1-2 BASIC CONCEPTS AND TERMINOLOGY

This section introduces the basic concepts of computer hardware structure, programming, and operating system terminology.

1-2.1 Computer Hardware Structure Terminology

Many different computer hardware configurations exist. Figure 1-1a depicts one possible configuration.

Instructions and data are stored in the *main storage*, historically called *core memory* (the terms *main storage*, *memory*, and *core* are often used interchangeably). Connected to the memory are various processors. A *processor* is a hardware device capable of interpreting instructions and performing the indicated operation (e.g., addition). Instructions as well as data are stored in the memory in coded form.

A computer system has one or more *Central Processing Units* (CPU). A *CPU* is a processor that manipulates and performs arithmetic operations upon data in the memory. It also executes instructions that control the other processors. For example, the CPU may execute a START I/O (SIO) instruction to initiate an I/O processor.

In many modern systems certain processors are specialized to perform specific tasks efficiently and/or inexpensively. For example, the *I/O* (Input and Output) *processors* are designed to control I/O devices and handle the movement of data between the memory and the I/O devices. These processors range from being specialized for a certain I/O device to being quite sophisticated, simultaneously executing several I/O instructions and controlling several I/O devices.

Almost any conceivable device from a laser to a milking machine can be (and has been) used as an I/O device. Many of the more common devices (card readers, printers, disks, drums, tapes, etc.) require control circuitry that is applicable to other devices. For reasons of economy, this common hardware is sometimes separated out into a device called a *control unit*.

There may be many user programs competing for the system resources, such as memory, CPUs, I/O processors, control units, devices, etc. What decides which users are to have access to the resources and how long a user may have a resource? What prevents the system from collapsing if a device becomes inoperable? What assigns the "path" to a device through the I/O processors and control units? What manages all of these resources? The answer to all of these questions is the operating system.

1-2.2 Programming Terminology

Figure 1-1a depicts the hardware of a computer system. In most installations there is an even larger dollar investment in software than in hardware. *Software* is the collection of programs or data that are used to perform certain tasks. A *program* is a

sequence of instructions. (Programs are sometimes loosely called "code"). Programs are placed into memory and interpreted (executed) by a processor. While not in use, programs are often stored in some secondary storage device, such as a disk, drum, or tape.

A system may have "prepackaged" programs available to users (routines to perform: square roots, searching, sorting, compilers, assemblers, translators). How are these prepackaged routines managed and who keeps track of where they are? Once again it is the operating system.

A lone user facing a formidable system such as the one shown in Figure 1-1a might become discouraged. Suppose this lone user is a Harvard student who has just learned how to program the CPU. He wants to execute a program to add 49 to numbers read from punched cards. How does he read cards? Must he also learn how to program the I/O processor? This is obviously too much to expect from a novice. Fortunately, the operating system usually provides routines for the I/O processors; in general, the operating system assists in the construction and execution of user programs.

The operating system provides an environment in which particular software packages can execute. An example is a compiler, which is a program that translates a *source program* (e.g., a FORTRAN program) in a high-level language to a corresponding machine language program, or *object program*. Most modern compilers interact closely with the operating system both during the compilation of source programs and the execution of the object code. During compilation the compiler needs the operating system to read the source program, print its listing, and punch its output decks. At execution time the object code produced by compilers such as PL/I requires a sophisticated *run-time environment*. ALLOCATE and FREE statements require the assistance of the operating system's memory management facilities. PL/I ON CONDITIONS need interrupt handlers provided by the operating system.

1-2.3 Operating System Terminology

A user submits his job to the operating system. A *user* is anybody that desires work to be done by a computer system. A *job* is the collection of activities needed to do the work required. A job may be divided into several steps (job steps). *Job steps* are units of work that must be done sequentially, for example, the three steps, compile, load, execute. Once the operating system accepts a user's job, it may create several processes. A *process* (or *task*) is a computation that may be done concurrently with other computations. Figure 1-2 depicts the relationship between user, job, process, and address space.

Another view of a process is the locus of points of a processor executing a collection of programs. The collection of programs and data that are accessed in a process forms an *address space*. Figure 1-2 depicts two sample address spaces—one for an I/O process, the other for a CPU process.

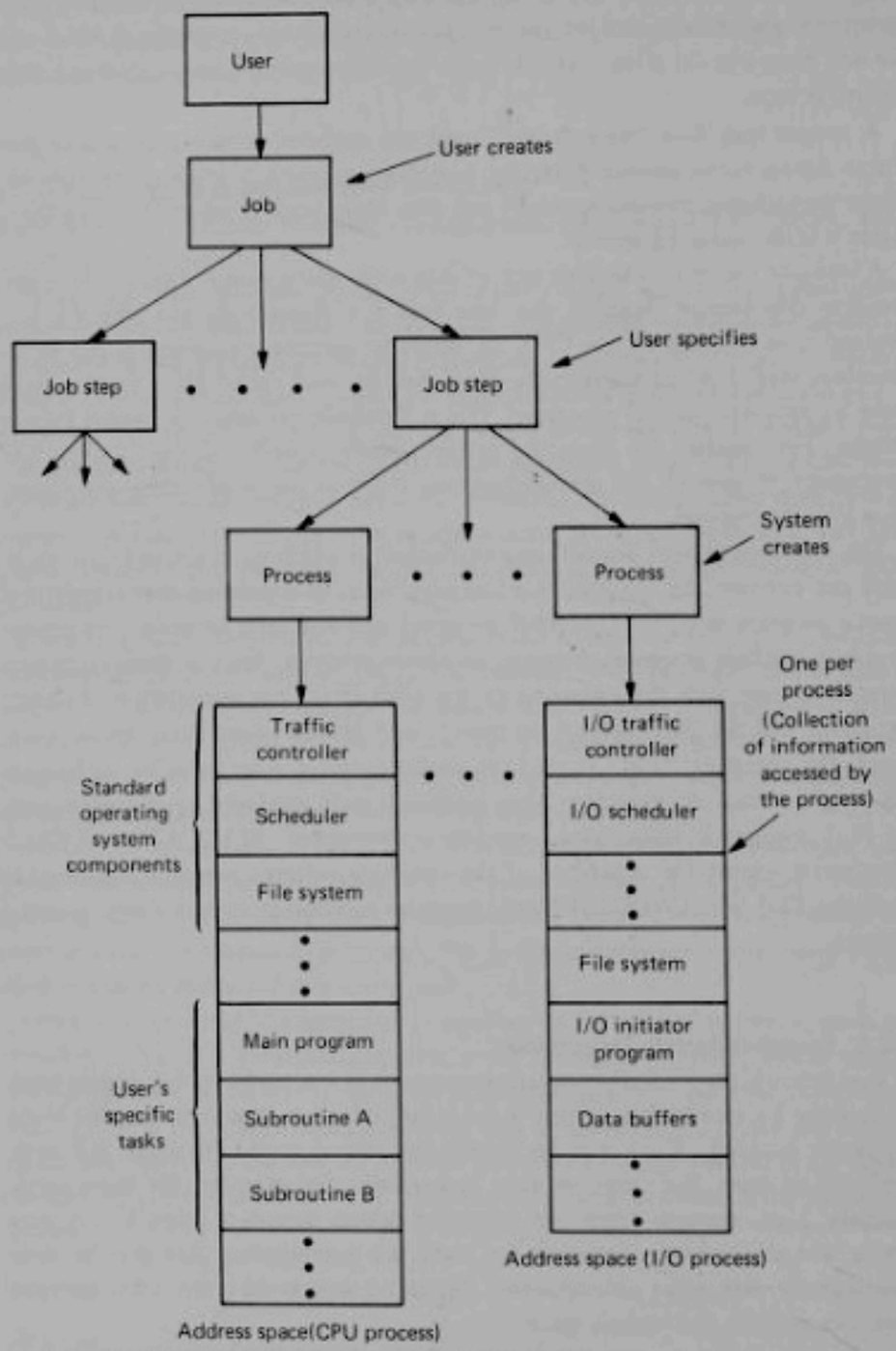


Figure 1-2 User, job, job step, process, and address space

Note that the actual code for the file system portion of the operating system may be the same for both address spaces. If code is to be shared in this manner, it must be pure, or locks must be set to prevent race conditions. *Pure code* (sometimes called *reentrant code*) is code that does not modify itself.

The operating system must map the address spaces of processes into physical memory. This task may be assisted by special hardware (e.g., a paged system) or it may be performed primarily by software (e.g., a swapping system).

Multiprogramming is a term given to a system that may have several processes in "states of execution" at the same time. A process is in a state of execution if the computation has been started but has not been completed or terminated (error completion). (Note: a process may be in a state of execution and not be executing; that is, some intermediate results have been computed but the processor is not currently working on this process.)

Another feature found in most contemporary computer systems is special hardware and instructions used by the operating system. These instructions, which are usually not available to the ordinary user, are commonly called *privileged instructions*. Most contemporary computers have at least two states of execution—a *problem state (user state, slave state)* and a *supervisor state (executive state, master state)*. In the supervisor state the processor can correctly execute privileged instructions. Obviously, one privileged instruction is to change the state of the machine. Others may start the I/O processors, change the protection rights of parts of memory, or change the interrupt status of the machine.

Protection hardware is often used to control access to parts of memory. For example, the operating system may deem certain parts of memory to be not writable. That is, a user program may be prohibited from altering the operating system programs.

Interrupt hardware allows the operating system to coordinate operations going on simultaneously; it also allows the nonsequential flow of a program. An *interrupt* is a mechanism by which a processor is forced to take note of an event (in much the same manner as you may be interrupted from reading this book by a relative or friend). Hardware may exist to mask certain interrupts. We have all "masked" out our mothers at times.

We should note here that many computer configurations do not have all these special instructions or hardware. (The IBM 1130 has no general masking facilities.) What limitations does the lack of these facilities place on the user or purchaser of such a system? On the other hand, many large systems have even more elaborate hardware facilities than those referred to here to assist the operating system, especially in the I/O processors. What does a user gain from these special features? This book will attempt to answer both these questions.

1-3 AN OPERATING SYSTEM RESOURCE MANAGER

We will present a framework for the study of operating systems based on the view that the operating system is a manager of resources. In this section we discuss these resources and define what the modules of the operating system must do to manage them; in Section 1-4 we present a model to demonstrate the sequencing of the management of these resources (process view); and in Section 1-5 we present a model to demonstrate the interrelationship and primitive functions of these modules (hierarchical and extended machine view).

These three views are not in conflict with one another; they represent three dimensions of one object—an operating system. Each view aids in the understanding and study of an operating system, and together they form the framework for this book. The models used are extensions of the work of others, most notably, Dijkstra (1968), Donovan (1972), Hansen (1970, 1971), Saltzer (1966), and Madnick (1969).

The primary view we take in this book is that the operating system is a collection of programs (algorithms) designed to manage the system's resources, namely, memory, processors, devices, and information (programs and data). All these resources are valuable, and it is the function of the operating system to see that they are used efficiently and to resolve conflicts arising from competition among the various users.

The operating system must keep track of the status of each resource, decide which process is to get the resource (how much and when), allocate it, and eventually reclaim it.

We distinguish between the resources of memory, processors, I/O devices (disks, tapes, teletypes) and information because of the inherent differences in their characteristics. We recognize that in a broad sense they are all devices and have many similarities. However, such abstractions do not appear to be useful at present for the development of relevant theory or techniques for the managing of the resources.

Viewing the operating system as a resource manager, each manager must do the following:

1 Keep track of the resources.
2 Enforce policy that determines who gets what, when, and how much.
3 Allocate the resource.
4 Reclaim the resource.

We have grouped all the programs of the operating system into four resource categories. On the following pages we have listed these categories together with their major functions and the typical names given to some of the routines that perform these functions.

1-3.1 Memory Management Functions

1 Keep track of the resource (memory). What parts are in use and by whom? What parts are not in use (called *free*)?
2 If multiprogramming, decide which process gets memory, when it gets it, and how much.
3 Allocate the resource (memory) when the processes request it and the policy of 2 above allows it.
4 Reclaim the resource (memory) when the process no longer needs it or has been terminated.

1-3.2 Processor Management Functions

1 Keep track of the resource (processors and the status of processes). The program that does this has been called the *traffic controller* (Saltzer, 1969).
2 Decide who will have a chance to use the processor; the *job scheduler* chooses from all the jobs submitted to the system and decides which one will be allowed into the system, i.e., have any resources assigned to it. If multiprogramming, decide which process gets the processor, when, and how much; this is called the *processor scheduler*.
3 Allocate the resource (processor) to a process by setting up necessary hardware registers; this is often called the *dispatcher*.
4 Reclaim resource (processor) when process relinquishes processor usage, terminates, or exceeds allowed amount of usage.

It should be pointed out that in making the decision as to which job gets into the system many factors must be considered. For example, a job that is requesting more memory or tape drives than the system has should not be allowed into the system at all. That is, it should not have any resource assigned to it. Why isn't the job scheduler a part of memory management or device management? Because historically the major consideration used in admitting a job to the system was the amount of processor time allocated to it. Furthermore, the record-keeping operations for job scheduling and processor scheduling are very similar. Therefore, we have chosen to place these modules in processor management.

1-3.3 Device Management Functions

1 Keep track of the resource (devices, channels, control units); this is typically called the *I/O traffic controller*.
2 Decide what is an efficient way to allocate the resource (device). If it is to be

shared, then decide who gets it, and how much he is to get; this is called *I/O scheduling*.

3 Allocate the resource (device) and initiate the I/O operation.

4 Reclaim resource. In most cases the I/O terminates automatically.

The method of deciding how devices are allocated depends on the flexibility of the device. Some devices cannot be shared (e.g., card readers) and must therefore be *dedicated* to a process. Others may be *shared* (e.g., disks); hence, there is more flexibility, and there are complications in the allocation of these devices. Others may be made into *virtual devices*. For example, the operation of punching on a card punch could be transformed into a "write" onto a disk (a virtual card punch), and at some later time a routine would copy the information onto a card punch. Virtualizing card reader, card punch, and printer devices is performed by *SPOOLing routines*. The virtual devices approach allows (1) dedicated devices to be shared, hence, more flexibility in scheduling these devices; (2) more flexibility in job scheduling; and (3) better match of speed of device and speed of requests for that device.

1-3.4 Information Management Functions

1 Keep track of the resource (information), its location, use, status, etc. These collective facilities are often called the *file system*.

2 Decide who gets use of the resources, enforce protection requirements, and provide accessing routines.

3 Allocate the resource (information), e.g., *open* a file.

4 Deallocate the resource, e.g., *close* a file.

1-3.5 Resource Management Summary

It is our contention that all the resource management functions of an operating system fall into the above four categories. We can use this framework to analyze existing operating systems, design new operating systems, and study theoretical work in context.

In Chapters 3, 4, 5, and 6, we will explore the management of these four resources in depth, examining operating systems from the bits to the broad overview. In Chapter 2 we will deal specifically with the bits.

Since our view of an operating system is based on the management of four separate resources—memory, processors, devices, and information—it is important to define the way these resources interrelate. Although, for simplicity, management of these resources can be viewed initially as being independent, there are subtle interdependencies that must be understood. Chapter 8 presents some of these.

The resources and the operating system's management of them are much like the wood, bricks, nails, etc. that go into building a house. There are various ways of "putting it together," all of which conform to acceptable standards; however, some ways are awkward and others elegant. Section 1-5 suggests one way of putting it together. A complete sample operating system based upon this view is described in Chapter 7. Besides serving as an example, that chapter implicitly presents some useful approaches to putting an operating system together, and does so on a level of detail not generally reached in other chapters. Except for the examples of computer structures in Chapter 2, this book is relatively machine-independent. Particular machine examples are used only to demonstrate concepts.

1-4 AN OPERATING SYSTEM—PROCESS VIEWPOINT (WHERE THESE RESOURCE MANAGERS ARE ACTIVATED)

In the last section we stated that an operating system consisted of a collection of programs to manage resources. What is the relationship among these programs? When in the life of a process does each of these programs come into play? To explain these interrelationships, and the sequencing between the resource manager modules, we present a view of the life of a process.

We will use an example to develop this view. Figure 1-3 depicts three processes (users' jobs) coexisting in a multiprogrammed system.

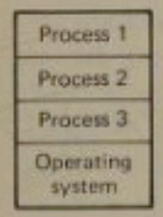


Figure 1-3 Multiple processes in a multiprogrammed system

The life cycle of a process can be represented by the transitions between the three states illustrated in Figure 1-4:

Run: The process has been assigned a processor and its programs are presently being executed.

Wait: The process is waiting for some event (e.g., an I/O operation to be completed).

Ready: The process is "ready to run," but there are more processes than processors and it must wait for its turn on a processor.

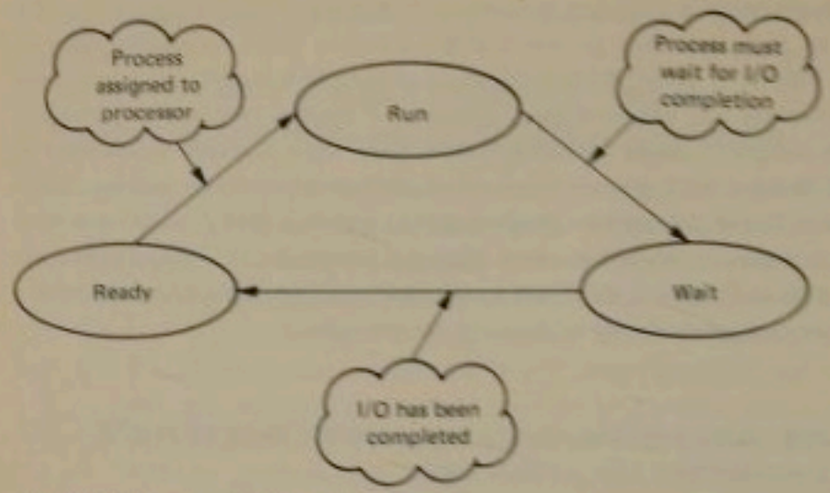


Figure 1-4 Simple states of a process

In Figure 1-4 sample reasons for the process state transitions are given; for example, if a "running" process must wait for I/O completion, it becomes a "waiting" process.

The process states of Figure 1-4 have been simplified. They assume that all the processes already existed in the system and that they will continue running forever. A more complete and realistic process life cycle is represented in Figure 1-5.

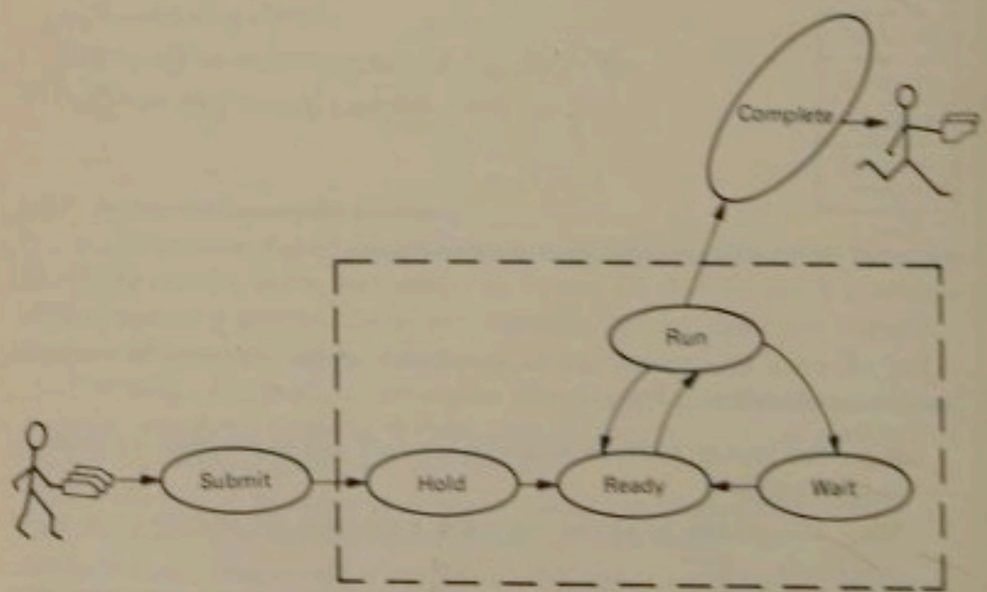


Figure 1-5 Model of process states

We have added three states:

Submit: A user submits a job to the system; the system must respond to the user's request.

Hold: The user's job has been converted into internal machine readable form, but no resources have been assigned to the process (the job has been SPOOLed onto disk). Resources must be allocated to move it to the ready state.

Complete: The process has completed its computation and all its assigned resources may be reclaimed.

In general, the operating system will take a process through all its states. These states and transitions are depicted in Figure 1-6, where the circles denote states and the clouds denote routines within the operating system that may be involved when a process switches from one state to another. There exists an identical diagram for each process in the system.

As Figure 1-6 shows, process management plays an important role in sequencing a process through its states. Let us go through an example using Figure 1-6. A user submits his job to the system, perhaps by placing his deck into a card reader (submit state). The job consists of several decks of programs preceded by job control cards. The job control cards pass information to the operating system as to what resources the job will need. The SPOOLing routine reads the job and places it onto a disk (*hold state*). To find storage space for the machine readable copy of the user's job, the SPOOLing routine must call information management, which keeps track of all information and available space in the system.

At some later time, the job-scheduling routine scans all the SPOOLed files on the disk and picks a job to be admitted into the system. In picking a job, the scheduler will call memory management to determine if sufficient main memory is available, and call device management to determine whether devices requested by the job are available. The job scheduler determines the device requirements from the user's job cards.

Once the job scheduler decides that the job is to be assigned resources, the traffic controller is called to create the associated process information, and memory management is called to allocate the necessary main storage. The job is then loaded into memory and the process is ready to run (*ready state*).

When a processor becomes free, the process scheduler scans the list of ready processes, chooses a process, and assigns a processor to it (*running state*).

If the running process requests access to information (e.g., read a file), information management would call device management to initiate the reading of the file. Device management would initiate the I/O operation and then call process management to indicate that the process requesting the file is awaiting the completion of the I/O operation (*wait state*).

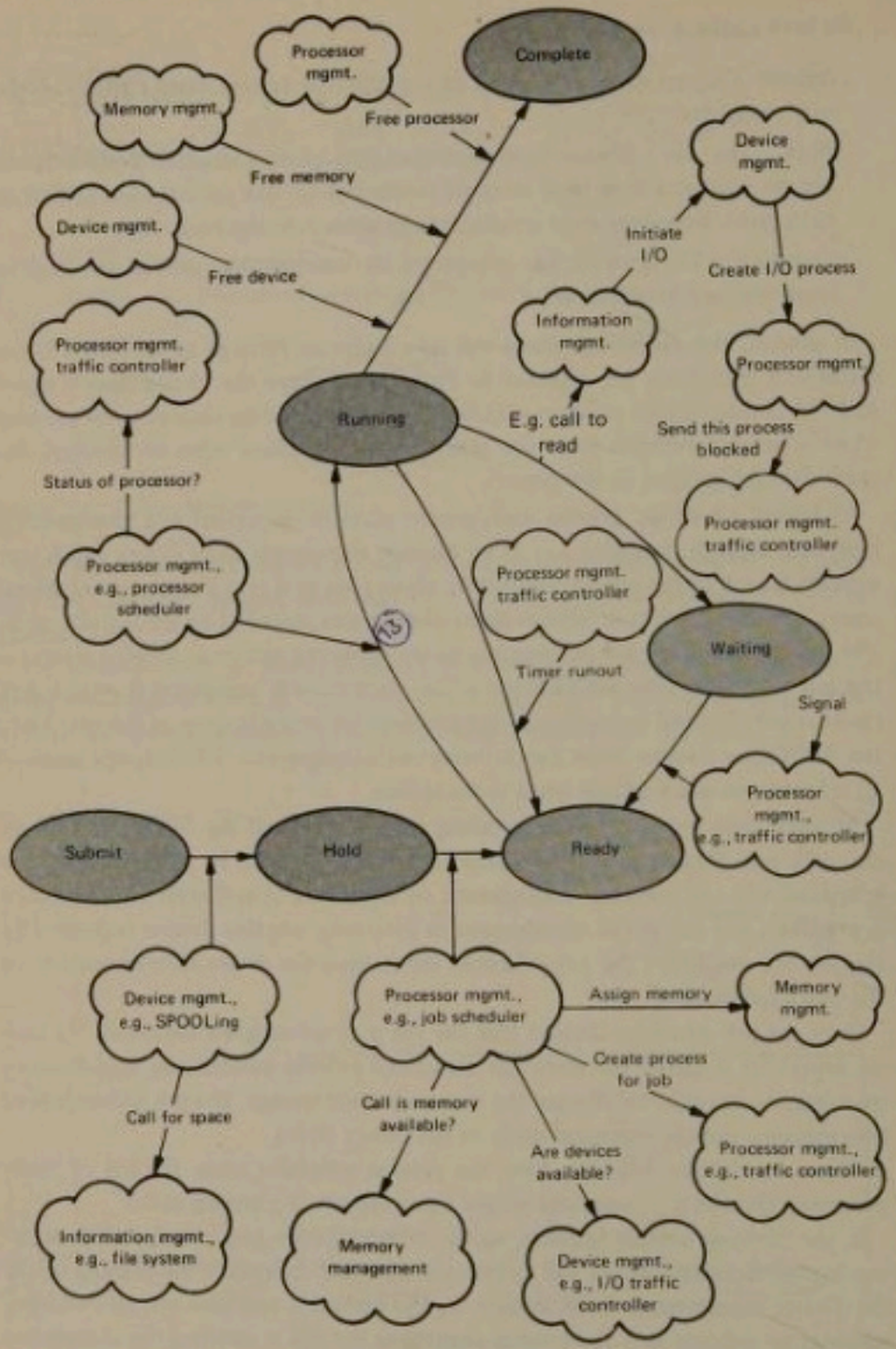


Figure 1-6 Operating system's handling of process state transitions

When the I/O is completed the I/O hardware sends a signal to the traffic controller in process management, which places the process back into the ready state.

