IBM Secureway Cryptographic Products
IBM 4758 PCI Cryptographic Coprocessor
CP/Q Operating System Overview for
Original Equipment Manufacturers

First Edition, June 1998

This edition of the Design and Implementation Guide is for CP/Q Version 1 Driver 16.

Comments or queries concerning this document should be addressed to:

Security Solutions and Technology Department
IBM Corporation
8501 IBM Drive
Charlotte, North Carolina 28262-8563

Contents

Figures

Preface

This book provides information on application design and implementation for the CP/Q Operating System.

Chapter 1, “Introduction” on page 1-1 discusses the purpose and functional characteristics of the operating system.

Chapter 2, “CP/Q Program APIs and Documentation” on page 2-1 contains information on program interfaces for CP/Q and their documentation.

Chapter 3, “CP/Q System Structure” on page 3-1 describes the basic kernal concepts and facilities of the CP/Q system.

Chapter 4, “RAS Considerations” on page 4-1 discusses the RAS features of CP/Q.

Chapter 5, “Dynamic, Modular System Considerations” on page 5-1 talks about considerations for programming applications and subsystems in the dynamic CP/Q system.

Chapter 6, “Fault Handlers” on page 6-1 discusses the Fault Handler facilities in CP/Q and how they handle failures.

Introduction

Chapter 1. Introduction

CP/Q is a high-performance 32-bit operating system developed by the IBM Research Division. It was designed and implemented for use in embedded system applications.

Examples of embedded systems are adapter cards, input/output (I/O) control units and system control applications. Typical requirements for these systems are critical I/O performance, high availability and robust recovery.

CP/Q directly addresses these requirements through optimized interrupt handling paths, extensive fault notification facilities and an architecture for coordinating resource recovery (both kernel implemented resources and application managed resources).

In addition, CP/Q is highly modular. The kernel provides core operating system functions, but has very few hardware requirements beyond a 32-bit central processing unit (CPU) with virtual memory management support (CP/Q is a virtual memory system, but does not implement demand paging to auxiliary storage; see "Virtual and Real memory management" on page 1-3). In today's speech, CP/Q can be considered a micro-kernel because it can act as the core of a more elaborate system built up in layers around the core.

The system's modularity allows it to scale well from relatively small applications to large, complex environments. Its modular design also allows CP/Q, when running on a PC workstation, to provide a rich code development environment, including such features as file systems, screen/keyboard support, TCP/IP (this is a port of the Berkeley Networking Distribution from 4.3BSD, and uses the Sockets API), an interactive command shell, and source-level debuggers. There is even a port of the MIT X-Window R11.4 Server (currently implemented on the Intel version of CP/Q), which allows X client programs on other systems to open windows on a system running CP/Q. This is useful, for example, when doing edit/compile/link work on a remote UNIX system.

When code development is complete, the extra subsystems are simply left out of the final system. There is no recompile necessary: Because of the system's modular design, all subsystems reside outside the kernel, and a system rebuild takes just a few seconds to collect individual load modules into an IPL image.

Because of the kernel's minimal hardware requirements, it is highly portable across 32-bit virtual memory processors. Originally implemented on the Intel *x*86 processors it has also been ported to the PowerPC architecture. Apart from their 32-bit address size, these two architectures are very different. The successful implementation of CP/Q on each is strong evidence of its portability.

CP/Q Kernel

The CP/Q kernel provides basic operating system services to all parts of the system. The processor is directly managed by the CP/Q kernel. The kernel provides the following types of services:

Process and Task management: In CP/Q terminology, a task is the dispatchable unit of execution. Tasks are grouped into processes. Processes are the boundary of resource ownership. Resource control and recovery are provided at the process level.

One of the most important resources is memory, so an address space can be equated with a process.

A high performance, preemptive dispatching mechanism is provided. It is interrupt driven with no time slicing, although time slicing can be optionally compiled into the kernel.

Tasks can share code and data with other tasks, either in the same process or in other processes.

Inter-task Communication: CP/Q provides queued, asynchronous message passing between tasks. Messages have a priority associated with them, and can be received out of band by filtering a receive on such things as the sender's identity or a user-defined message type field.

There are also two types of semaphores: synchronized and serialized. Combined with the powerful memory management facilities of CP/Q, these allow efficient implementation of cooperating tasks, while assuring a high level of integrity.

Device and Interrupt management: The only interrupt the kernel handles is the timer-tick. All other interrupts pass through the kernel's First Level Interrupt Handler (FLIH) code, but no further processing is performed beyond saving required state and reenabling interrupts to allow other interrupt levels to continue (if the hardware allows such processing)[1]. The FLIH code causes any Second Level Interrupt Handlers (SLIHs) which have registered for the particular interrupt level to be made dispatchable before the FLIH returns from the interrupt.

SLIHs are simply tasks running in the system which have I/O privilege and have registered with the kernel to become a SLIH for a particular interrupt level. When they register, they indicate which of three methods will be used to dispatch them:

- Sending a message to their task message queue
- Clearing a synchronization semaphore
- Completing a CPIntWait Supervisor Call (SVC) which the SLIH would have issued to block until the interrupt arrived

Since SLIHs are just tasks in the system, and they don't run on the interrupt context, all system services are available to them.

SLIHs can be added and removed from the system dynamically.

There are also facilities to install user-exits off the FLIH code for an interrupt level when performance is extremely critical. This is compiled into the kernel, and runs on the interrupt context so it is therefore more limited in what services are available.

[1] Clock and interrupt controller programming are the two areas that the kernel is inevitably hardware platform specific. This is recognized, and it is understood that CP/Q will need to be customized slightly in this area for certain platforms.

The CP/Q kernel is fully reentrant, and as far as possible runs with interrupts enabled to reduce interrupt latency.

The kernel does not architect a particular device driver model. It is felt that customers understand their particular devices best, and there is no reason for the operating system to dictate how devices should be presented in the system.

Timer: The timer tick rate is configurable at build time (assuming that is possible in the hardware platform). This allows control over the granularity of timer services. There are three types of timer services available:

- SVC timeouts
- CPSleep SVC
- Asynchronous timers

Any potentially blocking SVC in the kernel provides an optional timeout value as one of its parameters.

The CPSleep SVC allows a task to block for a specified period of time.

Asynchronous timers cause tasks to be notified when a time event occurs. Both single-shot and repetitive timers are provided.

CP/Q also provides current date and time in user accessible memory locations that are maintained by the kernel.

Virtual and Real memory management: CP/Q is a virtual memory system. This should not be confused with a demand paging system which uses auxiliary disk storage to over-commit real memory. While demand paging is possible with CP/Q, and may be added as an option in the future, it is not appropriate for most embedded applications because of the delays it introduces into system operation.

CP/Q makes use of virtual memory to achieve such things as address space separation for increased reliability (by minimizing the possibility of random data corruption, for example). Also, virtual memory eliminates such concerns as garbage collection. Physical pages need not be contiguous in a virtual memory range, so the Memory Manager can collect together any available physical pages when allocating new memory objects. Recovery of memory is also facilitated: when a process is removed from the system, the Memory Manager simply reclaims the physical pages in its address space.

Performance is not negatively affected by having virtual memory active: today's processors have very efficient virtual memory implementations which employ caches and pipelining to great effect. There is also the additional consideration of coding efficiencies that a virtual memory operating system allows, which can further offset any cost there might be in hardware execution time. One example would be the elimination of code to detect and recover from data corruption: address space protection can protect the data. Also, code is always read-only when loaded in CP/Q, so there is no possibility of its being corrupted. A real memory system cannot provide these facilities, so additional code must be added to applications to achieve the desired reliability (and in many cases, there is no possibility of reaching the reliability provided by virtual memory).

CP/Q has facilities for handling different types of physical memory, such as normal RAM, ROM, cachable memory and I/O mapped memory (device memory), so it is quite flexible in adapting to different hardware platforms.

It can also assist in dealing with Direct Memory Access (DMA) I/O. The system can return physical addresses of memory objects, including lists of physical memory blocks which comprise a single virtual object. It can also assist in managing contiguous physical memory regions for I/O purposes.

The Memory Manager provides such services as memory allocation, freeing, sharing and changing memory attributes (e.g., from read-only to read-write). Memory can be allocated in three different virtual regions:

Private Only appears in the current process. There are facilities for sharing memory between processes (aliasing), which can be used on Private memory.

Common Designed to be shared among processes, but the sharing is under application control. It is useful when there is a subset of processes that must all use a particular memory object (to share data, for example), but protection is desired from the remaining processes in the system. Common memory can either share underlying physical storage, in which case changes made by one process are visible in all other processes with access to that object, or the memory can be copied (effectively, a snapshot at the time that sharing is established with each new process). Once a copied Common object is accessed by a new process, each process is working on its own private copy of the memory.

Global Appears in all processes. Memory objects allocated in Global appear at the same location in all processes, and share the underlying physical storage.

Within an address space there is additional protection afforded by supervisor versus user page access. Supervisor pages cannot be written over by user-level tasks. Note that supervisor does not imply kernel: there can be supervisor application tasks and user application tasks. Supervisor tasks are usually associated with I/O privilege, and they also have additional privileges in terms of operating system services.

Resource management: The CP/Q Resource Manager coordinates resource allocation and recovery. The Resource Manager also monitors per-process allocation limits on such things as memory and kernel-defined objects (e.g., the number of subtasks a process can create). These limits are configurable, and can be specified as new processes are created.

In some sense, operating systems can be defined by the resources they manage. Since CP/Q is highly modular, and the kernel implements only core resources such as memory, tasks and semaphores, there is a need to involve additional components in kernel operation, particularly in the area of process creation and removal. These additional components are called Resource Providers, and there is an architecture in CP/Q for allowing Resource Providers to register with the Resource Manager and become part of process creation and removal.

For example, in the case of process removal, the Resource Manager will send a message to each Resource Provider indicating which process is being removed. The Resource Provider can then do whatever is appropriate for the resources it manages. In the case of the CP/Q File I/O Subsystem, all buffers can be flushed, and open files closed. The Resource Provider then sends a message back to the Resource Manager indicating the disposition of the process. When the Resource Manager has heard from all Resource Providers, it can continue with process removal, finally calling the Memory Manager to remove the address space.

Security and Integrity: CP/Q protects the system by providing mechanisms to control the use of resources in the system. Limits can be placed on resource utilization. Access to resources is controlled and only allowed when authorized. Failures are detected and appropriate resources are recovered.

Performance

The primary objective of CP/Q is high performance for basic kernel functions. We provide here a sampling of performance numbers for some common base operating system functions. The performance numbers are (rounded) measured results, not theory.

These measurements were made on two different systems:

PS/2 Model 80-071	PS/2 Model 70-B21
16 MHz 80386	25 MHz 80486
1 wait state / memory access	0-2 wait states
	no external 2nd level cache

Unless otherwise noted, times are measured from the call instruction which invokes the service to the return instruction which completes the service. This does not include time to set up registers or any high-level language overhead.

The following provides more detailed performance information.

Task switch: To perform a basic task switch takes about 15 μseconds on the above 386 Model 80, and about 3.4 μseconds on the above 486 Model 70. This time includes all kernel processing for a running task to give up control of the processor, a task switch to occur, and the next task to be given control of the processor.

Inter-task communication: To send a short[2] message to another task takes about 65 μseconds on the above 386 Model 80, and about 16 μseconds on the above 486 Model 70. To receive the message from the queue takes about 57 μseconds on the above 386 Model 80, and about 17 μseconds on the above 486 Model 70.

A common scenario is:

1. Task A sends a short message to task B
2. A task and process switch occurs, task B becomes active
3. Task B receives the message
4. Task B sends a short reply to task A
5. A task and process switch occurs, task A becomes active
6. Task A receives the reply

This complete scenario takes about 334 μseconds on the above 386 Model 80, and about 82 μseconds on the above 486 Model 70. The timing for this last scenario assumes that the tasks are in different CP/Q processes[3], and includes the time for a CP/Q process change occurring with each task switch.

Serialization: In this scenario, a high priority task waits on a semaphore. The system switches (task switch) to a lower priority task which clears the semaphore. The system task switches back to the higher priority task which resumes execution. The total scenario takes about 158 µseconds on the above 386 Model 80, and about 45 µseconds on the above 486 Model 70.

Software Interrupt Latency: Software Interrupt Latency is defined here as the time elapsed from the processor passing control to the first instruction in the first level interrupt handler to the time at which the second level interrupt handler is dispatched and begins executing user-supplied interrupt code. We do not have a set of measured results for interrupt response. However, for a task that has issued a CPIntWait SVC, the interrupt latency on the above 386 Model 80 is approximately 38 µseconds, and on the above 486 Model 70 it is approximately 15 µseconds.

The interrupt latency time can be further reduced by including a "user exit" routine linked directly into the SVC Handler first level interrupt handler code. In this case, the interrupt latency (from the first interrupt instruction to the "user exit" routine inside the SVC Handler first level interrupt handler) on the above 386 Model 80 is approximately 6 µseconds, and on the above 486 Model 70 it is approximately 2.5 µseconds.

[2] These are 28 byte messages, sent asynchronously; the time includes queuing services.

[3] CP/Q tasks run inside CP/Q processes, possibly many tasks within one process. The CP/Q process notion allows for allocation and recovery of resources such as system control blocks, memory, and address spaces.

Chapter 2. CP/Q Program APIs and Documentation

This chapter contains information on the program interfaces available in the CP/Q environment; these interfaces are called Application Programming Interfaces, or APIs.

Application Programming Interfaces

This section describes the APIs provided with CP/Q.

Kernel services

CP/Q provides high-level language callable functions to access all kernel services. These services include base supervisor functions (suspend, synchronization and serialization, inter-task messaging, resource naming and identification, task control, interrupt handler management, timer and some miscellaneous services), memory management functions, and resource management functions. Some of these functions require authorization.

These interfaces are documented in the manual *CP/Q Application Programming Reference*.

C Runtime library

CP/Q provides a set of ANSI “C” functions. These are the “C” functions that are documented in almost any book on “C” programming:. They include memory allocation and manipulation functions, file system functions, low-level input output functions, print functions, program control functions, scan functions, string functions, translate functions, data type functions, and miscellaneous functions.

The C library routines provided with CP/Q are documented in the manual *CP/Q C Runtime Library*.

The CP/Q++ interface details appear in *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference*.

Interface Types

CP/Q provides a wide variety of programming interfaces to services supplied by the kernel and system extensions. CP/Q can provide different levels of interfaces to the same functional area.

Lower-level interfaces are often message-based interfaces. A higher-level “C” interface is also available for the same set of functions.

Interfaces to functions and services in CP/Q come in two different styles, *call* and *message*.

