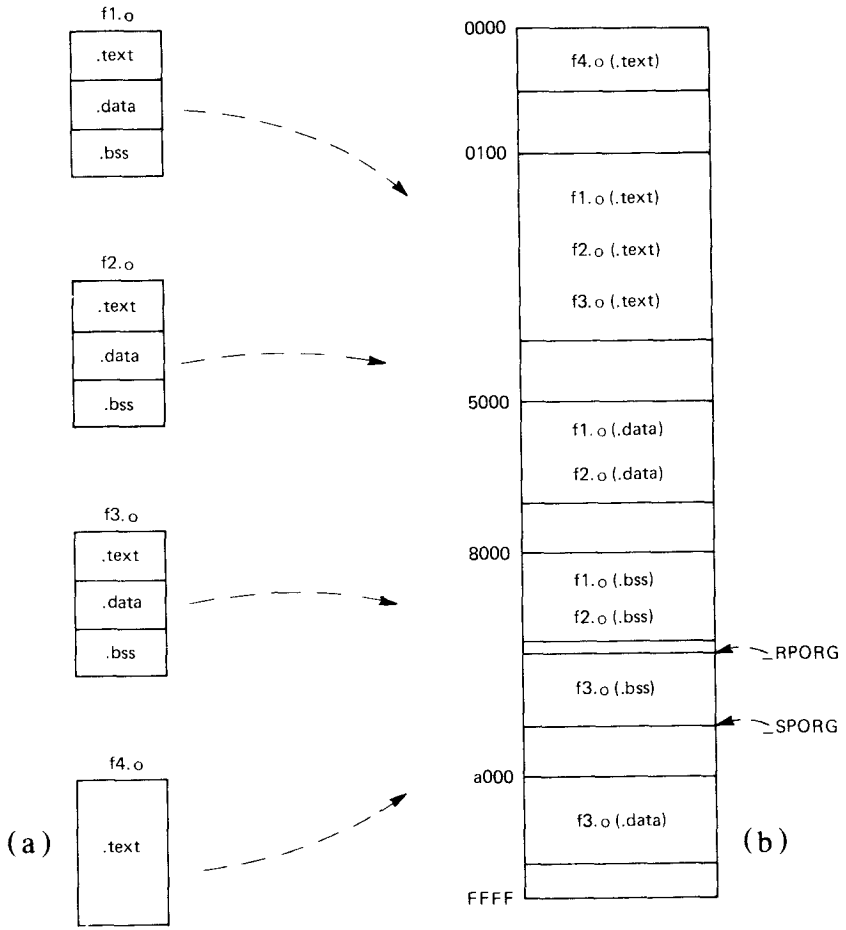

Fig. 5—Loader example.

```
when (23'func1 || glo == 4) {
        if ( flag'func1 ) {
                display i', w;
                userblock;
                continue;
        }
        else display ar'[0] : ar'[n], m'+6;
}
```

Fig. 6—MAC-8 simulator example.

determine initial values of the register pointer and stack pointer, respectively. The expression `.= ( .+ 7 ) & 0xfff8` aligns the `f3.o (.bss)` on a 8-byte boundary.

5.4 MAC-8 simulator

The MAC-8 simulator runs on the UNIX host system and permits debugging of MAC-8 programs without using MAC-8 hardware. The simulator is "source-oriented" and "symbolic," which means that programs can be debugged by referencing variables and function line numbers in terms used on the source listing (compiler or assembler). The symbolic debugging permits the debugging of a C program without worrying about the code generated by the compiler, as illustrated in Fig. 6. The simulator also allows conditional execution of pre-stored procedures of commands and the specification of C expressions containing both user-program and debug symbols, making possible the composition of debugging experiments in an interactive fashion. The C-like input language minimizes the difficulties in changing from one software tool to another. The major features of the MAC-8 simulator are:

(*i*) Loading a program and specifying a memory boundary.
(*ii*) Conditional execution of code and semantic processing when a break point is encountered.
(*iii*) Referencing C identifiers (both local and global) and C source-line numbers on a per-function basis.
(*iv*) Defining a command in terms of other commands to minimize typing.
(*v*) Displaying timing information.
(*vi*) Displaying items in various number bases.
(*vii*) Allocating non-program, user-defined identifiers.

(*viii*) Execution of input commands read from a file on an interactive basis.

(*ix*) Some structured programming constructs, including **if-else** and **while**.

The command illustrated in Fig. 6 will cause a break point when

the program executes line 23 of function **func1** or

the global variable **glo** is equal to 4.

When the break point occurs, if the value of local variable **flag** of function **func1** is non-zero, the values of local variable **i** and global variable **w** are printed, the user-defined block **userblock** is executed, and execution continues; otherwise the contents of local array **ar** for subscripts 0 through *n* and the value of the expression **m'+6** are printed.

5.5 Utilities

Utilities and a library are necessary parts of a support system. The MAC-8 system not only has utilities (like the UNIX system) for determining the size of an object file and the contents of the symbol table, but also a disassembler, a function line-number listing program, and a program to format an object file to permit "down-line" loading into the MAC-8-based application.

5.6 PLAID

A microcomputer-based application or target system typically differs from the host system on which it was developed, particularly in its periphery. Development of a microprocessor application requires hardware/software tools that allow development and debugging in real-time of the target processor and the periphery of its application. The PLAID (Program Logic Aid) system described below is such a tool.

The PLAID hardware includes two MAC-8 systems, each with full memory, and an associated hardware monitor system in a configuration that permits one MAC-8 system (the master) to closely monitor and control the other MAC-8 (the slave). Each MAC-8 has separate I/O, allowing connection to various peripheral devices from the master, and to the application hardware from the slave. The monitor hardware includes various debugging aids, as well as the MAC-cable that allows in-circuit control of any MAC-8 system.

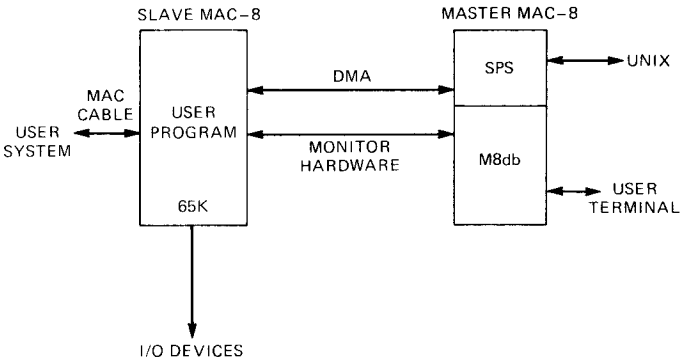

Fig. 7—Level-1 PLAID.

The PLAID software system includes the Satellite Processor System (SPS), which communicates with a host UNIX system, performing operating system functions for the master and monitor hardware, and **m8db**, a source-oriented symbolic debugger whose capability is similar to the MAC-8 simulator with the addition of real-time break points by use of the PLAID monitor system.

In the early stages of development, a fully-instrumented MAC-8 within the PLAID serves as the processor for the target machine, where in later stages, the PLAID monitors and controls the prototype system. Level-1 PLAID is illustrated in Fig. 7. Work is being done on level-2 PLAID, illustrated in Fig. 8. The fundamental difference in hardware between levels 1 and 2 is in the master system, which in level 1 contains a 32K PROM (programmable read-only memory) and a 16K RAM, while level 2 contains a 65K RAM and a dual-drive double-density floppy disk. The SPS executive is replaced in level 2 by a single-user UNIX system; the debugger can be swapped in from the floppy disk, as can other tools.

The satellite processing system of level 1, which is functionally similar to the system described in Ref. 9, resides in the master and controls the flow of program execution. SPS permits communication with a host UNIX system via a dial-up connection, and performs the operating system functions for **m8db**, such as control of the master and monitor hardware. Any UNIX command can be entered at the user terminal (see Fig. 7) and SPS determines whether the command will be processed by PLAID or must be transmitted to UNIX. The SPS interface to **m8db** consists of UNIX-like system calls.

The PLAID-resident symbolic debugger, **m8db**, has a command language which is a superset of the MAC-8 simulator. The additions to the language permit the referencing of the PLAID monitor system

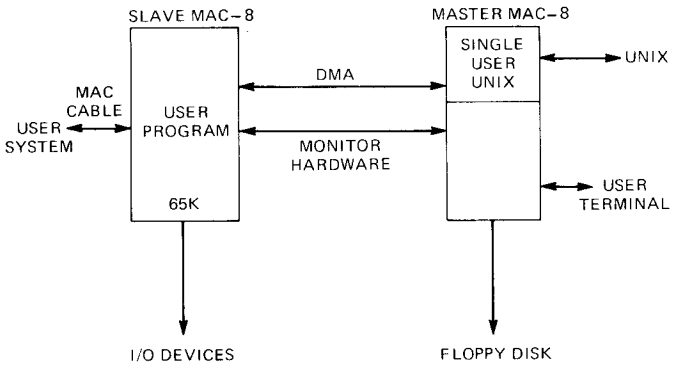

Fig. 8—Level-2 PLAID.

hardware to establish real-time breakpoints. **m8db** has all the features mentioned in Section 5.4, as well as facilities to help debug programs in real time.

The MAC-cable is the channel of communication between the PLAID and the application hardware, and permits control of the MAC-8-based system. During the initial stages in the development of an application, the MAC-cable permits testing out of the slave's MAC-8 and memory, while using the application's I/O. As the development progresses, the MAC-cable permits testing the application's hardware, including its MAC-8 and memory.

The PLAID monitor system keeps track of the actions of the slave or user system enabling the debugging of MAC-8 applications in a real-time environment. The major features of the PLAID monitor system include:

(*i*) *Memory monitor* contains a 65K by 4-bit dynamic RAM that enables trapping ("break-pointing") on a variety of conditions such as:
  (*a*) Memory read, write, or reference.
  (*b*) Arbitrary 8-bit pattern in memory.

(*ii*) *Register monitor* enables "break-pointing" on a read, write, or reference of any of the MAC-8 off-chip registers (see Fig. 1).

(*iii*) *CPU monitor* contains shadow registers that hold the current values of the slave/user MAC-8 on-chip registers (CR, SP, RP, PC), as shown in Fig. 1.

(*iv*) *Event counters* consist of three 16-bit counters and one 32-bit counter that enable "break-pointing" on a variety of events. The events include:

(*a*) Slave/user interrupt acknowledge.
(*b*) Slave CPU clock output.
(*c*) Slave/user memory read, write, or reference.
(*d*) Opcode fetch.
(*e*) Trap.
(*f*) User-supplied backplane signal.

(*v*) *Jump trace* contains a history table of the last 32 program counter discontinuities.

(*vi*) *Instruction trace* consists of a 64-entry trace table of the last 64 cycles executed by the slave.

(*vii*) *Memory access* circuitry permits the selective specification, on a block (256 bytes) basis, of read/write protection. Memory timing characteristics can also be specified on a block basis.

(*viii*) *Clock frequency* for the slave can be selected from a list of predefined frequencies.

VI. DESIGNER EDUCATION

Because of the nature of microprocessor applications, designers must have both hardware and software expertise. Many hardware designers must write programs for the first time, which poses an interesting educational problem. C is a difficult language for nonprogrammers because it is both powerful and concise. This problem can be partially remedied by giving seminars and supplying tutorials on C and on programming style. Offering workshops on the hardware and software aspects of the MAC-8 has also helped.

The MAC-tutor, an "electronic textbook," enables the designer to learn MAC-8 fundamentals. The MAC-tutor is a single-board computer with I/O and can be communicated with by a 28-function keypad or by a terminal. A connection to a UNIX system can also be established for use in loading programs and data into the MAC-tutor memory. The MAC-tutor also includes an executive to control hardware functions, 1K RAM expandable to 2K, sockets for three 1K PROMs, eight 7-segment LED displays, a PROM programming socket, and peripheral interfaces to a terminal and a cassette recorder. The tutor, besides being an educational tool, may be used to develop small MAC-8-based applications.

VII. SUMMARY

The MAC-8 was designed together with an integrated support system. The MAC-8 architecture was influenced by the C language, and

the support tools were influenced by UNIX. The consistent use of C-like language syntax permits easy transition from one tool to another. Software tools that began, in many cases, as spinoffs of existing UNIX tools have evolved to meet the needs of microprocessor applications. The MAC-8 support system continues to evolve to meet the growing needs of microprocessor-based applications.