If the process should complete its computation when it is run again, then it is placed into *completed state* and all allocated resources are freed.

The process states of Figure 1-6 represent one possible view of the life cycle of a process. In a more complex operating system it may be desirable to define even more states (see Chapter 4). On the other hand, if an operating system were designed without the process concept in mind, it might be difficult for an observer to determine clearly the state of a process at any given time.

1-5 OPERATING SYSTEMS—HIERARCHICAL AND EXTENDED MACHINE VIEW

In the previous sections we noted the need to *identify* the system resources that must be managed by the operating system and, using the process view, we indicated *when* these resource managers come into play. We must now answer the question, "*How* are these resource managers called into play, and *where* do they logically reside in regard to each other?" Does the scheduler call upon the services of memory management? Does memory management ever call the scheduler? Is the process concept only for the user or is it employed by the operating system also? This section presents a hierarchical view of an operating system to show how the modules relate. We realize that there is no absolutely right or wrong way to organize these modules. We feel, however, that the hierarchical approach results in a simple, clean structure that is easy to analyze and implement.

1-5.1 Extended Machine Concept

In Section 1-2.1 we introduced the basic hardware features of contemporary computers. This was the view of a *bare machine*—a computer without its software clothing. A bare machine is not the environment desired by most programmers. You might wish to write a program, such as:

```
1  MOVE C,B                 SET C = B
2  FIND-SPACE 80,X          FIND 80 BYTES OF "FREE" SPACE AND SET
                              X = LOCATION
3  READ-CARD X              READ A CARD INTO AREA X
4  COMPARE X(2),'/*'        ARE THE FIRST TWO BYTES OF AREA X A "/*"?
5  TRANSFER-MATCH END       IF SO, TRANSFER TO END
```

Statements 1, 4, and 5 correspond closely to typical computer instructions, as on a System/370. Statements 2 and 3, on the other hand, may require tens, hundreds, or even thousands of instructions to be correctly and efficiently accomplished on

a modern computer since they involve interaction with some of the key system resources, such as memory and I/O.

The instructions to perform these kinds of resource management functions are usually provided by the operating system. The user program can request these services by issuing special *supervisor call* instructions that act much like subroutine calls but transfer control to the operating system rather than to one of the user's subroutines. Thus as far as the user is concerned, statements 2 and 3 are legal instructions for the extended machine even though they do not correspond to instructions of the bare machine. The operating system provides many "instructions" in addition to the basic hardware instructions. The sum of these instructions is called the instruction set of the *extended machine*. This situation is graphically represented in Figure 1-7. The kernel of the operating system runs on the bare machine; user programs run on the extended machine.

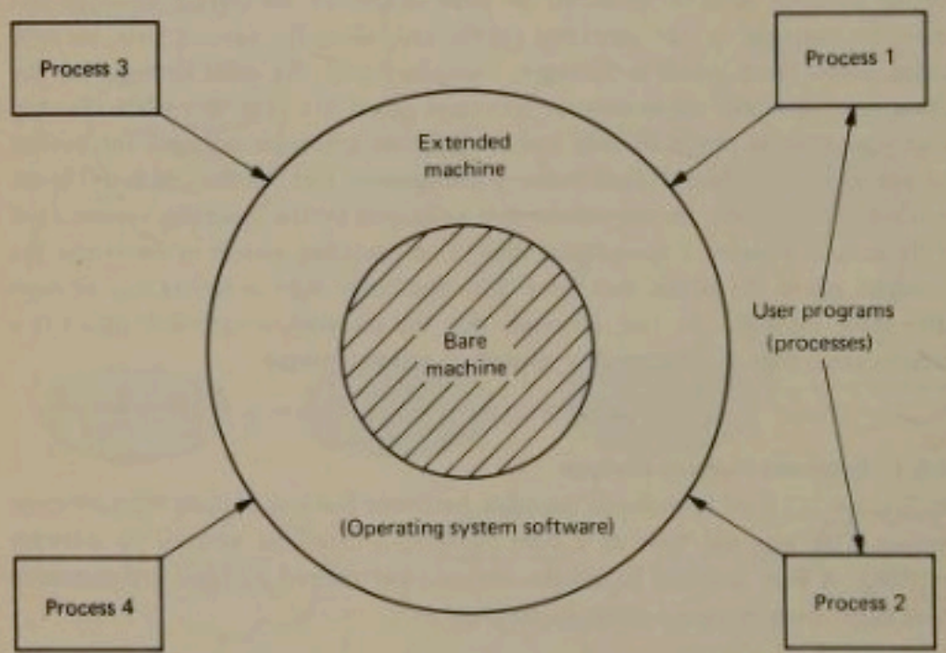


Figure 1-7 Extended machine

1-5.2 Hierarchical Machine Concept

We still have to explain how the operating system is put together. Most of the early operating systems consisted simply of one big program. As systems became larger and more comprehensive, this "brute force" approach became unmanageable. Eventually, it became clear that the extended machine approach could be applied to the operating system in two ways (see Figure 1-8): (1) key functions needed by many system modules could be separated into an "inner extended machine," and (2)

certain modules could be separated out and run on the extended machine in essentially the same way as user processes.

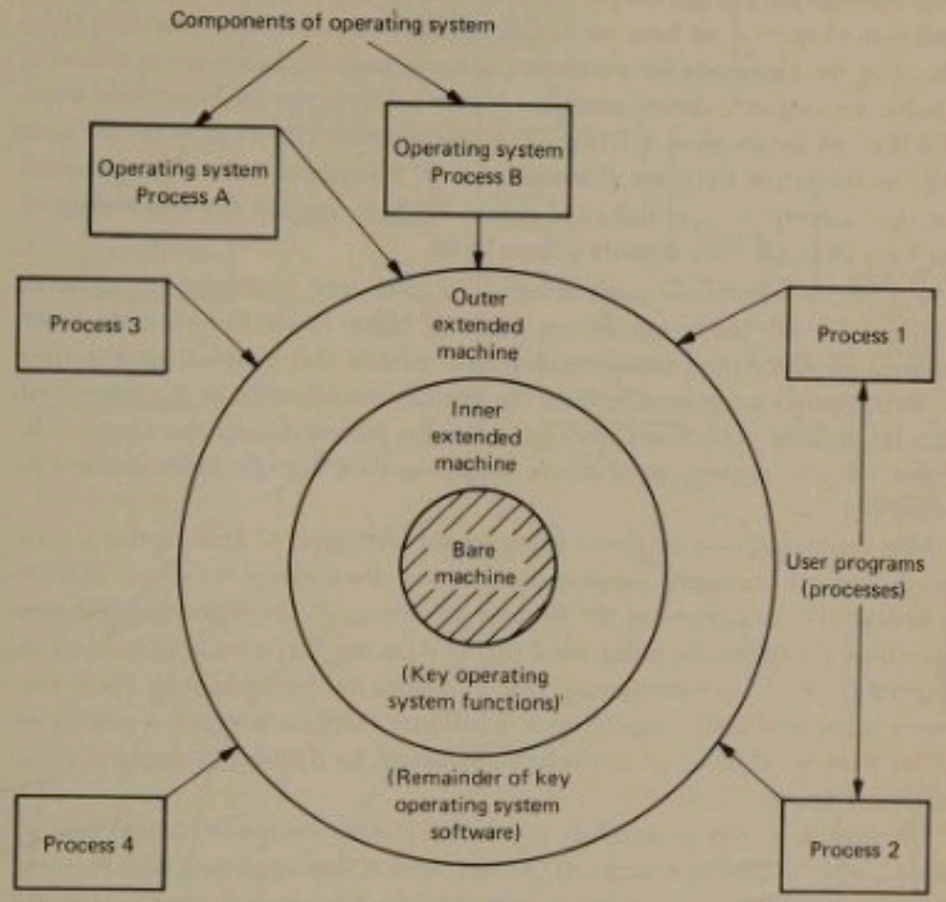


Figure 1-8 Simple hierarchical machine

In order to use the hierarchical approach, we must answer the original question: *Where* does each module of the operating system belong? In the inner extended machine, the outer extended machine, or as a process? Furthermore, the inner/outer extended machine concept can be generalized into *levels of extended machines* and the operating system processes can interrelate and be generalized into *layers of processes*. All those modules of the system that reside in the extended machine, as opposed to those that operate as process layers, are collectively called the *kernel* of the operating system. There is, at present, no firm rule to indicate how many levels should be used, what modules should go into which levels, what should be in the kernel, etc.

Two of the earliest designs using the hierarchical concepts were the T.H.E. operating system developed by Dijkstra (1968) and the modular file system design of

Madnick (1969). Similar strategies are now being employed in the design of complex application programs under such names as *modular programming* or *structured programming*. Both in this section and in our design of a complete sample operating system in Chapter 7, we have selected intrinsically clear and basic functions to be placed in the kernel and have separated as many tasks of the operating system as possible into separate system processes. Figure 1-9 illustrates the hierarchical structure that will be discussed in this book. All the processes (shown as boxes in Figure 1-9) use the kernel and share all the resources of the system. Some of the processes are the "parent" or "controller" of others. We have denoted this relationship by the wavy lines indicating separate process layers.

In a *strictly hierarchical* implementation, a given level is allowed to call upon services of lower levels, but not on those of higher levels. For example, if we designed an interprocess message management module that must call upon services of the memory management module, the memory module must be in a lower level than the message module and must not call upon process message management. In Figure 1-9, this requirement is shown by placing these modules in levels 2 and 3, respectively.

What function should be placed in the lowest level, level 1? The one that is used by all resource managers—keeping track of the allocation of resources. This requires the synchronization of the resource allocation. The corresponding primitive operations are frequently called the *P operator* (noting that a resource is seized or requested) and *V operator* (noting that a resource has been released). These synchronization primitives operate upon a software mechanism called a *semaphore* (from Webster's definition: "a telegraphic apparatus for signaling by means of arms, lanterns, flags, etc.").

The semaphore may be as simple as a switch (*binary semaphore*) or may have an integer value (*counting semaphore*). A semaphore is associated with each resource in the system. When a resource is requested, the P operator is used to test the corresponding semaphore; if it is off, P turns it on and returns. If the semaphore is already on, then P makes a note that the requesting process has been placed in WAIT state waiting for the resource associated with that semaphore. Later, a V operator issued by another process releases the resource, turns the semaphore off, and awakens any processes waiting for that resource.

In Chapter 4 the process synchronization operations will be more formally explained; a sample implementation is presented in Chapter 7. It is important to note that since most modules of the operating system need to use P's and V's, these functions are placed in the core of the kernel (level 1).

Examples of the primitive functions in the various levels of the kernel are:

Level 1: Processor Management Lower Level
The P synchronization primitive

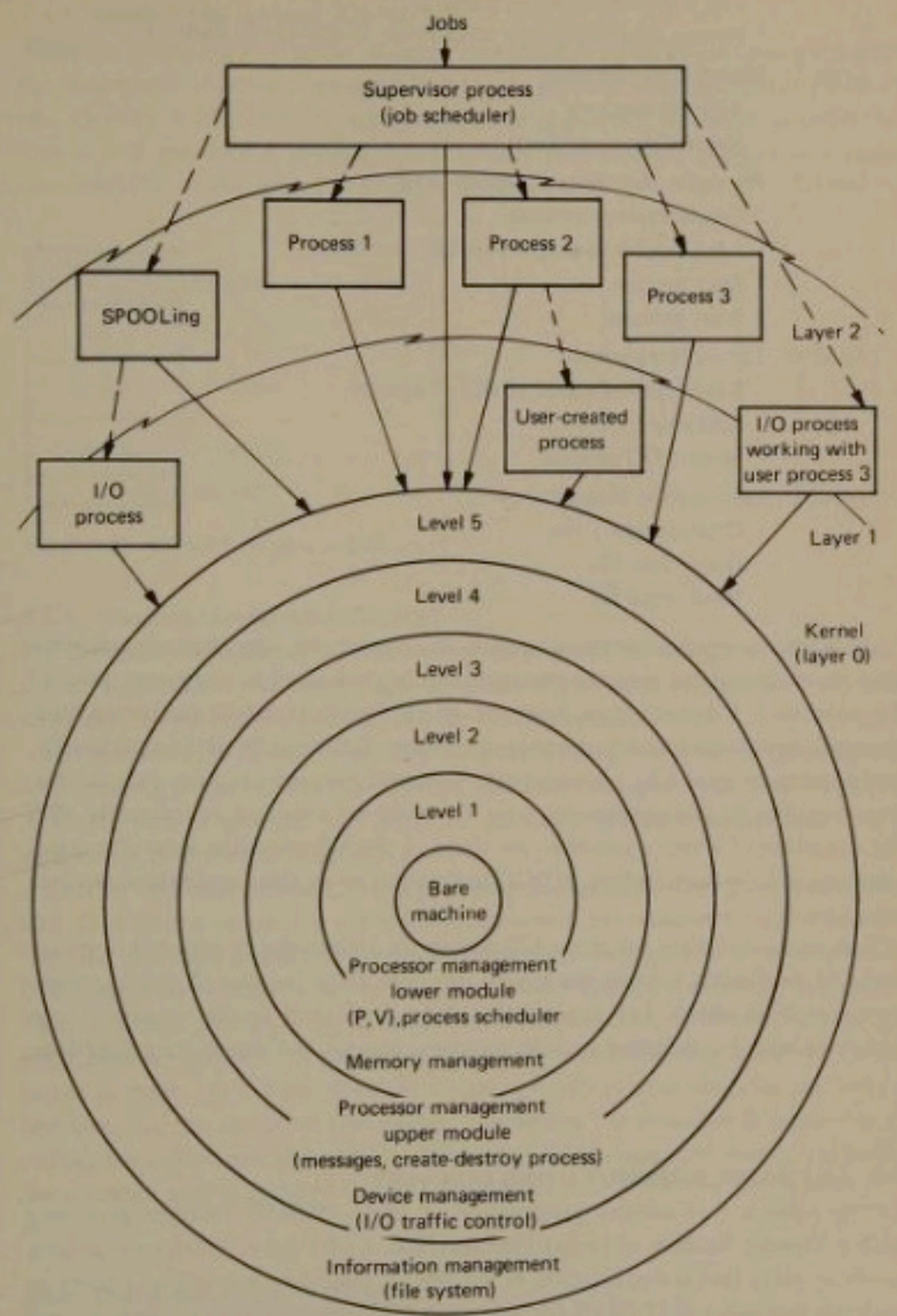


Figure 1-9 Hierarchical operating system structure

Memory Management

2

The memory management modules of an operating system are concerned with the management of primary memory. By *primary memory* we mean the memory that the processors directly access for instructions and data (see Figure 1-1). Primary memory is frequently called *core* memory in historical reference to the magnetic ferrite core technology that was used for many years. Specifically, memory management is concerned with four functions:

1. Keeping track of the status of each location of primary memory, i.e., each location is either allocated or unallocated ("free").
2. Determining allocation policy for memory, i.e., deciding to whom it should go, how much, when, and where. If primary memory is to be shared by one or more processes concurrently, then memory management must determine which process' requests will be satisfied.
3. Allocation technique—once it is decided to allocate memory, the specific locations must be selected and allocation information updated.
4. Deallocation technique and policy—handling the deallocation (reclamation) of memory. A process may explicitly release previously allocated memory, or memory management may unilaterally reclaim the memory based on a deallocation policy. After deallocation, status information must be updated.

The policies chosen for memory management may be influenced by the desire to keep memory management modules small and simple, or by the need to increase user flexibility or system efficiency. In this chapter we present a spectrum of policies and techniques; some of these make possible a greater utilization of memory, while others provide the user with more flexibility—though sometimes at the cost of greater complexity, higher hardware costs, or increased overhead. The techniques we describe are:

1 Single contiguous memory management
2 Partitioned memory management
3 Relocation partitioned memory management
4 Paged memory management
5 Demand-paged memory management
6 Segmented memory management
7 Segmented and demand-paged memory management
8 Other memory management schemes

Recalling our hierarchical extended machine viewpoint (Figure 1-9), we note that the memory management modules are usually neither on the highest level of an operating system (i.e., they are usually called by other levels) nor on the lowest. However, we have chosen to discuss memory management first, as it is presently a key distinguishing characteristic of most operating systems. We see that a clearly distinguishing feature between the SDS940 and the OS360/MVT operating systems is memory management (paged versus partitioned).

In this chapter we emphasize the role of memory management in relation to job processing. However, within a computing system there may be a variety of different requirements that relate to memory management. In some cases, these may involve wholesaler versus retailer relationships; in others, different algorithms may be chosen merely because they are more efficient or more appropriate for certain situations.

The particular techniques presented in this chapter cover most memory management applications. For example, in a particular operating system/user environment, the job processing memory management may be based on a demand-paged allocation strategy. The user's processes then may suballocate assigned memory into individual data areas based on a partitioned allocation technique. At the same time, the operating system may use a relocatable partition approach in handling its own dynamic system tables. For these reasons, an operating system may provide a multiplicity of memory management facilities that can be used for different purposes.

To keep the analysis as simple as possible, we will direct our attention to the job processing memory management requirements. The other cases can be analyzed in a similar manner. The sample operating system in Chapter 7 illustrates some of these other cases.

In order to illustrate the impact of memory management on system efficiency, the concept of multiprogramming is introduced in this chapter. A more extensive treatment of multiprogramming is found in Chapter 4.

For each of the memory management policies and techniques to be studied, the analysis is divided into four sections:

1 An overview of the approach and concepts employed
2 A description of any special hardware facilities required or recommended
3 A description of the particular software algorithms and processing required
4 A discussion of the advantages and disadvantages of the particular strategy

3-1 SINGLE CONTIGUOUS ALLOCATION

Single contiguous allocation is a simple memory management scheme that requires no special hardware features. It is usually associated with small stand-alone computers (or minicomputers) with simple batch operating systems, e.g., IBM OS/360 Primary Control Program, IBM 1130 Disk Monitor System, IBM 7094 Fortran Monitor System. In such systems there is no multiprogramming, and a one-to-one correspondence exists between a user, a job, a job step, and a process. Thus the terms user, job, or process may be used interchangeably. In this chapter, for the sake of simplicity, we will always assume the one-to-one correspondence: only the term "job" will be used. Memory is allocated to the job, as depicted in Figure 3-1.

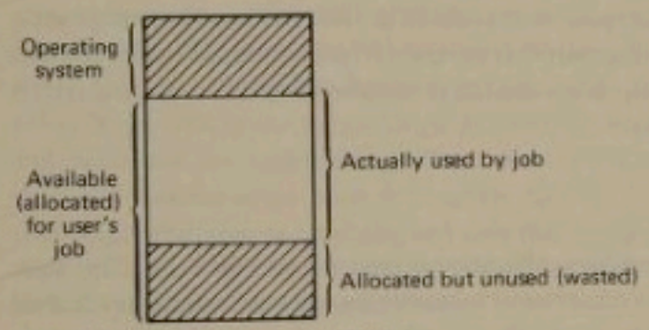


Figure 3-1 Single contiguous allocation

Memory is conceptually divided into three contiguous regions. A portion of memory is permanently allocated to the operating system. All of the remainder of memory is available (and allocated) to the single job being processed. The job actually uses only a portion of the allocated memory, leaving an allocated but unused region of memory. For example, if there are 256K bytes of memory, a simple operating system may require 32K bytes, leaving 224K bytes allocated for user jobs. If a typical job requires only 64K bytes, then 160K bytes of memory (over 50 percent of all memory) are unused. This unused memory cannot necessarily be returned to the manufacturer since there may be a few jobs that require having the entire 224K bytes available.

If we examine how the four functions of memory management are handled, we see the simplicity and hence the principal advantage of this scheme:

1 Keeping track of memory—it is allocated entirely to one job.
2 Determining factor on memory policy—the job gets all memory when scheduled.
3 Allocation of memory—all of it is allocated to the job.
4 Deallocation of memory—when the job is done, all memory is returned to free status.

3-1.1 Hardware Support

No special hardware is required for contiguous allocation. Sometimes a primitive hardware protection mechanism is desirable to ensure that the user's programs do not accidentally or maliciously tamper with the operating system. This mechanism may consist of a bounds register and a supervisor-user mode of the CPU. The bounds register contains the address of the protected area (which includes the operating system). If the CPU is in user mode, on each reference to memory the hardware checks to assure that it is not an access to the protected area. If an attempted access is made, an interrupt occurs and control is transferred to the operating system. In supervisor mode the operating system can access the protected area as well as execute privileged instructions that change the contents of the bounds register. Other techniques, such as the IBM 370's locks-and-keys protection mechanism, could be used to accomplish the same effect. The mode can be changed from user to supervisor only when control is transferred to the operating system (usually due to an interrupt).

3-1.2 Software Support

A flowchart of a single contiguous allocation is depicted in Figure 3-2. This algorithm is called when the job scheduler of processor management wishes to schedule a job to be run. The algorithm is called only when no other job is using memory.

3-1.3 Advantages

The major advantage of this scheme lies in its simplicity: an operating system using this scheme may require as little as 1K bytes in entirety in contrast to the 256K bytes or more needed by more sophisticated operating systems. Another advantage is that it does not require great expertise to understand or use such a system.

3-1.4 Disadvantages

A major disadvantage of this scheme lies in the fact that memory is not fully utilized. We note three inefficiencies. First, some memory is not being used at all (see the wasted area in Figure 3-1). Second, on most systems with channels, there are times when no portions of memory are actively being used by the CPU at all,

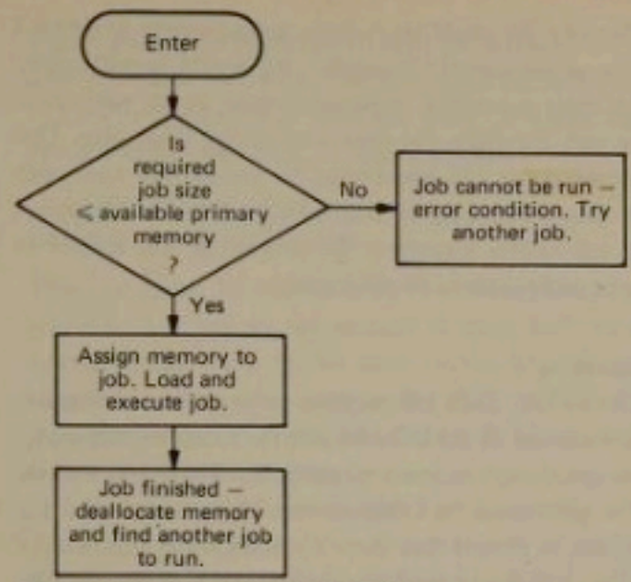


Figure 3-2 Single contiguous allocation

as when the job has initiated an I/O request and is waiting for the channel to complete its operations. The memory containing the user's CPU programs is not being utilized while CPU is waiting. What percent of the time is a CPU waiting for I/O? Every program is different, but a typical job on the IBM 360/65 may be waiting 65 or 70 percent of the time if single allocation is used. Therefore, both memory and processors are underutilized. Third, some portions of memory that contain the user's address space have information that is never accessed. For example, among the collection of programs and data that comprise the job, there exist routines, such as error routines, that will often not be executed. Why must these be loaded when they take up memory space?