Call interfaces use the standard program call or function invocation paradigm. This is a synchronous interface in which the calling program passes parameters to a called function or service. The called program usually sets a return code and optionally returns or alters passed arguments.

Message interfaces use the CP/Q “send message” kernel function to transmit information to the desired server (task or queue). The message is queued to the service that performs the function. This provides for asynchronous invocation of a service. The server receives the message from the queue, performs the requested function and sends a reply message to indicate the results. These interfaces are documented by describing the contents of these messages.

The minimal possible CP/Q system contains only the system kernel. The kernel provides a set of programming interfaces. All other interfaces in the system are optional and can be used only if the modules providing the corresponding services are configured in the system when it is built.

Design and Implementation

Chapter 3. CP/Q System Structure

CP/Q provides a basic kernel which implements the basic structure and programming model of the system. On top of this basic kernel, subsystems can be added, either by selecting ones provided with the CP/Q system and/or by user written ones. The notion of a kernel in CP/Q is much “leaner” than what many systems consider to be their kernel. For example, a file system or loader are both subsystems outside the CP/Q kernel.

The CP/Q kernel does define the basic programming model of the system. It defines the basic things like processes, tasks, the memory model, interrupt and fault handling, and inter-process/inter-task communication facilities. The remainder of this chapter contains sections describing the basic kernel concepts and facilities followed by a section on the subsystems provided with the CP/Q system.

Process/Task Structure

The purpose of this section is to provide information concerning the CP/Q process and task structure. Before continuing, some basic CP/Q concepts are presented:

Process A CP/Q *process* is a system entity that can acquire system resources. Such system resources include memory, tasks, semaphores, message queues, timers, and kernel extensions. Each process has associated with it an address space. Resource accounting is generally performed at the process level. Processes are arranged in a tree structured hierarchy, with the special process "SYSTEM" at the top of the tree. Apart from the process "SYSTEM", each process in the system also "belongs" to the process immediately higher in the tree (its "parent"). The process is the resource allocation, tracking, and recovery boundary.

Task A CP/Q *task* is the basic unit of runnable code within the system. A task is a single thread of execution that can be separately dispatched. The CP/Q system kernel keeps details of each task in an associated Task Control Block (TCB) and a Supervisor Table entry (SVT). There can be more then one task in a CP/Q process. The number of tasks that a process can own at one time is determined by the resource limits for that process.

Process/Task Organization

Processes are organized hierarchically (tree-structured); the hierarchical structure is defined by application programs and reflects the independence of sub-modules within the application for resource acquisition and recovery purposes. Each process can be named or unnamed; and both types of processes are accessible to other processes in the system.

The processes in a CP/Q system define the user's view of the system, just as files define the user's view of his stored data. The Resource Manager provides the interfaces required to manipulate processes and resources allocated to them.

An overview of the principal functions provided by the Resource Manager is presented in the following subsections.

Assume that the processes that are defined in a sample CP/Q system are arranged as follows:

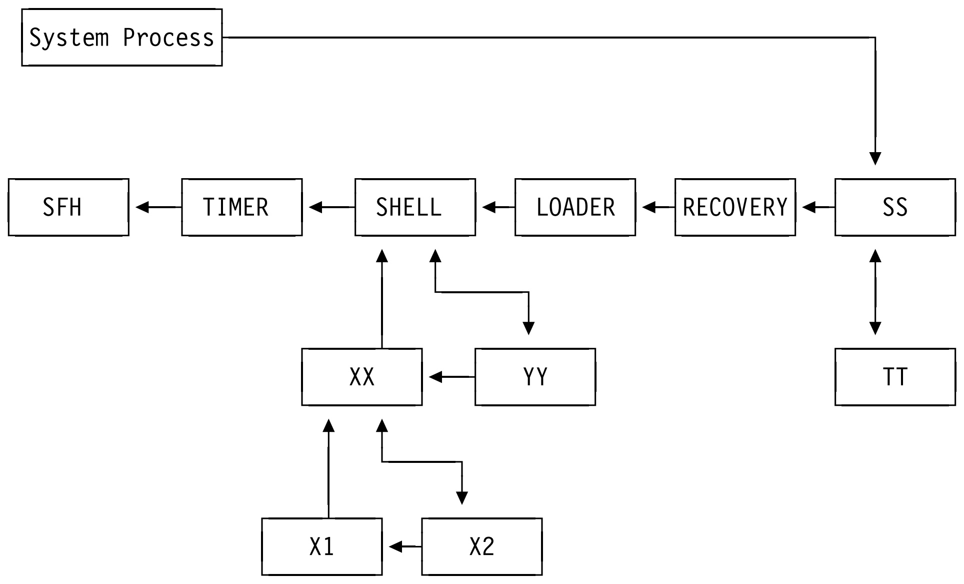


Figure 3-1. A sample process hierarchy

In Figure 3-1, a box represents a process. Boxes labelled with double capital letters are "named." (Thus, the named processes are XX, YY, SS and TT). The system process, containing the CP/Q Resource Manager and Memory Manager kernel tasks, points to the youngest process (SS) at the top level of the hierarchy. Sibling processes at each level are linked as shown by the horizontal arrows. Each process has a link to its older sibling. Vertical arrows connecting the boxes in the figure show parent/child relationships. For example, processes X1 and X2 are child processes of process XX. Process XX is the parent process for processes X1 and X2. Each process has a link to its parent process and a link to its youngest child process. In this discussion, "young" and "old" denote chronological order of creation.

In the sample illustration (Figure 3-1), assume that a command shell task runs in the SHELL process, a loader task runs in the LOADER process, that processes XX, YY and TT are all removable, and that SS is not removable. The SHELL process can "see" both named and unnamed processes in the system. Any request to remove SS is denied, because it is defined to be non-removable. The process TT can be removed. Process XX cannot be removed until its child processes X1 and X2 have been removed. If, however, a task fault occurs in SS, a fault report is sent to the fault handler for SS, who in this case (not shown in the figure) is the System Fault Handler (SFH). The action of the SFH issued with the default CP/Q system is to wait for a fault message. Once that message is received by the SFH, the SFH crashes the system. The default issued SFH can be replaced if so desired.

Each process has a base dispatch priority that is computed (at creation time) by adding a non-negative number to the base priority number of the parent process[1]. There exists an inverse relationship between the numerical value of the base

[1] The system process in the above figure has a base dispatch priority of 0. The priority number 0 is reserved for the system fault handler.

priority of a process and the dispatchability of tasks within that process. The higher the base priority number of a process, the less favored the dispatchability of tasks in that process relative to tasks of other processes. The dispatchability of any task within a process can be adjusted provided it does not exceed (or numerically be less than) the base dispatch priority computed above.

A task is a system resource owned by a process. Each process can have zero or more tasks. The number of tasks that a process can own at one time is determined by the resource limits for that process.

A task can reference a process directly (through its process ID) or indirectly (as the process which owns a certain resource). So it is not generally necessary to know the process ID.

System Resources

System resources, such as processes, tasks, message queues and semaphores can be marked “removable” or non-removable at creation time. “Removable” system resources can be dynamically created and removed by issuing requests to the Resource Manager. System resources marked “non-removable” are those whose removal causes error, such as the SYSTEM process, and cannot be removed for the lifetime of the running CP/Q system. If an attempt is made to create a non-removable system resource within a removable process, the created system resource is created removable.

Access Permissions

Some system resources, such as tasks and semaphores, have access permissions based upon task, process, or privilege level. Such access restrictions can be specified upon creation of the system resource.

A task can query information about any system resource within the running system, named or unnamed.

A task can create system resources by calling one of the Resource Manager system resource creation functions listed in the “Creating System Resources” chapter of the *CP/Q Application Programming Reference* manual. Memory is a special system resource handled by the Memory Manager. See the “Memory Manager Calls” chapter of the *CP/Q Application Programming Reference* manual for details of system calls for creating and manipulating memory.

Note: The memory creation function *malloc()* is not a Memory Manager system call, but a C Runtime Library call. *malloc()* may call the Memory Manager to request more memory.

The default operation of the Resource Manager system resource creation functions is to create the system resource as owned by the caller's process - the caller being the caller of the system resource creation function. If a new process is being created, by default it is created as a child of the caller's process.

The system resource creation functions do have options to allow the newly created system resource to be owned by another process in the process hierarchy rather than the caller's process. However, the Resource Manager does place restrictions on the scope where a new system resource can be created within the process hierarchy from a task's point of view.

A task can only create system resources in a process if at least one of the following is true:

- The process that is to own the system resource is the same as, or a descendant of, the caller's process.
- The process that is to own the system resource is the *effective process* of the caller.

Effective Process

Each CP/Q task has the concept of an effective process. Generally, the effective process for a task is the process owning the task. However, under special circumstances it can become necessary for a task to perform an operation on a different process than its own process. A task that has the privilege to change its effective process might act on other processes besides its own.

For example, a loader process, responsible for installing load modules into memory, has a need to change its effective process in order to carry out its function. For example, the loader receives a request from task A in process A to load a program into process B. The loader first validates that task A has permission to affect process B. If this validation was successful, then the loader task changes its effective process to process B in order to create code and data memory objects within process B. Process B is charged for these allocated memory objects. The loader then makes aliases to the created code and data memory objects in process B in order to insert the load module code and data sections into these created memory objects. After the loader task has finished processing the load request for process A, the loader task sets its effective process back to the loader process (the process owning the loader task) and wait for the next loader request.

The ability of a task to change its effective process is useful when creating memory aliases in other processes and also to create system resources in other parts of the process hierarchy. The effective process is charged for the system resources created on its behalf.

A task is required have “set effective process privilege” in order to have the ability to change its effective process. Effective process privilege can be specified at task creation time, provided the caller of the create task function has also set effective process privilege. Set effective process privilege can also be granted to a task by another task that already has this privilege.

Considerations

Because there can be more than one task in a process, some thought has to be given by system designers on how to organize their specific process/task hierarchy. One extreme is to have a single task in each process. The advantage is address space isolation, tasks cannot readily affect other tasks because each is in a different address space. Data is protected from potential corruption by tasks outside the current process. This can greatly reduce the scope of damage that a corrupt task can cause. A disadvantage of this scheme is that tasks cannot share memory without using specific features of the CP/Q Memory Manager Tasks within this setup use global memory or call the Memory Manager to alias or share memory between processes.

Another extreme is to have all user tasks attached to the same process. The advantage is that all data is available to every task because all tasks are in the same address space. The disadvantage is that a corrupt task can very well affect other tasks and cause them to be corrupted. The system designer is required to balance what is appropriate for their environment. See Chapter 4, “RAS Considerations” on page 4-1 for further discussion on this topic.

Another tradeoff involves privilege level, supervisor or user. System resources attached to a process, such as semaphores, tasks, and message queues that are marked for supervisor privilege only cannot be accessed by a user mode task within the same process. Supervisor privilege tasks can access both user and supervisor mode objects within their process.

Depending on the processor architecture running CP/Q, there can exist a performance tradeoff in terms of task switch time. "Task switch time" is defined in this context as the time actually required to change tasks once the decision to switch tasks has been made by the kernel. See “Call vs. Inter-Task Interfaces” on page 5-4 for further discussion on this topic.

There is a performance tradeoff on the Intel *x*86 architecture for a task switch with or without a process switch. A task switch with a process switch takes markedly longer than that for no process switch. A small part of this difference is due to executing a few extra instruction within the SVC Handler task switch code. However, most of the diffference is due to the side effects of the processor purging its page translation Translate Lookaside Buffer (TLB) when changing the address space on a process switch.

Supervisor/User Privilege

Virtually all modern processors, including the Intel *x*86 architecture, have at least two “modes” or “privilege levels.” These are referred to here as *supervisor mode* and *user mode*. Supervisor mode is reserved for such things as the operating system kernel, and “trusted” code that, for example, does input-output to hardware devices. User mode is used for “application” tasks. Supervisor mode programs and facilities are protected (by the processor) from corruption or use by user mode programs.

Some systems use supervisor mode for the system kernel, and user mode for everything else. CP/Q also uses supervisor mode for the kernel (which is defined to be the sum of the SVC Handler, the Memory Manager and the Resource Manager), but in addition supports both supervisor and user mode tasks in the system. Supervisor mode is used for tasks such as device drivers, which do I/O, or tasks which load or control other tasks in the system. User mode is used for everything else.

The Intel *x*86 architecture has defined four different privilege levels or “rings.” CP/Q uses ring 0 (the most privileged) as supervisor mode, and ring 3 (the least privileged) as user mode; CP/Q does not support the use of privilege levels 1 and 2.

Supervisor Table, SVid's, Id Translation

Elements of the CP/Q system are defined by entries in the Supervisor Table, henceforward referred to as the SVT. Such an element is specified in an SVC by its Supervisor Identifier or SVid, which consists of the index in the SVT of the entry for that system element and an “incarnation number.” If an SVT entry is deleted, and is then re-assigned to a new system element, the incarnation number is increased, so that the SVid of the same SVT entry has changed.

The index is held in the least significant 16 bits of a 32-bit value, and the incarnation number is held in the most significant 16 bits. The value 0xFFFF is never used for the incarnation number, because the value 0xFFFF*xxxx* is used in SVid fields for special purposes.

In general, SVT entries that are accessible to other elements of the system have names. Because a CP/Q system is completely configurable, and is built according to the specifications of the system designer, it is in general impossible to predict the SVid's of system elements. Therefore the SVC Handler provides a service of translating system element names to SVids.

There is a complication that can arise with the GET_ID SVC, particularly during system start-up, whereby the SVT entry that some task wishes to access does not exist, even though it is known that it is created at some point during system initialization (this is sometimes called a start-up “race” condition). Hence, the SVC Handler implements a queue associated with the GET_ID SVC, whereby a task seeking the SVid of something has the option of waiting until the required item is created. This GET_ID queue is a single system-wide queue; it is scanned (assuming it contains something) every time an SVT entry is created, to see if the new SVT entry satisfies a pending request. The GET_ID queue uses the normal wait queue pointers within the TCB (these are described below); it ignores the queue index fields required for the implementation of MUX_WAIT SVCs.

Access to a system entity (as defined by an SVT entry) can be restricted. In particular, access can be restricted to supervisor mode code, or to the system elements within the “process” to which the SVT entry belongs. For example, the SVT contains a semaphore, used by the SVC Handler to control access to the SVT. This semaphore is restricted to supervisor mode code. An attempt to access this semaphore by user mode code results in a software detected fault in the corresponding task.