VIII. ACKNOWLEDGMENTS

The design and development of the MAC-8 and its support system required the cooperative efforts of many people over several years. The author wishes to acknowledge the contributions of all the team members whose work is summarized here.

REFERENCES

1. J. A. Cooper, J. A. Copeland, R. H. Kranbeck, D. C. Stanzione, and L. C. Thomas, "A CMOS Microprocessor for Telecommunications Applications," IEEE Intl. Solid-State Circuits Conf. Digest, *XX* (February 17, 1977), pp. 138-140.
2. H. H. Winfield, "MAC-8: A Microprocessor for Telecommunications," The Western Electric Engineer, special issue (July 1977), pp. 40-48.
3. I. A. Cermak, "An Integrated Approach to Microcomputer Support Tools," Electro Conference Record, session 16/3 (April 1977), pp. 1-3.
4. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
5. D. M. Ritchie, "UNIX Time-Sharing System: A Retrospective," B.S.T.J., this issue, pp. 1947-1969.
6. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," B.S.T.J., this issue, pp. 2177-2200.
7. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," B.S.T.J., this issue, pp. 1991-2019.
8. H. D. Rovegno, "Use of the C Language for Microprocessors," Electro Conference Record, session 24/2 (April 1977), pp. 1-3.
9. H. Lycklama and C. Christensen, "UNIX Time-Sharing System: A Minicomputer Satellite Processor System," B.S.T.J., this issue, pp. 2103-2113.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

No. 4 ESS Diagnostic Environment

By S. P. PEKARICH
(Manuscript received December 2, 1977)

Maintenance and testing of the Voiceband Interface and the Echo Suppressor Terminal in No. 4 ESS are aided by software in the 1A processor. A No. 4 ESS diagnostic environment was needed during the development phase of both the hardware and diagnostic software. The minicomputer chosen to support this environment was also required to support the development of more than one frame at the same time. Because of this requirement and other reasons, the UNIX operating system was selected for software support and development. This paper describes how the UNIX operating system was applied to this project.*

I. INTRODUCTION

Software in the 1A processor is used to maintain and test the Voiceband Interface and the Echo Suppressor Terminal in No. 4 ESS.[1] These testing programs were written in a high-level language oriented to the special requirements of diagnostics in an ESS environment. A No. 4 ESS diagnostic environment was needed during the development phase of both the hardware and diagnostic software. Digital Equipment Corporation's PDP-11/40 minicomputer and the UNIX operating system[2] were chosen to support this environment.

* UNIX is a trademark of Bell Laboratories.

II. VOICEBAND INTERFACE AND ECHO SUPPRESSOR TERMINAL

The Voiceband Interface[3] (VIF) provides an interface between analog transmission systems and the digital time-division switching network of No. 4 ESS.[4,5] A VIF contains up to seven active and one spare Voiceband Interface Units (VIU's). Each VIU terminates 120 four-wire, voice-frequency analog trunks and performs the analog-to-digital and digital-to-analog conversions necessary for interfacing with the time-division network.

The Echo Suppressor Terminal[6] (EST) is inserted (when required) between the VIF and the time division network. Through use of digital speech processing techniques and by operating in the multiplexed DS120 bit stream, the EST achieves about a 10:1 cost reduction over the analog echo suppressor it replaced.

III. MAINTENANCE OF VIF AND EST

Maintenance software for No. 4 ESS[7,8] can be functionally divided into three categories:

(*i*) Detect and recover from software malfunctions.
(*ii*) Detect and recover from hardware faults.
(*iii*) Provide error analysis and diagnostic programs to aid craftspersons in the identification and replacement of faulty modules.

This paper discusses how the UNIX operating system was applied to aid the development of diagnostic programs for VIF and EST.

Figure 1 shows the maintenance communication path between the 1A processor and the VIF and EST. The 1A processor issues maintenance commands to the VIF through maintenance pulse points from a signal processor.[5] The VIF replies to the 1A via the peripheral unit reply bus (PURB). The EST communicates with the 1A through a full peripheral unit bus[5] (PUB). EST commands are issued from the 1A via the peripheral unit write bus (PUWB), and the replies return by way of the PURB.

IV. NO. 4 ESS DIAGNOSTIC ENVIRONMENT UNDER THE UNIX SYSTEM

During the hardware and diagnostic software development phase of both the VIF and EST, a No. 4 ESS diagnostic environment had to

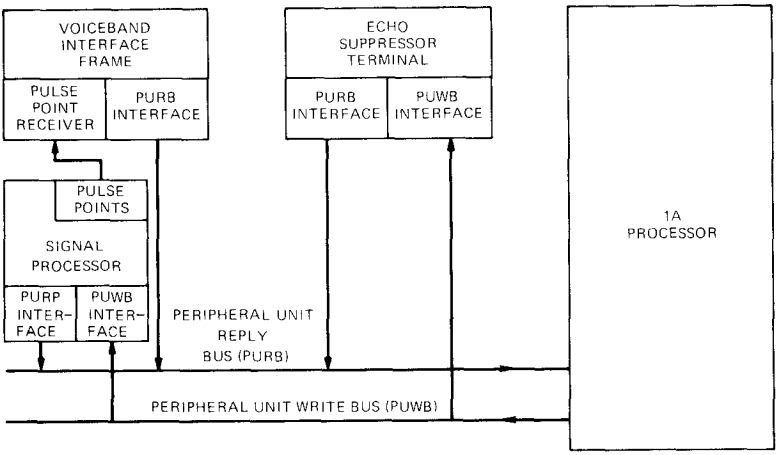

Fig. 1—Communication path between 1A Processor and VIF and EST.

be supported. Since a 1A processor was not available for the development of VIF and EST, a minicomputer was used to simulate the diagnostic functions. Figure 2 shows the No. 4 ESS diagnostic support environment which was created. Special hardware units

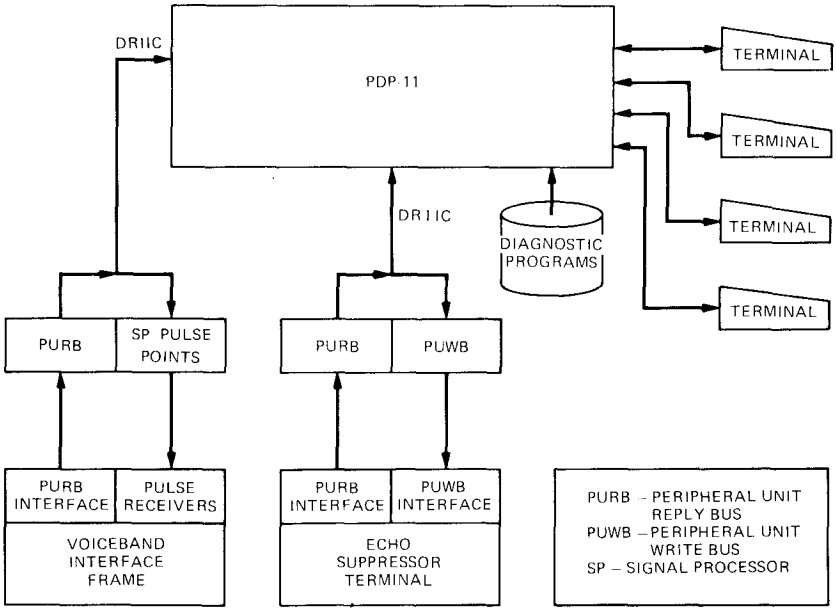

Fig. 2—No. 4 ESS diagnostic support environment.

were developed to provide electrically compatible interfaces representing the peripheral unit bus and the signal processor maintenance pulse points. These units are controlled by the minicomputer through standard computer interfaces. The minicomputer then performs the diagnostic functions of the 1A processor in a No. 4 office. By issuing commands to the special interface units, the minicomputer simulates the 1A processor's transmission of diagnostic instructions to the individual frames. The majority of the diagnostic software for VIF and EST was developed in this diagnostic environment.

Software development began under the disk operating system (DOS), supplied by the vendor. DOS is a "single-user" system; that is, only one software designer can use the machine at any given time. This limitation was acceptable early in the project when all the computer time was dedicated for software development. However, as support software became available, the hardware and diagnostic software test designers became heavy users of the system.

At the start of the project, it was realized that the minicomputer system was required to support more than one frame. In addition, support software development effort was still continuing, which now presented a problem in scheduling the minicomputer system. Two alternate solutions were considered. The first was to purchase another minicomputer to support the development effort on the second frame. A disadvantage of this proposal was that one of the minicomputer systems still had the scheduling problem with support software development and with supporting the frame. Also, supporting additional frames would cause the problem to arise again. The second alternative was to upgrade the minicomputer system so that it could support time-shared operations. This seemed a more economical way of supporting additional frames and support software development. The second alternative was chosen.

Two time-sharing systems were available for the PDP-11 computer, UNIX and RSX-11. The RSX-11 system was basically the single-user DOS, upgraded to support multiple users. Its main advantage was its upward compatibility with programs developed under DOS. The UNIX operating system, on the other hand, offered a better development environment and more support software tools than DOS. The C language was also available, which presented a very attractive alternative for developing new software. These advantages outweighed the disadvantage of having to modify the existing software developed under DOS. Therefore, the UNIX operating system was selected to support this project.

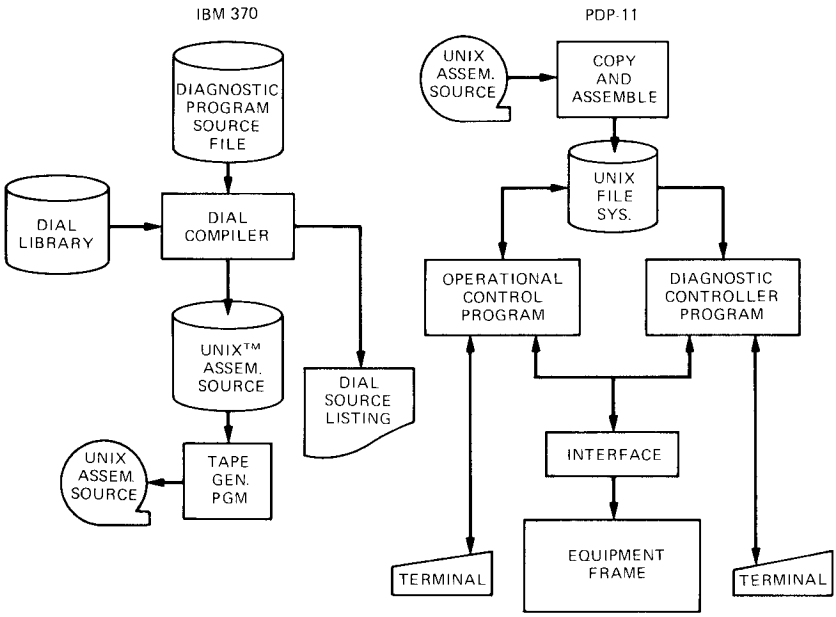

Fig. 3—Software for No. 4 ESS diagnostic environment.

V. THE SOFTWARE SYSTEM

Figure 3 is a block diagram of the overall software system used to support the No. 4 ESS diagnostic environment. It consists of an off-line compiler for a special diagnostic programming language (described below), a run-time monitor for execution and debugging of diagnostic programs, an on-line compiler for hardware and software debugging aids (denoted as Operational Control), and support programs for creating and maintaining the diagnostic data base. The entire software system on the minicomputer was named PADS (*P*DP-11 *A*ided *D*iagnostic *S*imulator).

5.1 DIAL compiler

DIAL (*Dia*gnostic *L*anguage)[8] is a high-level programming language that allows a test designer to write a sequence of test instructions with data using macro calls. The language was developed to meet the special requirements of diagnostics in the ESS environment. DIAL statements can be divided into two classes: testing statements and general purpose statements. Testing statements

are used to issue maintenance commands to a specified peripheral unit in a No. 4 ESS office. The general purpose statements are similar to most other high-level languages. They manipulate data, do arithmetic and logical functions, and control the program execution flow. The diagnostic programs for both the VIF and EST were written in DIAL.

Several DIAL compilers are available to diagnostic program designers. Each compiler produces code for a different application. The compilers of interest to the VIF and EST test designers are the DIAL/ESS and the DIAL/PADS compiler. The DIAL/ESS compiler, developed using the TSS SWAP (*S*witching *A*ssembler *P*rogram),[9] produces a data table which is interpreted by a diagnostic control program[8] in the No. 4 ESS office. The DIAL/PADS compiler, developed using VMSWAP (*V*irtual *M*emory SWAP), produces code for a PDP-11. This compiler, which runs on the IBM 370 computer, produces PDP-11 assembly language source code which is acceptable to the UNIX assembler. The assembly language code is subsequently transported to the UNIX file system where it is assembled. The resultant object modules are usable with the run-time monitor in PADS.