Another disadvantage is the lack of flexibility that this scheme offers the user. For example, his address space (total program size) must be smaller than main memory, or else jobs cannot be run.

In summary, the disadvantages are: (1) Poor utilization of memory, (2) poor utilization of processors (waiting for I/O), and (3) user's job being limited to the size of available main memory (with the exception perhaps of an overlay mechanism).

3-2 INTRODUCTION TO MULTIPROGRAMMING

Many of the inefficiencies associated with single contiguous memory allocation stem from the problem of matching a fixed set of available resources to a varying resource demand. The physical hardware resources of a computer system can be varied only over relatively long time periods—for example, by adding or removing an increment (say 64K) of memory. These physical changes usually are made only once or twice a year. At the same time, the resource requirements of the different

jobs to be run may vary significantly. We could try to force programmers to develop all jobs with identical resource requirements. However, this would be extremely difficult, if not impossible. A more successful approach would be to operate on more than one job at a time and distribute the resources among these jobs. This technique, called *multiprogramming*, is the interleaved or concurrent execution of two or more processes. Multiprogramming is examined in detail in Chapter 4 (Processor Management). We will briefly introduce the concept in this section to clarify the memory management techniques to be discussed.

3-2.1 Example of Multiprogramming

Figure 3-3 depicts three jobs to be run. Each job requires some processor time for its computation, C_i, as well as some use of the I/O channels for its input/output, I_i. In addition, each job's address space requires some memory, M_i. For simplicity, we have modeled each job as if it performed its computation, then did its I/O; this cycle is repeated continuously. Let us assume that there are 100K bytes of available memory. If only Job 1 were run, 30K bytes would be used, and the remaining 70K would be wasted. Furthermore, $\frac{I_1}{C_1 + I_1}$ of the processor time would be spent waiting for input/output.

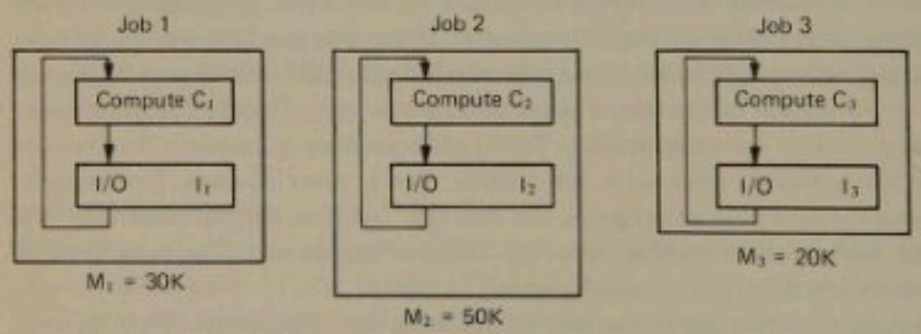


Figure 3-3 Models of three jobs to be multiprogrammed

Alternatively, we could place the address spaces for Jobs 1, 2, and 3 in main memory at the same time (30K + 50K + 20K = 100K bytes). This would fully utilize the 100K-byte main memory. Processor management would assign the processor to Job 1. After performing computation C_1, instead of stopping the processor while waiting for I_1 completion, processor management could assign the processor to Job 2. Likewise, when Job 2 reached I_2, the processor could be reassigned to Job 3. When Job 3 reaches I_3, we could immediately assign the processor again to Job 1 if I_1 had been completed (i.e., $I_1 \leqslant C_2 + C_3$). Thus, the processor would wait only $I_1 - (C_2 + C_3)$ time units instead of I_1. Through the use of multiprogramming, the processor utilization is much higher than when one job is run at a time. In many cases it is possible to complete two or three jobs in almost the same amount of time required to complete one uniprogrammed job.

3-2.2 Measures of System I/O Wait Percentage

Real processes are not as simple as the models in Figure 3-3. The computation and I/O are intermingled in complex ways. A useful aggregate parameter is the job's percent of I/O wait time, ω, which equals

$$\frac{\text{total I/O wait time}}{\text{total I/O plus total CPU time}}$$

This can easily be measured on most computers by running the process uniprogrammed and noting the percent of time that the processor is in the wait state. Several studies have shown that, on the average, typical jobs run on medium-to large-scale computers, such as the IBM 360/65 or 370/158, have an I/O wait time percentage of about 65 percent. Of course, individual jobs may deviate considerably from this average.

If we have two jobs, each with ω = 50 percent, and they are multiprogrammed, is the *effective system I/O wait percentage*, ω', reduced to 0? Well, if the model of Figure 3-3 held, we would have $C_1 = I_1 = C_2 = I_2$. Thus, C_2 could be executed concurrently with I_1 and likewise C_1 during I_2. However, as noted above, this model is oversimplified. Figure 3-4, for example, illustrates the processes as intermixed periods of computation and I/O. Each period, C_{11}, I_{12}, C_{13}, etc., is assumed to be of equal length in this example. Note that each job has an I/O wait percentage of 50 percent. If we tried to multiprogram them, we could match I_{12} with C_{21}, and I_{22} with C_{13}. However, when Job 1 wishes to perform input/output at I_{14}, we find that Job 2 also needs input/output at I_{23}. Thus the processor is idle while waiting for I_{14} or I_{23} to complete.

Generalizing the model of Figure 3-4, we can use probability theory to estimate

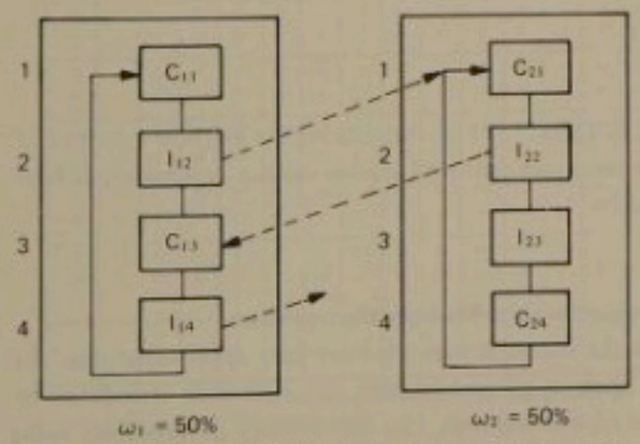


Figure 3-4 More detailed model of computation and I/O within a process

3-3 PARTITIONED ALLOCATION

Partitioned allocation, in its various forms, is one of the simplest memory management techniques for supporting multiprogramming. Main memory is divided into separate *memory regions* or *memory partitions*. Each partition holds a separate job's address space, as illustrated in Figure 3-5.

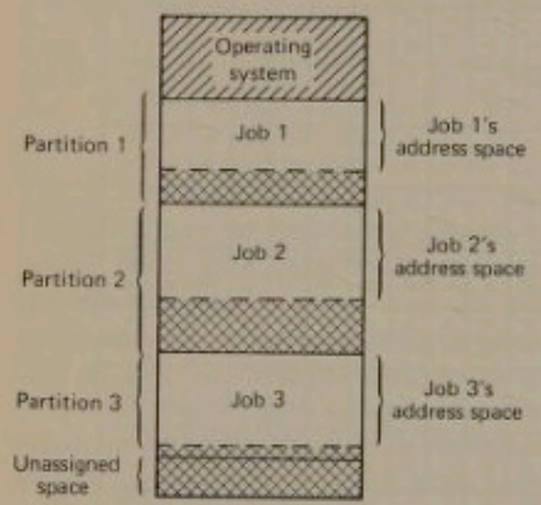


Figure 3-5 Partitioned allocation

The four functions of memory management are easily handled by this technique as follows:

1 Keeping track of the status of each partition (e.g., "in use" or "not in use," size).
2 Determining who gets memory—this is handled largely by the job scheduler.
3 Allocation—an available partition of sufficient size is assigned.
4 Deallocation—when the job terminates, the partition is indicated "not in use" and is available for future allocation.

Details of these functions are presented in the software support section.

3-3.1 Hardware Support

Very little special hardware is needed to support partitioned allocation. A memory protection mechanism is desirable to prevent one job from disrupting either the operating system or other jobs, whether accidently or maliciously. There are many ways to do this. We could use two bounds registers to bracket the partition being

used. If the job tried to access memory outside the partition, a protection interrupt would occur. There are two disadvantages to this scheme:

1 The bounds registers must be changed every time that the processor is reassigned for multiprogramming.
2 It is difficult to extend this protection to the I/O channel. A job in partition 1 may try to instruct the channel to read data into an area in partition 3. We could provide bounds registers for every I/O channel, with multiple registers required for multiplexor channels. Alternatively, the operating system that initiates all I/O operations could try to check the channel program by software, although this would be a time-consuming burden.

The 370 protection mechanism handles these requirements fairly well. Each partition is assigned a separate protection key (1 through 15); key 0 is reserved for the operating system. Since the hardware actually associates the keys with each 2K-byte block of memory, partitions must be multiples of 2K, and all locks within a partition are set to the same value. Then, whenever the processor is reassigned, the "key" field within the new Program Status Word is set to the corresponding partition's key. Similarly, the I/O channels also use the protection key mechanism.

3-3.2 Software Algorithm

The partitioned allocation approach, like most of the memory management techniques to be presented, has numerous variations. Two common versions are *static partition specification* and *dynamic partition specification*.

3-3.2.1 STATIC PARTITION SPECIFICATION

By *static specification*, we mean that memory is divided into partitions prior to the processing of any jobs. This is similar to the technique used in IBM's OS/360 MFT (Multiprogramming with a Fixed number of Tasks). The partition specification may be designated by the computer operator or it may be built into the operating system. An example of a static partition specification table is given in Figure 3-6.

Partition Number (protection key)	Size	Location	Status
1	8K	312K	IN USE
2	32K	320K	IN USE
3	32K	352K	NOT IN USE
4	120K	384K	NOT IN USE
5	520K	504K	IN USE

Figure 3-6 Static partition specification table

Each job step supplied by a user must specify the maximum amount of memory needed. A partition of sufficient size is then found and assigned. The software algorithm to handle the allocation and deallocation is a simple one.

The technique of static specification is especially appropriate when the sizes and frequency of jobs are well known. In such a case, the partition sizes are chosen to correspond closely to the most common job sizes. However, there can be considerable memory wasted if the sizes and frequencies of jobs are not known, or if they are diverse. For example, if many jobs of sizes 1K, 9K, 33K, and 121K are to be run, we could assign these to partitions as follows:

PARTITION	PARTITION SIZE	JOB SIZE	WASTED SPACE
1	8K	1K	7K
2	32K	9K	23K
3	32K	9K	23K
4	120K	33K	87K
5	520K	121K	399K
	712K	173K	539K

In this case, all the partitions are assigned in the best possible way, yet only 173K of the available 712K is actually used. Thus, over 75 percent of the available memory is wasted. This is, of course, an extreme case, but similar situations can and do occur in actual systems.

3-3.2.2 DYNAMIC PARTITION SPECIFICATION

By dynamic specification, we mean that partitions are created during job processing so as to match partition sizes to job sizes. Dynamic partition allocation schemes are used in many instances that do not involve the operating system. The techniques have been called partitioned memory, dynamic storage allocation, controlled storage, and garbage collection. (For example, when the PL/I ALLOCATE statement is executed, a new storage area must be allocated; conversely, when the FREE statement is executed, the storage is released. This type of activity occurs within the partition and need not concern the operating system.)

Numerous schemes exist for accomplishing dynamic partition allocation. Tables must be made up with entries for each free area and each allocated partition, specifying the size, location, and access restrictions to each partition.

Figure 3-7 depicts a possible table organization for accomplishing these purposes. Two separate tables are used, one for the allocated areas and the other for maintaining the status of the unallocated, or free, areas. Since the tables are not necessarily completely filled, the status fields indicate whether the corresponding entry is currently in use (containing allocated or available area information) or not in use (an "empty entry").

Partition Number (protection key)	Size	Location	Status
1	8K	312K	ALLOCATED
2	32K	320K	ALLOCATED
3	–	–	EMPTY ENTRY
4	120K	384K	ALLOCATED
5	–	–	EMPTY ENTRY
	---	---	

Allocated Partition Status Table

Free Area	Size	Location	Status
1	32K	352K	AVAILABLE
2	520K	504K	AVAILABLE
3	–	–	EMPTY ENTRY
4	–	–	EMPTY ENTRY
	---	---	---

Unallocated Area Status Table

Figure 3-7 Dynamic partition specification tables

An example of dynamic partition specification is presented in Figure 3-8. At some point three partitions are allocated, each containing a job of corresponding size (Figure 3-8a). Three additional jobs are then selected to be multiprogrammed, and new partitions of the appropriate sizes are created from the unallocated free areas (Figure 3-8b). Eventually, the partitions can be deallocated after the corresponding job is terminated. Figure 3-8c depicts the memory status after Jobs 2 and 3 terminate.

Various algorithms are available to accomplish the allocation and deallocation functions. Examples of algorithms, based on the tables of Figure 3-7, are shown in Figures 3-9 and 3-10. Certain obvious steps must be performed for allocation. First, a free area at least as large as the partition desired must be found. Second, if the area is larger than needed, it must be split into two pieces—one becomes the allocated partition, the other becomes a smaller free area. Conversely, when a partition is deallocated, we try to merge it with any adjacent free areas so as to make one contiguous free area rather than many small pieces. (See, for example, the handling of Job 3's partition in Figure 3-8b and c).

The tables illustrated in Figure 3-7 represent one way of keeping track of memory usage. In actual implementations a different approach is often used. The key problem is that the number of entries needed in the partition table and the free table actually depends upon the number of partitions and free areas that exist at any time—i.e., the number is not constant. The simple table approach is therefore somewhat inefficient since most of the entries are not used. A better scheme would be

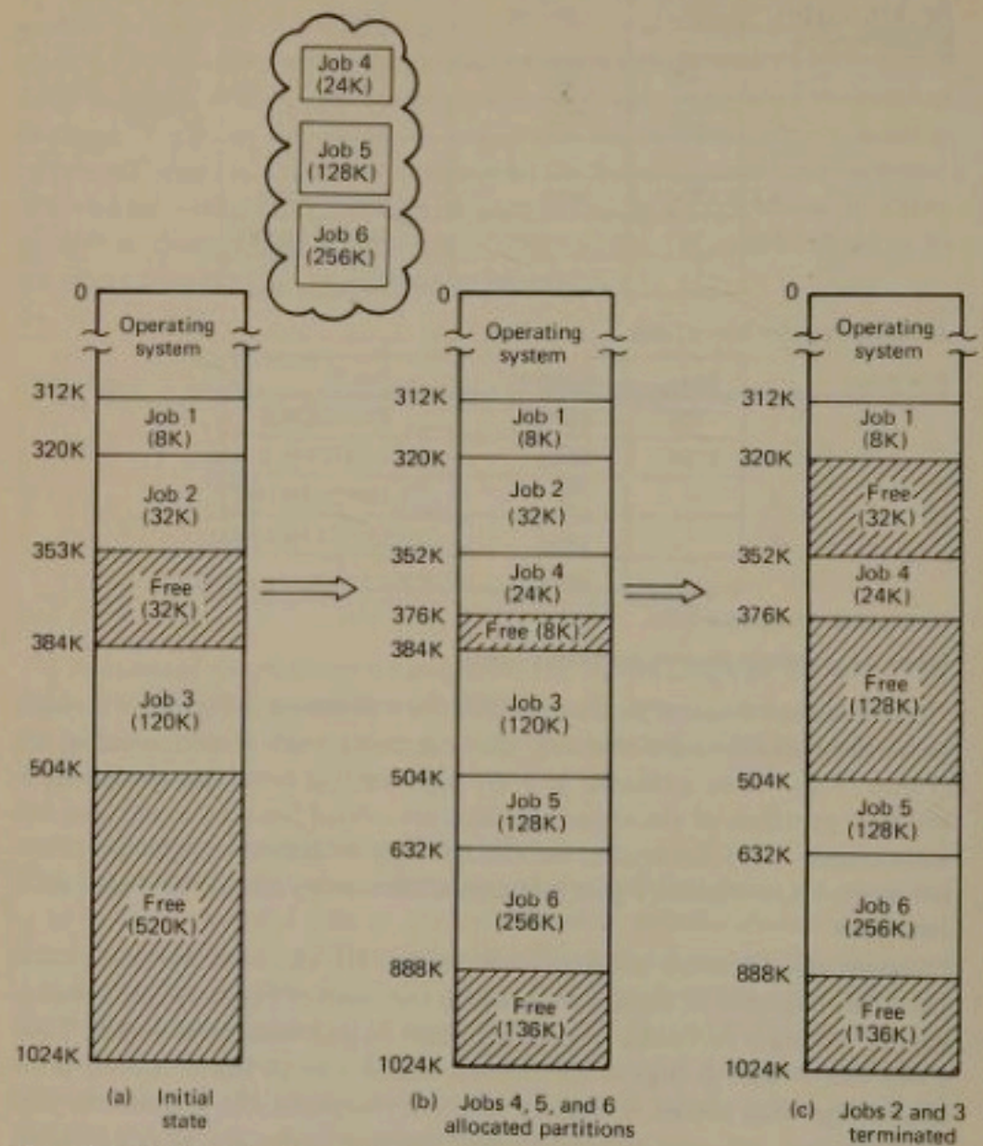


Figure 3-8 Partition allocation and deallocation

to append the table information to each partition or free area. The entries could then be chained together using address pointers. This approach is used in the sample operating system presented in Chapter 7. (The reader may wish to glance at Section 7-8 at this time.)

There are two common variations on the algorithms presented in Figures 3-9 and 3-10. These schemes are called *first fit* and *best fit* (Johnson, 1973; Knuth, 1968) and are outlined in the following sections.

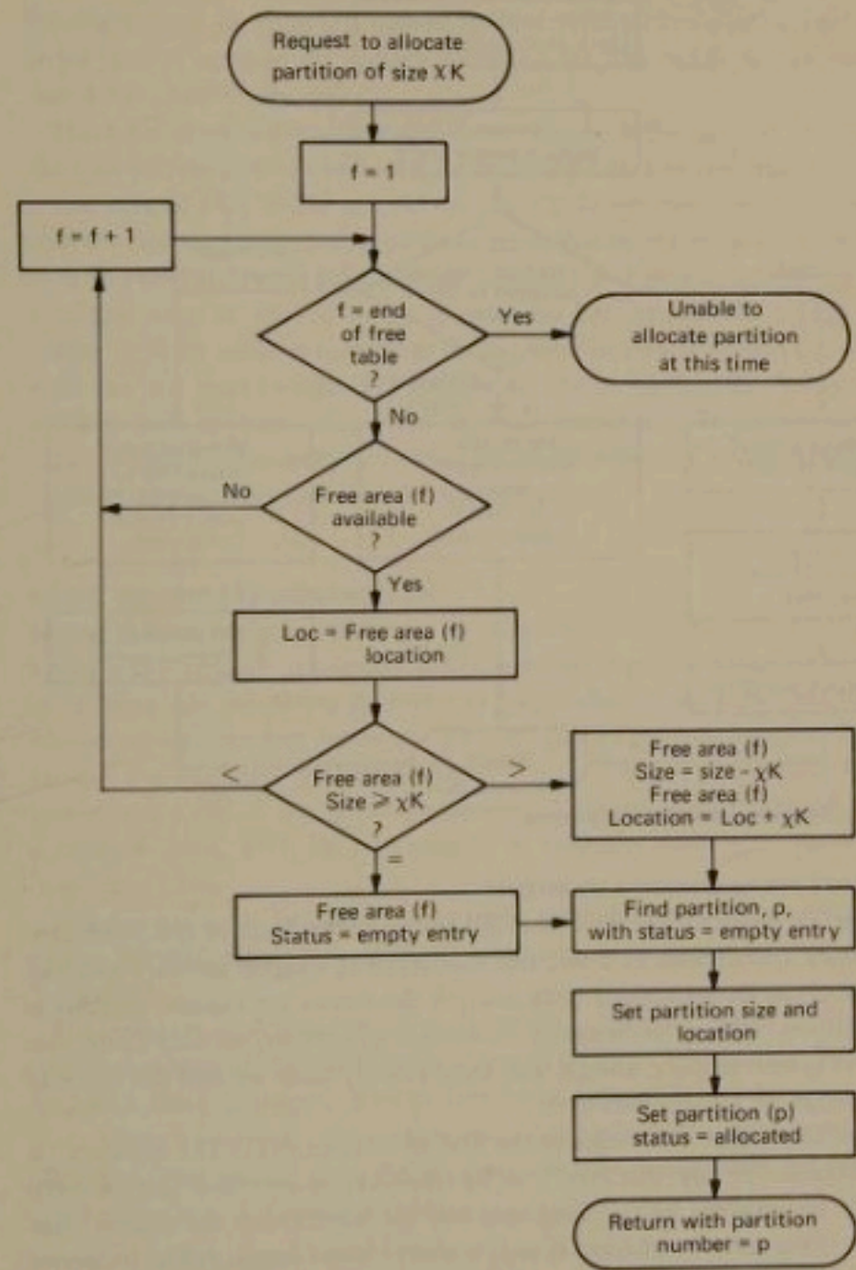


Figure 3-9 Partition allocation algorithm

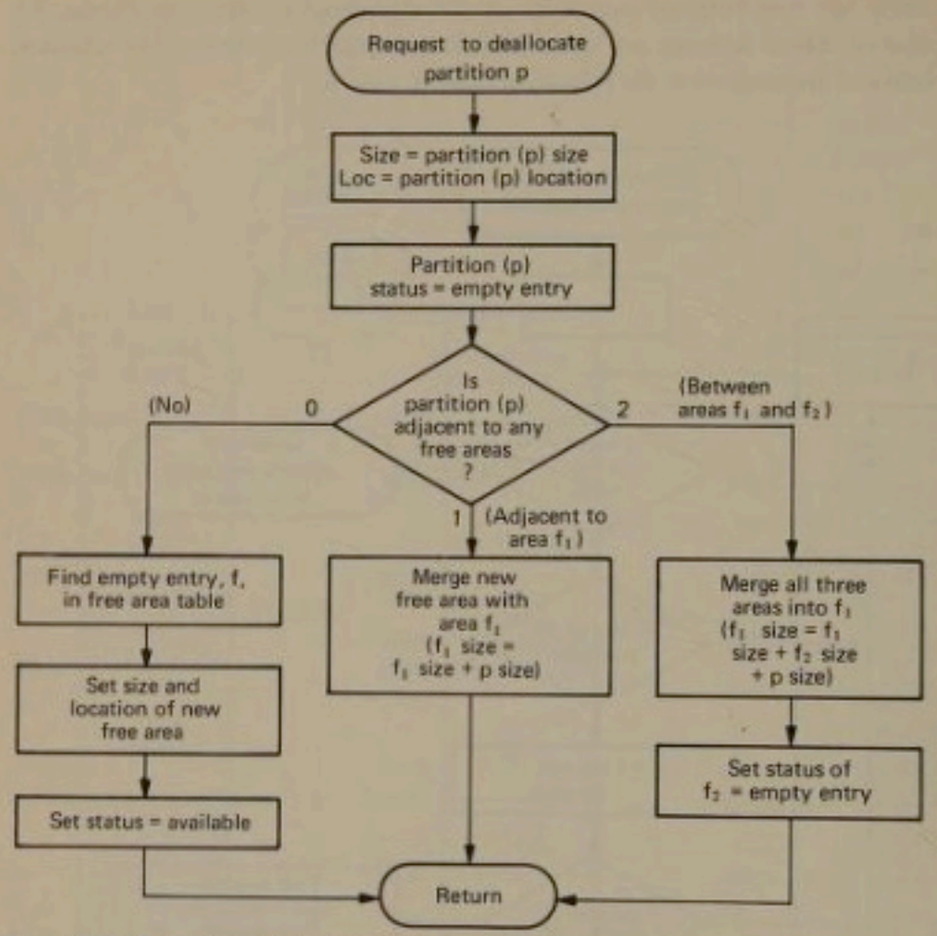


Figure 3-10 Partition deallocation algorithm

3-3.2.3 FIRST FIT PARTITION ALGORITHM

In the first fit algorithm the free table is kept sorted by location (i.e., each consecutive entry corresponds to a free area that starts at a higher memory address). This ordering can be maintained quite easily if the entries are chained together as suggested above. When it is necessary to allocate a partition, we start at the free area at the lowest memory address and keep looking until we find the *first* free area big enough for the partition to *fit*.