Inter-task Communication

The CP/Q tasks, with the assistance of the SVC Handler, can communicate with one another through the following ways:

- Messages to task and/or system message queues
- Shared memory
- Semaphores
- User defined SVT objects

Messages

Messages can be used both for passing data between tasks and for synchronization, simultaneously if required (note that semaphores can also be used for synchronization). Messages are sent by SEND_MESG and SPLIT_SEND SVCs. Messages are received by RECV_MESG and SPLIT_RECV SVCs. Messages are uni-directional; the system has no architected notion of a "reply" to message. If the target of a message is to reply, it merely sends another message back, but in general this is independent of the sent message.

Every message sent has a message id allocated by the system as it is sent (both the sender and receiver of the message know this id). Also, each message can have a "type" allocated by its sender. Using these two facilities enables us to define a convention whereby two cooperating tasks can communicate synchronously, that is the sender of a message (which is some request, for example) can wait for the reply to his request. This is achieved by the receiver of the first (request) message setting the "type" field of the second (reply) message to the id of the request message. The sender of the request can then do a RECV_MESG which specifies to wait for a message with the "type" set to the id of the request message.

There is a special SVC, namely SEND_RECV, which does both the send and the receive in a single SVC. For the cases where a reply is expected to a request (and there are many of them), using SEND_RECV in place of SEND_MESG followed by RECV_MESG saves one SVC Handler return to the caller plus one entry to the SVC Handler. This can markedly improve system performance.

Each message requires a system block called a Request Queue Element (RQE). The number of these is limited; this means that if a task goes into a loop sending messages that are not being received, then after a short while the system runs out of available RQEs, leading to system failure. There is a facility available to check on the number of messages sent by each task that have not been received, and fault the task if this number exceeds a system limit.

Shared Memory

Shared memory can be used for a variety of purposes, including assisting in inter-task communication. One method of obtaining permanent shared memory is, when the task or tasks concerned are added to the system, to place them within the same process, with the result that they have the same address space. Alternatively, explicit requests can be made to the Memory Manager. See the Memory Manager manual for more details.

Program code can be shared between multiple copies of the program by loading it into "common" or "global" class memory. See the Memory Manager manual and the loader manual for more details.

Semaphores

Semaphores come in two "flavors":

- *Resource semaphores*, also called *serialization semaphores*, are used to enforce serial access to some system resource, or simply to serialize the action of multiple tasks. Tasks claim the semaphore, but only one task is allowed to hold the semaphore at once; the others are queued (in chronological order), and suspended until the semaphore becomes available. When the task holding the semaphore has completed its activity, it releases the semaphore; the semaphore is then given to the next waiting task, which is restarted - the other waiting tasks continue to await their turn.

- *Synchronization semaphores* are used to notify tasks of some event. Any interested task sets the semaphore, and wait till it is cleared. When the semaphore is cleared, all tasks waiting for that semaphore are restarted.

User defined SVT entries

User defined SVT entries consist of an SVT entry with a name supplied by its creator, and 24 bytes of data that can be written and/or read by any task that has access to the SVT entry. This provides a means for a task to set up some data that can be found by other tasks by specifying the name of the SVT entry in a call to the CPResolveName function.

External Interrupt Handling

Hardware interrupts, except for those the timer ("decrementer" on POWER architecture) handles specially within the SVC Handler, are routed to a first level interrupt handler (FLIH) within the SVC Handler. The FLIH notifies every task that has been nominated to be a second level interrupt handler (SLIH) for that interrupt level, in the manner expected by that task (this can be by a message, by clearing a semaphore, or by terminating an INT_WAIT). The destination tasks can be of higher dispatch priority than the interrupted task, so that a task change then takes place.

The flow diagram for such an interrupt handler is shown in Figure 3-2 on page 3-9. The interrupt handler is running as the interrupted task up to the point at which the switch to the selected task occurs. When the interrupted task is next scheduled to run, the change to that task brings control to the "recover registers" near the end of the SVC Handler FLIH; the SVC Handler then returns control to the original task at the point at which it was interrupted.

The second level interrupt handler runs as an ordinary task (but possibly with a high dispatch priority), and is entered by being resumed at its current instruction, rather than being entered at the beginning through an interrupt or jump instruction. It receives the interrupt as a message, or by the clearing of a semaphore, or by issuing INT_WAIT SVCs. This approach makes for a simple design for the system kernel. It also makes for simple design of interrupt handlers - they are just "ordinary" tasks, issuing "normal" SVCs. When an interrupt occurs, the save and restore of the current context (i.e. registers) is handled automatically by the system - the interrupt handler does not have to save the registers of another task, or worry about changing stacks, for example.

There is an additional facility whereby a "user exit" routine can be called from within an SVC Handler FLIH. Such a user exit routine is coded by the user, and the SVC handler is recompiled with the appropriate items included in the configuration file - see the SVC Handler manual for further details. The routine is then included into the SVC Handler module at link edit time. The user exit routine is called from within the scan of the SLIB chain for that interrupt, as shown in Figure 3-2.

An SVC Handler FLIH does not normally interrogate any I/O ports, attempt to dismiss interrupts, or issue I/O commands to devices (such as "end of interrupt" to the interrupt controller. Actions of this nature are left to the interrupt handler tasks

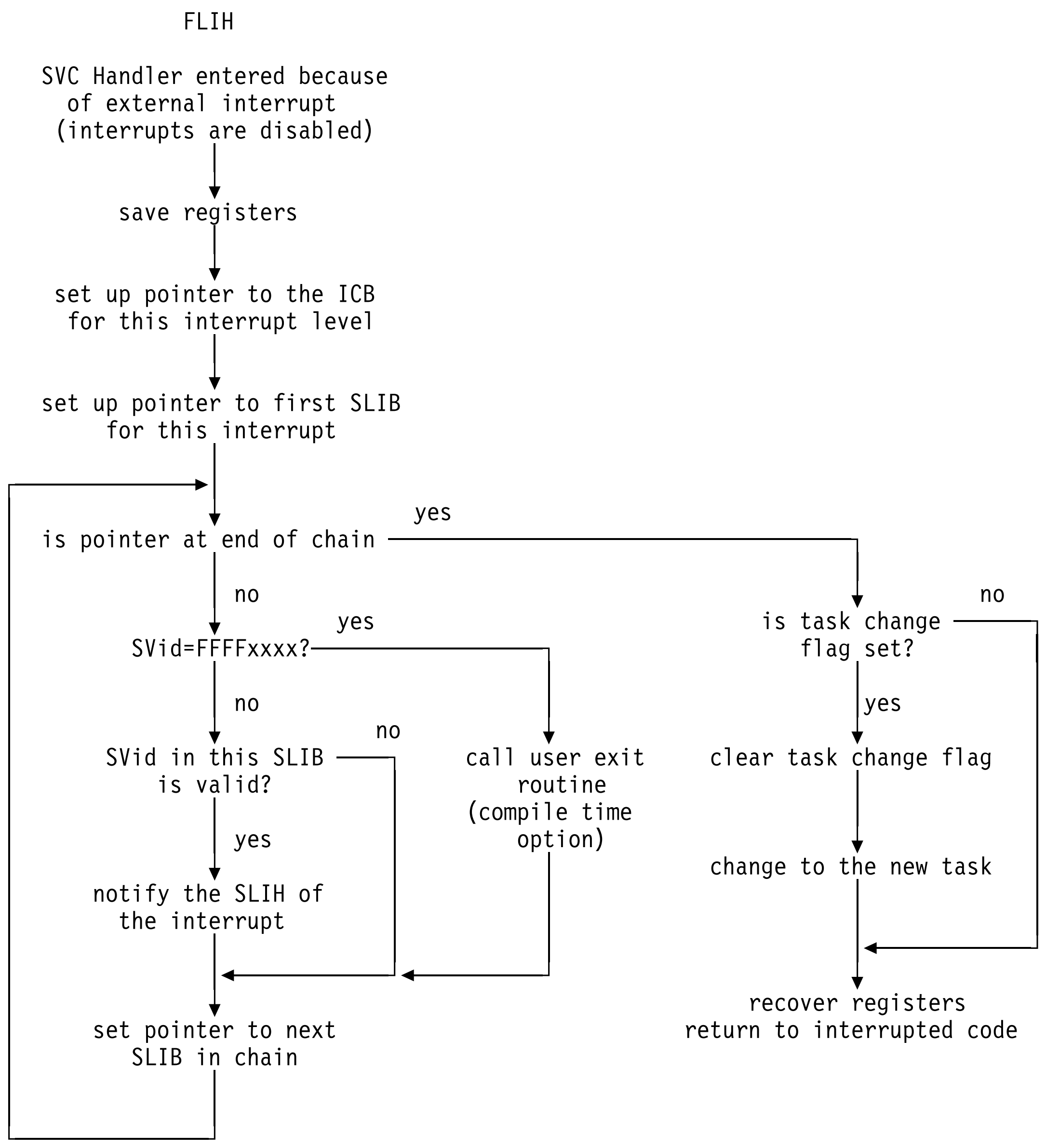

FLIH
SVC Handler entered because of external interrupt (interrupts are disabled)
save registers
set up pointer to the ICB for this interrupt level
set up pointer to first SLIB for this interrupt
is pointer at end of chain
yes
no
SVid=FFFFxxxx?
yes
no
SVid in this SLIB is valid?
no
yes
notify the SLIH of the interrupt
set pointer to next SLIB in chain
call user exit routine (compile time option)
is task change flag set?
no
yes
clear task change flag
change to the new task
recover registers return to interrupted code

Figure 3-2. Flow diagram for SVC Handler FLIH

(or the user exit routine). The SVC Handler FLIH's sole responsibility for a given interrupt is to post and start the nominated interrupt handler tasks. One obvious exception to this is the system timer, which is handled entirely by the SVC Handler. The only other exception to this rule arises when a hardware interrupt occurs from an interrupt level for which there is no SLIH, and for which either there is no compiled-in user exit routine, or the exit routine did not handle the interrupt. In this case, the SVC Handler sets the appropriate bit in the External Interrupt Mask Register (EIM) of the interrupt controller, and issue the necessary End of Interrupt (EOI) commands. This has two effects. First, no more interrupts arise from that level, and second, interrupts continue from lower priority interrupt levels.

Shared Interrupt Levels

The SVC handler allows for multiple SLIHs for a hardware interrupt level, either with or without a user exit routine called from the FLIH. That is, there are facilities to share a hardware interrupt level between more than one device driver task. When a hardware interrupt level is shared, it is up to the different device driver tasks to each decide if this interrupt is really from the device serviced by that device driver; if not, it can simply ignore the interrupt. The device driver that decides it is it's interrupt, after taking any necessary or appropriate action, issues the appropriate “End of Interrupt” command to the interrupt controller. Be aware that this can produce problems at run time. There are at least two particular cases:

- If there is one or more SLIHs for a level (either with or without a user exit routine), but none of them recognizes the interrupt as its own, and consequently does not clear it or issue the appropriate EOI commands to the interrupt controller, then no more interrupts are received from that level, or from lower priority interrupt levels.

  Currently, the SVC Handler does not recognise this problem.

- If there is a user exit routine and no SLIH for a level, and if an interrupt occurs which is not handled by the user exit routine, then the interrupt is masked by the SVC Handler, which results in the exit routine not receiving any further interrupts.

  The SVC Handler might easily detect this problem (currently it does not), but because there is no architected method of clearing an interrupt from a device because every device is different, the SVC Handler cannot readily correct this problem.

Fault Handling Structure

Almost all modern processors detect program or system faults (interrupts). These can in general be classified into two categories:

- System faults

  These are faults that indicate there is something wrong with the operating system, or there is a hardware problem.

  On Intel systems, these are:

  – Non-Maskable Interrupt (NMI), interrupt 2
  – Double exception, interrupt 8
  – Invalid task sate segment, interrupt 10

- Program faults

  These are faults specific to the current running task or program. This category includes such items as page faults, protection faults, floating point unit not available, and so on.

  Certain other program events, notably the task terminating, are handled in exactly the same way as the task faulting.

The issued CP/Q system does not handle system faults, they result in a system abend if they occur. Note that NMI on Intel systems, is used by the system debugger probe NAP to provide the facility of returning to the SLEEP debugger when the system is running.

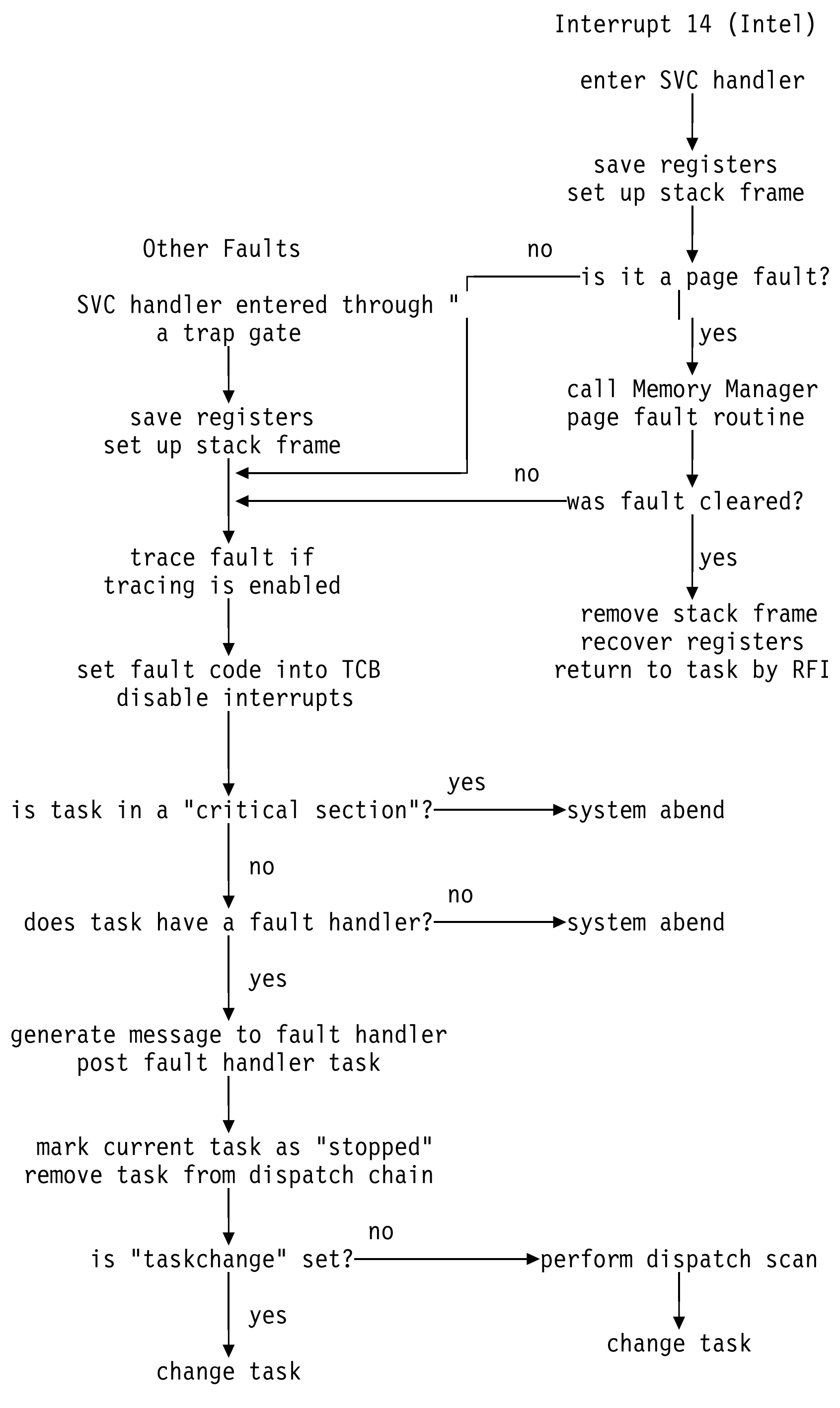


Figure 3-3. Flow Diagram for SVC Handler Fault Handling

The SVC handler contains code to receive notification of all program faults from the processor, and handles them in the following way (see figure Figure 3-3 on page 3-11):

- The task concerned is faulted, that is it is stopped, and information is put into the associated Task Control Block (TCB) to indicate why the task was stopped.
- A message is sent to the fault handler of the failing task, as specified by a field in the TCB.