Consideration was given to implementing the DIAL/PADS compiler directly on the UNIX system. This would have eliminated the need for the IBM 370 computer in the diagnostic development effort. All software development could have been performed on the UNIX system. However, because of the lack of the necessary staff and computer resources, this approach was abandoned.

5.2 No. 4 ESS run-time environment

The PADS system allows the test designer to execute a DIAL program with the aid of a run-time monitor and debugging software package called DCON (*D*iagnostic *Con*troller). The debugging package in DCON provides a tool for evaluating diagnostic algorithms and debugging DIAL programs. DCON facilities allow the user to:

(*i*) Trace the execution of DIAL statements.
(*ii*) Pause before execution of each DIAL statement.
(*iii*) Pause at DIAL statements selected at run time.
(*iv*) Display and modify simulated 1A memory during a pause.
(*v*) Start execution of the DIAL program at any DIAL statement.
(*vi*) Skip over selected DIAL statements at run time.
(*vii*) Loop one or a group of DIAL statements at run time.

This last feature was especially useful for hardware troubleshooting. Using this debugging package, the test designer can follow the execution of a DIAL program while it is diagnosing the VIF or EST.

DIAL programs compiled by the DIAL/PADS compiler communicate data and status information to the diagnostic controller program. Under DOS, this communication link was established at run-time via the PDP-11 "trap" instruction. After switching to the UNIX operating system, the "emulator trap" (**emt**) instruction was used. The DCON process used the **signal()** system call to catch the **emt** instruction executed by the DIAL program. However, because of the large number of **emt** instructions executed in the DIAL program and the operating system overhead to catch and handle this signal, this method of dynamic linking between DCON and DIAL programs had to be abandoned. It was replaced by the jump subroutine instruction and loading a register with an address at run-time.

Maintenance instructions are sent to the VIF and the EST through general purpose I/O ports (DR11Cs) on the minicomputer. Originally, DCON relayed the maintenance instructions of DIAL programs using the **read()** and **write()** system services of the UNIX operating system. Before executing a DIAL program, DCON opened the appropriate files (general purpose I/O ports) and retained their file descriptors. All subsequent maintenance instructions requested by the DIAL program were handled by DCON as **read()** or **write()** requests to the file. Measurement of the I/O activity on these ports revealed that the VIF diagnostic program sent a large number of maintenance instructions. Hence, a large portion of the diagnostic run time was operating system overhead. Based on this observation, special system service routines were added to the UNIX operating system. These routines directly read and write the general purpose I/O ports. After implementing the I/O for maintenance instructions in this manner, the run time for the VIF diagnostic was reduced by more than half.

The PADS system simulates the automatic running of diagnostics as performed in a No. 4 ESS office. A complete diagnostic for a peripheral unit is normally written in many functional blocks called phases. Each phase is a self-contained diagnostic program designed to diagnose a small functional part of the unit. The phases of the diagnostic program are stored as load modules in the UNIX file system. The PADS system automatically searches the directory containing the diagnostic phases for the peripheral unit. Each phase is loaded into memory by PADS and execution control is given to it. At the termination of each phase, control is returned to the run-

time monitor program which will determine the next phase to be executed. The diagnostic phases for different frames are stored in separate subdirectories. The files are named using a predefined naming convention. This allows DCON to automatically generate the file name for the next diagnostic phase to be loaded and executed.

5.3 Support programs

The output from the DIAL/PADS compiler is UNIX assembler source code. The DIAL assembly source code is placed on a magnetic tape for transport to the PDP-11 system. A shell procedure on the UNIX system reads the tape and invokes the UNIX assembler. The output from the assembler is then renamed to the file name supplied by the user in the shell argument.

The UNIX shell program[10] is also used to interpret shell procedure files which simulate the automatic scheduling of diagnostics in the central office. These procedure files call special programs which monitor the equipment for a fault indication. After a fault is detected, the shell procedure calls in the DCON program to run the frame diagnostics.

5.4 On-line diagnostic aids

A software package known as "Operational Control" was developed under the UNIX system using the C compiler.[11] This program allows the user to issue operational commands to the frame by typing in programming statements at his terminal. These statements are compiled directly into PDP-11 machine code and may be executed immediately. Test sequences may be developed directly on-line with the frames. These programs can then be saved in files for future usage. This last feature was extremely easy to implement in the UNIX operating system. By altering the file descriptor from standard input or output to a file, common code reads and writes the program from the terminal or a file.

The parser for the Operational Control language is recursive. This type of parser was exceptionally easy to implement in C since the language allows recursive programming. The management of storage variables needed by the parser in recursive functions was automatically performed by the C compiler. This would have been a horrendous bookkeeping operation if a nonrecursive programming language had been used. The block structuring features of C made it easy to implement the parser from the syntax definition of

Operational Control. Using C, the on-line compiler was defined, implemented, and operational within a short time period.

VI. UNIX SYSTEM SUPPORT

The UNIX time-sharing system enables the minicomputer system to support more than one frame at a time. Each frame has a dedicated, general-purpose I/O port from the computer. The user supplies the frame selection information to the controlling program (either DCON or Operational Control). Then the software opens the appropriate I/O port for future communications with the frame.

Another feature provided by PADS is to allow the diagnostic debugging program to escape to the on-line compiler program. This is done in a manner such that when the user exits the on-line compiler, he immediately returns to DCON at the state at which it was left. This feature was very easy to implement under the UNIX operating system. By using the **fork()** and **exec()** system calls, the parent program (DCON) sleeps, waiting for the death of its child process (Operational Control). When the user exits from Operational Control, DCON is awakened and will continue its execution from the last state it was left in. The concept of placing one process to sleep and invoking another was not available on RSX-11.

VII. SUMMARY

The UNIX time-sharing system had many advantages over the RSX-11 system that was considered for the PADS system. Originally, PADS was developed under Digital Equipment Corporation's single-user Disk Operating System (DOS) using the macro assembler MACRO-11. When it became apparent that a second frame had to be supported, a time-sharing system was considered. At that time only two time-sharing systems were available for the PDP-11 computer, UNIX and RSX-11.

DEC's RSX-11 system is a disk operating system which supports multiple users. The basic advantage of RSX-11 was its upward compatibility with programs written to run under the disk operating system. However, this advantage was outweighed by the advantages the UNIX operating system presented.

In our opinion, the UNIX operating system provided a much better software development environment for our purpose than RSX-11. In particular, the C language is much better suited to systems programming than a macro assembler or Fortran. Also, many excellent

software tools such as the text editor and the linker existed on the UNIX system. For example, it was felt that with the aid of the text editor, all the programs that were written in MACRO-11 assembly language could easily be translated into the UNIX assembler language. This allowed the existing software to come up under the UNIX system in a short time period. Then, as time allowed, the existing software could be rewritten in C. All future software was slated to be written in C. The need to rewrite the software gave an opportunity to rethink the entire software project, this time with some experience. In the end, this led to vast improvements in the software packages.

REFERENCES

1. "1A Processor," B.S.T.J., *56* (February 1977).
2. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
3. J. F. Boyle, et al., "No. 4 ESS: Transmission/Switching Interfaces and Toll Terminal Equipment," B.S.T.J., *56* (September 1977), pp. 1057-1098.
4. "No. 4 ESS," B.S.T.J., *56* (September 1977).
5. J. H. Huttenhoff, et al., "No. 4 ESS: Peripheral System," B.S.T.J., *56* (September 1977), pp. 1029-1056.
6. R. C. Drechsler, "Echo Suppressor Terminal for No. 4 ESS," Intl. Conf. on Communications, *3* (June 1976).
7. P. W. Bowman, et al., "1A Processor: Maintenance Software," B.S.T.J., *56* (February 1977), pp. 255-289.
8. M. N. Meyers, et al., "No. 4 ESS: Maintenance Software," B.S.T.J., *56* (September 1977), pp. 1139-1168.
9. M. E. Barton, et al., "Service Programs," B.S.T.J., *48* (October 1969), pp. 2865-2896.
10. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," B.S.T.J., this issue, pp. 1971-1990.
11. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," B.S.T.J., this issue, pp. 1991-2019.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

RBCS/RCMAS—Converting to the MERT Operating System

By E. R. NAGELBERG and M. A. PILLA
(Manuscript received February 3, 1978)

The paper presents a case history in applying the MERT executive to a large software project based on the UNIX system. The work illustrates some of the basic architectural differences between the MERT and UNIX systems as well as the problems of portability. Emphasis is on matters pertaining to software engineering and administration as they affect development and support of a manufactured product.*

I. INTRODUCTION

The Record Base Coordination System (RBCS) and Recent Change Memory Administration System (RCMAS) are two minicomputer-based products designed to carry out a record *coordination* function; i.e., they accumulate segments of information received from various *sources* on different media, filter, translate, and associate related data, and later transmit complete records to downstream *user* systems, also on an assortment of media and according to various triggering algorithms. The overall objective of record coordination is to assure that information stored and interchanged among a variety of related systems is consistent and accurately reflects changes that must continually occur in the configuration of a large, complex telecommunications network. To perform this function, RBCS and

* UNIX is a trademark of Bell Laboratories.

RCMAS both provide various modes of I/O, data base management, etc., and each utilizes a rich assortment of operating system features for both software development and program execution.

The RBCS project was initially based on the UNIX system,[1] making use of its powerful text processing facilities, the C language compiler,[2] and the program generator tools Yacc and Lex.[3] However, after a successful field evaluation, but just prior to development of the production system, it was decided to standardize using the MERT/UNIX* environment.[4] To a very large extent, this decision was motivated by the genesis of a second project, RCMAS, which was to be carried out by the same group using the same development facilities as RBCS, but which had much more stringent real-time requirements. The additional capabilities of the MERT executive, i.e., public libraries and powerful inter-process communication using shared segments, messages, and events, were attractive for RCMAS and, even though the RBCS application did not initially utilize these features, the MERT operating system was adopted here as well for the sake of uniformity.

The purpose of this paper is to present what amounts to a case history in transporting a substantial application, RBCS, from one operating system environment, the UNIX system, to another, the MERT/UNIX system. It is a user's view in that development and ongoing support for the operating system and associated utilities were carried out by other organizations. Of course, these programs were also undergoing change during the same period as the RBCS conversion. On the one hand, these changes can be said to confound the conversion effort and distort the results, but on the other hand a certain level of change must be expected and factored into the development process. This issue is referred to later as an important consideration in determining system architecture.

The paper begins with a discussion of the reasons for choosing a UNIX or MERT/UNIX environment, followed by analysis of the decision to utilize the MERT executive. The transition process is described, and a section on experience in starting with the MERT system, for purposes of comparison, is added. Throughout, the emphasis is on matters pertaining to software engineering and administration as they affect development and support of a manufactured product.

*MERT executive, UNIX supervisor program.

II. WHY THE UNIX AND MERT/UNIX OPERATING SYSTEMS?

With the exception of the interprocess communication primitives, the UNIX and MERT operating systems appear to the user as fairly equivalent systems. In large part, this is due to (*i*) the implementation of the UNIX system calls as a supervisory process under the MERT executive and (*ii*) the ability to use existing C programs such as the UNIX editor and shell with no modifications. Throughout this paper, unless emphasis of a particular MERT/UNIX feature is required, the phrase "UNIX operating system" will be used to mean the stand-alone UNIX operating system as well as the UNIX supervisor under MERT.

The facilities which the UNIX system provides for text processing and program development, the advantages of a high-level language such as C, and the power of program generators such as Yacc and Lex are described elsewhere in this issue. All these were factors in the final decision. However, the most compelling advantage of the UNIX system was the shell[5] program, which permitted the very high degree of flexibility needed in the RBCS project.

Because of the important role of field experience in the design process, developing a system such as RBCS or RCMAS is difficult, since a complete specification does not exist before trial implementation begins. Recognizing this fact, the developing department decided to make heavy use of the UNIX shell as the *primary* mechanism for "gluing together" various utilities. Use of the shell allows basic components to be bound essentially at run-time rather than early in the design and development stages. This inherent flexibility permits the system designer to modify and/or enhance the product without incurring large costs, as well as to separate the development staff into (*i*) "tool" writers, i.e., those who write the C utilities that make up the "gluable" modules and (*ii*) "command" writers who are not necessarily familiar with C but do know how a set of modules should be interconnected to implement a particular task.