There are two major advantages to the first fit technique: (1) The question "Is partition adjacent to any free area?" in the deallocation algorithm (Figure 3-10) can usually be answered by searching only half the table. (Also, the adjacent free area at the lower memory address, if any, is always found first.) (2) The technique essentially favors using the free areas at low memory address whenever possible. This tends to allow a large free area to form at high memory addresses. Thus, when a very large partition size is needed, there is a good chance of finding a large enough free area.

3-3.2.4 BEST FIT PARTITION ALGORITHM

The only major difference between the best fit algorithm and the first fit algorithm is that the free table is kept sorted by size (i.e., the first entry corresponds to the free area with the smallest size). Thus the first free area we find that is large enough for the desired partition is the *best fit* (i.e., it is the free area closest in size but larger than or equal to the desired partition size). The technique described in Section 7-8 is a best fit algorithm.

There are three major advantages to the best fit technique: (1) On the average, the best fit free area can be found by searching only half the table. (2) If there is a free area of exactly the desired size, it will be selected—this is not necessarily true for first fit. (3) If there is no free area of exactly the desired size, the partition is carved out of the smallest possible free area and does not destroy a large free area that may be needed intact to satisfy a later request for a large partition.

Best fit has a major disadvantage in that the free area is usually not exactly the right size and must be split into two pieces. Due to the best fit criteria, the resulting free area is often quite small—so small that it may be almost worthless. In other words, the tendency is toward the development of a large number of very small free areas rather than of one large free area.

3-3.2.5 FRAGMENTATION PROBLEM

Several factors must be considered in selecting a partitioned memory algorithm. Speed and simplicity are among them; however, these are fairly easy to judge. A more important concern is the effect of fragmentation, the development of a large number of separate free areas (i.e., the total free memory is fragmented into small pieces). For example, referring to Figure 3-8c, if a request were made for a partition of size 138K, it would not be possible to allocate such a partition. Although a total of 296K bytes of free memory is available, there is no single free area larger than 136K.

Fragmentation can be overcome by using techniques other than partitioned allocation. We shall discuss two of these techniques, relocation and paging, in Sections 3-4 and 3-5.

The problem of fragmentation in partitioned allocation can be minimized by a careful selection of the specific algorithms to be used. For each of the algorithms discussed—static partition, dynamic first fit, and dynamic best fit—there are situations where it is most appropriate and others where it is least appropriate. By "situation," we usually mean the sequence of partition allocation and deallocation.

For example, a sequence might be:

Job 1 needs 140K
Job 2 needs 16K
Job 3 needs 80K
Job 1 terminates

Job 3 terminates
Job 4 needs 80K
Job 5 needs 128K
and so on

A particular situation is considered "bad" for an algorithm if it cannot satisfy a request immediately (i.e., a partition cannot be allocated until some allocated partitions are deallocated) while other algorithms could satisfy the request. The reader should work through the sequence above for first fit and best fit starting with 256K bytes of free area. Is the sequence a bad situation for either algorithm? Why? The reader is encouraged to develop sequences that are good or bad for each algorithm. Static allocation should be included by choosing static partition sizes.

The choice of a "best" particular algorithm depends upon the expected sequence characteristics (e.g., all jobs are the same size, or jobs are mostly 64K with a few 256K jobs). There are many more algorithms in addition to first fit and best fit; the reader may wish to examine the references (Knuth, 1968; Madnick, 1966; Ross, 1967).

3-3.2.6 MULTIPLE PARTITION ALGORITHM

The use of multiple partition allocation can sometimes decrease the fragmentation problem. A job usually consists of several separate memory segment portions, that is, separate subroutines and data arrays. Each individual portion must be logically contiguous (e.g., A(i + 1) should follow A(i)) but the portions themselves need not be contiguous as long as adequate protection facilities are available. Thus a job requiring 100K of memory may actually consist of five 20K portions. This job can be allocated either one 100K partition, or five 20K partitions, or two 40K partitions and one 20K partition, and so on. This approach is called *multiple partition allocation* since more than one partition can be allocated to a job. Furthermore, with this kind of scheme it is possible for a job to request the allocation of additional memory during execution—for loading a special error-handling subroutine, for example—since this new portion does not have to be contiguous with any of the job's current portions. A memory allocation scheme similar to the above was used in the ill-fated IBM OS/360 VMS (Variable Memory System). (This system was never officially announced or delivered to customers; it was replaced by OS/MVT, which uses a dynamic partition scheme.)

Multiple partition allocation gives more flexibility to memory management (i.e., there is more than one way to satisfy the job's needs), but it doesn't eliminate fragmentation. Each free area fragment is, on the average, much smaller, but there are many more of them. In fact, the number of portions is actually a disadvantage.

In the case of a wide diversity in minimal portion sizes for jobs (e.g., an array of 50,000 elements, each of 4 bytes, would require a 200,000 byte contiguous area), it would be unlikely that a free area large enough for these large portions could be found. For example, it is possible for a single job requiring a mere 100K bytes allocated as fifty 2K portions to fragment a million-byte memory so that there would be no free area larger than 20K bytes; this might be smaller than the minimum portion size needed by most other jobs. This example is an extreme case, but such situations have been observed.

In cases where all portions were approximately the same size, multiple partition allocation would become much more attractive. Another variation on this approach, segmentation, is presented in Section 3-7.

3-3.3 Advantages

Partitioned allocation offers three major advantages:

1. It facilitates multiprogramming, hence, more efficient utilization of the processor and I/O devices.
2. It requires no special costly hardware.
3. The algorithms used are simple and easy to implement.

One question that may arise at this point is how much memory a system needs under a partitioned system. That is, how much memory should be bought? The answer depends on at least two factors: (1) average size of job and (2) percent of utilization of processor desired.

It has been observed on several 360/65 and 370/155 installations (Lehman, 1968) that a typical job requires about 128K bytes, and, if run alone, would spend 65 percent of its time in the I/O wait state. The performance model from Section 3-2.2 and Table 3-1 indicates that in order to keep average system I/O wait time below 10 percent, four jobs must be multiprogrammed to reduce the wait to 8 percent and five to decrease it to 3 percent. This suggests that about 640K bytes of memory would be adequate. There must, however, be a reserve to compensate for fragmentation. If we assume that one third of memory is wasted due to fragmentation, then our needs rise to 960K bytes. Thus, about one million bytes would be a good estimate of memory needs. Of course, memory must be even larger to hold the operating system itself. Typically, somewhere in the range of 256K to 512K would be needed for a sophisticated multiprogramming system, but a particular operating system may require less or more.

3-3.4 Disadvantages

There are several disadvantages to partitioned memory that reduce memory utilization:

1. Fragmentation can be a significant problem. It is possible to find job sequences that result in memory utilization well below 10 percent. The extent of this problem depends on the typical job sequences and algorithms used.

2 Even if memory is not fragmented, the single free area may not be large enough for a partition. For example, we have a 512K memory and a job sequence containing a few 300K jobs, but mostly 256K. If a 300K partition is allocated, the remaining 212K bytes are wasted since they are too small for any of the other jobs.

3 It does require more memory than a single contiguous allocation system (as well as a more complex operating system) in order to hold more than one job.

4 As with contiguous allocation, memory may contain information that is never used. Furthermore, a job's partition size is limited to the size of physical memory.

3-4 RELOCATABLE PARTITIONED MEMORY MANAGEMENT

An obvious solution to the fragmentation problem (as previously exemplified in Figure 3-8c) is to periodically combine all free areas into one contiguous area. This can be done by moving the contents of all allocated partitions so that they become contiguous, as illustrated in Figure 3-11. This process is called *compaction* (or *recompaction*, since it is done many times). The term "burping" the memory has also been used to describe this technique.

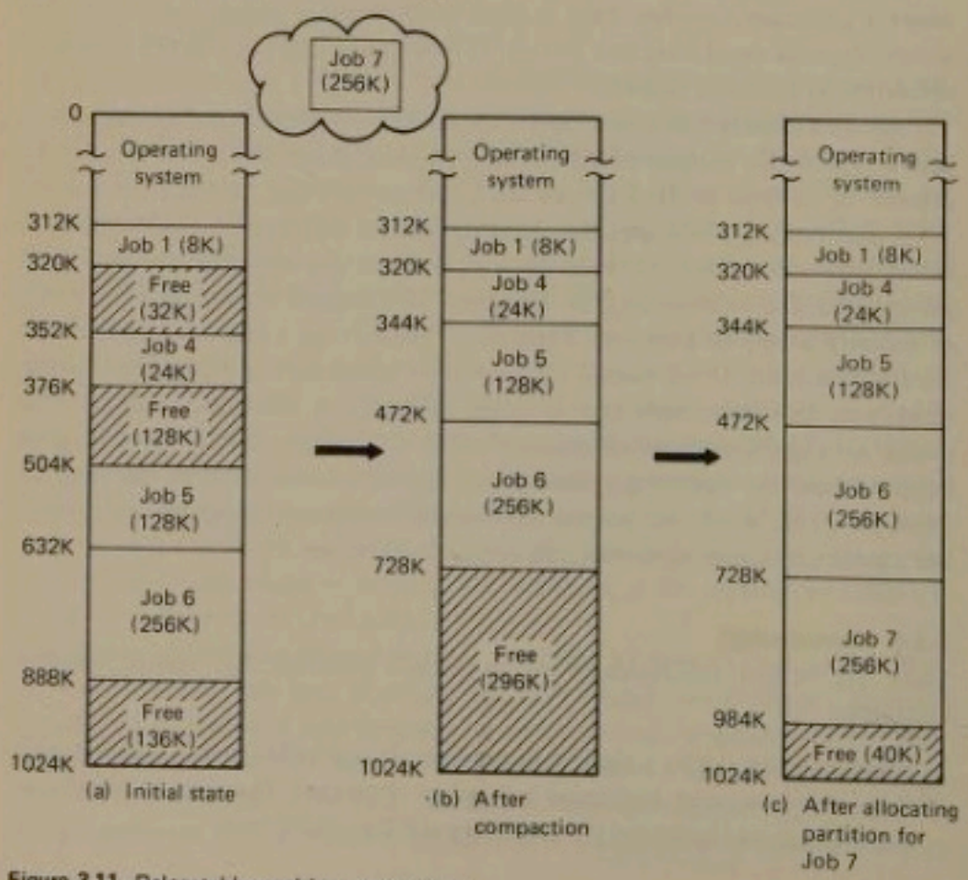


Figure 3-11 Relocatable partition compaction

Although it is conceptually simple, moving a job's partition doesn't guarantee that the job will still run correctly at its new location. This is because there are many location-sensitive items, such as: (1) base registers, (2) memory referencing instructions, (3) parameter lists, and (4) data structures (e.g., lists, chains, etc.) that use address pointers. To operate correctly, all location-sensitive items must be suitably modified. For example, in Figure 3-11, Job 4 is moved from location 352K to 320K. All addresses within Job 4's partition must then be decreased by 32K. This process of adjusting location-sensitive addresses is called *relocation* and is analogous to the operation of a relocating loader (Donovan, 1972; Presser, 1972).

Unfortunately, relocation of a job's partition can be quite difficult. In particular, how does the operating system identify the address pointers that must be altered? If the number 364000 appeared in Job 4's partition, is it an address pointer to be relocated or merely a variable representing the company's accounts payable? Some existing software techniques and conventions could be used (Reiter, 1967), but these are usually either inefficient or highly restrictive.

Another possible solution exists to the relocation problem: we could reload all jobs to be relocated and restart them from the beginning. The relocating loader is capable of handling the initial static relocation. However, besides being extremely crude (i.e., many hours of computation may have to be repeated), this restart may not be feasible if the program has already performed some irreversible actions (e.g., you may be issued two or more payroll checks, especially if your name is at the beginning of the alphabet).

Several hardware techniques have been developed to cope with this relocation problem. Most of them are based on the concept of a *mapped memory*, that is, the address space "seen" by a job is not necessarily the same as the physical memory address used. This concept is sometimes called virtual memory, but we shall reserve that term for a particular class of address mappings to be introduced in Section 3-6. Section 3-4.1 presents some techniques that can be used.

Except for the process of compaction and relocation, the relocatable partition allocation schemes are very similar to the previously discussed partitioned allocation techniques.

3-4.1 Hardware Support

There are two common approaches to the relocation problem, both of which require (or, at least, would benefit from) special hardware support. One approach, based upon the data typing concept, physically records the type of value stored in every memory location. For example, we could add 2 bits to every word to designate the value type (e.g., 00 = integer, 01 = floating point number, 10 = character, 11 = address pointer). These bits are never seen by the typical user program but are automatically set by the hardware. For example, the statement A = B not only sets A to the value of B but also copies the type information. Thus, address pointers may be manipulated and moved around within the program. When it is necessary to

compact and relocate, the operating system, using privileged instructions, can examine the type code information and thereby find each address pointer to be modified. Hardware of this type is available on computers such as the Burroughs 5500 and 6700.

Address mapping by means of data typing is possible because the user program has a somewhat special view toward address pointers. These cannot be arbitrarily examined or manipulated; thus the user cannot become location-sensitive because there is no way the program can determine its location (e.g., the question "Am I loaded at location X?" cannot be asked!).

The data typing approach has several disadvantages. It requires extra bits on every word, compaction may be slow since the type bits on every word must be examined, and it is incompatible with the general structure of most contemporary computers (in particular, the System/370).

Another common hardware solution uses dynamic relocation by means of base-bounds relocation registers. This technique is used on several computers, such as the DEC PDP-10, the Univac 1108, and the Honeywell 6000 series. Two special privileged registers, the *base relocation register* and the *bounds register*, are accessible only by the operating system. On every memory reference, the content of the base relocation register is automatically added to the effective address. (The effective address is the final reference address computed by the processor; this includes any index or 370-type base register adjustments.)

At the left of Figure 3-12a is the address space of Job 4 (from Figure 3-11a), which is separated from the physical location of its partition on the right. Before relocation, it is obvious that the instruction LOAD 1,370248 (Load Register 1 from Location 370248) at 360800 will load the value 015571 into register 1. (For ease of reading, we will use direct effective addresses in instructions rather than the 370 base plus displacement notation.)

After relocation, the LOAD instruction of Figure 3-12b is at location 328032. Since none of the instructions or registers have been changed, the effective address for the data to be loaded is still 370248 even though the intended data value has been moved to 337480. To produce correct results, the operating system must set the relocation register to −32768 (320K − 352K = −32K). When the instruction LOAD 1,370248 is encountered, −32768 is automatically added to the effective address, 370248, to determine the actual physical memory location to be accessed —in this case, 370248 − 32768 = 337480. Thus the value 015571 is loaded into register 1 as intended. Note that nothing within the job's address space has been changed, nor is the program aware of its actual physical memory location. Every instruction, LOAD, ADD, BRANCH, and so on, will behave as if the partition were at 352K, even though it has been moved to 320K.

Since this relocation adjustment is done automatically as each instruction is executed, it is called *dynamic relocation*.

Protection could be implemented by using techniques such as the 370 locks and keys, but more commonly a bounds register is set to the maximum address in the

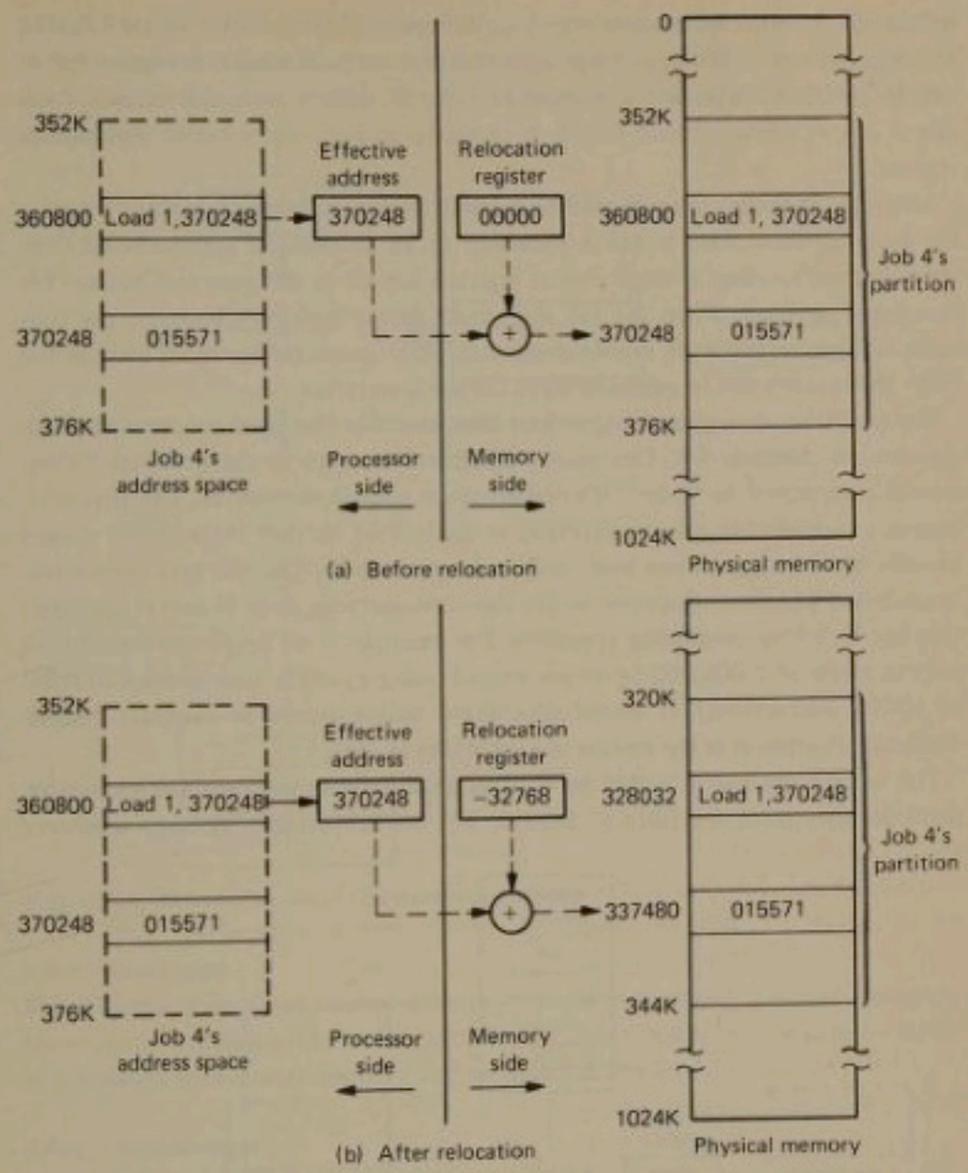


Figure 3-12 Use of relocation register

job's address space (376K in our example above). If the job attempts to access any location beyond the address specified in the bounds register, a protection interrupt will occur.

3-4.2 Software Algorithm

Since the starting point of a job's address space is unrelated to the physical location of the partition, it is useful to start the address space at location 0. This means that each job "thinks" that it is loaded at location 0 in memory. Thus the addresses

within Job 4 would be initially set up as in Figure 3-13, no matter where its initial partition is located. If this software convention is used, all relocation register values will be positive. Furthermore, protection is accomplished, since the hardware will detect any effective address that is negative or exceeds the value of the bounds register.

Although, logically, there should be a separate pair of base and bounds registers for each partition, this is not a necessary or an economical approach. Prior to starting a job running, a single pair of registers are set to the base and bounds for that job's partition. When the job must wait during multiprogramming, the contents of these registers are saved (along with other status information, such as the PSW) so that they can be reloaded when the job is restarted.

The partition management algorithms raise basically the same concerns as those discussed in Section 3-3. One major additional concern is the question "When should compaction be done?" Two alternatives suggest themselves. The first is to compact immediately after a partition is deallocated so that there would always be only one contiguous free area—and no fragmentation. The free area table maintenance and partition allocation would then become very easy. However, compaction can be a time-consuming operation. For example, if we could move partitions only at a rate of 1,000,000 bytes per second (using a facility such as the 370 MVC or MVCL instructions), it would take about half a second to compact a small deallocated partition in the middle of a 1024K-byte memory.

The second alternative would be to operate exactly as with the nonrelocatable partition algorithms described in Section 3-3, but to compact memory whenever

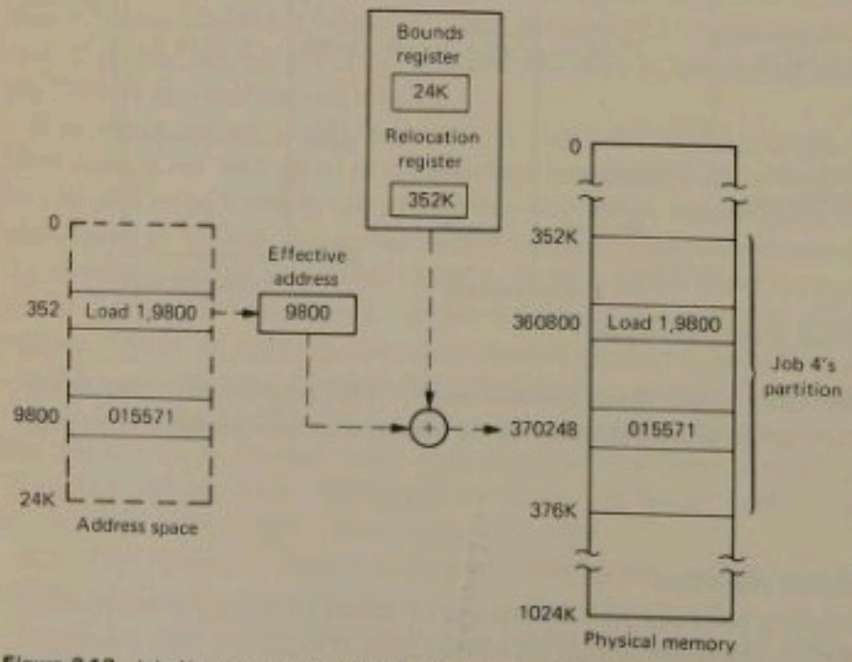


Figure 3-13 Job 4's address space if loaded at Location 0

a large enough free area was not available. Compaction would occur much less frequently than in the immediate compaction scheme above, but the table maintenance would be much more complex. Figure 3-14 depicts the overall flow of such an allocation algorithm.

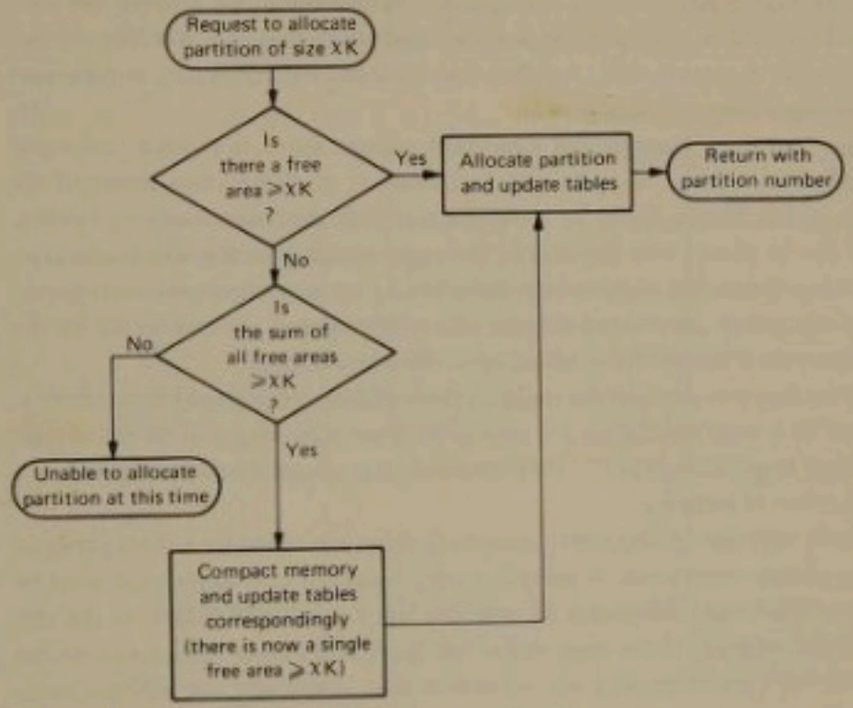


Figure 3-14 Overview of relocatable partition allocation

3-4.3 Advantages

The relocatable partition scheme eliminates fragmentation and thus makes it possible to allocate more partitions. This allows for a higher degree of multiprogramming, which results in increased memory and processor utilization.