Apart from this, the SVC Handler *takes no action whatsoever* on program faults. Each task has defined a fault handler. As far as the SVC Handler is concerned, the fault handler is simply a task which is notified if the task in question has faulted or terminated. For example, tasks that are loaded by the Command Shell issued with CP/Q have the Shell as the fault handler. It is also possible for a debugger task to take some other task under test; then the debugger is made the fault handler of the task being tested. It is left to the fault handler task to decide what to do about the faulted task; thus, if the shell is told that a task it loaded has terminated, the shell might well decide to remove that task from the system. A debugger, on the other hand, probably inserts break-points into the task it is testing, and then it is notified (as a form of program fault) when the task hits one of them. The action that follows this event is dependent on the commands given by the user to the debugger.

The SVC Handler assumes, if the faulted task is subsequently restarted, that some external agency (possibly the fault handler task) has taken whatever action is necessary to remove the cause of the fault, and the SVC Handler simply returns to the interrupted task. The worst that can happen (if the fault has not been repaired) is that the task immediately faults again.

The SVC Handler does contain code to handle all page faults, and simply fault the task when they occur. However, this page fault handler in the SVC handler is not normally used. Instead, a routine in the Memory Manager is called which handles page faults. The Memory Manager examines the fault, and can allocate a physical memory page if possible (that is, implement dynamic page allocation), in which case the task is restarted, or it can reply to the SVC Handler that the task is to be faulted.

CP/Q Memory Model Overview

The purpose of this section is to cover the basic memory model as implemented in all versions of CP/Q. The information in this section provides the necessary information for typical application programming. A following section covers the memory model in more detail, including Intel version specific aspects of memory management. Systems design and I/O programming requires consulting the section specific to the Intel version.

The Memory Management component is responsible for managing the different virtual address spaces as well as the physical address space (to include ROM, device memory, normal RAM memory, and not-in-use address space).

The basic operations which are performed by the memory management component are:

- Memory (object) allocation and de-allocation
- Change memory object attributes
- Sharing memory objects
- Fix/free memory for I/O
- Recover all memory resources from terminating process

All memory objects in CP/Q are in multiples of a page (4096 bytes) and are aligned on page boundaries. Requests for sizes of less than a entire page or that are not page multiples make inefficient use of real storage. Do not confuse the services provided by the CP/Q Memory Manager with functions, like the C library functions malloc() and free(), which manage memory contained within one or more memory objects.

Linear Memory Layout

The total address range that is available to a program, at any given instant, is 4 gigabytes (32-bit addresses). The virtual memory mechanism provides a mapping between these addresses and real storage addresses. This mechanism provides a vehicle to implement a a virtual address space for each process. Having a separate address space for each process provides isolation and protection between processes in CP/Q. The use of the virtual memory mechanism of the processor does not imply demand paging. CP/Q implements virtual memory but does not do demand paging, in other words, paging to disk, and so on.

Each address space in CP/Q is made up of three different classes of memory. The general layout of the classes is shown in Figure 3-4 on page 3-14. The classes are made up as follows:

Global Objects in this class appear in all address spaces and have the same attributes in all address spaces. This provides a simple system wide method of memory sharing between processes. Typical objects in this class include: the CP/Q kernel, the System Data Area (SDA), and system kernel extensions and device drivers.

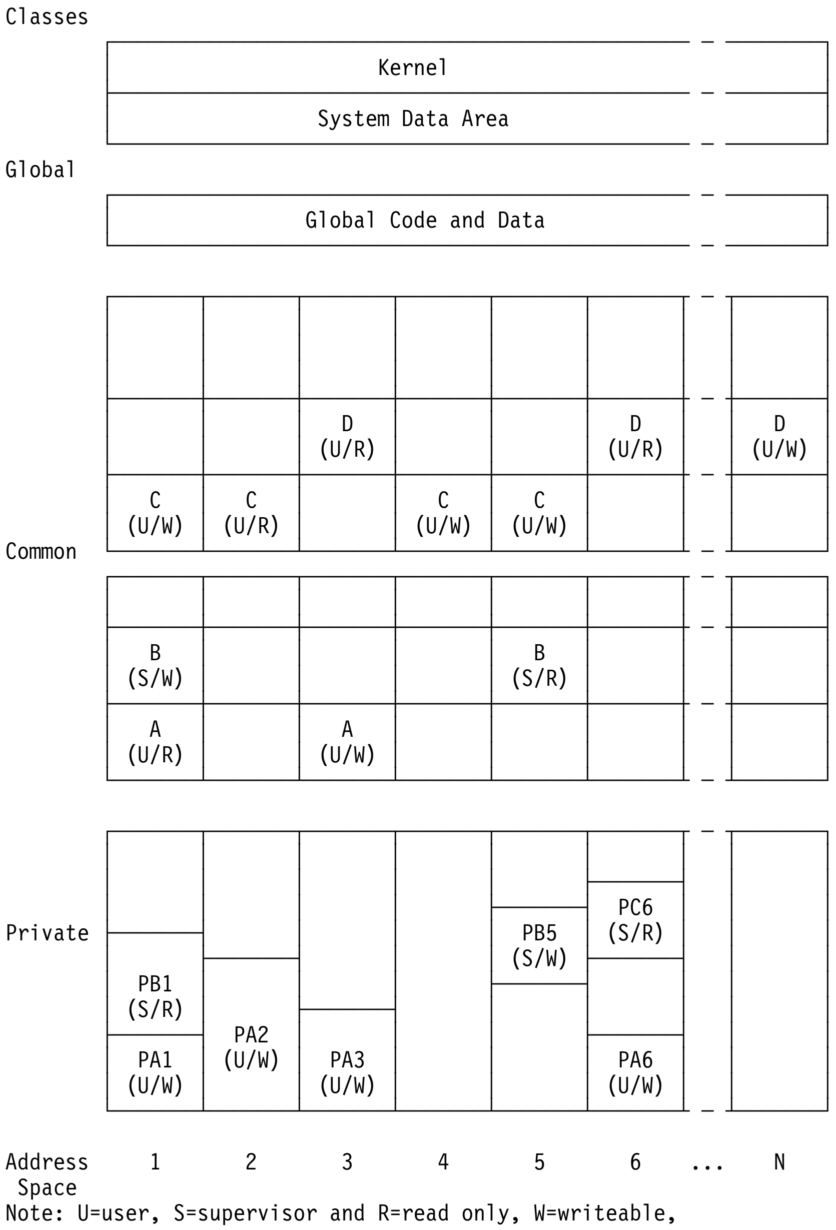
Note: In some implementations the kernel and SDA might not be visible.

Common Objects in this class can appear in multiple address spaces, and can have different attributes in different address spaces. The user has control over which processes can access common objects and what type of access they are allowed. This enables a more selective memory sharing mechanism than simple global memory sharing. Depending on version, common objects might or might not be located at the same offset in each address space where they appear. There are two modes in which common class objects manifest themselves in an address space:

1. Shared Mode - for each address space in which the object exists, reference is made to the same underlying physical pages. Each address space in which the object exists, shares a single copy of physical storage. The individual address spaces can have different access attributes for the object (for example, Read/Only versus Read/Write).
2. Copy Mode - In copy mode, a unique copy of the actual underlying data is made for each address space in which the object exists.

Shared mode common objects behave in a manner quite similar to System V shared memory.

Private Objects in this class appear only in a single address space. Aliases can be created which maps private objects into another address space, but in this case the object appears at a different offset in each address space. Typical objects in this class include: stack, heap, and private copies of code and data.



Note: U=user, S=supervisor and R=read only, W=writeable,

Figure 3-4. Linear Memory Layout

The basic unit of memory is the page (4096 bytes). This is the unit with which the hardware virtual memory mechanism operates. It is also the basic unit on which protection is implemented.

Memory sharing

There are three levels of memory sharing provided by CP/Q's memory manager.

1. Global objects - These objects are located in the global class and appear uniformly in all address spaces. Such objects are typically permanent with respect to an IPL.
2. Shared objects - These objects are located in the common class. When created, the object is only visible in the address space in which it is created.

Making an object visible in any other particular address space is done through one of two styles of sharing:

a. Give - The creating process gives the object to another process through a **QMgivemem** system call, which causes the object to appear in the target process.
b. Get - A process requests access to an object through a **QMgetmem** call, which causes the object to appear in the process.

When a common class object is created it is specified to be shareable by either the *give* or *get* method. Secondly, the mode of the object is defined. These affect how the underlying storage is managed.

Shared mode All users of a shared mode object view the same underlying memory. This mode is a peer sharing approach. The underlying memory remains until the last user departs. It is not necessary for the creating process to remain. In the *give* style, however, no new appearances of an object are possible once the creating process has deleted the object. In contrast, with *get* mode, new users can get access to an object as long as it exists in any address space. The views in different process' address spaces need not be the same, but the attributes of a view cannot be less restrictive than the original attributes.

Copy mode In copy mode, a unique copy of the underlying memory is made for each address space in which it exists. This is not a peer style arrangement in that there is the notion of the original object, a copy of which one is obtaining. Once the creator has deleted the object, new copies cannot be obtained by either the *give* or the *get* mechanism. However, those users who have already obtained a copy, can retain it even after the original is gone. Note that the copy of the original is made at the time the copy is requested. This means that different users, in general, obtain different data in their copy if the creator has updated the data because creation.

3. Aliases - This is a capability to provide access to an object (or part of an object) located in another address space. The aliased area does not appear at the same offsets in the two address spaces. These aliases are created by a QMalias call and are intended for such functions as:

   - Access to I/O buffers for file subsystems or device drivers.
   - Debuggers or other system tools.

   Such sharing is typically transitory and between a single pair of address spaces.

Privilege levels

The CP/Q system implements two levels of privilege, supervisor and user. Supervisor privilege is referred to as being more privileged than user level. When making a Memory Manager function call, the privilege level of the caller is determined by the privilege level of the code making the actual Memory Manager call.

CP/Q Memory Protection and Access Hierarchy

CP/Q uses the protection facilities of the processor architecture to provide protection. Each CP/Q memory object has access attributes defined when they are created. In order to maintain system integrity, a number of memory management calls have restrictions about relaxing access to a memory object. The four possible levels of accessibility of a memory object are listed below in order from most restrictive to least restrictive.

1. Supervisor/Read
2. Supervisor/Write
3. User/Read
4. User/Write

Depending on the version and specific processor, not all levels of accessibility might be distinguishable/operable. At a minimum however, user level code cannot alter any supervisor object, nor can user code alter a read/only object.

Memory Ownership

The resource ownership boundary in CP/Q is the process. When a memory object is created, it's ownership is recorded in a specific process. This is the effective process of the caller of the memory management call creating the object. Under normal circumstances the effective process is the actual process of the caller. It is possible to change effective process by using a Resource Manager function if required. However, this is not normally the case in application code. Once an object is created in an address space, it is visible to all tasks within that address space, subject of course, to the protection mechanisms discussed above. Removal of an object and certain other operations are restricted to the objects owner.

Process Memory Limits

CP/Q has per process and system wide limits which it manages on certain resources. Virtual memory is one such resource and the Memory Manager ensures that a process' virtual memory usage conforms to these limits.

Real Storage Allocation and Sparse Objects

When a memory object is created, one has two options with respect to the assignment of real storage page frames to the object.

1. Assignment at allocation - when this option is requested page frames are assigned for the entire object when it is allocated. This has the advantage of synchronously ensuring, at the time of allocation, that sufficient real storage is available to satisfy the request. It has the disadvantage that real storage is consumed immediately and without regard to whether it is actually used or not. Use this method, however, for critical items which cannot tolerate page faults, and, on non-demand paged systems, for critical items which cannot tolerate the risk of being faulted asynchronously, due to the Memory Manager being unable to provide a page frame during page fault processing.

   Implementers Note: Assignment on allocation is available as an option on all memory manager allocation calls. It can also be forced as the standard behavior by a compile time switch in the Memory Manager.

2. Assignment on use - when this option is requested no real storage is assigned to the object when it is allocated. When a page within the object is referenced

for the first time, a page fault occurs and the Memory Manager assigns at that time a page frame. This has the advantage that no real storage is used until/if the page is actually needed. This has the effect of reducing the overall demand for real storage. It has the disadvantage of a possible asynchronous fault that can occur in a non-demand paged system if the Memory Manager did not have a free page to clear the page fault.

Sparse objects do not follow either of the normal methods, described above, for the assignment of page frames. When a sparse object is created no page frames are assigned. In addition, the pages are marked in such a way that the Memory Manager does not assign page frames when a page fault occurs. Instead, the Memory Manager considers a page fault to be a fault and the task is faulted. The only way that page frames are assigned are by explicit calls to the memory manager to commit page frames to an area within a sparse object. A Memory Manager call is also provided to decommit pages from a sparse segment. Sparse objects are intended for things like system data structures where room can be provided for dynamic growth of the data structures, but only in response to explicit requests. This has the advantage of allowing offsets to be compiled in, or tables built directly without the possible performance implications of resorting to linked lists to manage dynamic growth of performance critical system data structures.