In actual practice, some overlap will exist between the two groups if for no other reason than to properly define the requirements for the particular C utilities. The RBCS and RCMAS developments followed this approach and the evaluations of the projects overwhelmingly support the ease of change and administration of features. The response of the end users to the RBCS field evaluation system, in particular, has been most encouraging. In fact, they have written a few of their own shell procedures to conform more easily to local methods without having to request changes to the standard product.

In addition to the shell, heavy use of both Yacc and Lex was made to further reduce both projects' sensitivity to changes of either design or interface specification. For example, some processing of magnetic tapes for RBCS has been accomplished with parsers that produce output to an internal RBCS standard. For RCMAS, various CRT masks are required for entry and display of important information from its data base. An example is shown in Fig. 1. These masks, being character strings tailored to the format of the specific data to be entered or displayed, lend themselves very naturally to design and manipulation using Yacc and Lex program generators. This allows rapid changes (even in the field) to the appearance of the CRT masks without incurring the penalty of modifying the detailed code for each customized mask.

Documentation for manufacture and final use was provided using the nroff[6] text formatting system. In addition, to make the system easier to use and to obtain uniformity of documentation, the UNIX form[7] program was utilized to provide templates for the various documentation styles.

With two development projects, short deployment schedules, and limited computer resources in the laboratory, it was necessary to *develop*, *test*, and *manufacture* software on the same machine. The software engineering required to support such simultaneous operations would have been difficult, if not impossible, without the various UNIX features.

III. WHY THE MERT SYSTEM?

If the UNIX operating system performed so well with its powerful development tools and flexible architecture possibilities, why, then, was RBCS converted from the UNIX to the MERT operating system and RCMAS directly developed under the MERT operating system? To answer this question, it is important to examine the underlying software engineering and administration problems in a project such as RBCS.

The organization responsible for end-user product engineering and design for manufacture should regard themselves *as users of an operating system,* with interest centered around the application. Once end-user requirements are defined, architectural and procedural questions should be resolved at all levels (*i*) to minimize change and (*ii*) to avoid the propagation of change beyond the region of immediate impact. Once the development is under way, changes of

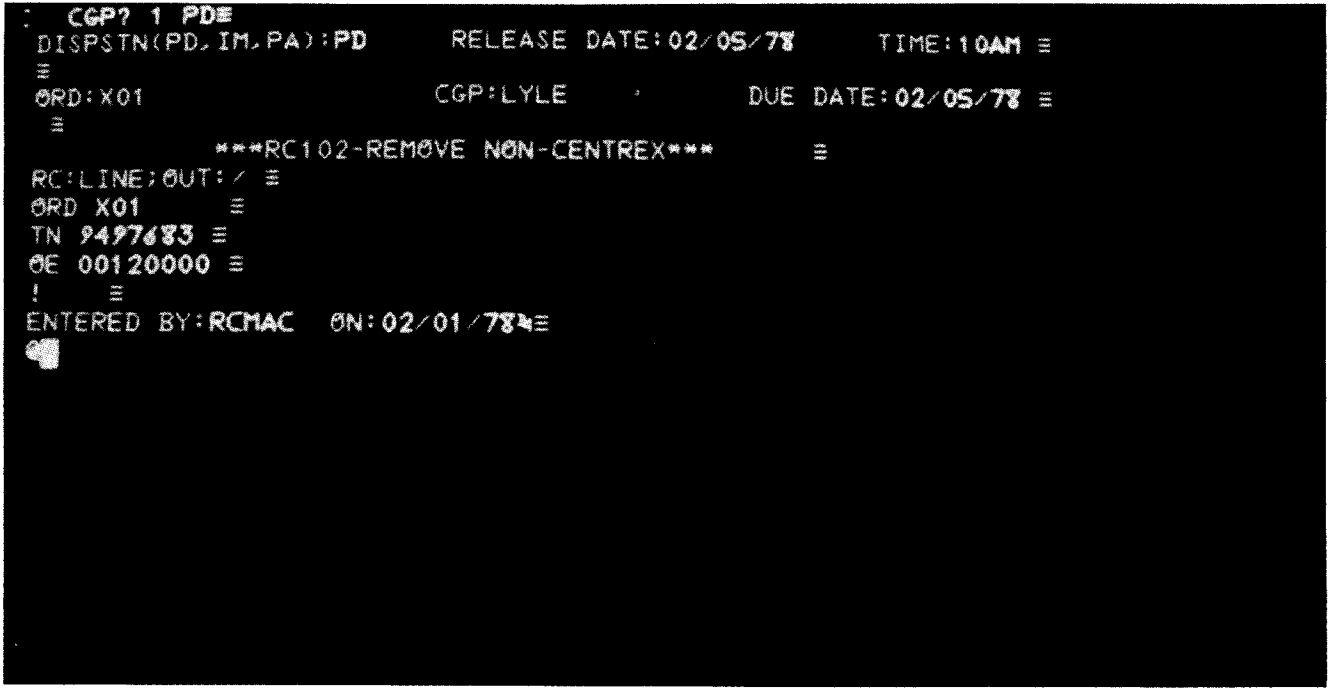

Fig. 1—RCMAS CRT mask.

any sort are undesirable if adequate administrative control is to be exercised, schedules met, and the product maintainable.

The nature of the RBCS project, however, necessitated changes to the UNIX operating system in three important areas: (*i*) a communications interface to support Teletype Corporation Model 40 CRTs, (*ii*) multifile capability for magnetic tape (e.g., standard IBM-labeled tapes), and (*iii*) interprocess communication for data base management. It was the problem of satisfying RBCS requirements in these areas under an ever-increasing rate of modifications to the "standard" UNIX system that led to a consideration of the MERT operating system.

Figure 2 graphically illustrates the problem of making slight modifications to the executive program of the UNIX operating system, especially in the area of input/output drivers. Because the UNIX system, including all drivers, is contained in a single load module, *any* modifications, even those at the applications interface, require careful scrutiny and integration, including a new system generation followed by a reboot of the system. On the other hand, referring again to Fig. 2, modifying a driver in a MERT environment is considerably easier because of the modular architecture of the MERT executive. In particular, the MERT executive is divided into several components, which can be administered and generated separately, frequently without a reboot of the system.

The experience over an 18-month period with the MERT operating system is that no RBCS or RCMAS driver was affected by changes to the MERT system. RBCS modifications to the magnetic tape and communications drivers were frequently tested *without* generating another system and, quite often, without rebooting. The advantages of not having to regenerate or reboot become obvious after a few iterations of change.

Furthermore, as shown in the encircled area of Fig. 2, it should be feasible, ultimately, for a project such as RBCS or RCMAS to receive the UNIX supervisor, MERT file manager, and MERT executive as object code rather than as source code. Both RBCS and RCMAS change only sizing parameters in these modules; such parameters could be administered after compilation, thus freeing a project from having to track source code for the major portions of the operating system.

With regard to interprocess communication, the translation and data-base management aspects of RBCS place a heavy strain on any operating system, but on the UNIX system in particular. For example, since the UNIX system was designed to provide a multi-access,

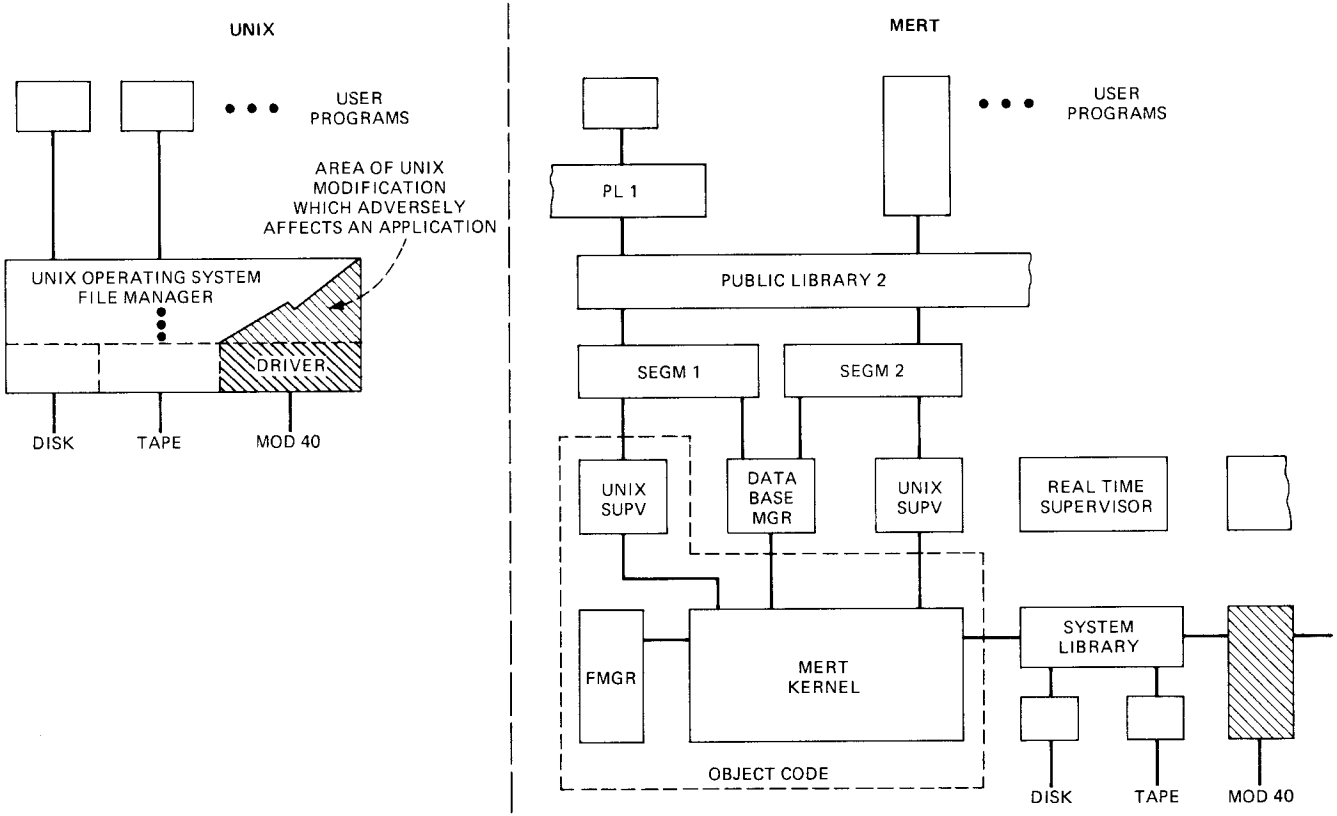

Fig. 2—UNIX™ and MERT executive architectures with possible application interfaces.

time-sharing environment in which a number of independent users manipulate logically separate files, a large body of common code, such as a data base manager, could only be implemented via a set of subroutines included with each program interacting with the data base. With such a structure, no natural capability exists to prevent simultaneous updates, and special measures become necessary. In general, it is difficult to allow any asynchronous processes (nonsibling related) to communicate with each other or to synchronize themselves without modifying the operating system or spending inordinate amounts of time with extra disk I/O.

Several measures were taken to try to overcome these limitations. Where sibling relationships existed by virtue of processes having been spawned by the same shell procedure or special module, UNIX pipes could be used, but these are quite slow in a heavily loaded system. Where nonsibling related processes were involved, specialized file handlers had to be used since large amounts of shared memory (within the operating system) were out of the question. It is important that the shared memory be contained in *user* space since different applications, at different points during processing, may place inordinate demands on a system-provided mechanism for sharing memory. This point is discussed further in the next section.

After considerable experience with the RBCS field evaluation, the time came to incorporate what had been learned and design a production system. It became quite clear that interprocess communications was our primary bottleneck and that this would be even more the case for RCMAS. An operating system was needed that could (*i*) support better interprocess communication than the UNIX operating system, (*ii*) support feature development (modifications to the operating system) without tearing into the kernel of the system or unnecessarily inconveniencing users by system generations, reboots, etc., and (*iii*) still provide the powerful development tools of the UNIX system. The MERT operating system met these three criteria quite satisfactorily.