3-4.4 Disadvantages

Several potential disadvantages must be considered:

1 Relocation hardware increases the cost of the computer and may slow down the speed (although usually only slightly).

2 Compaction time may be substantial.

3 Some memory will still be unused because even though it is compacted, the amount of free area may be less than the needed partition size. This wasted space is approximately equal to one-half the average job partition size.

4 As with partitioned allocation, memory may contain information that is never used. Furthermore, a job's partition size is limited to the size of physical memory.

3-5 PAGED MEMORY MANAGEMENT

The relocatable partitioned allocation approach represented one solution to the fragmentation problem through the use of address mapping to allow the individual free areas to be coalesced into one contiguous free area. This was necessary because we required that a partition be a contiguous area. We were able to relax the contiguity requirement in the multiple partition algorithm variation, but this did not always decrease fragmentation. Another way to avoid the contiguity requirement is through paged memory management.

For paged memory management each job's address space is divided into equal pieces, called *pages*, and, likewise, physical memory is divided into pieces of the same size, called *blocks*. Then, by providing a suitable hardware mapping facility, any page can be placed into any block. The pages remain logically contiguous (i.e., to the user program) but the corresponding blocks are not necessarily contiguous. As in the relocatable partitioned scheme, the user programs are unaffected by the mapping because it has no visible effect upon the address space.

For the hardware to perform the mapping from address space to physical memory, there must be a separate register for each page; these registers are often called *page maps* or Page Map Tables (PMTs). They may be either special hardware registers or a reserved section of memory.

Since each page can be separately relocated, there is no need for a job's partition to be completely contiguous in memory; only locations in a single page must be contiguous. Obviously, the choice of page size has a substantial effect on the usefulness of this scheme. If the page size is too large, it becomes comparable to the size of the job's partition, and the scheme is then essentially one of relocatable partition. If the page size is too small, it is necessary to have many page map registers, which greatly increases the cost of the computer system. As a result of these and other concerns, most paging systems use a page size of from 1K to 4K bytes.

A simple example, using a 1000-byte page size, is shown in Figure 3-15. Job 2, which has an address space of 3000 bytes, is divided into three pages. The page table associated with Job 2 indicates the location of its pages. In this case, page 0 is in block 2, page 1 is in block 4, and page 2 is in block 7. The LOAD 1,2108 instruction at location 0518 (page 0, byte 518) in Job 2's address space is actually stored at physical location 2518 (block 2, byte 518). Likewise, the data 015571 logically located at 2108 is stored at physical memory location 7108.

The paged memory management approach solves the fragmentation problem without physically moving partitions. For example, in Figure 3-15 there are 2000 bytes of available memory, but they are not contiguous. If there were a fourth job requiring 2000 bytes, we could compact memory, as is done with relocatable partitions, to produce a single 2000-byte free area. Alternatively, we can assign Job 4's two pages to the available blocks, such as page 0 = block 3 and page 1 = block 9. If the PMT is set correspondingly, the address space will still appear contiguous without making it necessary to physically move any partitions. A scheme of this type is

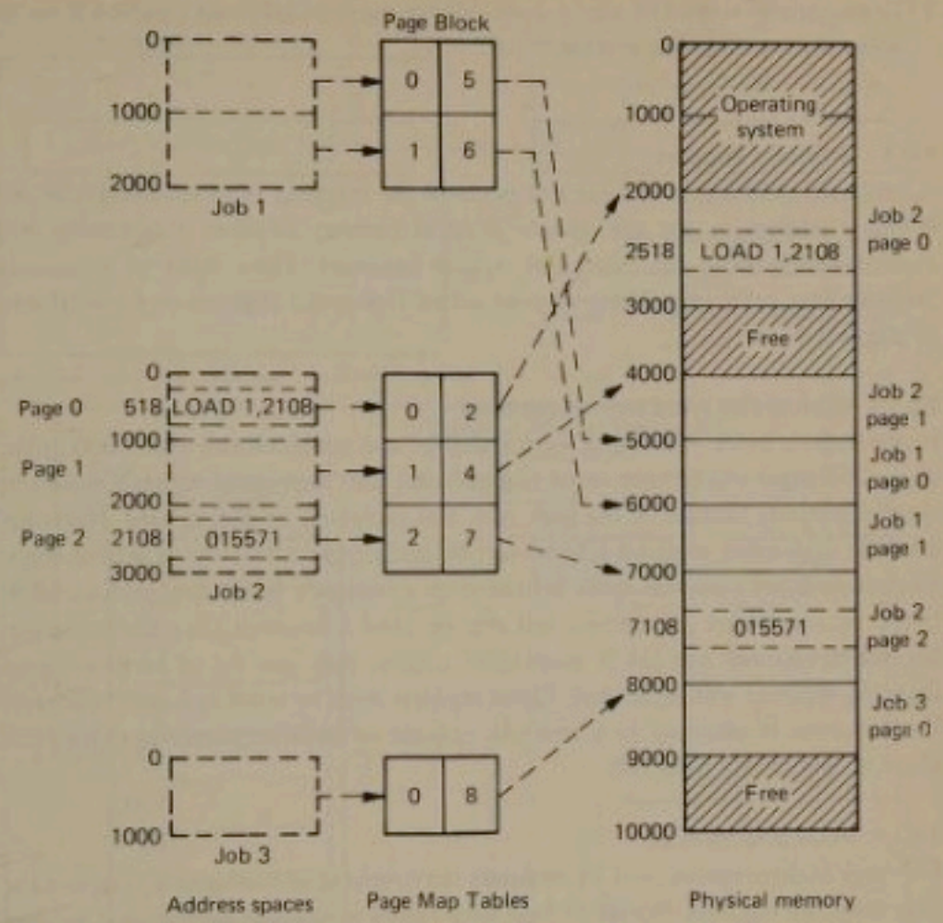


Figure 3-15 Page mapped memory

used in several contemporary computers, for example, the XDS 940, XDS Sigma 7, and CDC 3300.

The four functions of memory management are performed as follows:

1 Keeping track of status—accomplished through two sets of tables:
   a Page Map Tables—one per address space, each containing one entry for each page.
   b Memory Block Tables—one in system, containing one entry for each memory block with information on the use of that block (e.g., allocated or available).
2 Determining who gets memory—this is largely decided by the job scheduler. Memory may be assigned to the user simply by giving him the first set of free blocks found.
3 Allocation—all pages of the job must be loaded into assigned blocks and appropriate entries made in the Page Map Table and Memory Block Table.

4 Deallocation—when the job is done, blocks must be returned to free status by adjusting entries in block table.

3-5.1 Hardware Support

A hardware mechanism is needed to perform the mapping from each instruction's effective address to the appropriate physical memory location, functioning in a manner analogous to the relocation register hardware. There must be a separate "register" for each page. These may be actual high-speed registers or a special area of physical memory.

3-5.1.1 HIGH-SPEED PAGE MAP REGISTERS

If one million bytes of memory were available, and the block size were 4000 bytes, up to 250 pages would have to be mapped. But 250 high-speed registers would be quite expensive; because of the high cost, this technique is seldom used. Nevertheless, the high-speed registers scheme can be made more attractive in several ways. First, if each job's address space is limited to a relatively small size, such as 100K bytes, no more than 25 registers will ever be needed for each job's address space. Second, since only one job is running at a time, only one set of hardware page mapping registers will be needed. These registers must be saved and reset whenever the processor is switched to a new job because of multiprogramming. This technique is used on the XDS 940.

3-5.1.2 PAGE MAP TABLES

For large address spaces, and to eliminate the overhead of continually resetting the page mapping registers, the use of Page Map Tables in physical memory has become popular. The following hardware description is a slightly simplified version of the Dynamic Address Translation (DAT) facility—another name for page mapping—available on the IBM System/370.

To simplify address mapping, the page size is usually chosen to be a power of two, e.g., 1024 (1K) bytes, 2048 (2K) bytes, or 4096 (4K) bytes. In this way, if the page size is 4K bytes on the 370, the page number is precisely bits 8 through 19 of the effective address (see Figure 3-16). The effective address is 24 bits long, as generated by the processor. The DAT mechanism automatically separates the effective address into two parts: bits 8 through 19 become the 12-bit page number and bits 20 through 31 become the 12-bit byte offset within the page. Using the PMT, the page number is replaced by the block number to produce the resultant physical memory address to be used.

If the Page Map Table is to be stored in main memory, the hardware must know the location. We could assign a specific location, as done for the "old" and "new" PSWs, CAW, CSW, etc. This would, however, necessitate changing the entire table every time the processor switched jobs. Instead, the PMT is allowed to be stored

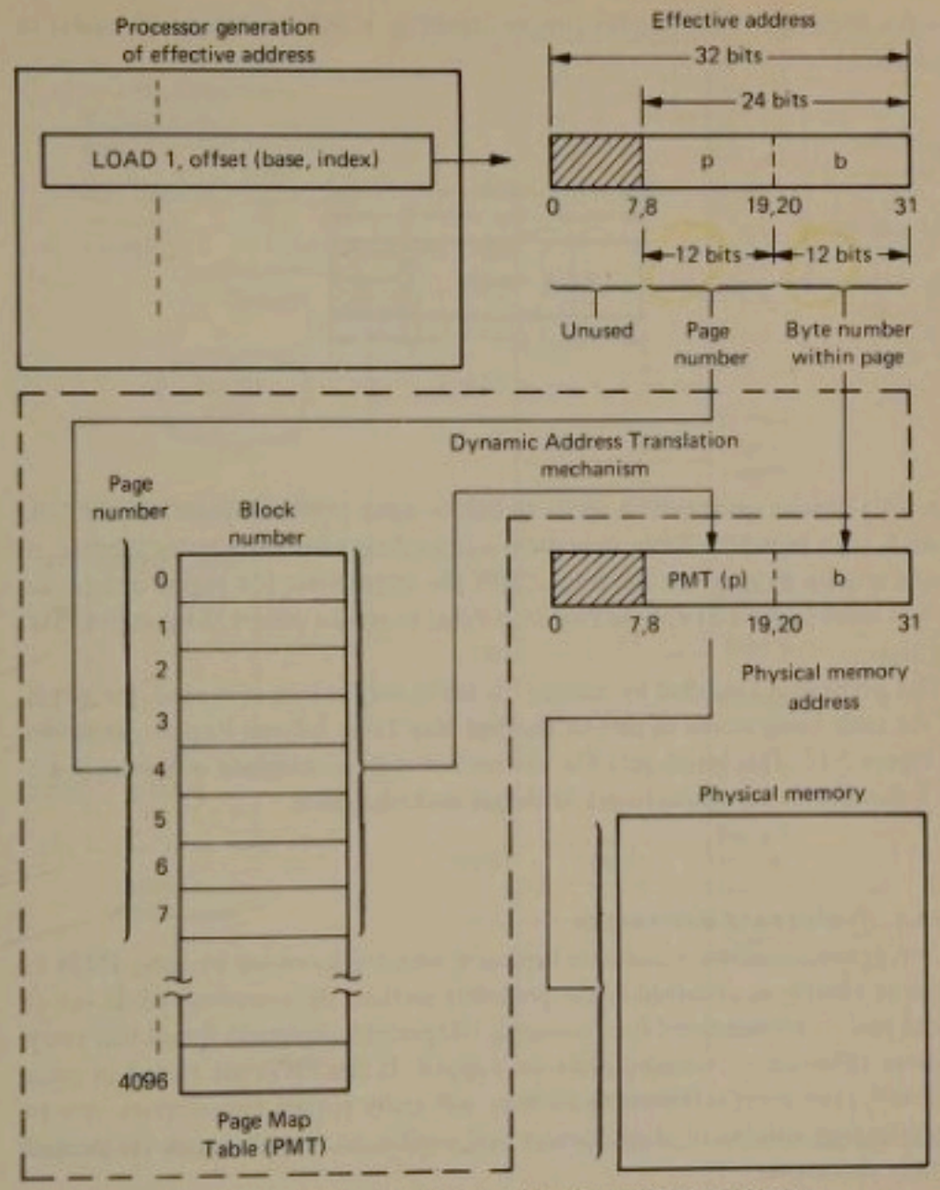


Figure 3-16 Determination of physical memory address from effective address

at any location in memory, the specific location used being indicated by the contents of a special Page Map Table Address Register (PMTAR). When the processor is switched to a new job, only the PMTAR has to be changed to indicate the location of the new job's PMT.

Portions of the simple example of Figure 3-15 are redrawn in Figure 3-17 to indicate the relationships between the pages, the blocks, the PMTAR, and the PMTs. The page size used in this diagram is 4096 (4K) bytes. There are various ways to encode the PMT entries. The block number stored in each entry must be 12 bits

It is sometimes desirable to have the address spaces overlap partially in order to make some subroutines and/or data areas accessible to all jobs. For example, there may be an area in the operating system where the current date is stored for reference. This overlapping can be accomplished as shown in Figure 3-20. All jobs are loaded starting at 8K in their address spaces; locations 0 to 8K are part of the operating system's address space. If address spaces overlap, it is necessary to provide additional protection in addition to the address mapping. This can be accomplished by the lock and key mechanism. Alternatively, the mapping hardware could be extended so that some of the unused bits in each page table entry could be used to indicate whether reading and/or writing are permitted in that page.

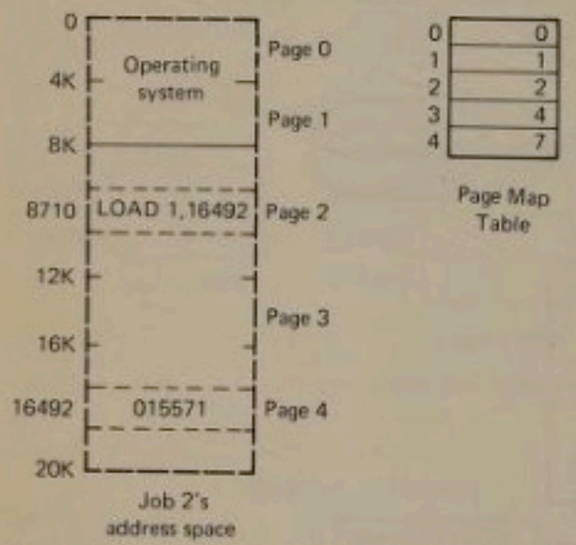


Figure 3-20 Overlapping address spaces (see Figure 3-17)

3-5.3 Advantages

The paged memory scheme eliminates fragmentation and thus makes it possible to accommodate the address spaces for more jobs simultaneously. This allows for a higher degree of multiprogramming, which results in increased memory and processor utilization. The compaction overhead required for the relocatable partition scheme is also eliminated.

3-5.4 Disadvantages

Several potential disadvantages must be considered:

1 The page address mapping hardware usually increases the cost of the computer and at the same time slows down the processor.

2 Memory must be used to store the various tables, principally the PMT. Processor time (overhead) must be expended to maintain and update these tables.

3 Even though fragmentation is eliminated, a similar phenomenon, called *internal fragmentation* or *page breakage*, does occur. Jobs are assigned whole blocks of memory. If a job requires 5K bytes and the block size is 4K, two blocks must be allocated (i.e., 8K is allocated but only 5K was needed). On the average, half a page is wasted for each job. A tradeoff must be made between cutting wastage due to page breakage, which favors small pages, and reducing the number of entries in the Page Map Tables, which favor large pages. Knowledge of the average job's size is important in resolving this conflict.

4 Some memory will still be unused if the number of available blocks is not sufficient for the address spaces of the jobs to be run. For example, if two blocks totaling 8K bytes are available, they cannot be used if the minimum job size to be run is over 8K bytes. This wasted space, exclusive of page breakage, is approximately equal to one-half the average job's address space.

5 Finally, as in the previous memory management techniques, memory will often contain information that is seldom used. Furthermore, a job's address space is limited to the size of physical memory.

3-6 DEMAND-PAGED MEMORY MANAGEMENT

In all the previous schemes a job could not be run until there was sufficient available memory to load its entire address space. This restriction often resulted in unused free areas even though there were jobs waiting to be loaded and run. Furthermore, the pressure on programmers to keep their job's address space small was often counterproductive. Several studies have shown that if a job must be designed to operate in a constrained address space size, the cost of programming increases rapidly (Boehm, 1973).

These problems could be resolved by using extremely large main memories. At present this approach, although simple, is not usually economically feasible. Another approach is to use the operating system to produce the illusion of an extremely large memory. Since this large memory is merely an illusion, it is called *virtual memory* (Denning, 1970). These are two major virtual memory techniques: demand-paged memory management and segmented memory management. Segmented memory management will be discussed in Section 3-7.

Whereas the preceding memory management techniques attempted to approximate 100 percent memory utilization, demand-paged memory management can logically attain a utilization greater than 100 percent! That is, the sum of all the address spaces of the jobs being multiprogrammed may exceed the size of physical memory. This feat is accomplished by removing the requirement that a job's entire address space be in main memory at one time; instead only portions of it have to be loaded.

Consider Figure 3-15 where three jobs are loaded in memory with two 1000-byte blocks unused. How could we handle a fourth job that requires 4000 bytes (i.e., it needs four blocks but only two are available)? We could do as indicated in Figure 3-21, that is, load only two of Job 4's pages. Job 4 would then run correctly as long as it only references information in these two pages (e.g., the LOAD 1,1120 instruction at location 100 in Job 4's address space would work correctly).

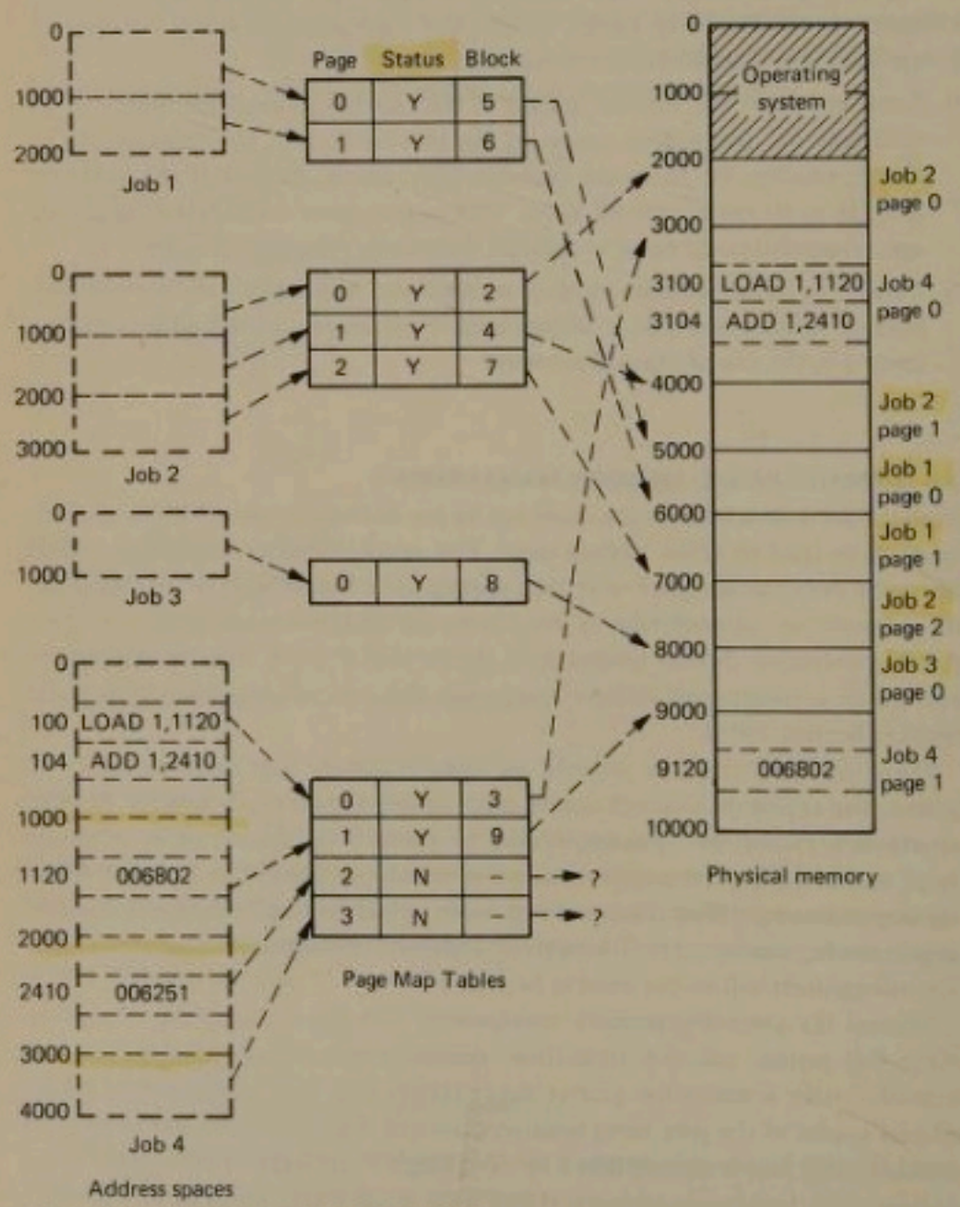


Figure 3-21 Demand-paged-mapped memory

Is it reasonable to try to run a job without loading its entire address space? Yes, because most programs use only a small amount of their entire address space during any particular run of the job. There are many reasons for this, such as:

1 User-written error-handling routines are used only when an error occurs in the data or computation (hopefully a rare event).
2 Certain options of the programs are mutually exclusive or not requested on every run (e.g., sort the payroll by name or by employee number).
3 Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used (e.g., if a symbol table for an assembler is set to handle 1,000 symbols and only 100 symbols are used in a particular run, 90 percent of the table is unused).
4 Many routines are commonly used at mutually exclusive times during a run (e.g., the input phase routines, the computation phase routines, and, finally, the output phase routines). Even if all these routines are used, they do not have to be in physical memory at the same time.

Two key questions remain to be answered: What do we do if a job references an area of address space not in physical memory? How do we decide which pages to keep in memory? Notice in Figure 3-21 that the ADD 1,2410 instruction at location 104 in Job 4 references a page that is not in memory. To handle this case, we extend the PMT hardware to include a status bit (i.e., Y = yes, reference is okay; N = no, reference is impossible).

If the address mapping hardware encounters a page table entry with status = N, it generates a *page interrupt*. Note that this is different from a protection interrupt, in that it is not a user program error. The operating system must process this interrupt by loading the required page and adjusting the page table entries correspondingly. The instruction can then be restarted. We say this page was "loaded on demand" (i.e., page 2 of Job 4's address was loaded because the page interrupt indicated that the job needed–demanded–the page). Thus, this scheme is called *demand-paged memory management*. In particular, when a job is initially scheduled for execution, usually only its first page is actually loaded. All other pages needed by the job are subsequently loaded on demand. This guarantees that an unnecessary page is not loaded.

Where are these pages loaded from when demanded? A copy of the job's entire address space is stored on a secondary storage device (e.g., magnetic disk or drum). When a page is needed in main memory, it is read from the secondary storage device.

The demand-paged memory management technique is very attractive because it does not constrain the address spaces to the size of physical memory. But there are some complications. In Figure 3-21 when we get to instruction ADD 1,2410 at location 104, page 2 must be loaded into physical memory for Job 4. Since all the memory blocks are now in use, where can page 2 be placed? This requires a technique

called variously *page swapping, page removal, page replacement, page turning,* or *page cannibalizing*.

Once memory has become filled with pages it is possible to load another page only by first removing one of the pages presently in memory. The *replaced* page is copied back onto the secondary storage device before the new page is loaded (i.e., the two pages *swap* places between memory and secondary storage). How do we decide which page to *remove* from memory? Replacement algorithms have been the subject of considerable research (Belady, 1966; Coffman, 1968; Denning, 1970, 1972; Gustavson, 1968; and Randell, 1968). (See Section 3-6.2.)

Removing a page from memory and then immediately needing it again due to a page fault referencing that page would be very unfortunate. The phenomenon of excessively moving pages back and forth between memory and secondary storage has been called *thrashing* since it consumes a lot of the computer's energy but accomplishes very little useful results. Thrashing and its avoidance have received considerable research attention (Denning, 1968, 1970). In order to assist the operating system software in making these page replacement decisions, the hardware usually maintains some information on page usage. An example of such information is presented in Section 3-6.1, which deals with hardware support.