CP/Q Memory Model Specifics

Linear Memory Layout

The total linear address range that is available at any given instant is 4 gigabytes. The paging mechanism provides a mapping between linear addresses and real storage addresses. This mechanism uses a Page Directory (PD) and one or more Page Tables (PT) to provide the translation between linear addresses and real addresses. The processor's Control Register 3 (CR3) points to the Page Directory. By changing the PD, by changing the contents of CR3, a new linear address mapping can be made operative. Each mapping, or Page Directory, constitutes an address space or virtual address space. A separate address space is provided for each process in CP/Q.

Each address space in CP/Q is made up of three different classes of memory. In turn, each class can be made up of one or more ranges of linear addresses. Each class (and range within class) is aligned on four Mb boundary. The layout of the classes is shown in Figure 3-5 on page 3-19. The classes are made up as follows:

Global Objects in this class appear in all addresses spaces and have the same attributes in all address spaces. These objects are mapped with a single set of page tables, and each address space's page directory points to these page tables. Typical objects in this class include: IPL ROM; the CP/Q kernel; the System Data Area (SDA); a 1:1 linear to real mapping for the page manager; and system kernel extensions and device drivers.

Common Objects in this class can appear in multiple address spaces, and can have different attributes in different address spaces. However, the object appears at the same offset in any address space in which it appears. In other words, common class objects always appear at the same linear address range in each address space in which it appears.

Each address space has it's own page tables for this class, and there are two modes in which common class objects manifest themselves in an address space:

1. Shared Mode - for each address space in which the object exists, the address space's page tables point to the same physical pages. Each address space in which the object exists share a single copy of storage. The individual page tables define the attribute of the address space's access to the object.
2. Copy Mode - In copy mode, the linear address range is shared, but a unique copy of the actual underlying data is made for each address space in which the object exists. In addition, in copy mode, there are two possible mechanisms for implementing the copy function: immediate copy, and copy on write. These are discussed further in the next section.

Typical objects in this class include programs, code and data. By creating shared mode code objects and copy mode data objects for a program, it is possible to share programs across multiple address spaces with unique data objects in each process. In this architecture this is the only class in which programs which have instance data, the normal situation, can be shared.

Private Objects in this class only appear in a single address space. Address spaces have their own page tables for this class. Page aliases can be created which map private objects into another address space, but in this case the object appears at a different offset in each address space. Typical objects in this class include: stack, heap, and private copies of code and data.

The Global class has a range of at least 128K located at 4G to accommodate the power on/reset ROM addressing. All other ranges are configurable to allow the system to be tailored to particular hardware configurations. In the actual implementation the V=R mapping range starts at linear offset 0.

The basic unit of memory is the page (4096 bytes). This is the unit with which the hardware, page directory and page tables, operate. It is also the basic unit on which protection is implemented. A page can be user or supervisor and can have read only or read/write access.

Memory sharing

There are three levels of memory sharing provided by CP/Q's memory manager.

- Global objects - These objects are located in the global class and appear uniformly in all address spaces. Such objects are generally permanent with respect to an IPL.
- Shared objects - These objects are located in the common class. When created their range of linear offsets are reserved in all address spaces. This reservation of offsets does not make the object visible in any other address space beside the creator's. Making an object visible in a particular address space is done through one of two styles of sharing:
  1. Give - The creating process gives the object to another process through a QMgivemem call, which causes the object to appear at the reserved offsets in the target process.

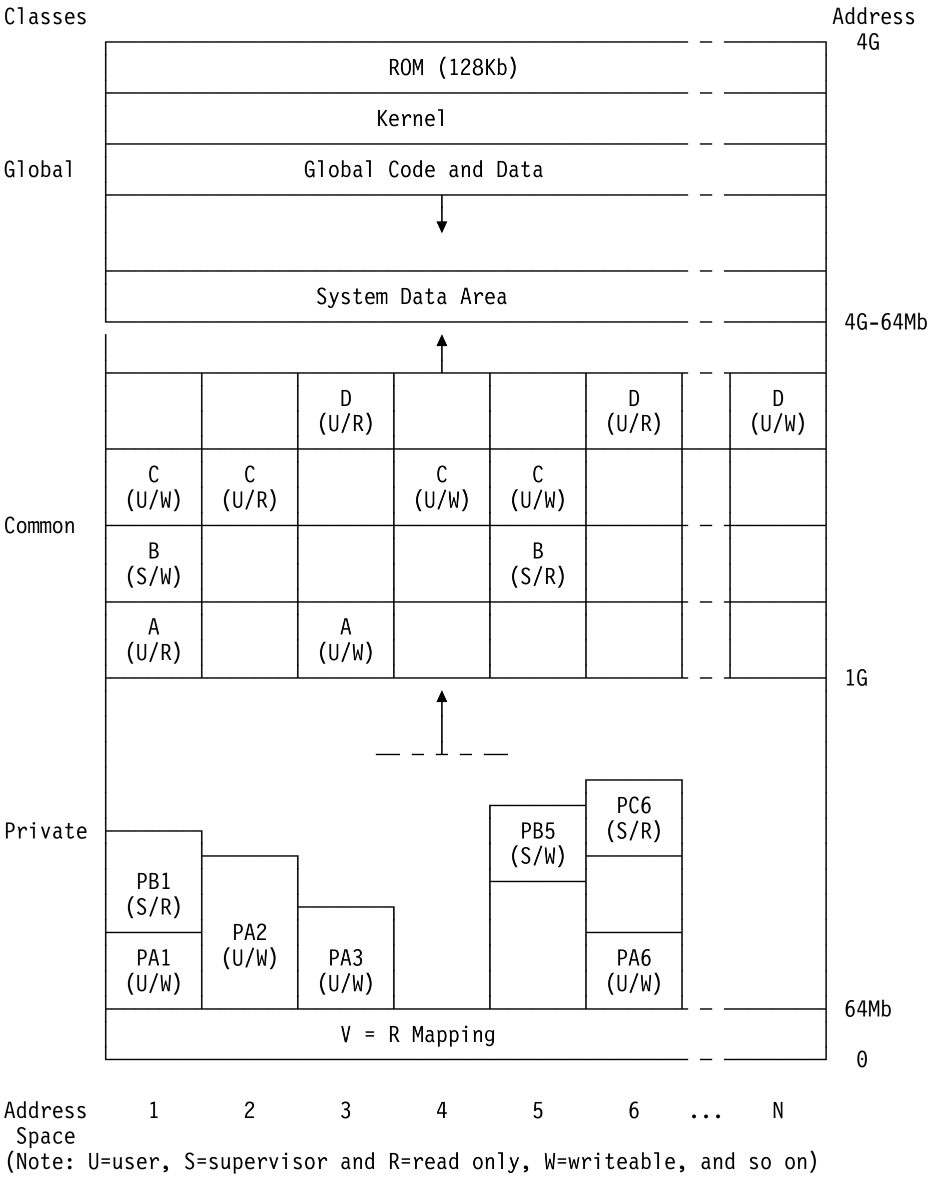


Figure 3-5. Linear Memory Layout

2. Get - A process requests access to an object through QMgetmem call, which causes the object to appear at the reserved offsets in the process.

When a common class object is created, it is specified to be shareable by either the give or get method. Secondly, the mode of the object is defined. These affect how the underlying storage is managed.

Shared mode All users of a shared mode object view the same underlying memory. This mode is a peer sharing approach. The underlying memory remains until the last user departs. It is not necessary for the creating process to remain. In the give style, however, no new appearances of an object are possible once the creating process has deleted the object. In contrast, with get mode, new users can get access to an object as long as it exists in any address space. The views in different process' address spaces need not be the same, but the attributes of a view cannot be less restrictive than the original attributes.

Copy mode In copy mode, a unique copy of the underlying memory is made for each address space in which it exists. This is not a peer style arrangement in that there is the notion of the original object - a copy of which one is obtaining. Once the creator has deleted the object, new copies cannot be obtained by either the give or the get mechanism. However, those users who have already obtained a copy, can retain it even after the original is gone. The address range does not become free until all users have deleted the object. Note that the copy of the original is made at the time the copy is requested. This means that different users, in general, obtain different data in their copy if the creator has updated the data since creation.

- Page Aliasing - This is a capability to provide access to an object (or part of an object) located in another address space. The aliased pages do not appear at the same offsets in the two address spaces. These aliases are created by a QMalias call and are intended for functions such as the following:
  - Access to I/O buffers for file subsystems or device drivers.
  - Safe send messaging of large messages.
  - Debuggers or other system tools.

  Such sharing is typically transitory and between a single pair of address spaces.

Copy on Write (COW) - there are a several ways in which to implement copy mode objects. The copy on write technique offers some attractive features. This has no direct underlying Intel processor architectural feature corresponding to this attribute. When a common class object is created in an address space by a get or give function with the copy mode, the page table entries for the object in the new process are set up with the read only attribute, but the object is noted as being copy on write. The original process' page tables are also changed to read only. As long as all the processes access the object through reads, they continue to share the same real storage page frames as the original object. If a process writes into the object, a page fault occurs. The page fault handler recognizes that this is a copy on write page frame, and create a unique copy of that page frame and enter it in the page table of the process that did the write.

Note: The processor features needed to support Copy-on-Write are not available on the 80386 processor. When running on an 80386, physical copying is always performed. Note that DMA I/O into a page frame does not use the Page Tables and therefore compromises COW. Therefore, DMA I/O cannot be performed directly into Common Copy-on-Write objects. Make a page alias which forces a physical copy to be made.

The net result is that each process has it's own logical copy of the object, but only has a unique physical copy where the actual contents of a page frame have been modified. This is very useful for creating independent data objects for use by code across processes. The data objects appear at the same offset as required by the single shared copy of the code object, but each data object has it's own copy of the data object. If the data object has a large amount of constants, text, and so on, or much of it is not typically used, then substantial savings in real storage are possible. It is also possible to use copy mode with common code objects to create a unique copy of a piece of shared code so that a debugger can insert breakpoints that only stops a single thread of execution of that shared code.

Warning
Copy-on-Write mode copy objects cannot be used for PL0 stacks as this causes double faults.

Code Sharing/Loader

A specific use of memory sharing is in the loading and management of programs. With the memory management facilities described in this volume it is possible to write a loader which enables programs to be shared among processes using only a single copy of the code (and a single copy of any read only data pages). A linear memory layout to support such a loader is shown in Figure 3-6 on page 3-22.

The loader runs as a separate process. When requested to load a shared program module, it creates a common memory object, in it's own address space using Memory Manager calls, for the code and data objects of the module. It reads the objects in from the program module and fix up any relocatable offsets in the objects to reflect the offsets where the code and data are loaded. It then gives shared mode read only access to the code object to requesting process, and copy mode write access to the data object. Copy on write is an attractive method of producing the copies of data objects, particularly if they contain a large amount of constants. It then creates in the address space of the requestor any stack and/or heap called for by the module. The loader can then perform the other processing required to complete the loading operation for that process. The loader retains it's copy of the code and data objects as well as other module information, so that it can create additional copies in other address spaces. Note that, for debugging of shared code objects, it might be appropriate to create copy mode code objects so that the debugger can insert breakpoints into the process code instance being debugged.

If a process requires a second copy of a program module to be loaded and executed in the same address space, as a separate task, it is necessary to load a second copy of the code and data objects into the private class for that process. It is not possible, in any completely general way, to have a single code object refer to two different data objects in the same address space, as the offsets of items in the data objects are contained in the code object. It is possible, however, to create with some care and forethought, programs in which there is either no data object or the data object is read only. Such programs use automatic variables on the stack, or dynamically allocated memory from either the heap or Memory Manager calls for all instance data. Alternately, by programming convention the read/write data object within the data module can contain only shared global data for the tasks implemented by the module with all truly instanced data being automatic/dynamic. Finally, the loader can also load a purely private copy of a program module into a process' private memory. Ensure that the linker and/or language processors have the capability to flag data objects as being read only or shared, so that the loader can recognize whether the module can be shared by tasks within a process or whether it has to load a completely separate copy.

The CP/Q Memory Manager facilities have been designed to enable a loader to offer the facilities described above. These facilities also enable a dynamic link capability to be implemented.

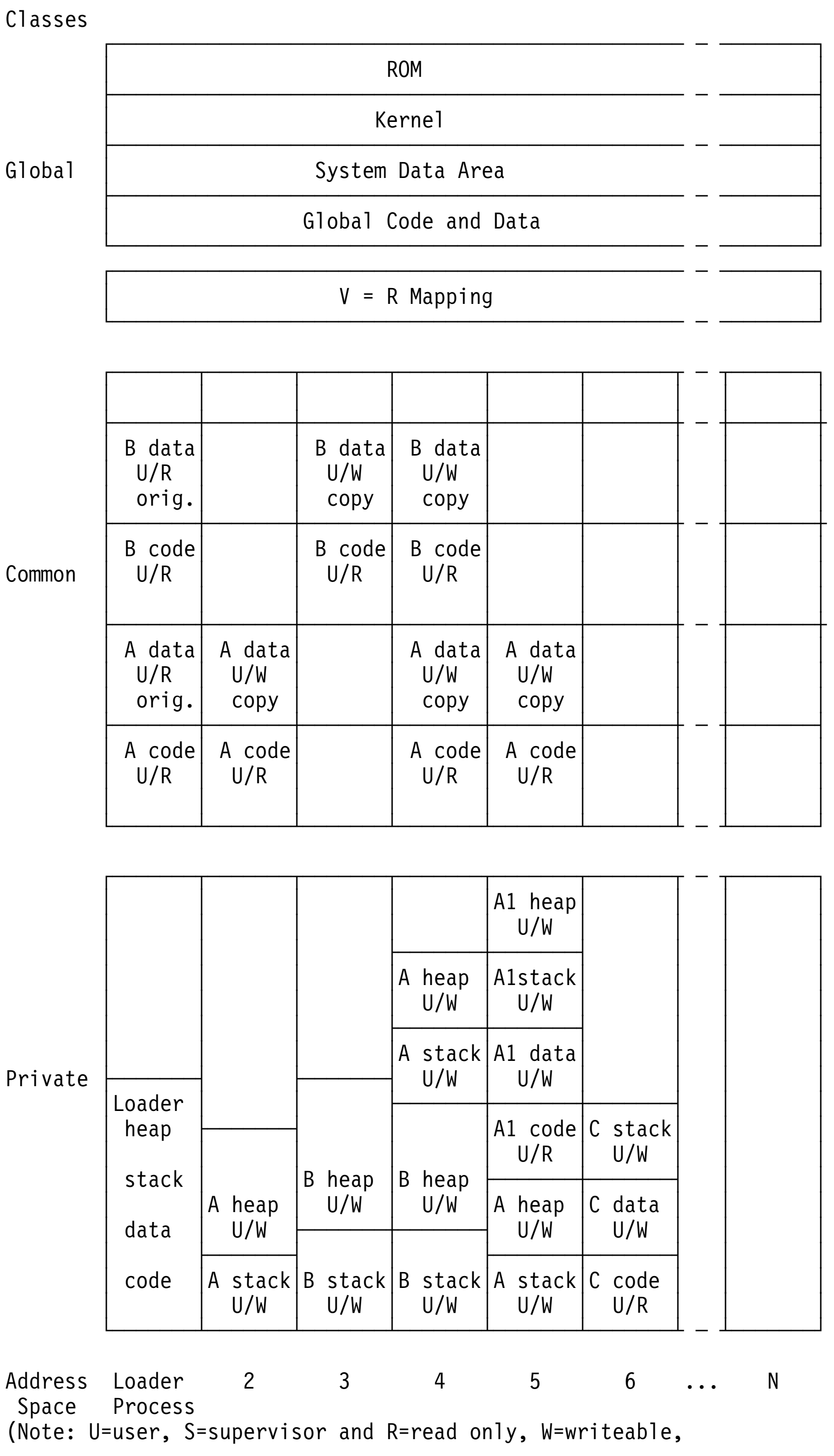


Figure 3-6. Loader - Code Sharing

Intel Privilege level terminology

The Intel architecture defines four privilege levels in protect mode operation of the iAPX 80386 processor. These privilege levels are encoded as a two bit field in a number of internal processor registers as well as in Intel defined memory descriptors. The highest, most privileged level is defined as the numeric value of 0. The lowest, least privileged level is defined as the numeric value of 3. Secondly, the paging mechanism of the Intel processors has a second privilege mechanism at the page level. At the page level, a page can be a user page or a supervisor page. Programs executing with a CPL of 3 (PL3 selector in CS) are running at user level and can only access user pages. Programs at all other privilege levels can access both supervisor and user pages. This terminology can lead to some confusion. This manual has adopted the following conventions for describing privilege level:

- Where possible the descriptive privilege level terminology of user and supervisor is used.
- Qualitative references to privilege level like; "Requires more privilege" refer to higher privilege levels which correspond to lower numeric values.
- Quantitative references to privilege level that refer to specific instances of privilege level information like the IOPL, DPL, CPL or RPL fields refers to the numeric value of the privilege level. Therefore, statements like "CPL are required to be less than or equal to IOPL" refer to the numeric values, in this case implying that CPL is required to have the same or greater privilege level than IOPL.

Supervisor Level vs IOPL

The CP/Q Memory Manager, as written, does not assume that IOPL is necessarily set to PL0, supervisor level. Therefore, some memory manager calls have supervisor level as a prerequisite, while others have IOPL. The standard system does, in fact, have IOPL set to PL0.

CP/Q Memory Protection and Access Hierarchy

CP/Q use the page level protection facilities of the Intel architecture to provide protection. The access attributes of a CP/Q memory object is defined in terms of the Intel architecture. In order to maintain system integrity, a number of memory management calls have restrictions about relaxing access to a memory object. The four possible levels of accessibility of a memory object are listed below in order from most restrictive to least restrictive.

1. Supervisor/Read[2]
2. Supervisor/Write
3. User/Read
4. User/Write

An executing program has a protection level defined, either supervisor or user. The level is defined as the current CPL, privilege levels 0-2 are defined as supervisor, and privilege level 3 as user. A supervisor level program can access both user and supervisor memory objects. A user level program can only access user level objects. In either case the type of access is further limited by the setting of the R/W bits[3].

[2] Not available when running on a 80386, as the the 80386 chip does not implement read-only/read-write protection at supervisor level. All memory is writable at supervisor level. This also means that Copy-on-Write cannot be implemented on the 80386, physical copies are done automatically.

[3] Only at user level on 80386.

Memory Ownership

The resource ownership boundary in CP/Q is the process. When a

Real storage management

Real storage regions and categories

The total real address space is divided into regions which fall into three possible categories. The categories are:

- General purpose storage - physical memory which can be used to provide real memory to back normal requests for memory objects. It consists of any one or more general purpose physical memory regions which have defined to Memory Manager. One or more initial regions of general purpose storage are defined during the system build and IPL process. Additional regions can be defined by Memory Manager calls and, as parameters to the memory task. At any given moment each page frame in a general purpose region of storage is either on the freelist (a list of pages available to satisfy normal allocation requests) or is actually allocated to provide backing store to one or more memory objects.
  Current Implementation Assumption: All general purpose storage is assumed to be cachable and, is required to be located below 64MB in the physical address space.
  Standard Memory Configuration: When using the standard system on a PS/2 two regions of general purpose storage are defined through the system build/IPL process. The system is built with a 2Mb region of memory beginning at 1Mb. The size of this region can be specified through the SLEEP SYSPARM MEMsize parameter. The IPL process, whether through a SLEEP go command or an IPL through IPLCP386, defines an additional region from the top of SLEEP/IPLCP386 to 636Kb (640 - the 4k extended BIOS data area).
- Adapter storage - memory which is not intended for general use. Typically, such things as I/O adapter memory or other forms of memory mapped I/O. Adapter storage is not available to satisfy normal memory allocation requests. It can only be used to satisfy requests for specific physical addresses - that is through function 97, QMallocbase. The system includes a memory object to access the 386KB from 640k-1MB. The offset of this object is contained in the first page of the CDA and can be obtained through a GET_CDA call to the SVC Handler. Additional regions can be defined by Memory Manager calls. Adapter storage is normally defined as non-cachable. There is a predefined 128Kb adapter range and object located a physical address and offset 0xFFFE0000 for ROM.
- Free real address space - physical address space which is not defined as general purpose or adapter storage. Additional ranges of general purpose storage or Adapter storage can be defined from free real address space through Memory Manager calls.

Real storage types

The CP/Q memory manager also supports multiple real memory types. Memory types are not to be confused with the categories of real memory - adapter and general purpose discussed above. Memory types apply strictly to general purpose memory. By real memory types, one normally thinks in terms of different types having different speeds like DRAM vs SRAM. &thing, however, does not apply any interpretation to the memory types. CP/Q supports 3 types of real memory denoted as 1,2, and 3. There is also a default type (0) which in fact is type 1, but the distinction becomes clear shortly. The system is built in the default memory type. Memory of other types is only introduced into the system at run time through the CPCreateRange call. When creating a new memory object that can be backed by real general purpose memory that is the calls CPAllocMem, CPGetMem(copy mode), and CPGiveMem(copy mode), a real memory type can be requested. If a specific type (1,2,3) is requested then ONLY that type is used to provide real storage page frames - whether at allocation time or through page fault. If the default type is specified, then type 1 memory is used if available, but if none is available, types 2 and 3 are used, in that order, to satisfy the request.

Memory object granularity and alignment

All memory objects in CP/Q are in multiples of a page (4096 bytes) and are aligned on page boundaries. This is dictated by the Intel processor architecture when using a flat model. Perform management of memory at a smaller level of granularity (such as a heap) through library, subsystem or application code.

Chapter 4. RAS Considerations

Reliability, Accessibility, Seviceability (RAS) is a very key point in today's systems. Our own quality goals, and the need to be very competitive, make it extremely important to have reliable, robust end products that achieve continuous availability in the face of various types of failures. This is not a simple goal. Keep it in mind throughout the entire product development cycle.

While features such as hot-pluggable hardware and redundant components leap to mind when considering high RAS designs, many people tend to ignore the negative or positive effect operating system choice can have on the overall product. All attention is focused on performance and size which, while extremely important, make up only two thirds of the formula.

This chapter discusses some of the RAS features of CP/Q with the goal of raising people's awareness about how to take advantage of these features in improving the RAS of their end products.

Process Separation

One of the key features of CP/Q is virtual memory support. Many of the RAS features inherent in the system are implemented through the virtual memory system. Process separation refers to the protection afforded through multiple virtual address spaces, or processes.

Processes allow data to be protected from potential corruption by tasks outside the current process. This can greatly reduce the scope of damage that a corrupt task can cause.

Also, because processes are the boundary of resource ownership in CP/Q, and because cleanup is performed on a process basis, process separation can help manage the removal and replacement of failing components.

Supervisor vs. User protection

Each virtual page in an address space is marked as either a supervisor page or a user page. In addition, each task in the system is marked either supervisor or user privilege. Supervisor privilege is usually associated with I/O privilege, in addition to some additional privileges on certain SVCs.

While a supervisor task can read and write user level pages, a user task can not alter supervisor pages. This means that, even within a single address space, an additional level of protection is available.

Supervisor privilege is usually given to "trusted" code because supervisor privilege tasks can potentially cause a great deal more damage than user privilege tasks. "Trusted" simply means that the code is well tested and the source is known. You would not normally allow supervisor privilege to be given to any new piece of code, regardless of its source.

Privilege change

Because a good design normally results in a mixture of user and supervisor privilege tasks, and because it is often the case that important functions are provided by supervisor code, there has to be a means for user privilege tasks to request services from supervisor code.

When the supervisor code is being executed by a separate task, then inter-task communication facilities can be used, such as messages or semaphores. The privilege change is handled by the kernel as part of the task change, and the supervisor code is completely protected because the action of the user task is completely passive: it has caused the supervisor task to wake from some blocking operation (for example, receiving a message), and has perhaps made some data available to it.

When the supervisor code is designed to be called directly by the user task, then something is required to occur which allows the user task to change to a supervisor task (at least, temporarily). The CP/Q kernel interfaces are an example of this type of interface. The kernel is written to be reentrant, and for performance reasons it is desirable to provide a call interface rather than a message interface. The privilege change is effected through the Supervisor Call (SVC) mechanism.

What is key here is that the SVC mechanism provides a controlled interface to supervisor code: only a limited set of execution points are exposed, and the entire privilege transition and return is handled by trusted kernel code.

In addition to kernel calls, the system also allows users to add supervisor routines that can be accessed through the SVC mechanism. These can either be statically linked into the kernel, using a reserved range of function numbers, or they can be dynamically added to the system through the CPCreateUserDefinedSVC call.

Read-only code

Most of the code in the system is in read-only pages. This is true regardless of whether the code was loaded from the IPL image, or dynamically loaded at some later time. The normal default is for read-only code and read/write data.

By having read-only code, there is no possibility of code being corrupted by a task in error. This means that no additional code is required to detect damaged code, and that tasks that go wild and try to damage code are immediately caught and identified through a page fault. This is something that can not be achieved in a real memory only operating system.

Resource recovery

All resources in CP/Q are associated with a particular process. Processes are the boundary of ownership and of recovery. When a process is to be removed, either because of normal or abnormal termination, all resources associated with that process are recovered. This includes all kernel objects (for example, semaphores), all memory (with some specific exceptions), and any additional resources being managed by Resource Provider subsystems, This last category is open-ended and could potentially be anything.

In a continuously available system, processes are bound to be created and removed. Complete resource recovery becomes key to allowing the system to run indefinitely, because over time incomplete recovery could result in exhaustion of resources, and hence, failure of the system.

Resource limits

The kernel has certain resources that are tracked for each active task and process. These include such items as the total number of Task Control Blocks (TCBs) in use for a process, or the total amount of memory allocated. The limits on these resources are configurable, and the system can detect when these limits are exceeded. There are both global default limits, and individual per-process limits that can be set when a process is created.

By proper tuning of these limits, runaway allocations can be caught and recovered from.

Access protection for kernel objects

The CP/Q kernel is placed in supervisor pages, so user privilege tasks can not access it. In addition, on the POWER implementation, the entire kernel code and data space is placed in a separate segment which is left inaccessible unless an SVC has been made to the kernel, at which point the kernel segment is made visible for the duration of the SVC. This affords an additional level of protection for the kernel for added reliability.

Chapter 5. Dynamic, Modular System Considerations

CP/Q was designed to be a highly modular system. The kernel provides base operating system services; all additional services, such as file I/O, screen/keyboard management, and so on, are added as *subsystems* outside the kernel. They are packaged in separate load modules and can be installed or left out of the system at the user's option.

This level of modularity allows CP/Q to be highly customized for any application, without the need to modify kernel code. It also means that the programming environment is more dynamic than in other systems, and that applications and subsystems alike are required to be coded appropriately.

Dynamic Subsystems

The term subsystem refers to a collection of code that provides a set of related functions, and which is normally considered to be part of the operating system. This is true regardless of whether the subsystem is built into the kernel or is started dynamically.

In a typical CP/Q installation, the File I/O and Session Manager components are considered subsystems, as is the Loader and the Sockets Server (TCP/IP support). Each of these is packaged as separate load modules (in some cases more than one load module), and none of them are built into the kernel.

Start-up Race Conditions

One type of problem that can occur in CP/Q which is not an issue for other, more static systems is start-up race conditions. This happens when there are interdependencies between subsystems. For example, the CP/Q Shell requires the File I/O subsystem when it starts, because it processes its initial command file. There is a race condition which requires the File I/O subsystem to be up a running before the Shell requests its services.

To handle this situation, CP/Q provides a queued, name resolution SVC called CPResolveName. The File I/O subsystem has a well-known name assigned to it called **QFROUTER**. Before the Shell can make a request to the File I/O subsystem, the name **QFROUTER** is resolved. The CPResolveName SVC blocks the Shell's task until the name is created.

This means the Shell does not continue until the File I/O subsystem is available. When CPResolveName does complete, it returns the SVid of the File I/O subsystem, which is needed to send it messages.

The File I/O subsystem is an example of a message-based interface. Simply by having the task created, the necessary message queue is present and requests can begin to arrive. A call-based interface needs a little more attention when handling start-up race conditions: for one thing, there is no task associated with called code; the caller's task is used to execute the function. Also, there is typically some initialization phase that is required before the code can be called: perhaps some data areas need to be initialized, or some devices need to be programmed.

Even if a separate task is used just to perform the initialization, it is not the creation of the task which is required to be completed, it is the completion of the initialization. Therefore, resolving the name of an initialization task does not achieve the desired result: queue on something else. A good candidate might be a semaphore that is part of the interface, or some other object whose name can be resolved, but which can be created last in the initialization phase.

If nothing else is obvious, a User Item can be created. User Items are just kernel objects for storing some small amount of data, perhaps a couple of pointers. This often works out well with called code that is dynamically placed in the system because there needs to be some way of making the addresses of the called routines know to applications. User Items are perfect for advertising entry points, global data locations, and so on through a well-know name. An application resolves the name of the User Item, and when the CPResolveName completed, the application knows that the interface was available. It then retrieves the entry points, and so on from the User Item and begin making calls.