IV. CONVERTING TO THE MERT OPERATING SYSTEM

It was essential in converting RBCS from the UNIX operating system to the MERT operating system to make as few application changes as possible; RBCS was already running so that only modifications necessary for enginering the production system, as opposed to a field evaluation, were incorporated. Confounding this, however, was a desire to upgrade to the latest UNIX features such as

(*i*) the latest C compiler, (*ii*) "standard" semaphore capability, (*iii*) "standard" multifile magnetic tape driver, and (*iv*) the Bourne shell.[5] There was also, of course, a desire to investigate and exploit those MERT features that were expected to yield performance inprovements.

The conversion was carried out over a period of 3 to 4 months by a team of three people. As expected, the major changes took place in the I/O routines, including (*i*) the CRT driver, (*ii*) the CRT utility programs, (*iii*) the magnetic tape utilities, and (*iv*) the RBCS data base manager, which allocated physical disk space directly. The procedure was straightforward and algorithmic, albeit a tedious process. Some manual steps were required, so that the conversion was not totally mechanized, but since the four basic subsystems mentioned above were edited, tested, and debugged in only 3 to 4 months, the effort was considered minimal.

At this point, an attempt was made to convert the remainder of the RBCS data base manager subsystem to the MERT public library feature.[4] Under the UNIX operating system, the RBCS data base manager routines were link edited into a substantial number of application utilities at compile time. Any changes required approximately two weeks for recompilation of the affected programs (the subroutine dependencies were kept in an administrative data base). It was felt, during the conversion planning effort, that the most common file manager routines were taking an inordinate amount of space; approximately 10 to 12K bytes duplicated among on the order of 100 routines; each with different virtual addresses for these common routines. The MERT public library feature appeared to solve the common space problem by allowing the most frequently used routines to be placed in a shared text segment. As of this writing, the basic system without the public library feature has been completed. Work is still underway to convert the RBCS data-base manager routines to the new MERT public library format and should be completed in a few months.

With the confidence gained by the initial success, work proceeded on obtaining compatibility with the latest version of the C language compiler, on the standard I/O library, and on engineering the manufacture of the production RBCS system.

The program which proved most difficult to convert was the "transaction processing" module. This large, complex body of code is responsible for (*i*) restricting write-access to the data base manager to avoid simultaneous updates and (*ii*) maintaining internal consistency between successive states of the numerous RBCS files.

These functions require considerable interaction with other shell procedures, generally on an asynchronous basis. Since the system was written under the UNIX file manager, the transaction processing module carried out this interaction through specially created files with a large number of "open" and "close" operations, characteristic of a time-sharing system. The MERT file manager restricted the number and frequency of these operations with the result that considerable effort was necessary first to analyze the problem and then to carry out the necessary design changes.

V. STARTING WITH THE MERT OPERATING SYSTEM

Probably the most important reason for using the MERT operating system, in addition to the above-mentioned architectural advantages, is the interprocess communication capabilities; in particular, shared segments, public libraries, and guaranteed message integrity.[4]

In Fig. 3a, the first module represents a "data base read" module, the second a "CRT interactive" package, the third a "data verification" module, and the fourth a "data base write" module. Fig. 3a illustrates the necessary pipeline flow of data using the UNIX "pipe" mechanism via the shell. Reverse flow of data (from cooperating modules, for example) is not possible at the shell level, and control flow is difficult without some form of executive. Furthermore, in a heavily loaded system, the pipes degenerate rather quickly to extra disk I/O; the result is a total of 4 reads and 4 writes for a simple data base transaction.

A typical transaction for the architecture illustrated in Fig. 3a would be for a keyboard query (module 2) for a particular data base record (module 1) to be displayed (module 1). Following local modification of the record using CRT masks, a request to send the record to the data base (module 4) is made. Before sending the record to the data base manager, the sanity check process (module 3) verifies the "within record" data integrity for such violations as nonnumerical data or illegal data. If the data check is unsuccessful, then module 2 should be activated to blink the fields in error as indicated by data obtained from module 3. Control flows alternately between modules 2 and 3 until either the record passes the data integrity check or local management overrides the process for administrative purposes. Only at that time would the record proceed to the data base write process (module 4). With a shell syntax, it is only possible for data to flow left to right; reverse flow requires files specifically allocated for that purpose.

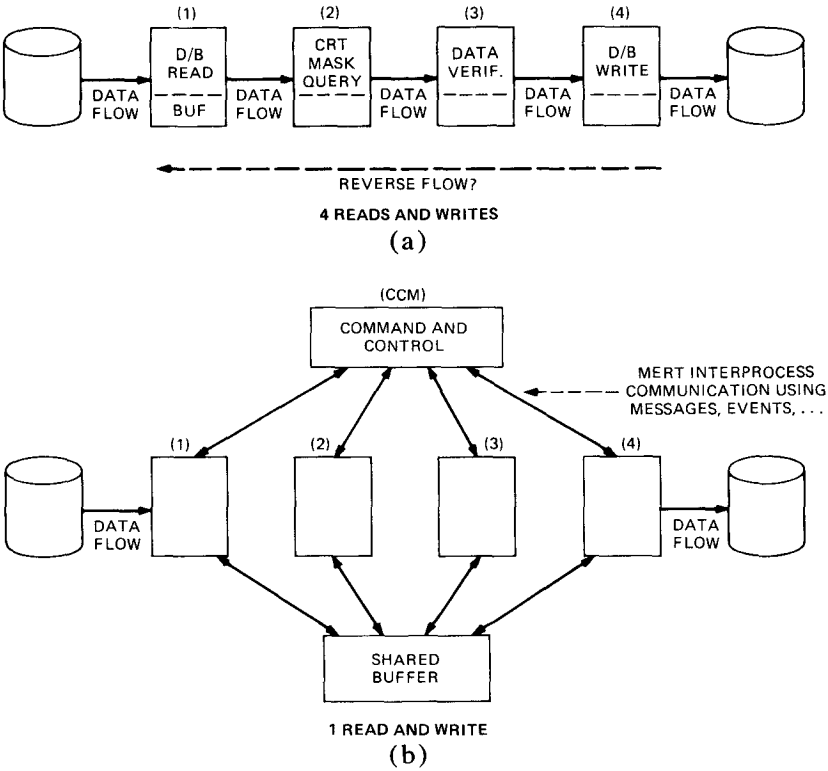

Fig. 3—(a) UNIX™ shell architecture. (b) MERT shell architecture.

Figure 3b illustrates the MERT approach with a single shared-data segment used for a minimum 1 read and 1 write. Each cooperating process is shown with a virtual attachment to the shared data segment indicating access to the data without disk I/O. Thus, reverse flow of data is accomplished by each process writing into the shared segment at any given time. Flow of control and process synchronization is accomplished in the example shown by the upper process called a "Command and Control Module" (CCM), in RCMAS. The shared data segment can be as large as 48K bytes in *user* space with MERT support provided so that sibling-related or even nonsibling-related processes can be adequately interfaced or isolated as the case requires *without* system modifications on our part.

The large user segment size allows individual RBCS or RCMAS transaction data to be supported with a minimum of disk I/O. The message capability, along with the segment management routines, allows

the data to *remain* in a segment; the processes modify the *single* copy of the data instead of passing the data around through pipes or files, as with the UNIX operating system implementation shown in Fig. 3a.

Rather than write an elaborate C module for the "Command and Control" process of RCMAS, the latest version of the shell written by S. R. Bourne[5] was extended to include the MERT interprocess communications primitives.[8] In particular, the "Command and Control Module" for RCMAS has been implemented entirely by means of one master shell procedure and several cooperating procedures. This has the advantage of easier readability to nonprogrammers, flexibility in the light of frequent changes, and ease of debugging (since any shell procedure can be run interactively from a terminal, one line at a time). Measured performance to date has not indicated any penalty great enough, from a time or space viewpoint, to necessitate rewriting the shell procedures as C modules.

It is clear from observations of RCMAS performance and discussions with interested people that considerably more flexible architectures are possible than the simple "star" network illustrated in Fig. 3b. However, it was felt that such a simplified approach was necessary to retain administrative control during the initial design and implementation stages until sufficient familiarization was achieved with asynchronous processes served by the elaborate MERT interprocess primitives.

VI. ACKNOWLEDGMENTS

The work of converting the RBCS project from the UNIX to the MERT operating system fell primarily on D. Rabenda, J. P. Stampfel, and R. C. White, Jr. The task of coordinating the design work for the RCMAS project was the responsibility of N. M. Scribner. Throughout the RBCS and RCMAS developments, especially under the MERT operating system, D. L. Bayer and H. Lycklama were most helpful with the operating system aspects and S. R. Bourne with the shell.

REFERENCES

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
2. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," B.S.T.J., this issue, pp. 1991-2019.
3. S. C. Johnson and M. E. Lesk, "UNIX Time-Sharing System: Language Development Tools," B.S.T.J., this issue, pp. 2155-2175.

4. H. Lycklama and D. L. Bayer, "UNIX Time-Sharing System: The MERT Operating System," B.S.T.J., this issue, pp. 2049-2086.
5. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," B.S.T.J., this issue, pp. 1971-1990.
6. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," B.S.T.J., this issue, pp. 2115-2135.
7. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975, section form(I).
8. N. J. Kolettis, private communication.

THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

The Network Operations Center System

By H. COHEN and J. C. KAUFELD, Jr.
(Manuscript received January 27, 1978)

The Network Operations Center System (NOCS) is a real-time, multiprogrammed system whose function is to survey and monitor the entire toll network. While performing this function, the NOCS supports multiple time-shared user work stations used to interrogate the data base maintained by the real-time data acquisition portion of the system. The UNIX operating system was chosen to support the NOCS because it could directly support the time-shared user work stations and could, with the addition of some interprocess communication enhancements and a fast access file system, support the real-time data acquisition. Features of the UNIX operating system critical to data acquisition include signals, semaphores, interprocess messages, and raw I/O. Two features were added, the Logical File System (LFS), which implements fast application process access to disk files, and the Multiply Accessible User Space (MAUS), which allows application processes to share the same memory areas. This paper describes these features, with emphasis on the manner in which UNIX operating system features are used to support simultaneously both real-time and time-shared processing and on the ease with which features were added to the UNIX operating system.*

I. INTRODUCTION

This paper explains how the Network Operations Center System

* UNIX is a trademark of Bell Laboratories.

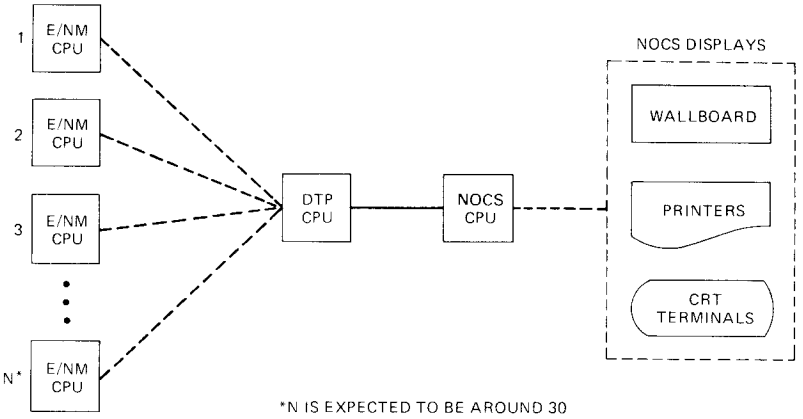

Fig. 1—Network management data network.

systems and the NOCS as the nodes (see Fig. 1). The NOCS, which executes on a Digital Equipment Corporation PDP-11/70 minicomputer system, is physically co-located with the DTP. This network is synchronized in time to utilize telephone traffic data accumulated periodically (every 5 minutes) by data collection systems. When an E/NM detects a problem or control on a trunk group or office of potential concern to the NOCS, the data describing that problem or control are sent to the DTP which passes them on to the NOCS. In addition, this network is used to pass reference data describing the telephone network configuration to the NOCS.

2.1.4 NOCS data

During the first 150 seconds or so of each 5-minute interval, the NOCS can receive as many as 6000 dynamic data messages, i.e., data describing the current state of the network, through the link connecting the NOCS and DTP computers. About 2000 of these messages concern the trunk status data needed continuously by the NOCS in order to find spare capacity for alternate routes. Most of the rest are trunk group or office data messages which occur only when a trunk group or office is overloaded beyond its design capacity. Each message contains identity information and several processed traffic data fields, which are retained for several intervals in files in the NOCS data base. Dynamic data messages which give trunk group control, code block control, and office alarm status are

such as a large metropolitan area, a state, telephone operating company, or switching region. It is expected that 30 or more E/NMs will blanket the U.S. telephone network by the early 1980s.

received asynchronously in time by the NOCS, and are retained in the NOCS data base until a change in state occurs.