Demand paging is used in many contemporary operating systems, such as VS/1, VS/2, and VM/370 on the IBM System/370, MULTICS on the Honeywell 6180, and VMOS on the UNIVAC Series 70/46.

The four functions of memory management become both more complex and more flexible in a paged memory environment:

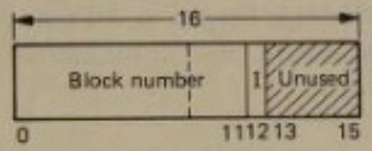
1 Keeping track of status—this is accomplished through three sets of tables:
   a Page Map Tables—one per address space
   b Memory Block Tables—one in system
   c File Map Tables—one per address space
2 The policy of who gets memory and when—partially determined by the job scheduler, but, on a dynamic basis, it is also determined by the demand page interrupts.
3 Allocation—when a block must be allocated, an available block must be found and the status of the block altered.
4 Deallocation—if it is not possible to find an available block for allocation, one of the allocated memory blocks must be deallocated and reassigned. When a job terminates, all the blocks it was using become available.

3-6.1 Hardware Support

The address mapping hardware via Page Map Tables, as described in Section 3-5, is needed for demand paging. Three key additions to the hardware are required.

1 A status bit in the Page Map Table to indicate whether the page is in main memory or secondary storage.
2 Interrupt action to transfer control to the operating system if the job attempts to access a page not in main memory
3 Record of individual page usage to assist the operating system in determining which page to remove, if necessary.

These requirements are met on the IBM System/370 (when operating in Extended Control Mode—i.e., when address translation is turned "on") as follows.[1] The PMT entry described earlier in Section 3-5.1.2 is altered so that one of the previously "unused" bits is used to indicate whether to generate an interrupt if that entry is accessed (I = 1 means generate interrupt).



If a *page interrupt condition* (also called *page exception interrupt* or *page fault*) is encountered, a program interrupt with code = 17 occurs. Furthermore, the logical address that was referenced is stored in the word at location 144 in memory. The operating system can determine from this word which page needs to be loaded.

There are 2 additional bits associated with each block of memory. (These 2 bits are actually stored as part of the lock register on each 2K memory block.) The *reference bit* is automatically set to 1 whenever any location in that block is referenced (i.e., load, store, or execute). The *change bit* is set to 1 whenever any location in the block is changed (i.e., stored into). The operating system software can reset these bits using the Set Storage Key privileged instruction. On other hardware implementations, such as the Honeywell 6180, these bits are stored in the PMT entry.

Other facilities may be provided by the hardware to assist the operating system. For example, the operating system usually runs in *absolute mode* or *basic control mode* (i.e., like a 360 without dynamic address mapping). The address mapping can be selectively turned on by instructions such as Load Real Address (LRA), which translates a virtual address into its corresponding physical memory address.

[1] The Extended Control Mode is in effect when the "E" bit in the PSW (bit 12) is set to 1. The System/370 provides hardware support for either 2K or 4K pages. The mode of operation (2K or 4K) is determined by the setting of bits 8 and 9 of control register 0. In this chapter only the 4K mode is described.

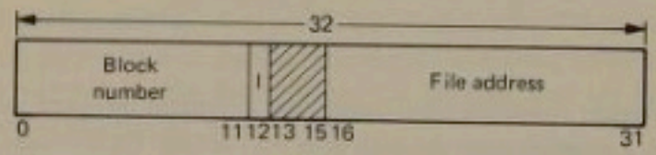
If the page is not in memory, a specific condition code is set rather than an interrupt being generated. This facility is important for handling input/output requests, as explained in the next section.

3-6.2 Software Algorithm

Demand-paged memory management provides tremendous flexibility for the operating system and, consequently, has received considerable attention. In this section we can discuss some of the major algorithms only; the reader is directed to the references for more extensive coverage.

3-6.2.1 FILE MAP REQUIREMENT

Demand-paged memory management must interact with information management to access and store copies of the job's address spaces on secondary storage (information management is discussed in Chapter 6). We will assume that each page has a unique secondary storage address, called its *file address*. This address logically is part of each PMT entry, such as



if the file address were 16 bits long. Although the Page Map Table is tied directly to the address mapping hardware, the format of the file address (and related status information) is handled solely by the software, and may be variable. Thus, the file information is usually stored in a separate table called the File Map Table, which is not used by the hardware.

The relationship of the File Map Table to the Page Map Table and the Memory Block Table is illustrated in Figure 3-22. Recall that these tables all reside as part of the operating system area in main memory and that there are separate Page Map and File Map Tables for each address space. In this diagram Job 2 has only two of its three pages in main memory. A copy of all three pages is available on a secondary storage device.

3-6.2.2 OVERVIEW OF PAGE INTERRUPT PROCESSING

In the previous memory management schemes, allocation of memory was completely static. That is, memory was allocated or rearranged only when a job started or terminated. In demand paging, it is necessary to allocate and deallocate memory during the job's execution. The memory management routines are invoked by means of the page interrupt.

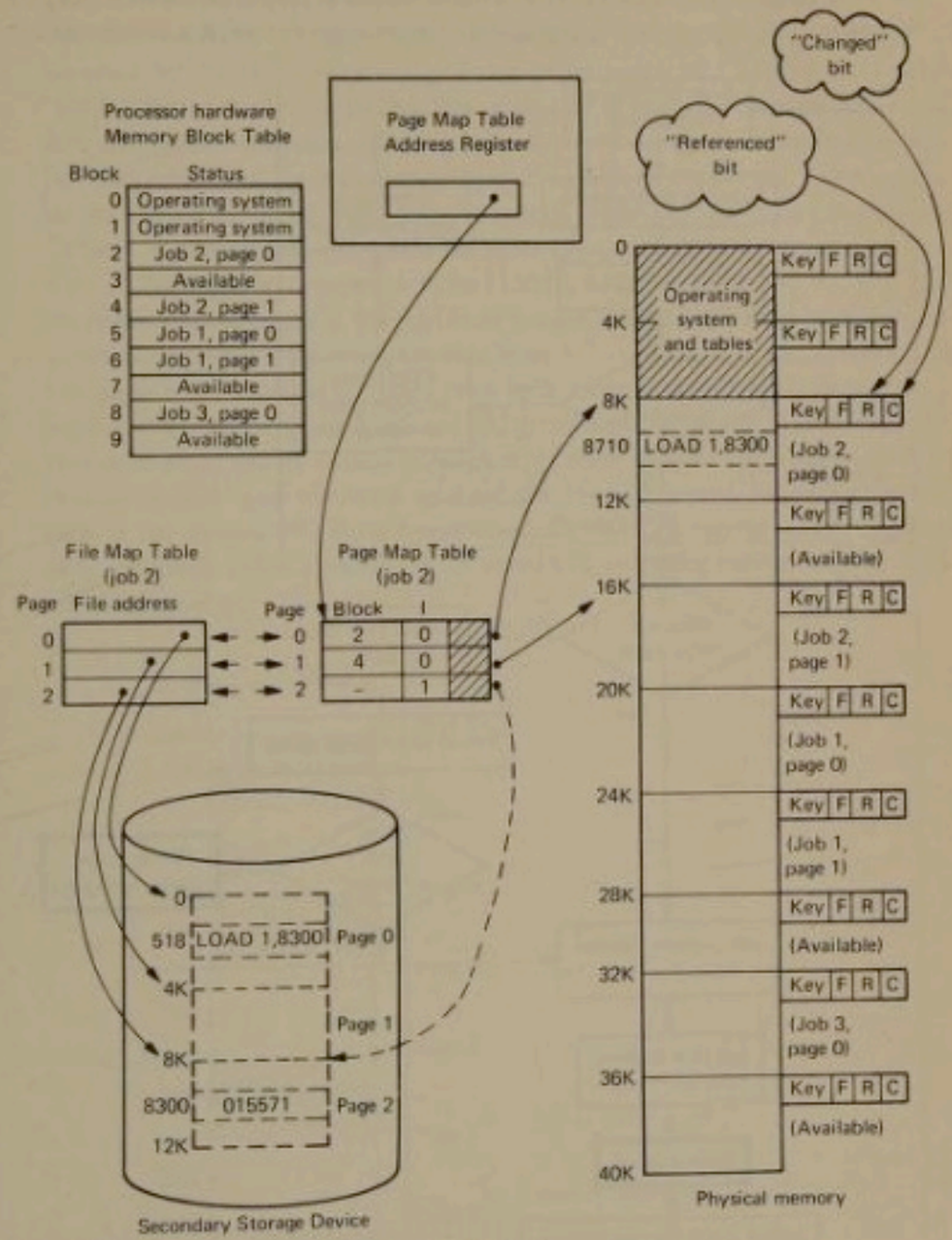


Figure 3-22 Relation of File Map Table to other tables

management. There is no sense in assigning a processor to a job if insufficient memory is available to hold that job's working set. Chapter 8 presents an analysis of this phenomenon.

3-6.3.4 LOCALITY

Efficient operation of a virtual memory system is dependent upon the degree of *locality of reference* in programs. Locality can be divided into two classes: temporal locality and spatial locality (Madnick, 1973).

Temporal locality refers to an observed property of most programs, i.e., once a location—data or instruction—is referenced, it is often referenced again very soon. This behavior can be rationalized by program constructs such as loops, frequently used variables, and subroutines.

Spatial locality refers to the probability that once a location is referenced, a nearby location will be referenced soon. This behavior can be rationalized by program constructs such as sequential instruction sequencing, linear data structures (e.g., arrays), and the tendency of programmers to put commonly used variables near one another.

The degree of locality varies from program to program. In general, it is desirable to develop programs with a high degree of locality. Appropriate programming techniques are discussed in Morrison, 1973 and Brawn, 1968.

3-6.4 Advantages

Demand-paged memory management has all the advantages discussed earlier with regard to paged memory management. In particular, fragmentation is eliminated and compaction is not necessary.

There are several additional advantages:

1. *Large virtual memory* A job's address space is no longer constrained by the size of physical memory. This provides better compatibility between large and small computers (e.g., a 512K-byte job that is normally run on a 1024K computer can be run on a 256K computer if necessary).
2. *More efficient use of memory* The portions of a job's address space that are not used—or seldom used—will not be kept in physical memory. Often 25 percent or more of a job's address space need never be loaded since these parts are used only under peculiar circumstances (e.g., error routines).
3. *Unconstrained multiprogramming* The degree of multiprogramming was usually limited by the size of physical memory in the previously discussed schemes. That is, the sum of the jobs' address space sizes must be less than or equal to physical memory size. In demand paging, there is no such limit. It is possible, though not necessarily desirable, to multiprogram 50 or 100 jobs on a computer with only 256K bytes of memory.

3-6.5 Disadvantages

In addition to the disadvantages mentioned for paged management in Section 3-5.4 (hardware costs, page breakage, processor overhead, and memory space for tables), new concerns arise in the case of demand-paged management:

1. The number of tables and amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management technique.
2. Due to the lack of an explicit constraint on a job's address space size or amount of multiprogramming, it is necessary to develop approaches to prevent thrashing (i.e., an exorbitant amount of processor overhead is being consumed by the system to handle paging interrupts while very little progress is being made on user jobs). Under extreme thrashing situations, over 99 percent of processor time may be consumed by overhead activities and less than 1 percent by user jobs. (Some techniques for thrashing avoidance are presented in Chapter 8.)

3-7 SEGMENTED MEMORY MANAGEMENT

In the previous memory management approaches the physical memory was handled in various ways, e.g., partitioned, relocated, paged, etc.; yet these actions were invisible to the user's program. A linear and contiguous address space was always provided. Are there other ways of presenting the address space so as to facilitate efficient memory utilization and programmer convenience? Yes, segmentation is such a way.

A *segment* can be defined as a logical grouping of information, such as a subroutine, array, or data area. Thus, each job's address space actually consists of a collection of segments, as illustrated in Figure 3-38. *Segmentation* is the technique for managing these segments.

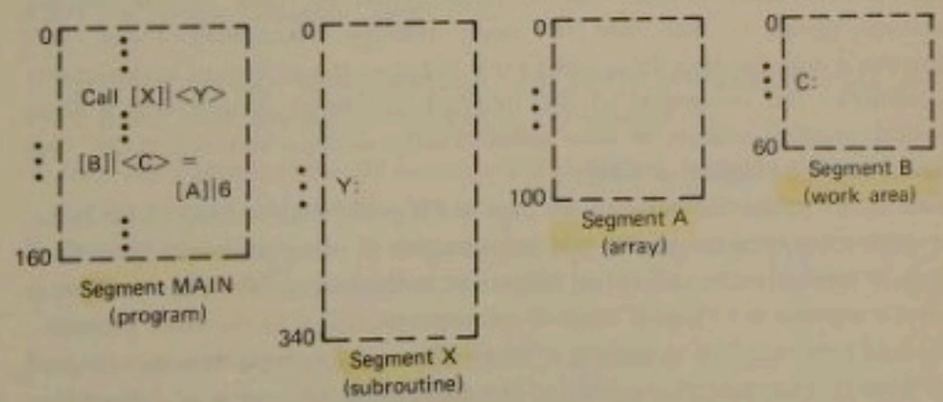


Figure 3-38 Segmented address space

In a segmented environment all address space references require two components: (1) a segment specifier and (2) the location within the segment. For example, references may be as follows:

CALL	[X]|⟨Y⟩	TRANSFER TO ENTRY POINT Y WITHIN SUBROUTINE X.
LOAD	1,[A]|6	LOAD THE 6TH LOCATION OF ARRAY A INTO REGISTER 1.
STORE	1,[B]|⟨C⟩	STORE REGISTER 1 INTO LOCATION C WITHIN SEGMENT B.

Although the segmented address space is explicit and "visible" to the programmer, the differences between this space and conventional address space need not affect his programming. In most programming languages there are statements, such as:

CALL Y;

C = A(6);

In a single linear address space the compiler and/or loader must convert these segmentlike references into a single address. For example, if Y is an entry point at location 120 within subroutine X and subroutine X is loaded at location 3460, then CALL Y becomes CALL 3580. Likewise, if A is an array of bytes at location 3800, A(6) refers to location 3806. Notice that the original structure is lost in this transformation. If an error occurred at location 3803, there is no explicit indication of the fact that 3803 is really A(3).

Typically, in a segmented address space, each segment is assigned a number, such as [X] = 3, [A] = 5. Then, CALL Y becomes CALL [3]|120 and A(6) becomes [5]|6, explicitly indicating the segment and location within the segment being referenced.

A segmented address space can be implemented by using address mapping hardware similar to that used for paged memory management. Figure 3-39 illustrates a Segment Map Table (SMT) that indicates the address of each segment in memory. The conversion of the two-part segmented address into a linear physical memory address is done automatically during execution rather than statically by the compiler or loader.

Due to the similarities between the page and segment address mapping hardware, the distinction between paging and segmentation is often confused. We will attempt to emphasize the conceptual differences in this section. The major difference is that a segment is a "logical" unit of information—visible to the user's program—and is of arbitrary size. A page is a "physical" unit of information—strictly used for memory management, invisible to the user's program—and is of a fixed size, e.g., 4K bytes. In retrospect, a segment is a more precise form of the job "portion" notion introduced in the multiple partition allocation discussion (Section 3-3.2.6).

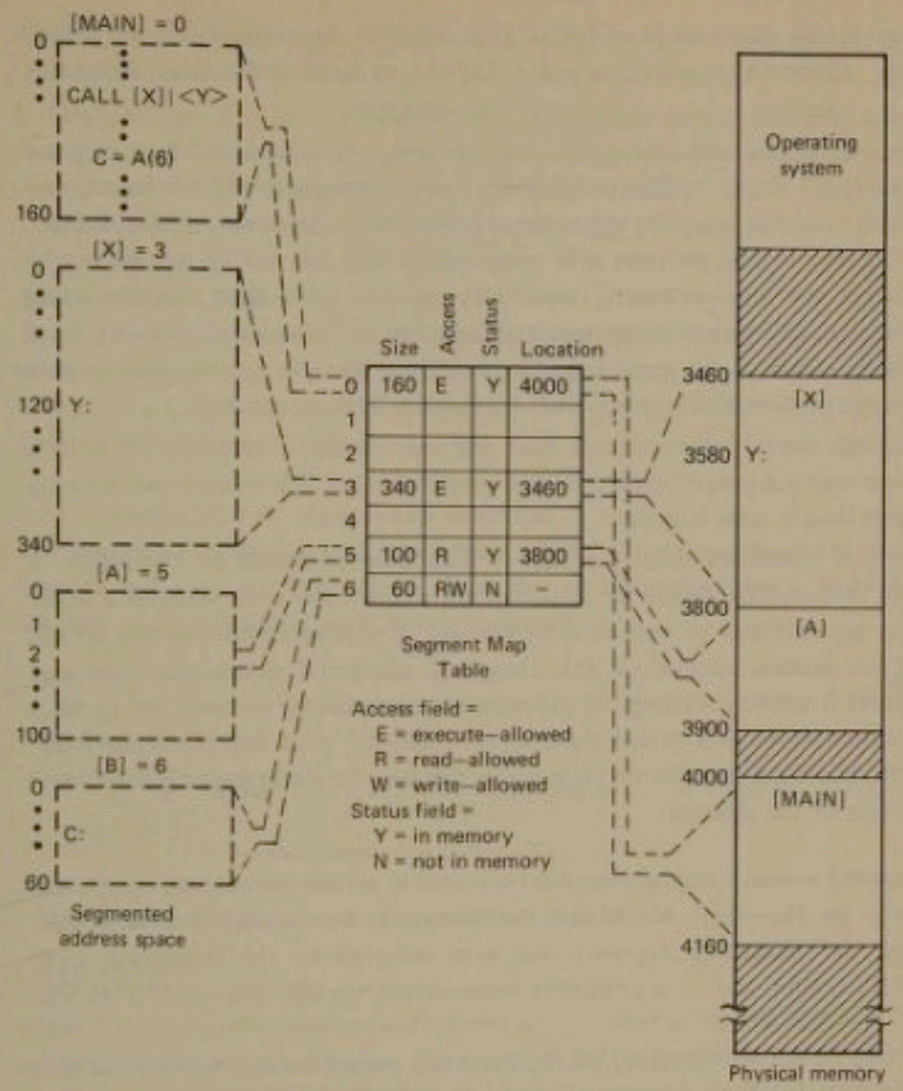


Figure 3-39 Segmented-mapped memory

Segmented memory management can offer several advantages (Dennis, 1965, 1966; Daley, 1969; Randell, 1967; Denning, 1970). It can:

1 *Eliminate fragmentation* By moving segments around, fragmented memory space can be combined into a single free area.

2 *Provide virtual memory* By keeping only the actively used segments in main memory, the job's total address space size may exceed the physical memory size.

3 *Allow dynamically growing segments or automatic bounds checking* If a segment's size must be increased during execution (e.g., table space full), the "out-of-range" reference can be detected by the size component of each

entry in the Segment Map Table. This detection can trigger either dynamic space allocation–increased segment size–or an error indication, depending on the operating system's handling of this condition.

4 *Dynamic linking and loading* In most systems, subroutines and data areas are located and added to the job's address space statically during the linking and loading process prior to the start of execution (Donovan, 1972; Presser, 1972). For a large program with many subroutines this linking process can be complex and time-consuming, sometimes requiring more time than the actual execution. It frequently happens that many of the subroutines linked are not actually called during any particular run. Thus, by deferring the linking until the segment is explicitly referenced, unnecessary linking is avoided.

5 *Facilitate shared segments (data areas and procedures)* If two jobs are using a square root routine, it is wasteful and possibly undesirable to have two separate copies exist in main memory.

6 *Enforced controlled access* Access to each segment should be controlled. A table of constants should be restricted to read accesses only, whereas a work-space segment may be written as well as read. A pure procedure segment should not be written because of the danger of accidental destruction, nor read because it allows "stealing" of procedures; access should be restricted to allow execution only. In a shared segment environment, these key privileges, Read, Write, and Execute, may be assigned on the basis of the segment involved and the user of the segment.

Segmented memory management has been used in certain commercial computers, primarily the Burroughs B5000 and its successors, since the early 1960s. More recently, the segmented approach has been adopted by the Honeywell 6180 MULTICS system, and, to a somewhat lesser extent, the IBM System/370 OS/VS2 system.

The four functions of memory management in a segmented environment combine some of the aspects of relocatable partitioned and demand-paged memory management:

1 Keeping track of status is done through four main sets of tables:
   a Segment Map Tables–one per address space
   b Unallocated Area Table–one in system
   c Active Reference Table–one per address space
   d Active Segment Table–one in system
2 The policy of who gets memory and when may be determined statically by the job scheduler if virtual memory use is limited (i.e., virtual memory size is comparable to physical memory size). In a general virtual memory environment, memory requirements are determined on a dynamic basis by segment "demand" interrupts.

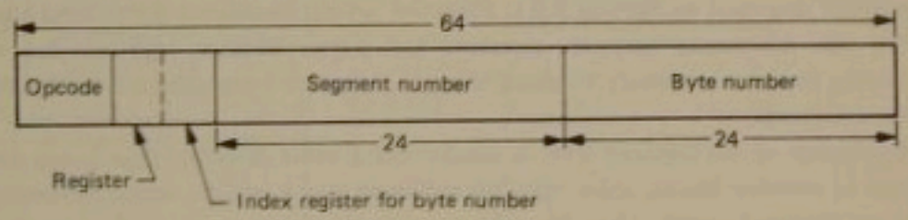
3 Allocation–when a segment must be allocated, a large enough available area must be found. Compaction may be performed, if necessary.
4 Deallocation–if it is not possible to find a large enough available area for segment allocation, one or more of the currently allocated segments must be deallocated. The memory released becomes available space. When a job terminates, all the memory used by its segments becomes available.

3-7.1 Hardware Support

There are numerous variations on the hardware implementation of segmented memory management. We will present the key hardware support concepts in this section. The references provide more details and alternatives (Dennis, 1966, 1967; Daley, 1969; Denning, 1970).

3-7.1.1 FORMATION OF SEGMENTED ADDRESS

In a segmented environment it is necessary to generate a two-component, segment number and byte number address. One way to accomplish this is to have two separate address fields in each instruction, such as:

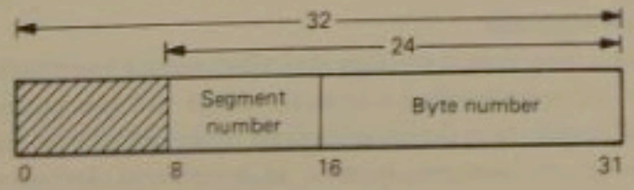


This allows easy formulation of instructions, such as:

LOAD 1,[5]|(8)

where 5 is the segment number and 8 is the byte number within the segment. In order to shorten the length of such an instruction as well as to provide more flexibility, segment base registers and byte base registers can be used, possibly with segment and/or byte offsets in the instruction. This addressing approach is used in the Burroughs 5500 and Honeywell 6180.

Another approach is to generate the effective address, using linear address formation techniques, and to interpret certain bits of the resulting effective address as a segment number and the remaining bits as the byte number. For example, the IBM System/370 can interpret the 24-bit effective address as an 8-bit segment number and a 16-bit byte number as follows:



allowing up to 256 separate segments, each up to 64K bytes long. By using a 32-bit effective address, as provided on certain versions of the IBM System/360 Model 67 (a special model of the System/360 series designed for timesharing usages), it is possible to provide more and larger segments–up to 4096 segments, each up to one million bytes long. The Honeywell 6180, in comparison, allows up to 256K segments, each up to one million bytes.

The latter approach, using a single address field within the instruction, has the advantage (or disadvantage–depending on viewpoint) that the segmented address space can be made to be invisible to the user programs. Due to the single address formation, the last byte of one segment (if it is maximum size) is logically adjacent to the first byte of the next segment. This means that four adjacent 64K segments can be treated as a single 256K segment. This is not usually permitted when there are separate segment and byte address fields in the instruction.

3-7.1.2 ADDRESS MAPPING

The segment address mapping hardware required is similar to the page mapping hardware described in Section 3-6.1. However, several significant differences arise from the differences between segments and pages. Because segments are of arbitrary size, it is necessary to check that the reference byte address is within the segment's range.