Dependencies on Dynamic Subsystems

Having handled the case of start-up race conditions between subsystems, address the more difficult problem of subsystems which are dynamically stopped and possibly re-started. Using the CP/Q Shell as an example again, we'll examine its dependency on the Console subsystem. The Console is used as a write-only screen for tasks that normally do not have a screen associated with them (for example, error messages from the File I/O subsystem).

The Shell sends messages to the Console during normal execution, to print out information about faults, and so on. If the Console were to crash, or be explicitly removed, the SVid that the Shell is sending messages to is no longer be valid. This causes the next CPSend SVC to fail with a return code indicating a bad SVid.

At this point, the Shell can do a number of different things, depending on the exact nature of its dependency on the Console. As it turns out, the Shell does not have a very hard dependency, and can run without the Console installed at all. So it can choose to just let the failed CPSend pass, and mark that the SVid is no longer valid.

Another alternative is to see if the Console has restarted by again resolving the name as it did earlier to handle any race conditions. Because the Shell does not have a hard dependency on the Console, it doesn't want to block on the CPResolveName if the Console is not present. Therefore the Shell specifies a timeout of zero on the CPResolveName SVC, and the SVC returns immediately whether the Console was present or not: a return code indicates if the name was resolved or not.

Alternatively, if the Shell had a hard dependency on the Console, it can issue a blocking CPResolveName, and block until the Console reappeared.

The Console is a message-based interface, like the File I/O subsystem, so queuing on the task name (and hence, the task message queue) is sufficient to reestablish a binding. However, as with start-up race conditions, called code presents a more difficult problem.

Short of removing the memory the code resides in, thereby causing a page fault on the next call, there is no passive way to learn that called code has become stale (perhaps replaced by a newer version somewhere else in memory). Even though a User Item which describes the called code might have been updated to point to new code, causing all new bindings to use the new code, previously bound instances continue to call the old code, unless they do something to actively check for the possibility that the code has changed.

From a performance perspective, the worst case is to re-resolve the User Item each time a call is about to be made, and even that permits the possibility of a window during which the old code can be replaced, so it might even be necessary to introduce a semaphore for serialization. This is not a practical example, but it does illustrate a couple of the problems that can arise.

Because performance dictates that it isn't practical for the caller to confirm the validity of the entry point before each call, the alternative is to have the caller be told when the interface is changing. This is what occurs with a message-based interface when the target SVid becomes invalid.

However, callers are anonymous in a call-based interface unless they make their identity known. A mechanism can be provided by the subsystem whereby the caller can explicitly register as a user of the interface, and provide enough identification (perhaps nothing more than the caller's SVid) to allow the subsystem to send notification when the interface is changing. This notification might consist of the subsystem initialization task recognizing that it is not the first instance of the subsystem, and retrieving a list of registered callers from a global memory area that the subsystem manages. The subsystem's well-known User Item can achieve both of these items: by its presence, it indicates the subsystem was already installed, and it can store the address of a global area that new invocations can access.

Another possibility to consider is the case where a subsystem does not start at all. For example, it is perfectly acceptable to have a CP/Q system configured with the Shell, but without the Console.

Because the Shell only has a soft dependency on the Console, everything runs just fine, but the Console is simply absent. To achieve this, the Shell is required not to block when resolving the name of the Console.

An example of a hard dependency is the Loader's requirement for the File I/O subsystem. The Loader can not function without the File I/O interface installed in the system. Therefore, the Loader is required not to block when resolving the File I/O subsystem's name. If the File I/O subsystem never gets installed, the Loader never become active.

The problem with this example is that the Loader has been created, and therefore can be receiving requests on its message queue. To handle those requests, the Loader might instead not block on the name resolution, or block with a timeout, and then check its queue for pending requests. It can then reject the pending requests with a return code indicating the File I/O subsystem was not present. This prevents the queue from growing too large, thereby using up system resources.

The issues of dynamic subsystems can become complex, particularly when dealing with called code. This section tried to explain the problem and present some solutions to a few simple cases, but primarily, this section was intended to make the reader aware of the issue and point out why it is to be dealt with thoroughly in the design of any code.

Call vs. Inter-Task Interfaces

The decision of whether to implement a direct called interface or an inter-task interface is a complex one, and many trade-offs need to be resolved before deciding. Because performance is always desirable, and often critical, it is a good starting point. Note that it is faster to make a direct call to a piece of code than to send a message or use some other type of inter-task communication. That will never change.

With that base assumption stated, we can begin to examine some of the other issues and trade-offs that, in some cases, lead to the conclusion that a called interface might be used, and in other case lead to the conclusion that an inter-task interface (whether it be messages, semaphores or some other mechanism) is the right solution despite its apparent performance penalty.

As just discussed above, dynamic binding and re-binding of an interface, and the related problem of cleanly removing an interface, are more difficult with called interfaces than with inter-task interfaces. Therefore, called interfaces are more appropriate for static situations. For example, the CP/Q kernel is primarily call-based because we want the highest level of performance, and the kernel is static: it will not be replaced during the execution of the system.

In contrast, the various subsystems supplied with CP/Q, which tend to be less performance critical than the kernel, are message-based. This allows them to be dynamically stopped and started (unless some other restriction prevents that), and to be replaced. With today's continuous availability requirements, that is a primary goal when designing subsystems.

Concurrency is another major factor in the decision between called interfaces and inter-task interfaces. When designing a component, pay careful attention to finding areas that benefit from concurrent execution. For example, let's assume that a component performs two functions: A and B, in that order. If it is the case that each function requires that the previous function be complete before it can commence, then there is really no advantage to assigning separate tasks to each function as there is no concurrency to take advantage of: the tasks execute serially. In this case, a called interface could be used, and a single task can execute the calls serially.

However, let's assume that function B can begin execution anytime after function A reaches a particular point (B's dependency on A is on some intermediate action, not on the completion of A). In this case, assigning B to a separate task might allow it to begin execution before A completes (e.g., A blocks, or A causes B to become dispatchable by clearing a semaphore, and B is of a higher dispatch priority so it preempts A). This can allow B to execute while A is blocked, or allow B to execute as soon as A completes its critical actions, and the rest of A can complete later. Instead of both A and B being held up by A blocking, B is able to execute because it has its own task.

Another scenario might be that A can begin processing the next request in its queue before B has completed the current request. In this case, the concurrency in achieved by allowing more than one request to be in process at the same time, even if the individual functions A and B might need to operate serially for each individual request.

Whether an interface is synchronous or asynchronous is another characteristic. Called interfaces are inherently blocking, whereas inter-task interfaces can be either. Message interfaces, in particular, are typically non-blocking: sending a message queues a request to another task, but the sending task does not need to wait for the reply synchronously (some systems have synchronous send/receive primatives). In CP/Q, if a synchronous send/receive operation is desired, one can follow the convention of having the destination task send the message ID in the *type* field of the reply. The origin task can then receive by *type*, using the original message ID as the type value (the message ID was returned to the sender by the original CPSend SVC).

Having an asynchronous interface is again related to finding the concurrency in a particular application. Perhaps the requestor is of higher priority than the service, and can be allowed to continue after queuing a request for the service. Then there is a real advantage to having an asynchronous interface with two tasks.

Asynchronous interfaces can be designed using called code, but it usually involves call-back routines, which can expose the service's task to unknown, or possibly missing code (if the call-back routines are removed from memory before the service completes).

Finally, if an interface needs to be serialized among a number of tasks, the queued nature of either message-based or serialization semaphore controlled interfaces handle that naturally. Called interfaces normally need to incorporate serialization and locking mechanisms to avoid reentrancy problems.

Chapter 6. Fault Handlers

The Fault Handler facilities in CP/Q allow applications to handle serious or unexpected failures. Both hardware and software faults are detected. Hardware faults include such failures as divide-by-zero. Software faults are generated by the operating system in response to such errors as insufficient privilege when accessing system services.

Fault Handlers are another tool for achieving 100% availability. There will always be error conditions, so a truly reliable system must anticipate and handle the unexpected.

Assigning Fault Handlers

Every task in a running CP/Q system has a Fault Handler assigned to it. Assignment is done at task creation time, by passing the Supervisor ID (SVid) of the intended Fault Handler task as a parameter to the CPCreateTask SVC. Passing a value of zero causes the new task to inherit the Fault Handler of the parent task (the task calling the CPCreateTask SVC).

Other than their fault handling responsibilities, there is nothing special about Fault Handler tasks: they are normal CP/Q tasks. They even have Fault Handlers of their own assigned.

Halt and Fault Messages

When a task faults, the kernel sends a message to the task's Fault Handler. The format of these messages is described in *APL Volume 1: Application Programming Reference*. The primary action of a Fault Handler is to receive these messages from its task message queue using the CPReceive SVC. It can then examine the message to determine the exact condition.

There are actually two types of messages that are sent to Fault Handlers: the kernel also sends Halt messages, which notify of task completion (in contrast to task failure). Note that task completion does not necessarily imply success: a Halt message can contain a non-zero return code. The return code is set by a parameter to the CPHaltTask SVC. Typically the CPHaltTask SVC is issued by the C Runtime Library as a result of calling exit() or returning from main().

Fault Handler Hierarchy

There is a loose hierarchy of Fault Handlers in a running CP/Q system, starting with a System Fault Handler at the top of the hierarchy. The System Fault Handler should be global in scope, designed to handle failures from any part of the running system (there is a further discussion of System Fault Handlers below). Next might be less broadly defined Fault Handlers which each cover a subset of the running system. Any LEEDS application is a good example of this, as it covers only the tasks created by itself.

Finally, there might be application-specific Fault Handlers which are concerned only with the application of which they are a part.

CP/Q does not enforce any strict hierarchy of Fault Handler tasks: one task can act as the Fault Handler for any number of tasks in the system, at any number of different points in the task hierarchy tree. Each Fault Handler is an independent entity, with no kernel imposed relationship to any other Fault Handler in the system.

However, because of the hierarchy of scope that often occurs in a typical set of Fault Handlers (as described above), design a system such that there is a logical hierarchy of Fault Handlers. The lowest level, application-specific Fault Handlers might only handle a few specific faults that the application might be able to recover from, and all other faults would be handled by upper-level Fault Handlers. To achieve this behavior, it is necessary to *percolate* the fault up.

Percolation

Since the kernel does not enforce any relationship between Fault Handlers, the system designer must enforce the relationship. In order for an upper-level Fault Handler to process a fault, the lower-level Fault Handler needs to forward the fault message (that is, percolate it up).

This implies that the lower-level Fault Handler needs to know the identity of the next highest Fault Handler in the hierarchy (that is, the SVid of that Fault Handler task). There are many methods to pass such information. One example might be to send the lower-level Fault Handler a message with with SVid. If it is the next highest Fault Handler sending the message, then no data would be necessary: the SVid of the sender is always inserted in the header of a CP/Q message.

One additional note: when a halt or fault message arrives from the kernel, the sender field of the message header is set to **x'FFFFFFFF'**. However, when percolating fault messages, the sender field will be set to the SVid of the sending task, in this case the lower-level Fault Handler. This means the higher-level Fault Handlers should not check the sender identity, unless they are prepared to receive messages from other than the kernel. If security is a concern, the higher-level Fault Handlers can be made aware of the specific Fault Handlers below them, and verify that any halt or fault messages arrive either from the kernel, or from the proper lower-level Fault Handlers.

Supplied Fault Handlers

CP/Q is shipped with a System Fault Handler which takes actions that follow the host/coprocessor interaction rules. These include writing error codes to the mailboxes. The details appear in *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference*.

SCC Manager

The SCC Manager is a special task included in the CP/Q system for the Cryptographic Coproccessor, which is responsible for loading programs (tasks) from flash memory during system start-up, and is the fault handler for these tasks. When notified of the termination of one of these tasks, the SCC manager arranges for its removal from the system.

Glossary and Abbreviations

A

ASCIIZ. An ASCIIZ string consists of a sequence of ASCII characters, terminated by a NUL character (0x00).

B

BIOS. Basic Input/Output System. This is the device driver and interrupt handler code provided with an IBM compatible PC, in a ROM or EEPROM on the system board and certain device adapter boards. BIOS includes code to test the system on power on, to provide interrupt handlers for standard devices, and to IPL an operating system.

C

Common. Common Memory Objects. This is a "class" of memory object defined by the Memory Manager, which can be shared by different processes in a controlled manner. If a common memory object is visible in two (or more) different processes, it has the same linear address offset in each of the processes.

See also Global and Private.

COW. Copy On Write. This is a mechanism whereby programs running in different address spaces may share a data area, but have private copies of any parts (for example, pages) that are written to. This is achieved by placing read-only page references to the data in each address space; when the program attempts to write to a location, a page fault occurs. The page fault handler recognizes this situation, and creates a private writable copy of just the affected data page, leaving the other pages as read-only access to the shared data.

Safe and reliable operation of copy on write requires the supervisor mode read-only page protection facility that is present in 486 and later processors, but omitted from a 386. For this reason, CP/Q does not provide the COW facility when the system is running on a 386 (if COW mode is requested, it is implemented as copy mode, that is a physical copy is made).

CPL. Current Privilege Level. The Intel *x*86 processors have the concept of privilege levels. There are 4, numbered from 0 (most privileged) to 3 (least privileged). If a program is running at, say, privilege level 2, it is allowed to access items specified as being at privilege level 2 or 3, but not those at privilege level 0 or 1. The privilege level associated with the code currently running is called the CPL, and is defined by the two low order bits of the CS register.

The Intel specification denotes CPL 3 as "user" mode, and CPL 0-2 as "supervisor" mode. CP/Q supports only CPL 3 (for "user" programs) and CPL 0, for the system kernel, device drivers and certain other privileged modules.

creator process. The creator process, defined in the Resource Manager creation parameter lists, is the process to which the newly created object is to be attached. If a new process is being created, it is a child of the creator process.

creator task. The creator task is the caller of the create function.

D

Descriptor. The Intel processors provide memory protection facilities at the memory segment level. Each physical segment has an associated *descriptor*, which defines the segment type, access, base address and size. Other descriptors identify control blocks defined by the Intel *x*86 architecture, which determine the action taken by the processor when interrupts occur (both hardware and as a result of INT instructions), or provide and control access to code modules within a system. Descriptors are held in tables, namely the GDT, the LDT and the IDT. Consult a specification of the Intel processors for further information.

DMA. Direct Memory Access. This is the name given to the access to system memory by an I/O controller, as requested by the controller. This is used by I/O controllers, for example disk controllers, to read and/or write directly from or to system memory without requiring any intervention by the main processor; this drastically reduces the system overhead of I/O, and can produce equally dramatic increases in I/O performance.