The telephone network configuration of interest to the NOCS is specified by reference data describing approximately 300 toll switching offices and up to 25,000 of their interconnecting trunk groups. All the trunk group reference data needed by the NOCS are derived algorithmically from the reference data base of the E/NMs and are transmitted to the NOCS via the DTP. Hence, reference data messages that contain updates to the network configuration are received at irregular intervals from E/NM systems.

2.1.5 NOCS data base

The NOCS data are arranged into a data base consisting of several hundred data files containing all the reference data necessary to describe the network configuration and the dynamic data needed to reflect the current state of the network. Some of the files are record-oriented, describing all the information about individual data base entities such as an office or trunk group. Others are relational in nature, containing sets of related information such as all the trunk groups between geographic areas or all the trunk groups with the same type of problem condition. This arrangement allows complex data inquiries to be answered quickly by combining relations with a standard set of operations. It also allows per-entity data to be accessed very quickly by a simple indexing operation into a file.

2.2 NOCS design

2.2.1 Philosophy

The UNIX operating system was selected as the basis for design to take advantage of prior experience with that system. During the design phase, every attempt was made to use existing UNIX operating system features to implement the required NOCS functions and to minimize the number of modifications necessary to the UNIX operating system. As a result of this philosophy, the final set of features needed in the operating system by the NOCS included only two features not in the standard UNIX system. If the UNIX operating system modifications had turned out to be too extensive to be implemented locally, another operating system would have been investigated.

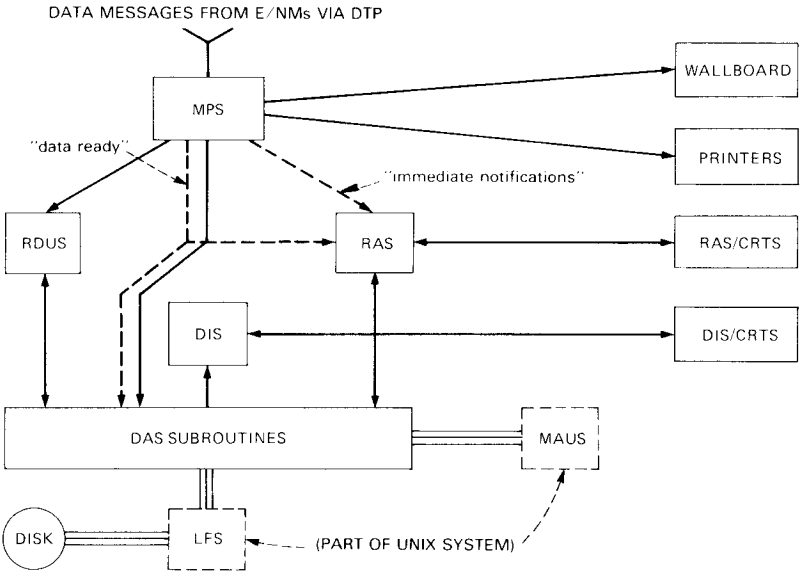

Fig. 2—Major NOCS subsystems.

2.2.2 NOCS subsystems

The application software for the NOCS is organized along the lines of functional requirements into the major subsystems listed below (for graphic representation, see Fig. 2). Each of these subsystems consists of one or more UNIX processes.

(*i*) The Message Processing Subsystem (MPS) receives all incoming data from the DTP. Some incoming data from the DTP are reference data messages which are not handled in real-time. These reference data messages are passed to the Reference Data Update System (RDUS), a background process that maintains the NOCS reference data base. The remainder, dynamic data messages, are handled by the MPS and are entered into the NOCS data base. If an "immediate notification" request has been made for a data item by some other NOCS subsystem, an interprocess message containing that data item is sent to that subsystem. When all the dynamic data for an interval have been received, the MPS notifies all interested subsystems of the availability of new data, updates the data indirection pointers to make the new data available, updates the wall display to indicate the latest network problems, and begins printing the latest data reports.

(*ii*) The Reroute Assist Subsystem (RAS) analyzes new data each 5-minute interval to determine if there are any interregional final trunking problems in the network that are possible candidates for traffic reroutes. If any are found, RAS looks through the available trunk data to determine if spare capacity exists to relieve the problem. Any problems and suggested solutions are displayed to the network managers through CRT terminals. The RAS also requests the MPS to notify it immediately if any further data are received relating to an implemented reroute. If such notification is received, the RAS immediately examines the data given it by the MPS to determine if any changes should be made to that reroute.

(*iii*) The Data Inquiry Subsystem (DIS) provides the network managers with access to the NOCS data base from CRT work stations. From any of these work stations, a manager can display network data in a variety of ways. Each way presents the data from a unique perspective in a rigorously formatted and easily interpreted manner. In addition, CRT data entry displays are also used in maintaining the nontrunking portions of the reference data base.

(*iv*) The Reference Data Update Subsystem (RDUS) processes all changes relating to the configuration of the network. Inputs to this system can come from the E/NMs via the MPS via the DTP, CRT displays or as bulk input through the UNIX file system. It uses these inputs to create and maintain the reference data base needed by the MPS, RAS, and DIS to effectively interpret, analyze, and display the dynamic data.

(*v*) The Data Access Subsystem (DAS) handles all requests for information from the NOCS data base. The DAS consists of a large set of subroutines that provide access to the NOCS dynamic and reference data files. Hence, in the UNIX sense, DAS is not a process but a set of routines loaded with each UNIX process that accesses the NOCS data base. Because multiple processes simultaneously need quick access to the same files for both reading and writing, the DAS maintains a large area of common data and synchronization flags, which is shared by all NOCS processes using the DAS.

2.2.3 Operating System Problem Analysis

2.2.3.1 File system requirements. The initial processing of data

messages by the MPS involves a trunk group identity translation, a possible wallboard indicator translation and storage of the data items contained in the message. A simple analysis of the relationship among message content, the expected message volume, and the data display requirements reveals that the NOCS must be able to do at least 12,000 translations on the set of incoming data messages and place them into the correct disk files in an elapsed time of about 150 seconds. Potentially, then, a very large number of disk accesses are possible unless careful attention is given to the algorithms and data layouts used. Thus, a data storage mechanism is required with as little overhead as possible from the point of view of system buffering, physical disk address calculation, and disk head positioning. This simple analysis does not take into account the substantial number of disk accesses necessitated by DIS, RAS, and other NOCS background processes.

In addition, analysis of data inquiry displays reveals that some DIS processes would need simultaneous access to more data files than the UNIX file system allows a process to have at one time. Therefore, in order to hold response times down, some mechanism for overcoming this limitation without numerous time-consuming opening and closing of files is necessary.

2.2.3.2 Scheduling requirements. The application processes in the NOCS impose three types of scheduling criteria on the UNIX operating system. First, the MPS and RAS are required to analyze the incoming data in "near" real-time to provide network managers with timely notification of potential problems and recommended solutions. Since the MPS is responsible for collecting and processing incoming data, it must execute with enough frequency and for long enough periods to prevent data overruns and/or delays. Second, the CRT work stations, at which network managers interact with the DIS, have classical time-share scheduling needs. Third, background programs, exemplified by the processing of reference data messages by RDUS, require neither real-time nor time-share scheduling. Processes of this third type can be scheduled to run whenever no processes of the real-time or time-share variety are waiting to run.

2.2.3.3 Interprocess communication requirements. Several types of interprocess communications mechanisms are needed by NOCS subsystems. First, the MPS must send interprocess messages to other NOCS processes upon reception of "immediate notification" requested data. These data must be passed quickly but are not large in volume. The use of an interprocess message facility implies that the process identifications (process IDs) assigned by the UNIX

operating system* be communicated among processes. Another mechanism mentioned in the MPS description was the need to be able to notify other processes of the availability of new data. This mechanism must be able to interrupt the actions of the target process so that any necessary processing can occur before resuming the interrupted activity. Last, it was also foreseen that the implementation of the DAS would require multiple processes to share their knowledge of the state of files in the data base and would require a mechanism for synchronization and protection of these files.

2.2.4 Operating system problem solutions

2.2.4.1 File system. The file system requirements led to the conclusion that the standard UNIX file system was inadequate for the NOCS. However, the raw I/O facility in the UNIX operating system has the following features:

(*i*) Control of data block placement within the raw I/O area.
(*ii*) Transfer of data directly from disk to user buffer.
(*iii*) Access to a large contiguous disk area through a single UNIX file.

These features provide precisely the capabilities required by the NOCS in a file system. Hence, they were used as the foundation for a new type of file system, known as the Logical File System (LFS), which was added to the UNIX system.

The LFS controls logical file to physical disk address mapping and the allocation of space on the disk. The LFS can be requested by the user to read or write 512-byte sectors of a file, create or delete a file, or copy one file to another. It keeps all files contiguous to simplify logical to physical address mapping and minimize disk head movement. Also, it transfers data directly from disk to user buffer areas.

The entire NOCS data base is implemented using the LFS. Since all access to data from NOCS processes is through DAS subroutines, the DAS has total semantic responsibilities for the contents of the files which are in the LFS. The DAS remembers what NOCS data are in which file, the size of the file in bytes, and the current usage status of the file.

2.2.4.2 Scheduling. The NOCS scheduling requirements are such that the standard UNIX facilities can handle them. All real-time

* The process identification is the only way of uniquely identifying a job once it has been started by the UNIX system. The message mechanism uses process identifications as its addressing mechanism.

processes are given a base priority that is higher than any time-share process. Thus, in the case of competition for the processor, the real-time processes will be scheduled first. Within the real-time priority range, the MPS is given the highest priority to ensure that it will always be able to process the incoming data. Within the time-share priority range, the CRT work stations assigned to reroute assist interaction are given the highest priority. Finally, background processes, like RDUS, are given lower priority than any time-share process to ensure that background processes will only be run if no other work needs to be performed.

2.2.4.3 Interprocess communications. The final design for the NOCS system relies on the following interprocess communication mechanisms:

(*i*) A mechanism for communicating process identifications.
(*ii*) An interprocess message facility for passing small amounts of data between unrelated processes.
(*iii*) An interrupt mechanism for interprocess notifications of events.
(*iv*) A synchronization/resource protection mechanism.
(*v*) A mechanism for sharing large amounts of data between processes.

Given the existence of item (*i*), the UNIX interprocess message facility can handle item (*ii*) and the UNIX signal facility can handle item (*iii*). However, items (*i*), (*iv*), and (*v*) required additions to the UNIX system. Item (*iv*), the synchronization/protection mechanism, was solved by expanding the semaphore capability of the UNIX system. Semaphores existed in the UNIX system, but processes were restricted to five semaphores, and about 400 were needed by the DAS simultaneously to effectively use the LFS capability. One way in which item (*v*), sharing of data, could be handled by the standard UNIX system would be to establish a file (either in LFS or the UNIX file system) whose contents were read by each process when necessary. In addition, a series of semaphores would have to be established so that processes could have exclusive access to this file for the purpose of changing it. A system for data sharing of that design would have many problems in terms of simplicity, synchronization, and speed, so it was decided to make a major addition to the UNIX operating system known as MAUS, or Multiply Accessible User Space. MAUS allows processes to directly share large portions of their data space. MAUS also provides a solution for item (*i*).

2.2.4.4 Other. A variety of other problems were encountered, most of which were solved without any modifications of the UNIX operating system. However, the following other additions were made to the operating system:

(*i*) A DA11 UNIBUS link driver for high-speed interprocessor communications. This is used for the DTP-to-NOCS communications link.
(*ii*) A DH11 asynchronous line multiplexer driver for half-duplex DATASPEED® 40 terminals.
(*iii*) Disk I/O priorities so that the priority of a disk request matches the priority of the process needing the disk.

All the above modifications were necessary, but considered sufficiently straightforward to need no further explanation.

III. UNIX FEATURE ADDITIONS

The Logical File System and the Multiply Accessible User Space features were implemented for the NOCS. These features are now available and being used by other UNIX-system-based application systems needing real-time operating system features.

3.1 LFS (Logical File System)

The LFS is a mechanism that enables processes to use the raw I/O facility in the UNIX operating system without having to manage the disk space within the disk area reserved for raw I/O. It establishes a file system oriented around 512-byte blocks within which it can create, write, read, and delete files.