Protection at the segment level is usually much more desirable than protecting pages or memory blocks, since segments represent logical entities that are meaningful to the programmer. Thus, the address mapping mechanism must check that access is permitted.

Since segments may be removed from main memory and placed on secondary storage, it is necessary to indicate whether the segment is currently in memory. If it is not, an interrupt to the operating system must occur. To help the operating system decide which segments to keep in memory and which to remove, "referenced" and "changed" bits may be used in each Segment Map Table Entry (SMTE).

Figure 3-39 depicts a general Segment Map Table format and Figure 3-40 indicates the sequence of operation that occurs in the process of address mapping. Specific implementations may omit certain features, such as the access protection or reference/change information, or may include additional features, such as the "ring protection" mechanism of the Honeywell 6180 (Graham, 1968).

3-7.2 Software Algorithm

Many of the operating system software algorithms needed to support segmentation are similar to the corresponding actions associated with demand paging. In this section we will explain how the six advantages of segmented memory management mentioned earlier are attained. It must be understood that many, if not all,

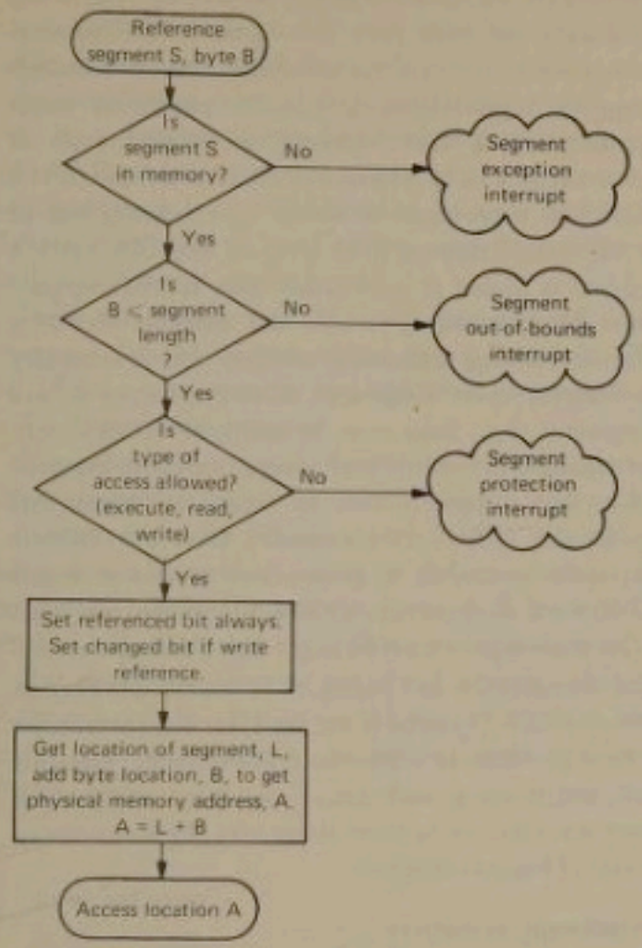


Figure 3-40 Segment address mapping

of these advantages can be gained by using other memory management techniques. However, segmentation, in general, makes the implementation much simpler.

3-7.2.1 ELIMINATION OF FRAGMENTATION

The Segment Map Tables act as multiple relocation/bound registers. By the readjustment of the location field in an entry, a segment may be moved around in memory without any apparent change to the job. In Figure 3-39, the three free areas can be combined into one large area by moving segments X, A, and MAIN.

3-7.2.2 PROVIDING VIRTUAL MEMORY

Virtual memory can be accomplished in a manner similar to demand paging. Copies of all segments are kept on secondary storage. Whenever a segment not in memory is referenced, a *segment exception interrupt* occurs. The operating

segments are shared, the memory requirement drops to 256K + 10 × 32K = 576K—an 80 percent reduction.

3-7.2.6 ENFORCED CONTROLLED ACCESS

If each job had a separate and disjoint address space, then protection of segments would serve primarily to detect programming errors, such as an attempt to alter a procedure segment. Extensive use of sharing, as described above, increases the importance of controlled access. If a pure procedure segment is being shared, we must prohibit any job from modifying this segment—either accidentally or maliciously. In addition, we usually prohibit any job from "reading" a procedure segment. There are two primary reasons for this restriction: (1) an attempt to read a procedure segment usually indicates a programming error and (2) the procedure may be *proprietary* (i.e., it is "owned" by someone who charges for use of that program), in which case we wish to prevent a user of this program from "stealing a copy." If a segment has only *execute access* status, it may be called as a procedure to be executed, but reading and writing (i.e., LOADs and STOREs) are prohibited.

Not only procedures, but also data segments may be shared. For example, a segment containing a table of major cities and their population may be used by several jobs simultaneously. To prevent unintentional alteration or programming error, this segment should have only *read access* status—attempted writing or execution of this segment would be prohibited.

Since the access control information is contained in each Segment Map Table (see Figure 3-39), it is not necessary for each job to have the same access restrictions. If we have a data segment that contains the current stock market prices, we can allow one job *write access* to periodically update the prices while restricting all other jobs to *read access*. Methods of determining the appropriate access permitted to each user are described in more detail in Chapter 6. A conceptually simple scheme shown in Figure 3-47 employs a matrix where one side lists all jobs (or, more typically, users) and the other side lists all segments. The intersection indicates the access privileges of a user to a particular segment. Access control has been the subject of considerable research; an extensive summary can be found in Graham (1972) and Hoffman (1969).

3-7.3 Advantages

The advantages of segmentation have been described in detail earlier in this section. In summary, segmentation (1) eliminates fragmentation, (2) provides virtual memory, (3) allows dynamic segment growth, (4) assists dynamic linking, (5) facilitates shared segments, and (6) enforces controlled access. Most of these features contribute directly to efficient utilization of memory.

Segments \ Users	Donovan	Madnick	Ziering	Bad guy	• • •
SQRT	E	RWE	E	E	
PL/I	RWE	E	E	None	
Stock price	RW	R	R	R	
Payroll	R	RW	R	None	
•					
•					
•					

Figure 3-47 Access control matrix

3-7.4 Disadvantages

Segmentation has many of the same disadvantages as the other advanced memory management schemes: increased hardware cost, processor overhead for address mapping, the need for additional memory space for tables, and increased complexity in the operating system. Additional problems are caused by the following:

1. Considerable compaction overhead is incurred in order to support dynamic segment growth and eliminate fragmentation.
2. There is difficulty in managing variable size segments on secondary storage.
3. The maximum size of a segment is limited by the size of main memory.
4. It is necessary to develop techniques or constraints to prevent segment thrashing (the counterpart of page thrashing).

3-8 SEGMENTED AND DEMAND-PAGED MEMORY MANAGEMENT

One way to gain the logical benefits of segmentation and remove many of the disadvantages is to combine the segmentation and paging mechanisms. Instead of treating each segment as a single contiguous entity, each one can be subdivided into pages. By physically manipulating these pages, rather than the entire segment, problems of compaction, secondary storage handling, and limitation on segment size are removed. This approach has been employed in the IBM System/370 and the Honeywell 6180.

3-8.1 Hardware Support

There are, as usual, numerous variations possible for the hardware implementation. We will briefly describe the System/370 hardware; the Honeywell 6180 is quite similar and is described in Bensoussan (1969), Daley (1968), and Organick (1972).

segments are shared, the memory requirement drops to 256K + 10 × 32K = 576K–an 80 percent reduction.

3-7.2.6 ENFORCED CONTROLLED ACCESS

If each job had a separate and disjoint address space, then protection of segments would serve primarily to detect programming errors, such as an attempt to alter a procedure segment. Extensive use of sharing, as described above, increases the importance of controlled access. If a pure procedure segment is being shared, we must prohibit any job from modifying this segment–either accidentally or maliciously. In addition, we usually prohibit any job from "reading" a procedure segment. There are two primary reasons for this restriction: (1) an attempt to read a procedure segment usually indicates a programming error and (2) the procedure may be *proprietary* (i.e., it is "owned" by someone who charges for use of that program), in which case we wish to prevent a user of this program from "stealing a copy." If a segment has only *execute access* status, it may be called as a procedure to be executed, but reading and writing (i.e., LOADs and STOREs) are prohibited.

Not only procedures, but also data segments may be shared. For example, a segment containing a table of major cities and their population may be used by several jobs simultaneously. To prevent unintentional alteration or programming error, this segment should have only *read access* status–attempted writing or execution of this segment would be prohibited.

Since the access control information is contained in each Segment Map Table (see Figure 3-39), it is not necessary for each job to have the same access restrictions. If we have a data segment that contains the current stock market prices, we can allow one job *write access* to periodically update the prices while restricting all other jobs to *read access*. Methods of determining the appropriate access permitted to each user are described in more detail in Chapter 6. A conceptually simple scheme shown in Figure 3-47 employs a matrix where one side lists all jobs (or, more typically, users) and the other side lists all segments. The intersection indicates the access privileges of a user to a particular segment. Access control has been the subject of considerable research; an extensive summary can be found in Graham (1972) and Hoffman (1969).

3-7.3 Advantages

The advantages of segmentation have been described in detail earlier in this section. In summary, segmentation (1) eliminates fragmentation, (2) provides virtual memory, (3) allows dynamic segment growth, (4) assists dynamic linking, (5) facilitates shared segments, and (6) enforces controlled access. Most of these features contribute directly to efficient utilization of memory.

Segments \ Users	Donovan	Madnick	Ziering	Bad guy	• • •
SORT	E	RWE	E	E	
PL/I	RWE	E	E	None	
Stock price	RW	R	R	R	
Payroll	R	RW	R	None	
⋮					

Figure 3-47 Access control matrix

3-7.4 Disadvantages

Segmentation has many of the same disadvantages as the other advanced memory management schemes: increased hardware cost, processor overhead for address mapping, the need for additional memory space for tables, and increased complexity in the operating system. Additional problems are caused by the following:

1. Considerable compaction overhead is incurred in order to support dynamic segment growth and eliminate fragmentation.
2. There is difficulty in managing variable size segments on secondary storage.
3. The maximum size of a segment is limited by the size of main memory.
4. It is necessary to develop techniques or constraints to prevent segment thrashing (the counterpart of page thrashing).

3-8 SEGMENTED AND DEMAND-PAGED MEMORY MANAGEMENT

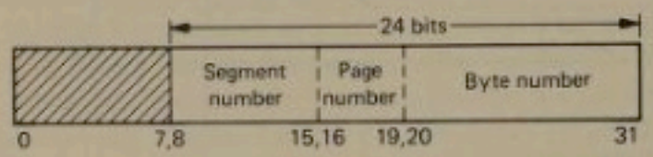
One way to gain the logical benefits of segmentation and remove many of the disadvantages is to combine the segmentation and paging mechanisms. Instead of treating each segment as a single contiguous entity, each one can be subdivided into pages. By physically manipulating these pages, rather than the entire segment, problems of compaction, secondary storage handling, and limitation on segment size are removed. This approach has been employed in the IBM System/370 and the Honeywell 6180.

3-8.1 Hardware Support

There are, as usual, numerous variations possible for the hardware implementation. We will briefly describe the System/370 hardware; the Honeywell 6180 is quite similar and is described in Bensoussan (1969), Daley (1968), and Organick (1972).

The IBM System/370 Dynamic Address Translation (DAT) feature supports either 2K- or 4K-byte pages and segments of maximum size of either 64K or 1024K bytes. The choice of page and segment sizes is determined by control register settings made by the operating system. The following description applies to the 4K-byte page and 64K-byte segment settings.

The 24-bit 370 effective address is split into three parts: (1) segment number, (2) page number, and (3) byte number, as follows:



The hardware thus supports an address of up to 256 segments where each segment may consist of up to 16 pages (64K bytes).

The mapping from virtual address to physical address is accomplished by a series of tables, as depicted in Figure 3-48. The Segment Table Register (control register 1) indicates the location and length of the current Segment Map Table. Each Segment Map Table Entry indicates the location and length of a Page Map Table—unless the segment exception interrupt indicator is set. Each Page Map Table Entry (PMTE) indicates the corresponding block number for each page of that segment—unless the page exception interrupt indicator is set.

To decrease hardware costs as well as save bits in the various table entries, certain restrictions are imposed, as noted in Figure 3-48. For example, the Page Map Table address field in each SMTE should require a full 24-bit address. However, by requiring PMTs to start at an address that is a multiple of 8, only 21 bits are needed in the SMTE (i.e., the last 3 bits of the address are always zeros). The unused bits of each entry provide a convenient means for future changes to the hardware—such as protection bits in SMTEs, reference/change bits in PMTEs, and an increased number of segments by extending the length field in the Segment Table Register.

In order to prevent serious performance degradation due to accesses to the SMT and PMT, a *Translation Lookaside Buffer* (TLB)—an associative scratch pad memory—is automatically used by the hardware to eliminate most SMT and PMT accesses. The use of the TLB can generally keep address translation overhead below 10 percent. (See Section 3-10.3 for a related discussion.)

3-8.2 Software Algorithms

The algorithms presented in both the demand-paged and segmented memory management sections are directly applicable. Only minor modifications are

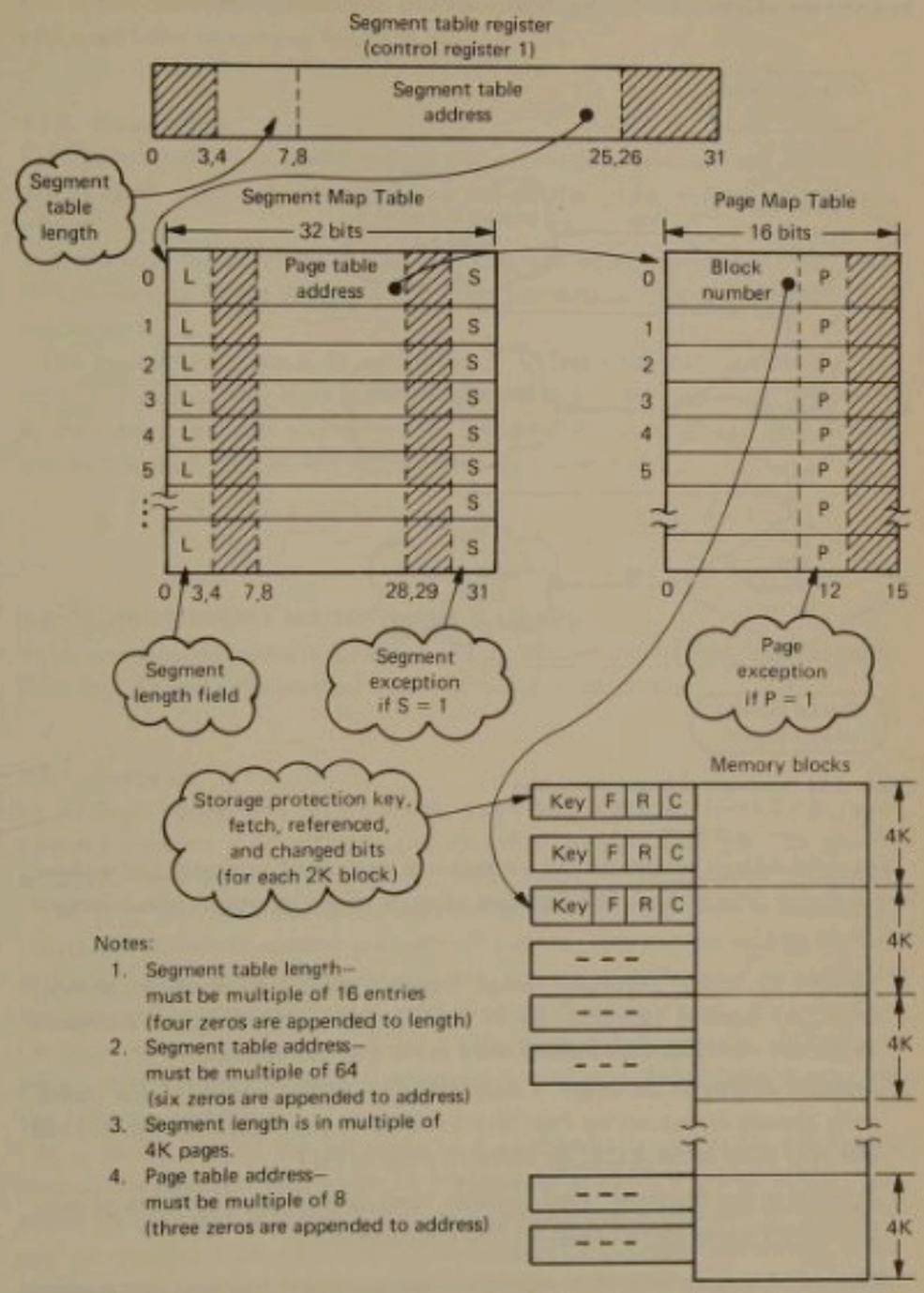


Figure 3-48 System/370 segmentation-page mapping hardware

necessary to eliminate duplication of effort. Figure 3-49 illustrates the interaction between the address translation hardware and the software algorithms.

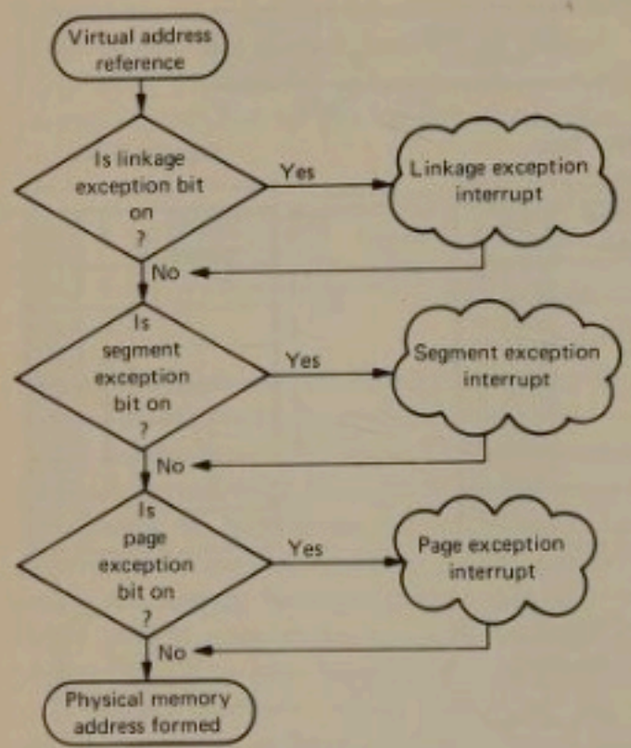


Figure 3-49 Hardware-software interaction

For most address references the translation is performed automatically without any software intervention. The handling of each of the interrupts is summarized below.

1. *Linkage Exception Interrupt* Assign segment number to symbolic segment name, set segment exception bit in SMTE, make entry in Active Reference Table, and convert linkage indirect word to use segment number.
2. *Segment Exception Interrupt* Create entry in Active Segment Table (unless entry already exists), set up Page Map Table (with page exception bits on) and File Map Table in the ASTE, set SMTE to point to PMT.
3. *Page Exception Interrupt* Find available block (removing a page from memory if necessary), load needed page, set PMTE appropriately.

Many of the operations of a segmented demand-paged memory management system closely approximate the operations of information management described in Chapter 6.

3-8.3 Advantages

The combination of segmentation and demand paging provides all the advantages previously listed in Sections 3-6.4 and 3-7.3

3-8.4 Disadvantages

The disadvantages of increased hardware cost, processor overhead, and complexity can be quite significant in a segmented and demand-paged memory management scheme.

In addition, the problems of page breakage and memory needed for SMTs and PMTs persist, as do the dangers of thrashing prevalent in both demand paging and segmentation.

The successful operation of systems such as IBM's OS/VS-2 and Honeywell's MULTICS indicate that these disadvantages can be kept to a minimum. Segmented demand-paged memory management is currently the most general and flexible approach to large-scale general purpose memory management.

3-9 OTHER MEMORY MANAGEMENT SCHEMES

There are many variations to the basic memory management schemes that have been presented. Several key variations will be discussed in this section.

3-9.1 Swapping

In demand paging and segmentation, pages and segments, respectively, were swapped between main memory and secondary storage. This approach can be employed in conjunction with the other memory management techniques.

The early M.I.T. Compatible Time-Sharing System, CTSS, used a basic single contiguous allocation memory management scheme—only one job was in memory at a time. After running for a short period, the current job's entire address space was swapped onto secondary storage to allow another job to run in main memory. (A more detailed discussion of the CTSS memory management algorithm is presented in Chapter 9.) Many contemporary small-scale timesharing systems use this *swapping technique*.

In a similar manner the partitions of a partitioned or relocatable partitioned memory management system can be swapped. In OS/360, if a high priority job arrives for processing and insufficient main memory is available, one or more jobs may be removed from main memory and swapped onto secondary storage. (This process is called *roll-out*.) At a later time, when the high-priority job has completed, the low-priority jobs may be reloaded into main memory. (This is *roll-in*.) The Time-Sharing Option (TSO) of OS/360 uses extensive swapping of user partitions to

provide a CTSS-like facility. Roll-out/roll-in can be awkward in a partitioned environment since the partition must be reloaded into exactly the same location from which it was removed. In a relocatable partitioned environment, as provided by the DEC PDP-10 and Honeywell 6000 computer series, the partition may be reloaded into any available space that is sufficiently large, as long as the relocation base register is correctly set.

Swapping as described in this section is quite crude since the job's entire address space must be swapped back and forth. If job sizes are kept very small, as is often true of timesharing application programs (such as the editor), the swapping overhead may be minimal. For large jobs, the overhead incurred in swapping may be significant.

3-9.2 Overlays

A more refined form of the above swapping technique, which swaps only portions of the job's address space, is called *overlay management*. Overlays, normally used in conjunction with single contiguous, partitioned, or relocatable partitioned memory management, provide essentially an approximation to segmentation—but without the segment address mapping hardware.

Figure 3-50a illustrates the relationship between the procedures in a job's address space. This diagram indicates that procedure A calls only procedures B and C, procedure B calls only procedure F, whereas procedure C calls only D and E. As a result of this relationship, we notice that B never calls C and C never calls B. Thus, procedures B and C do not have to be in memory at the same time. By assigning procedure addresses as shown in Figure 3-50b, instead of using 190K bytes to hold the job's address space in memory, a maximum of only 100K bytes are needed. Whenever an interprocedure reference is made, control is transferred to a special *overlay supervisor* that loads the destination procedure, if necessary.

In most automatic overlay systems, as used in the IBM 1130 Disk Monitor System and OS/360, the user must explicitly state the overlay structure in advance. The swapping of overlays can become excessive if there are many alternating calls to mutually exclusive procedures. For example, a loop such as:

```
DO I = 1 TO 1000;
    CALL B(I);
    CALL C(I);
END;
```

in procedure A would cause procedures B and C to be swapped 1,000 times each.