DPL. Descriptor Privilege Level. The Intel *x*86 processors have the concept of privilege levels. There are 4, numbered from 0 (most privileged) to 3 (least privileged). If a program is running at, say, current privilege level 2, it is allowed to access items specified as being at privilege level 2 or 3, but not those at privilege level 0 or 1. Associated with any descriptor is a value, called the DPL, which specifies the maximum privilege level for code that may access the item (such as a segment) defined by that descriptor.

DRMM. Direct Real Memory Map. The linear address range 0-*n*K (where *n*K is the size of the physical memory of the machine) is reserved, and is available in all address spaces in supervisor mode as a 1-to-1 mapping of linear address to real memory, that is virtual = real.

The probe for the system debugger (NAP) uses the DRMM (it runs in virtual = real address space).

F

FLIH. First Level Interrupt Handler. A code module that is entered by the processor when a hardware interrupt occurs. In a CP/Q system, this term is used more particularly for the FLIHs within the SVC Handler, that generate interrupt notifications for second level interrupt handlers (SLIHs). There can be only one FLIH for any particular hardware interrupt level.

G

GDT. Global Descriptor Table. A table of descriptors, architected by Intel as part of the specification of the *x*86 processors. There is only one active GDT in a system, which is set up during initial system load, and cannot be easily changed once the system is running. This table is used to define the memory segments or code modules accessible to all tasks in the system. Consult a specification of the Intel processors for further information.

Global. Global Memory Objects. This is a “class” of memory object defined by the Memory Manager. Global memory objects are visible in *all* processes, at the same linear address offset in each process.

See also Common and Private.

I

ICB. Interrupt Control Block. A type of control block used by the CP/Q system kernel, to control the use of hardware interrupt levels. The format is defined in the SVC Handler manual.

IDT. Interrupt Descriptor Table. A table of descriptors, architected by Intel as part of the specification of the *x*86 processors. This table is used to determine the action taken by the processor when hardware interrupts occur, or INT instructions are executed. Consult a specification of the Intel processors for further information.

Interrupt Handler. A term for the code or task that is executed when an external interrupt is recognized, upon processor detected fault, or as a result of the software INT instruction.

The particular interrupt handler for a given interrupt is defined by a descriptor in the IDT.

IOPL. Input/Output Privilege Level. The minimum (that is, numerically largest) processor privilege level at which I/O may be performed. The use of instructions to perform I/O to devices is restricted by the *x*86 processors to code for which CPL ≤ IOPL. This means that many programs running in the system simply are not permitted to perform I/O. The IOPL is held in two bits within the FLAGS register, but only privileged code can successfully alter the IOPL. In CP/Q systems, IOPL is set to 0. Instructions that are considered I/O type instructions include the IN and OUT instructions as well as the ability to change the setting of the Interrupt Enable bit in the flags register.

K

Kernel. The CP/Q Kernel is the SVC Handler, Memory Manager and Resource Manager together with their associated data areas (the System Data Area).

L

LDT. Local Descriptor Table. A table of descriptors, architected by Intel as part of the specification of the *x*86 processor. There may be more than one LDT in a system; each task may have no LDT, its own private LDT, or may share an LDT with a number of tasks. This table is used to define the memory segments or code modules local to those tasks that have this table as their LDT. Items in an LDT are not accessible to tasks having a different LDT, unless special provision is made by to create a segment alias. Consult a specification of the Intel processors for further information.

CP/Q does not use LDTs, nor does it support their use.

N

NAP. The probe program required to enable the system level debugger SLEEP to load and debug a remote CP/Q system.

NDA. Nucleus Data Area. This is the fixed area at the start of the System Data Area that holds various items of information, such as the time and date, pointer to the current task, the system GDT, the system TSS, and so on, and also holds the pointers and chain headers for the other control blocks held in other parts of the System Data Area.

NMI. Non-Maskable Interrupt. Almost all processors provide facilities for the processor to be *interrupted* by external events, such as I/O operations completing. Such external interrupts can be disabled (or *masked*) by software running on the processor; for example it is common practice, while handling one interrupt, to prevent other interrupts from occurring.

The Intel *x*86 processors also have a special interrupt which is completely independent of the above mentioned external interrupt mechanism (it is raised by asserting a signal on a special pin of the processor), and which cannot be disabled within the processor. This is called a Non-Maskable Interrupt. An NMI can occur at any time, including while handling an I/O interrupt with interrupts disabled.

The NMI is used in IBM compatible systems for reporting memory parity failures. It can also be used by a debugger to cause an entry to the debugger whenever an NMI is raised (by use of an external push button switch). On some IBM PCs, NMI *can* be masked, through special I/O logic on the system board that is external to the processor.

O

OCB. Object Control Block. A type of control block used by the Memory Manager to record information about every memory “object” in the system.

P

PCB. Process Control Block. A type of control block used by the CP/Q Resource Manager to hold information relevant to a particular process. The format is defined in the Resource Manager manual.

PFD. Page Frame Descriptor. A type of control block used by the Memory Manager to maintain a chain of every allocatable page of physical memory in the system.

Private. Private Memory Objects. This is a “class” of memory object defined by the Memory Manager. Private memory objects are private to the address space of the process that owns them, that is, such objects are not visible in other processes.

See also Common and Global.

process. In CP/Q, a **process** is something that can acquire system resources, such as memory. Everything in the system (that is, all memory, tasks, SVT entries, and so on) belongs to some process. Each process is associated with an address space; conversely, all tasks within a given process share the same address space. Accounting is in general performed at the process level. Processes are arranged in a tree structured hierarchy, with the special process SYSTEM at the top of the tree. Apart from process SYSTEM, each process in the system also “belongs” to the process immediately higher in the tree (its “parent”).

A process is not a code module that can be run on the processor; the process probably contains one or more *tasks* which can be dispatched. The code of these tasks is also not part of the process; however, the memory occupied by that code is assigned to the process.

R

Resource Provider. A resource provider (RP) is any system extension or sub-system, such as a screen manager or file system, which provides a service (or “resource”) to client processes. Providing a service may involve the acquisition of memory or other system resources either directly by the RP or on behalf of the client (using the notion of effective process). The resource tracking interface to the Resource Manager allows the RP to respond to the removal of clients by recovering resources allocated to such clients or otherwise adjusting its internal state.

RPL. Requested Privilege Level. The Intel *x*86 processors have the concept of privilege levels. There are 4, numbered from 0 (most privileged) to 3 (least privileged). If a program is running at, say, privilege level 2, it is allowed to access items specified as being at privilege level 2 or 3, but not those at privilege level 0 or 1. Access to descriptors is performed by having appropriate selectors in segment registers; the privilege level associated with the attempted access is specified by the two low order bits of the segment registers, and is called the RPL.

RQE. Request Queue Element. A type of control block used by the CP/Q system kernel to form message (or request) queues. The format is defined in the SVC Handler manual.

S

SCB. Signal Control Block. A type of control block used by the SVC Handler to record the requested action for specific signals for a task.

SDA. System Data Area. This an area of memory that holds the control blocks, pointers and control variables used by the system kernel code to control the running of the CP/Q system. The SDA consists of a fixed header or anchor area, called the Nucleus Data Area or NDA, and other areas containing control blocks.

The format of the NDA is predefined, and is known at compile time. The areas containing the control blocks may be of variable size (initially set at system build time, but possibly changed when the system is running), and is at offsets that are set at system build time - that is, they are unknown when the system kernel code is compiled. The NDA contains pointers to the other variable areas, and also such items as the pointers to the start of the free block chains.

SLIB. Second Level Interrupt Block. A type of control block used by the CP/Q system kernel for forming lists of second level interrupt handlers. The format is defined in the SVC Handler manual.

SLIH. Second Level Interrupt Handler. A code module (currently a task in a CP/Q system) that is entered as a result of some action taken by a first level interrupt handler (FLIH). There may be more than one SLIH corresponding to a given hardware interrupt level.

SLEEP/R. System Loader and Error Environment Process/Remote A special program designed and written specifically to build and test CP/Q systems. It uses a probe program, called NAP, to remotely debug a system. This program has its own manual, to which the reader is referred for further information.

SVC. SuperVisor Call. This is the mechanism whereby a task requests services of the “system” or of other elements in the system, or performs services for other elements in the system. In a CP/Q system on an Intel processor, an SVC is made by an INT 0x7A instruction (for a privilege level 3 program), or an indirect call for privilege level 0 programs. In a CP/Q system on a RIOS processor, an SVC is made by an SVC instruction.

The CP/Q code module that is entered when an SVC is made is called the SVC Handler.

SVid. The SVid of a CP/Q entity is the name used to specify this entity to the SVC Handler. It consists of an SVT index (that is, the number of the SVT entry within the SVT) and an incarnation number.

SVT. SuperVisor Table. An array of elements, representing system objects known to the supervisor and containing sundry information concerning object type (task, semaphore, other), access permission, and various pointers to associated control blocks and queues.

T

Task. In CP/Q, a dispatchable program or code module is termed a **task**. The system kernel keeps details of each task in an associated Task Control Block and SVT entry.

The Intel processors have architected into them the notion of a task. The processor provides protection between tasks, and also provides facilities to swap tasks, both voluntarily by jump or call instructions, and involuntarily, on interrupts for example. The CP/Q system uses the protection mechanisms, but for performance reasons does not perform hardware task switches, except in special cases such as system abends, or other traps to SLEEP/R.

TCB. Task Control Block. A type of control block used by the CP/Q system kernel, to hold all the information relevant to a particular task. The format is defined in the SVC Handler manual.

TSS. Task State Segment. A type of processor control segment architected by Intel in the design of the *x*86 processors. In particular, it includes the register save area for a task.

In a CP/Q system, there is one TSS used by all tasks (the task change is performed by software within this TSS), and there are certain other TSS's set up by SLEEP which are used when starting CP/Q and also when the system traps back to the debugger.

X

XPT. Auxiliary Page Tables. Are used by the Memory Manager to manage virtual memory. For each active segment that has any virtual memory allocate, there is a pointer to an XPT Directory block.

Index

Artwork Definitions

id	File	Page	References
TXTQ	CPQSET	i	

Figures

id	File	Page	References
PROC1	NSLRMOVR	3-2	3-1
			3-2, 3-2
SVCFL2	RSLDTCOM	3-9	3-2
			3-8, 3-8
SVCFL3	RSLDTCOM	3-11	3-3
			3-11
GLIN	RSLMMGEN	3-14	3-4
			3-13
LINMEM	NSLMODEL	3-19	3-5
			3-17
LOADMEM	NSLMODEL	3-22	3-6
			3-21

Headings

id	File	Page	References
INTRO	RSLBINTR	1-1	Chapter 1, Introduction
			v
MM	RSLBINTR	1-3	Virtual and Real memory management
			1-1
PROG	LSLDPROG	2-1	Chapter 2, CP/Q Program APIs and Documentation
			v
CPQSS	RSLDMOD	3-1	Chapter 3, CP/Q System Structure
			v
MODGEN	RSLMMGEN	3-12	CP/Q Memory Model Overview
GACCH	RSLMMGEN	3-16	CP/Q Memory Protection and Access Hierarchy
MMOD	NSLMODEL	3-17	CP/Q Memory Model Specifics
IPRIV	NSLMODEL	3-22	Intel Privilege level terminology
ACCH	NSLMODEL	3-23	CP/Q Memory Protection and Access Hierarchy
MOWN	NSLMODEL	3-24	Memory Ownership
RAS	RSLDRAS	4-1	Chapter 4, RAS Considerations
			v, 3-5
DYNSYS	RSLDDYN	5-1	Chapter 5, Dynamic, Modular System Considerations
			v
INTF	RSLDDYN	5-4	Call vs. Inter-Task Interfaces
			3-5
FHCHAP	RSLDFLT	6-1	Chapter 6, Fault Handlers
			v

Index Entries

id	File	Page	References
ADA01H	LSLDPROG		
		2-1	(1) CP/Q
			(2) Application Programming Interfaces
ADA060A	RSLDDYN		
		5-1	(1) CP/Q
			(2) dynamic system considerations

Footnotes

id	File	Page	References	
FLIH	RSLBINTR			
		1-2	1	
				1-2
SHORT	RSLBINTR			
		1-6	2	
				1-5
CPQPROC	RSLBINTR			
		1-6	3	
				1-6
FN1	NSLRMOVR			
		3-2	1	
				3-2
FNORD	NSLMODEL			
		3-23	2	
				3-23
FNORD1	NSLMODEL			
		3-23	3	
				3-23

Spots

id	File	Page	References
RPLYMSG	RSLDTCOM		
		3-7	Messages

Processing Options

Runtime values:

Document fileid	LSLDESGD SCRIPT
Document type	USERDOC
Document style	IBMXAGD
Profile	EDFPRF40
Service Level	0032
SCRIPT/VS Release	4.0.0
Date	98.06.17
Time	14:01:16
Device	PSA
Number of Passes	3
Index	YES
SYSVAR G	INLINE
SYSVAR X	YES

Formatting values used:

Annotation	NO
Cross reference listing	YES
Cross reference head prefix only	NO
Dialog	LABEL
Duplex	YES
DVCF conditions file	(none)
DVCF value 1	(none)
DVCF value 2	(none)
DVCF value 3	(none)
DVCF value 4	(none)
DVCF value 5	(none)

DVCF value 6	(none)
DVCF value 7	(none)
DVCF value 8	(none)
DVCF value 9	(none)
Explode	NO
Figure list on new page	NO
Figure/table number separation	YES
Folio-by-chapter	YES
Head 0 body text	(none)
Head 1 body text	Chapter
Head 1 appendix text	Appendix
Hyphenation	NO
Justification	NO
Language	ENGL
Keyboard	395
Layout	OFF
Leader dots	YES
Master index	(none)
Partial TOC (maximum level)	4
Partial TOC (new page after)	INLINE
Print example id's	NO
Print cross reference page numbers	YES
Process value	(none)
Punctuation move characters	.,
Read cross-reference file	(none)
Running heading/footing rule	NONE
Show index entries	NO
Table of Contents (maximum level)	4
Table list on new page	YES
Title page (draft) alignment	RIGHT
Write cross-reference file	(none)

Imbed Trace

```
Page 0      CPQSET
Page 1      RSLBINTR
Page 1-6    LSLDPROG
Page 2-3    RSLDMOD
Page 3-1      NSLRMOVR
Page 3-5      RSLDTCOM
Page 3-12     RSLMMGEN
Page 3-17     NSLMODEL
Page 3-25   RSLDRAS
Page 4-3    RSLDDYN
Page 5-4      RSLDINTF
Page 5-5    RSLDFLT
Page 6-2    RSLGLOSS
```