3.1.1 Overview

The file system which the LFS provides sacrifices a number of features of the standard UNIX file system for simplicity of implementation. For instance, the standard UNIX file system provides access protection at the individual file level; in the LFS, access protection is only provided once for the set of files managed by the LFS. Another difference is that file names in the standard UNIX file system are character strings which can be descriptive of file contents; in the LFS, file names are numbers.

The important features of the LFS are listed below.

(*i*) Treatment of the LFS as a single UNIX file which, when opened, allows access to all LFS files.

(*ii*) File names that are indices into an array which lists the starting block and size of each file, thereby minimizing the time required to "look up" the physical mapping of a file.

(*iii*) Contiguous space allocation for all files, thereby minimizing the time required to copy file data into memory.

(*iv*) Integrated file positioning with read or write, thereby eliminating separate file positioning system calls which are necessary for accessing normal UNIX files.

(*v*) Adherence to the UNIX principle of isolating the application processes from the vagaries of the physical device — with the restriction that any physical device used for the LFS must appear to have 512 bytes per block.

In order to access the files managed by the LFS, the unique UNIX file name associated with the LFS must be opened using the special routine **lfopen**. The routines **lfcreate**, **lfwrite**, **lfread**, and **lfdelete** then can be used by processes to create, write, read, and delete files within the LFS. Each of these routines expects the LFS file number as an argument. In addition, **lfcreate** expects to be passed the size of the file being created; **lfwrite** and **lfread** expect a data buffer address, data buffer size, and starting position in the file.

The LFS has one additional feature which the NOCS software uses. The **lfswitch** routine takes two LFS file numbers as arguments and switches the physical storage pointers for the files. This feature enables files to be built in an offline storage area and then be quickly switched online; it is especially useful during data base updates.

3.1.2 Implementation

A restriction existed in the UNIX operating system which had to be eliminated before the LFS could be effective. The interface between the LFS and the disk is through the raw I/O routine, **physio**. **physio**, in the standard UNIX operating system, allows only one raw I/O request to be queued for each device; since almost all NOCS processes make raw I/O requests via the LFS, this restriction would have resulted in a severe bottleneck. The remedy was to allow **physio** to queue raw I/O requests for different processes by providing a pool of raw I/O headers analogous to the pool of system buffers available for standard UNIX file I/O.

One code module containing the LFS routines was added to the

UNIX operating system. Of course, the module containing **physio** was modified as outlined above. In addition, the **physio** modification requires minor changes in several device-handling routines. Also, several data structures were modified to allow for the pool of raw I/O headers and to define the LFS file system structure. In total, the modifications necessary to install the LFS required about 300 lines of C code to be added or modified.

3.2 MAUS (Multiply Accessible User Space)

MAUS (unpublished work by D. S. DeJager and R. J. Perdue) is a mechanism that enables processes to share memory. It is not necessary for the general time-sharing environment, but for multiprogramming real-time systems such as NOCS it is almost essential.

3.2.1 Overview

MAUS consists of a set of physical memory segments which will be referred to as MAUS segments; a unique UNIX file name is associated with each MAUS segment. A MAUS segment is described to the system by specifying its starting address (a 32-word block number relative to the start of MAUS) and its size in blocks (up to 128 blocks or 4096 words). The physical memory allocated for MAUS starts immediately after the memory used by the UNIX operating system and is dedicated to MAUS. Any process may access MAUS segments.

A process may access a MAUS segment in two ways. The preferred MAUS access method is to make the MAUS segment a part of the process's address space by using a spare segmentation register. This method can be used by any process which has at least one memory segmentation register left after being loaded by the UNIX operating system.* If the process has no free memory segmentation registers, then access to the MAUS segment may be obtained under an alternate method which uses the standard file access routines such as **open**, **seek**, **read**, and **write**. The alternate method is slower than the preferred method and has potential race problems if more than one process tries to write data into a MAUS segment.

* Each process has a fixed number of memory segmentation registers available for its use. For processes running on a Digital Equipment Corporation PDP-11/70 under the UNIX operating system, eight memory segmentation registers are available for mapping data and MAUS. Each segmentation register is capable of mapping 4096 words, i.e., one MAUS segment. One of these registers is always used for the process stack and at least one other is used for the data declarations within the process. Thus, a maximum of six segmentation registers are available for accessing MAUS.

3.2.1.1 Preferred access method. To access a MAUS segment by the preferred method, a process must first obtain a MAUS descriptor using the MAUS routine **getmaus** in a manner similar to the standard UNIX **open**. The UNIX file name associated with the MAUS segment and the access permissions desired are given in the **getmaus** call. **getmaus** makes the necessary comparisons of the access desired with the allowable access for the process making the call and returns either an error or a MAUS descriptor which has been associated with the requested MAUS segment. A process is allowed to have up to eight MAUS descriptors at the same time. When a valid MAUS descriptor is given to the **enabmaus** routine, a virtual address, which may be used by the process to access data within MAUS segment associated with the MAUS descriptor, is returned. This virtual address is also used to detach the MAUS segment with the **dismaus** routine. The MAUS descriptor can be deallocated using the **freemaus** routine. Obtaining a MAUS descriptor is very slow relative to attaching and detaching MAUS segments; thus processes which cannot simultaneously attach all the MAUS segments they need to access can still rapidly attach, detach, and reattach MAUS segments using MAUS descriptors. Any number of processes can have the same MAUS segment simultaneously attached to their virtual address space.

3.2.1.2 Alternate access method. To use the alternate access method, the UNIX file name associated with the MAUS segment desired is **opened** like any normal UNIX file. The file descriptor returned can be used by the **read** or **write** routines to access the MAUS segment as if it were a file.

3.2.2 Implementation

One code module containing the MAUS routines and some minor modifications to several existing functions within the UNIX operating system were all that was necessary to install MAUS. In addition, several minor modifications were made to UNIX data structures to store MAUS segment descriptions and MAUS descriptors. In total, less than 150 lines of code were added or modified. In the final analysis, the most difficult part of the MAUS implementation was arriving at a design which was compatible with existing interfaces within the UNIX operating system.

IV. STANDARD UNIX FEATURE USAGE

The standard UNIX system provides a complete environment for

the entry, compilation, loading, testing, maintenance, documentation, etc., of software products. It is impossible to categorize all the ways in which this environment aided in the design and development of the NOCS; however, a few examples are illustrated here.

4.1 Development and testing

The same version of the UNIX operating system is used both for NOCS development in the laboratory and for the NOCS application. In fact, because of the versatility of the UNIX system and the design of the NOCS software, most of the NOCS system is left running continuously during software development. New versions of NOCS subsystems can be installed without the need for restarting the entire system. In essence, a continuous test environment exists so that developers can integrate their programs as soon as module testing is completed without having to schedule special "system test" time.

4.2 Software administration

The NOCS software is maintained in source form on two UNIX file systems. One of these file systems is mounted read-only and may not be modified by software developers; the other initially contains a copy of the first and is used for software development. Upon completion of a successful development milestone, an updated version of the software is moved to the read-only file system by a program administrator and a new development file system is created. Standard UNIX utilities are used to keep track of changes between the read-only and development file systems.

In order to generate the NOCS binary from the source, UNIX shell procedures have been developed. There is one "build" procedure for each NOCS subsystem. The procedures are part of the NOCS software and are administered in the same manner as the rest of the NOCS software. The structural similarity between the read-only and development file systems allows the complete testing of software "build" procedures before they are copied to the read-only file system.

4.3 System initialization

The standard UNIX init program is used to start the NOCS functions. Each NOCS process is assigned to a run level or to a set of run levels. When init is told, by an operator, to enter a particular run

level, the NOCS processes assigned to that run level are started. The assignment of processes to run levels is made in such a way that critical NOCS processes may be isolated both during development and in the field for testing.

4.4 Documentation

All NOCS documentation is done under the UNIX system. This documentation consists of a user's manual which describes the inputs and outputs for the system and a developer's guide which is a description of the NOCS software. The **nroff** UNIX program along with NOCS-developed **nroff** macro packages is used to format the documentation for printing. The text is entered using the UNIX **ed** program and is stored in standard UNIX files.

V. CONCLUSIONS

The NOCS system has real-time multiprogramming requirements that make operating system demands very much counter to the basic UNIX time-share philosophy. However, these demands were met quite readily with some feature additions because of the adaptability and generality inherent in the UNIX operating system. The available scheduling parameters were flexible enough to handle the three kinds of scheduling demands; "near" real-time, time-share, and background, imposed on the UNIX operating system by the NOCS. The interprocess communication mechanisms were rich and varied enough that only one addition was necessary to provide all the features needed by NOCS. The UNIX operating system was modular enough so that a completely new kind of file system was interfaced in a very short time with almost none of the timing bugs that might be expected in a typical operating system. The total UNIX environment provided software tools to support the complex development effort needed to implement the NOCS. Finally, the use of the same operating system for both development and application certainly minimized friction between the coding and testing phases of development and allowed the smooth integration of all system functions.

Contributors to This Issue

Douglas L. Bayer, B.A., 1966, Knox College; M.S. (physics), 1968, and Ph.D. (nuclear physics), 1970, Michigan State University; Rutgers University, 1971-1973; Bell Laboratories, 1973—. At Rutgers University, Mr. Bayer was involved in low-energy nuclear scattering experiments. At Bell Laboratories, he has been involved in computer operating systems research on mini- and microcomputers. Member, ACM.

S. R. Bourne, B.Sc. (mathematics), 1965, University of London; Diploma in Computer Science, 1966, and Ph.D. (applied mathematics), 1969, Cambridge University; Bell Laboratories, 1975—. While at Cambridge Mr. Bourne worked on the symbolic algebra system CAMAL, using it for literal calculations in lunar theory. This work led to his interest in programming languages and he has been an active member of IFIP Working Group 2.1 on Algorithmic Languages since 1972. He has written a compiler for a dialect of ALGOL 68 called ALGOL68C. Since coming to Bell Laboratories he has been working on command languages. Member, RAS, CPS, and IFIP Working Group 2.1.

L. L. Cherry, B.A. (mathematics), 1966, University of Delaware; M.S. (computer science), 1969, Stevens Institute of Technology; Bell Laboratories, 1966—. Ms. Cherry initially worked on vocal tract simulation in Acoustics and Speech Research. After a brief assignment at Whippany and Kwajalein where she was responsible for the Safeguard utility recording package, she joined the Computing Science Research Center. Since 1971, she has worked in the areas of graphics, word processing, and language design. Member, ACM.

Carl Christensen, B.S. (electrical engineering), 1960, and M.S. (electrical engineering), 1961, University of Wisconsin; Bell Laboratories, 1961—. Mr. Christensen is currently a member of the Digital Switching and Processing Research Department.

Harvey Cohen, B.A. (mathematics), 1967, Northeastern University; M.S. (applied mathematics), 1970, New York University; Bell Laboratories, 1968—. Mr. Cohen began work in software development on the Automatic Intercept System (AIS). He later worked on

the system design and software development of the EADAS family of traffic collection and network management minicomputer systems. He is currently supervising the Network Engineering Support Group in the Advanced Communication System project. Member, Phi Kappa Phi.

T. H. Crowley, B.A. (electrical engineering), 1948, M.A., 1950, and Ph.D. (mathematics), 1954, Ohio State University; Bell Laboratories 1954—. Mr. Crowley has worked on magnetic logic devices, sampled-data systems, computer-aided logic design, and software development methods. He is presently Executive Director, Business Systems and Technology Division. Fellow, IEEE.

T. A. Dolotta, B.A. (physics) and B.S.E.P., 1955, Lehigh University; M.S., 1957, Ph.D. (electrical engineering), 1961, Princeton University; Bell Laboratories, 1960-1962 and 1972—. Mr. Dolotta has taught at the Polytechnic Institute of Grenoble, France, and at Princeton University. His professional interests include the human interface of computer systems and computerized text processing. Until March, 1978, he was Supervisor of the PWB/UNIX Development Group. He is now Supervisor of the UNIX Support Group. He served as ACM National Lecturer and as Director of SHARE INC. Member, AAAS and ACM; Senior Member, IEEE.

A. G. Fraser, B.Sc. (aeronautical engineering), 1958, Bristol University; Ph.D. (computing science), 1969, Cambridge University; Bell Laboratories, 1969—. Mr. Fraser has been engaged in computer and data communications research. His work includes the Spider network of computers and a network-oriented file store. Prior to joining Bell Laboratories, he wrote the file system for the Atlas 2 computer at Cambridge University. In 1977 he was appointed Head, Computer Systems Research Department. Member, IEEE, ACM, and British Computer Society.

R. C. Haight, B.A. (English literature), 1955, Michigan State University; Bell Laboratories, 1967—. Mr. Haight was with Michigan Bell from 1955 to 1967. Until 1977, he was Supervisor of the PWB/UNIX Support Group; since then, he has been Supervisor of the Time Sharing Development Group.

S. C. Johnson, B.A., 1963, Haverford College; Ph.D. (pure mathematics), 1968, Columbia University; Bell Laboratories, 1967—. Mr. Johnson is a member of the Computer Systems Research Department. His research interests include the theory and practice of compiler construction, program portability, and computer design. Member, ACM and AAAS.

J. C. Kaufeld, Jr., B.S., 1973, Michigan State University; M.S., 1974, California Institute of Technology; Bell Laboratories, 1973—. Mr. Kaufeld has participated in the design, programming, testing, and maintenance of minicomputer-based network management systems and in UNIX operating system support for operations support systems. Currently, he is working in operating system support for the Advanced Communications System.

Brian W. Kernighan, B.A.Sc., 1964, University of Toronto; Ph.D., 1969, Princeton University; Bell Laboratories, 1969—. Mr. Kernighan has been involved with heuristics for combinatorial optimization problems, programming methodology, software for document preparation, and network optimization. Member, IEEE and ACM.

M. E. Lesk, B.A., 1964, Ph.D. (chemical physics), 1969, Harvard University; Bell Laboratories, 1969—. Mr. Lesk is a member of the Computing Mathematics Research Department. His research interests include new software for document preparation, language processing, information retrieval, and computer communications. Member, ACM and ASIS.

Gottfried W. R. Luderer, Dipl. Ing. E. E., 1959, Dr. Ing. E. E., 1964, Technical University of Braunschweig, Germany; Bell Laboratories, 1965—. Mr. Luderer has worked in the field of computer software; his special interests include operating systems and their performance. Mr. Luderer has taught at Stevens Institute of Technology and at Princeton University. He currently supervises a group engaged in the development and support of minicomputer operating systems for real-time applications. Member, IEEE and ACM.

H. Lycklama, B. Engin. (engineering physics), 1965, and Ph.D. (nuclear physics), 1969, McMaster University, Hamilton, Canada; Bell Laboratories, 1969-1978. As a member of the Information Processing Research Department, Mr. Lycklama was responsible for the

design and implementation of a number of operating systems for mini- and microcomputers. In 1977 he became supervisor of a software design group at Holmdel responsible for message switching and data entry services for a data communications network. He is currently associated with the Interactive Systems Corporation in Santa Monica, California.

Joseph F. Maranzano, B.Sc. (electrical engineering), 1964, Polytechnic Institute of Brooklyn; M.Sc. (electrical engineering), 1966, New York University; Bell Laboratories, 1964-1968 and 1970—. Mr. Maranzano is a Supervisor in the Small Systems Planning and Development Department. He has been engaged in the centralized maintenance, distribution, and support of the UNIX system since 1973.

J. R. Mashey, B.S. (mathematics), 1964; M.S. 1969, Ph.D. (computer science), 1974, Pennsylvania State University; Bell Laboratories, 1973—. Mr. Mashey's first assignment at Bell Laboratories was with the PWB/UNIX project. There, and later, in the Small Systems Planning and Support Department, he worked on text-processing tools, command-language development, and UNIX usage in computer centers. He is now a Supervisor in the Loop Maintenance Laboratory. His interests include programming methodology, as well as interactions of software with people and their organizations. Member, ACM.

M. D. McIlroy, B.E.P. (engineering physics), 1954, Cornell University; Ph.D. (applied mathematics), 1959, Massachusetts Institute of Technology; Bell Laboratories, 1958—. Mr. McIlroy taught at M.I.T. from 1954 to 1958, and from 1967 to 1968 was a visiting lecturer at Oxford University. As Head of the Computing Techniques Research Department, he is responsible for studies of basic computing processes, aimed at discovering the inherent capacities and limitations of these processes. He has been particularly concerned with the development of computer languages, including macro languages, PL/I, and compiler-compilers. Mr. McIlroy has served the ACM as a National Lecturer, as an editor of the Communications and Journal, and as Turing Award Chairman.

L. E. McMahon, A.B. 1955, M.A. 1959, St. Louis University; Ph.D. (psychology), 1963, Harvard University; Bell Laboratories, 1963—. Mr. McMahon's research interests include

psycholinguistics, computer systems, computer text processing, and telephone switching. He has been head of the Human Information Processing Research Department, the Murray Hill Computer Center, and the Interpersonal Communication Research Department, and is now in the Computing Science Research Center.

Robert Morris, A.B. (mathematics), 1957, A.M., 1958, Harvard University; Bell Laboratories, 1960—. Mr. Morris was on the staff of the Operations Research Office of Johns Hopkins University from 1957 to 1960. He taught mathematics at Harvard University from 1957 to 1960, and was a Visiting Lecturer in Electrical Engineering at the University of California at Berkeley during 1966-1967. At Bell Laboratories he was first concerned with assessing the capability of the switched telephone network for data transmission in the Data Systems Engineering Department. Since 1962, he has been engaged in research related to computer software. He was an editor of the Communications of the ACM for many years and served as chairman of the ACM Award Committee for Programming Languages and Systems.

Elliot R. Nagelberg, B.E.E., 1959, City College of New York; M.E.E., 1961, New York University; Ph.D., 1964, California Institute of Technology; Bell Laboratories, 1964—. Mr. Nagelberg is currently Head, CORDS Planning and Design Department, with responsibilities in the area of data base management for operations support and Electronic Switching Systems. Member, IEEE, Sigma Xi.

Joseph F. Ossanna, Jr., B.S. (electrical engineering), 1952, Wayne State University; Bell Laboratories, 1952-1977. During his career Mr. Ossanna did research on low-noise amplifiers, and worked on feedback amplifier theory and on mobile radio fading studies. He developed methods for precise prediction of satellite positions for Project Echo. He was among the earliest enthusiasts for time-sharing systems and took part in the development of the MULTICS system. More recently he was concerned with computer techniques for document preparation and with computer phototypesetting. Mr. Ossanna died November 28, 1977. Member, IEEE, Sigma Xi and Tau Beta Pi.

S. P. Pekarich, B.S. (electrical engineering), 1972, Monmouth College; M.S. (computer science), 1975, Stevens Institute of Technology; Bell Laboratories, 1967—. Mr. Pekarich first worked on support software for the maintenance of the Voiceband Interface and the Echo Suppressor Terminal. He is currently working for the MAC-8 microprocessor department. Member, Eta Kappa Nu, Sigma Pi Sigma, ACM.

Michael A. Pilla, B.S.E.E. and M.S.E.E., 1961, and Ph.D., 1963, Massachusetts Institute of Technology; Bell Laboratories, 1963—. Since joining Bell Laboratories Mr. Pilla has been involved with minicomputer software development. Initial projects in the Human Factors Engineering Research Department were concerned with the development of tools and techniques for real-time process control. Next, he supervised a group responsible for developing a computer-aided reorder trap analysis system. Subsequently, he supervised a group responsible for advanced software techniques for projects in the CORDS Planning and Design Department. He presently supervises a group responsible for the design and development of a product to perform Recent Change Memory Administration for Nos. 1, 1A, 2, and 2B ESS.

Elliot N. Pinson, B.A., 1956, Princeton University; M.S., 1957, Massachusetts Institute of Technology; Ph.D. (electrical engineering), 1961, California Institute of Technology; Bell Laboratories, 1961—. From 1968 to 1977, Mr. Pinson was Head of the Computer Systems Research Department where he supervised work on programming languages, operating systems, computer networks, and computer architecture. Included in these activities was the development of the C programming language and portions of the UNIX operating system. Since October 1977, Mr. Pinson has been Director of the Business Communications Systems Laboratory in Denver, where he is responsible for software design and development for the DIMENSION® PBX system. Member, ACM, Phi Beta Kappa, Sigma Xi.

Dennis M. Ritchie, B.A. (physics), 1963, Ph.D. (applied mathematics), 1968, Harvard University; Bell Laboratories, 1968—. The subject of Mr. Ritchie's doctoral thesis was subrecursive hierarchies of functions. Since joining Bell Laboratories, he has worked on the design of computer languages and operating systems. After contributing to the MULTICS project, he joined K. Thompson in the creation of the UNIX operating system, and designed and

implemented the C language, in which UNIX is written. His current research is concerned with software portability, specifically the transportation of UNIX and its software to a variety of machines.

Helen D. Rovegno, B.S. (mathematics), 1969, City College of New York; M.S. (computer science), 1975, Stevens Institute of Technology; Bell Laboratories, 1969—. Ms. Rovegno, a former member of the Microprocessor System Department, worked on the design of the MAC-8 architecture and its support system and implemented the MAC-8 C compiler. Subsequently, she was involved in software coordination of UNIX-based MAC-8 tools. She currently supervises a group responsible for all programming languages and compilers for ESS software development and for exploratory work on languages and compilers. Member, Phi Beta Kappa, ACM.

J. D. Sieber, B.S. (computer science) 1978, Massachusetts Institute of Technology; Bell Laboratories, 1974—. Mr. Sieber's first contact with UNIX was as part of an Explorer Scout Post which met at Bell Labs in 1972. Since then he has worked on various UNIX and MERT projects as a summer employee and consultant. He is primarily interested in using low-cost microprocessors to design distributed processing networks that afford large computational resources and real-time response.

Arthur R. Storm, B.A. (physics), 1969, Fairleigh Dickinson University; M.S. (materials research), 1975, Rutgers University; Bell Laboratories, 1956-1959, 1960—. Mr. Storm began working at Bell Laboratories in the Semiconductor Physics Research Department. In 1959, he worked in the Low Temperature Physics Department at the University of North Carolina. In 1960 he returned to Bell Laboratories in the Materials Research Laboratory, where he has been involved with X-ray analysis of materials and studies of the automation of such experiments.

Berkley A. Tague, B.A. (mathematics), 1958, Wesleyan University; S.M. (mathematics), 1960, Massachusetts Institute of Technology; Bell Laboratories, 1960—. Mr. Tague worked in Systems Research and Computing Science Research on stochastic simulation, languages for symbolic computation, and operating systems. From 1967 to 1970, he supervised language-processor and operating-systems development for military systems. In 1970, he formed a department to plan and review Bell Laboratories computing services.

In 1973, his department assumed the additional responsibility for central support and development of the UNIX system. Member, IEEE, ACM, Sigma Xi, and Phi Beta Kappa.

Ken Thompson, B.Sc. and M.Sc. (electrical engineering), 1965 and 1966, University of California, Berkeley; Bell Laboratories, 1966—. Mr. Thompson was a Visiting Mackay Lecturer in Computer Science at the University of California at Berkeley during 1975-76. His current work is in operating systems for telephone switching and in computer chess. Member, ACM.

B. C. Wonsiewicz, B.S., 1963, and Ph.D. (materials science), 1966, Massachusetts Institute of Technology; Bell Laboratories, 1967—. Mr. Wonsiewicz has worked on a wide variety of materials problems and has special interests in areas of mechanical and magnetic properties, materials characterization, and the application of computer techniques to materials science problems.

This issue of The Bell System Technical Journal was composed, including all tabular and displayed material and final page makeup, using the document preparation software described on pages 2115-2135. It was phototypeset using the **troff** program, which was written by the late Joseph F. Ossanna, Jr.

THE BELL SYSTEM TECHNICAL JOURNAL is abstracted or indexed by *Abstract Journal in Earthquake Engineering, Applied Mechanics Review, Applied Science & Technology Index, Chemical Abstracts, Computer Abstracts, Computer & Control Abstracts, Current Contents/Engineering, Technology & Applied Sciences, Current Contents/Physical & Chemical Sciences, Current Index to Statistics, Current Papers in Electrical & Electronic Engineering, Current Papers on Computers & Control, Electrical & Electronic Abstracts, Electronics & Communications Abstracts Journal, The Engineering Index, International Aerospace Abstracts, Journal of Current Laser Abstracts, Language and Language Behavior Abstracts, Mathematical Reviews, Metals Abstracts, Science Abstracts, Science Citation Index,* and *Solid State Abstracts Journal.* Reproductions of the Journal by years are available in microform from University Microfilms, 300 N. Zeeb Road, Ann Arbor, Michigan 48106.

CONTENTS (continued)