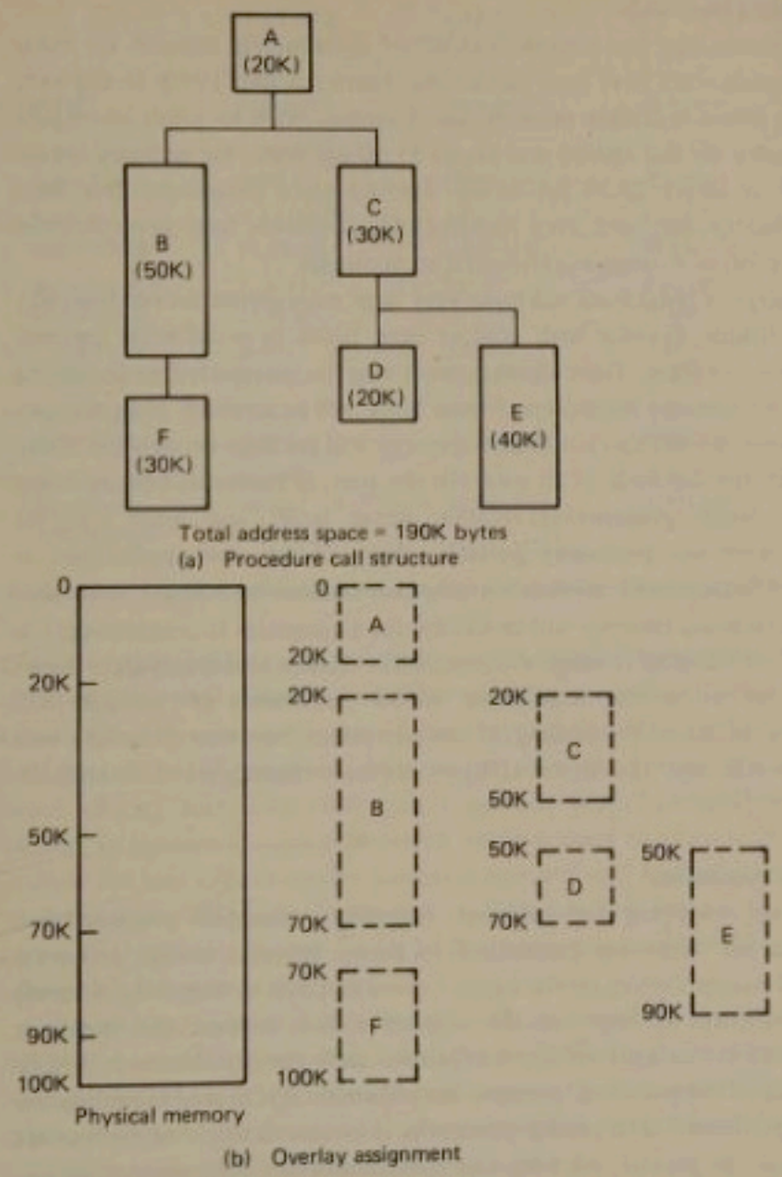


Figure 3-50 Overlay memory management

3-10 FUTURE TRENDS IN MEMORY MANAGEMENT

Computer system technology is a rapidly changing and evolving field. The memory management techniques presented in this chapter will undoubtedly undergo change in the future. In this section some identifiable trends are discussed. The three key topics discussed are: (1) larger main memories, (2) storage hierarchies, and (3) more operating system functions in hardware.

Processor Management

3 ~~4~~

Processor management is concerned with the management of the physical processors, specifically, the assignment of processors to processes. Examining Figure 4-1, we can identify the following modules: the job scheduler, which creates the processes and, in a nonmultiprogrammed environment, would decide which process is to receive a processor; the processor scheduler, which, in a multiprogramming environment, decides which of the ready processes receives a processor, at what time, and for how long; and the traffic controller, which keeps track of the status of the process. There is a similar state diagram for every process in a system. In most systems it is necessary to synchronize processes and jobs, a task that is also performed by modules of processor management.

We discuss three topics in this chapter: job scheduling (Abell, 1970; Browne, 1972; Coffman, 1968), process scheduling–including traffic controller (Baskett, 1970; Hansen, 1971; Kleinrock, 1970; Lampson, 1968; Oppenheimer, 1967; Varney, 1971; Buzen, 1971), and synchronization of jobs and processes (Bernstein, 1969; Dijkstra, 1968; Haberman, 1971; Horning, 1973; Spier, 1969).

Figure 4-1 depicts the domains of job scheduling and process scheduling. The job scheduler can be viewed as a macroscheduler, choosing which jobs will run. The process scheduler can be viewed as a microscheduler, assigning processors to the processes associated with scheduled jobs.

The user views his job as a collection of tasks he wants the computer system to perform for him. The user may subdivide his job into job steps, which are sequentially processed subtasks (e.g., compile, load, execute). The system creates processes to do the computation of the job steps.

Job scheduling is concerned with the management of jobs, and processor scheduling is concerned with the management of processes. In either case, the processor is the key resource.

In a nonmultiprogramming system no distinction is made (and none need be) between process scheduling and job scheduling, as there is a one-to-one correspond-

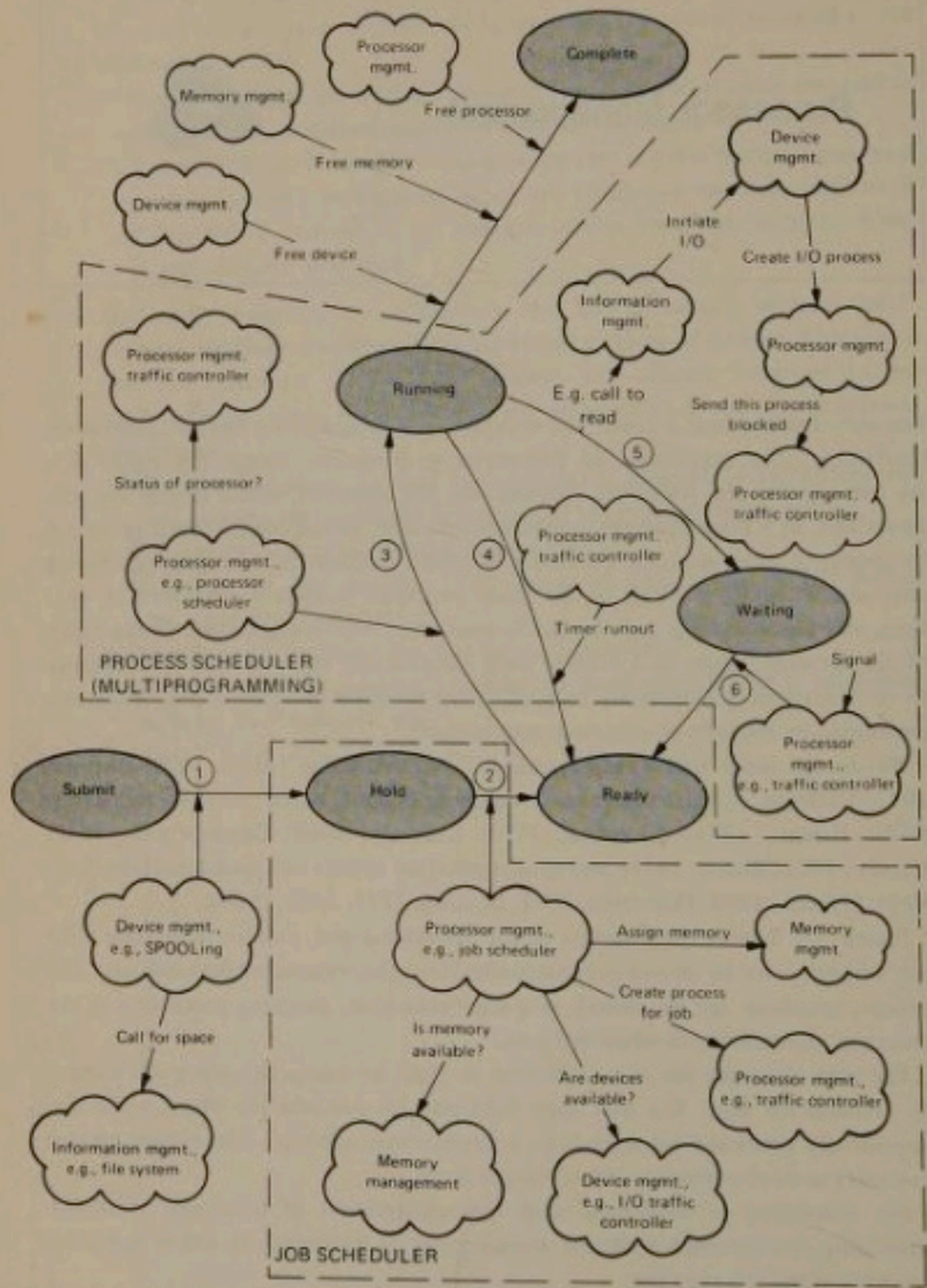


Figure 4-1 Process view of an operating system

ence between a job and its process and only one job is allowed in the system at a time. In such a simple system the job scheduler chooses one job to run. Once it is chosen, a process is created and assigned to the processor.

For more general multiprogramming systems, the *job scheduler* chooses a small subset of the jobs submitted and lets them "into" the system. That is, the job scheduler creates processes for these jobs and assigns the processes some resources. It must decide, for example, which two or three jobs of the 100 submitted will have any resources assigned to them. The *process scheduler* decides which of the processes within this subset will be assigned a processor, at what time, and for how long.

4-1 STATE MODEL

Let us pretend we are a job and follow our path through the states of Figure 4-1.

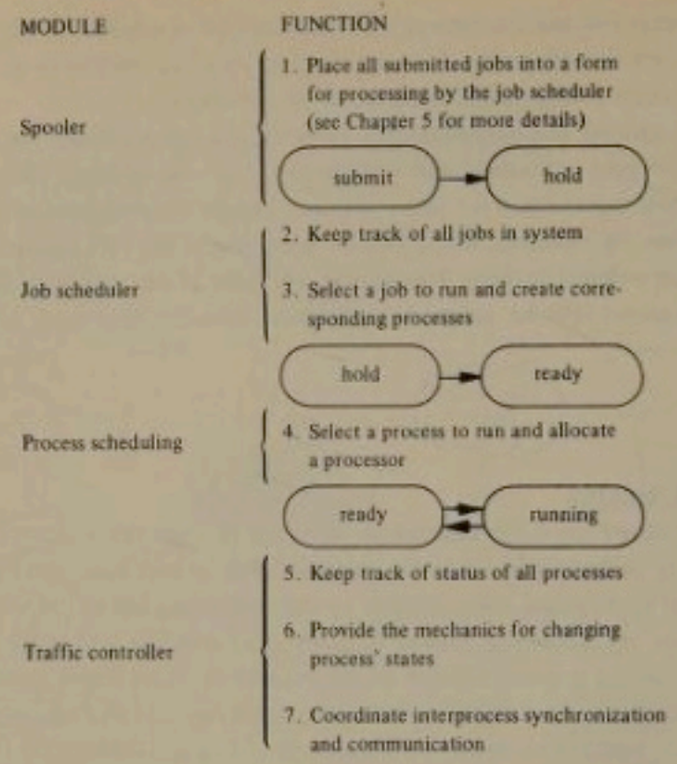
My friendly programmer wrote my program code several days ago. He submitted my code and appropriate control cards to the computing center (*submit state*). An operator took my code and caused it to be read into the computer and copied onto a disk where a job scheduler could examine it (*hold state*). Eventually, the job scheduler chose me to run and set up a process for me (*ready state*). To get my process into a ready state faster on a manually scheduled system, my friendly programmer might have inserted a green card (a five-dollar bill) in front of my deck. After being scheduled and set up by the job scheduler, my process was finally assigned a processor to execute my code (*running state*) by the process scheduler module. At that time I was given a "time quantum," i.e., a time limit for running. When that time quantum was up, I still was not completed. Therefore, my process was sent back into the ready state.

Sometime later the process scheduler gave my process another time quantum and assigned a processor (running state). During this time quantum my process almost finished but still had to print my answer. Therefore, my process initiated another process, an I/O process, and had to wait (*blocked state*) until my I/O was completed.

My I/O process has a similar state diagram—it became *ready* when my CPU process initiated I/O. When the I/O process finished, it signaled my blocked process that it was through.

When the operating system received the signal that my I/O process was finished, the traffic controller module changed my CPU process from the blocked state to the ready state again. This time when the process scheduler assigned a processor to my process (running state), I finished all my work (*completed state*).

We can identify the following modules of processor management.



4-1.1 Job Scheduler

We have shown the process view of an operating system in Figure 4-1, noting the scope of job scheduling. The job scheduler is the "super" manager, which must:

1 Keep track of the status of all jobs. It must note which jobs are trying to get some service (hold state) and the status of all jobs being serviced (ready, running, or blocked state).

2 Choose the policy by which jobs will "enter the system" (i.e., go from hold state to ready state). This decision may be based on such characteristics as priority, resources requested, or system balance.

3 Allocate the necessary resources for the scheduled job by use of memory, device, and processor management.

4 Deallocate these resources when the job is done.

4-1.2 Process Scheduling

Once the job scheduler has moved a job from *hold* to *ready*, it creates one or more processes for this job. Who decides which processes in the system get a processor, when, and for how long? The process scheduling modules make these decisions.

Specifically, the following functions must be performed:

1 Keeping track of the status of the process (all processes are either running, ready, or blocked). The module that performs this function has been called the *traffic controller* (Saltzer, 1966).

2 Deciding which process gets a processor and for how long. This is performed by the *processor scheduler*.

3 Allocation of a processor to a process. This requires resetting of processor registers to correspond to the process' correct state. This task is performed by the traffic controller.

4 Deallocation of a processor, such as when the running process exceeds its current quantum or must wait for an I/O completion. This requires that all processor state registers be saved to allow future reallocation. This task is performed by the traffic controller.

4-1.3 Job and Process Synchronization

On the job level there are usually mechanisms for passing conditions between job steps. For example, we might want to prevent job step 4 from being performed if job step 1 had failed.

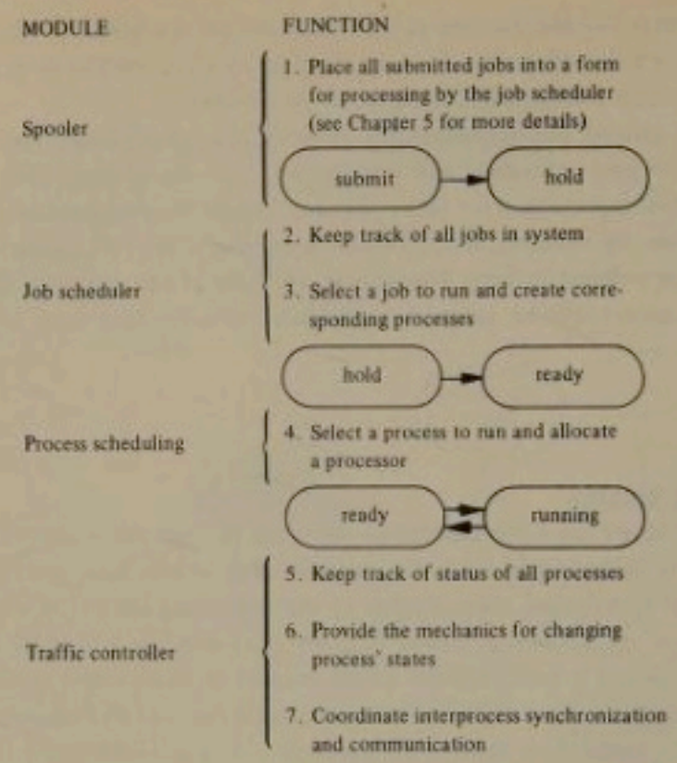
On the process level there must be mechanisms to prevent race conditions. A race condition exists when the result of a computation varies, depending on the timing of other processes. For example, one process requests a printer while another process is printing; if the printer were to be also assigned to the second process, the output would be intermixed between the two processes. We must have a mechanism to stop the second process from running until the first process releases the printer.

The P and V operators and semaphores are one set of mechanisms for coordinating the assignment of processors to processes (Dijkstra, 1968). In addition, we must be aware of possible deadlock situations in cases where there are two processes, each of which is waiting for resources that the other has and will not give up. In this situation, no processor can be assigned to either process.

4-1.4 Structure of Processor Management

The three main sections of this chapter, job scheduling, process scheduling, and synchronization, are all concerned with the assignment of processors to processes. Hence, they are all included in this chapter on processor management. In some cases, especially with job scheduling, interaction with the other resource managers takes place.

In summary, processor management operates on two levels—assigning processors to jobs and assigning processors to processes.



4-1.1 Job Scheduler

We have shown the process view of an operating system in Figure 4-1, noting the scope of job scheduling. The job scheduler is the "super" manager, which must:

1 Keep track of the status of all jobs. It must note which jobs are trying to get some service (hold state) and the status of all jobs being serviced (ready, running, or blocked state).

2 Choose the policy by which jobs will "enter the system" (i.e., go from hold state to ready state). This decision may be based on such characteristics as priority, resources requested, or system balance.

3 Allocate the necessary resources for the scheduled job by use of memory, device, and processor management.

4 Deallocate these resources when the job is done.

4-1.2 Process Scheduling

Once the job scheduler has moved a job from *hold* to *ready*, it creates one or more processes for this job. Who decides which processes in the system get a processor, when, and for how long? The process scheduling modules make these decisions.

Specifically, the following functions must be performed:

1 Keeping track of the status of the process (all processes are either running, ready, or blocked). The module that performs this function has been called the *traffic controller* (Saltzer, 1966).

2 Deciding which process gets a processor and for how long. This is performed by the *processor scheduler*.

3 Allocation of a processor to a process. This requires resetting of processor registers to correspond to the process' correct state. This task is performed by the traffic controller.

4 Deallocation of a processor, such as when the running process exceeds its current quantum or must wait for an I/O completion. This requires that all processor state registers be saved to allow future reallocation. This task is performed by the traffic controller.

4-1.3 Job and Process Synchronization

On the job level there are usually mechanisms for passing conditions between job steps. For example, we might want to prevent job step 4 from being performed if job step 1 had failed.

On the process level there must be mechanisms to prevent race conditions. A race condition exists when the result of a computation varies, depending on the timing of other processes. For example, one process requests a printer while another process is printing; if the printer were to be also assigned to the second process, the output would be intermixed between the two processes. We must have a mechanism to stop the second process from running until the first process releases the printer.

The P and V operators and semaphores are one set of mechanisms for coordinating the assignment of processors to processes (Dijkstra, 1968). In addition, we must be aware of possible deadlock situations in cases where there are two processes, each of which is waiting for resources that the other has and will not give up. In this situation, no processor can be assigned to either process.

4-1.4 Structure of Processor Management

The three main sections of this chapter, job scheduling, process scheduling, and synchronization, are all concerned with the assignment of processors to processes. Hence, they are all included in this chapter on processor management. In some cases, especially with job scheduling, interaction with the other resource managers takes place.

In summary, processor management operates on two levels—assigning processors to jobs and assigning processors to processes.

On the job level, processor management is concerned with such questions as which jobs will be run and which will run first. At this first level processor management is not concerned with multiprogramming. It assumes that once a job is scheduled, it will run. Job scheduling can run concurrently with other users' programs or other system functions. Thus job scheduling may itself be a separate process, as shown in Figure 4-2.

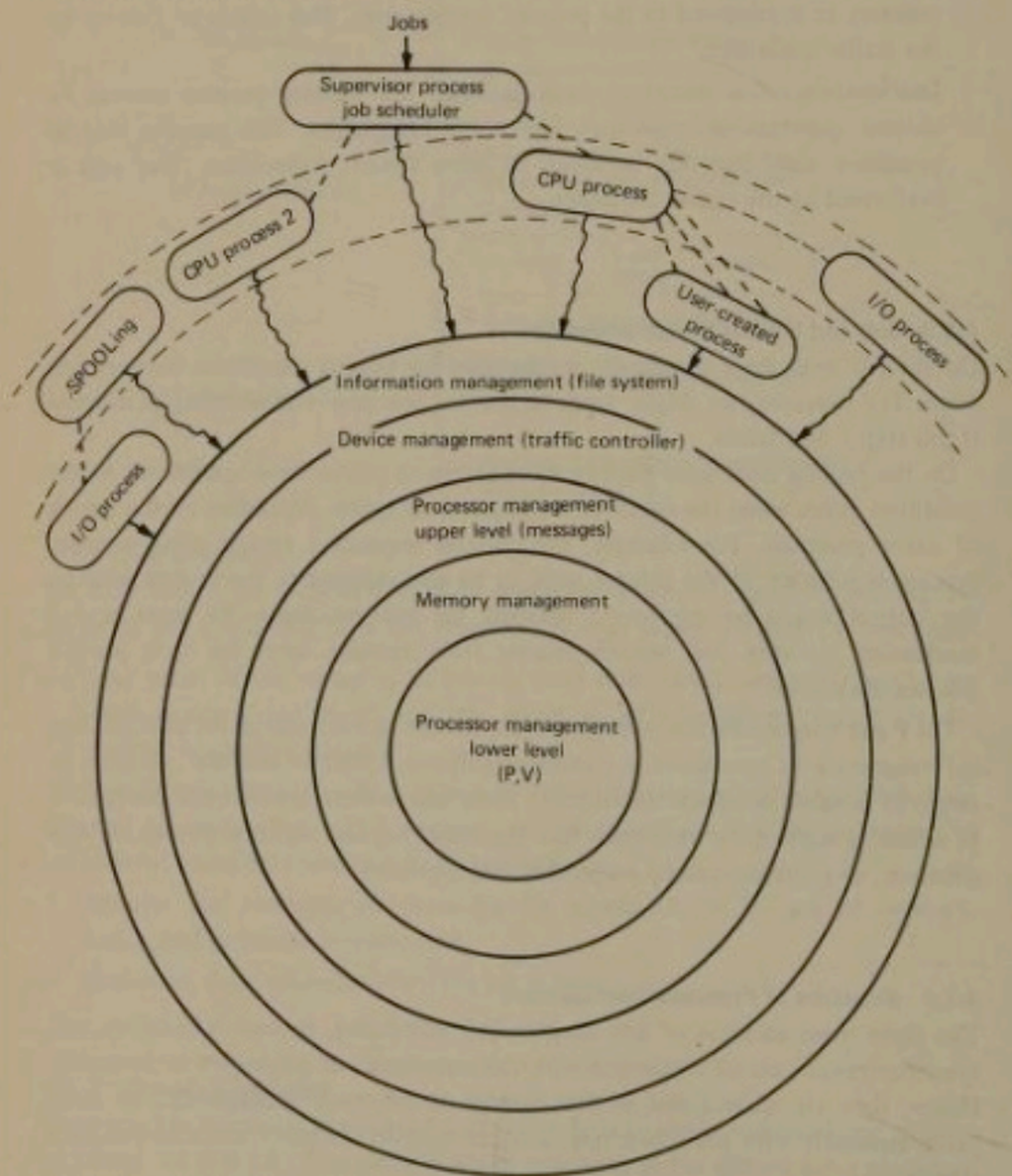


Figure 4-2 Hierarchical view of computer-based operating system levels for a real or extended machine

Once a job is scheduled, the system must perform the functions of creating processes, destroying processes, and sending messages between processes. In a general system these functions may be requested by users' processes, job scheduler, and other processes. Thus the functions may be common to all address spaces (see Figure 4-2–Processor Management Upper Level).

In a multiprogramming environment the process scheduler and the synchronization mechanisms may be called by all modules of the system. Thus they form the very center of the kernel of a system (Figure 4-2–Processor Management Lower Level). This model of system structure is the basis for the sample operating system presented in Chapter 7.

The job scheduler is like the coordinator of a contest; he decides who will enter the contest but not who will win it. The process scheduler decides which participant will win. In a nonmultiprogramming environment only one contestant is allowed to enter.

4-2 JOB SCHEDULING

The functions of the job scheduler may be done quite simply, quite elaborately, or dynamically.

In a timesharing system such as the Compatible Time Sharing System (CTSS) (Crisman, 1965), the job scheduling policy may consist of admitting the first 30 users that log in (i.e., arrive). A simple two-level priority algorithm allows a user with a higher priority to force the logout (i.e., termination) of a lower priority user. The user to be logged out is chosen on the basis of the processor time he has used: the one who has used the most processor time gets logged out.

The job scheduling algorithm of a batch system such as OS/VS-2 can be sophisticated. It takes into account not only the time a job arrives but also priority, memory needs, device needs, processor needs, and system balance.

In this section we discuss some simple job scheduling policies. (In Chapter 9 we discuss the sophisticated job schedulers of some other systems.) We focus on the policies of job scheduling and their implications, dealing first with systems without multiprogramming, then with more general multiprogramming systems.

4-2.1 Functions

The job scheduler can be viewed as an overall supervisor that assigns system resources to certain jobs. Therefore, it must keep track of the jobs, invoke policies for deciding which jobs get resources, and allocate and deallocate the resources.

We have chosen not to spend a great deal of time on extremely detailed mechanisms for performing the functions of job scheduling. The reader may

refer to Chapters 7 and 9 for as much detail regarding a particular scheduler as he wishes (right down to the code). We are mainly concerned here with the factors that go into scheduling, and the consequences that result. For example, if job scheduling is done by reference to memory needs only, what is the effect on turnaround time as compared with other scheduling policies?

Before discussing different policies, let us briefly address ourselves to the function of keeping track of jobs in a system. One mechanism for keeping track of jobs is to have a separate Job Control Block (JCB) for each job in the system. When a process is placed in hold state (Figure 4-1), a JCB is created for it with entries regarding its status and position in a job queue.

Job identification
Current state
Priority
Time estimate
etc.

Figure 4-3 Job Control Block

Figure 4-3 illustrates the type of information that may be contained in a Job Control Block. Some information, such as priority and time estimate, is obtained from the job control cards submitted with the job. Other information, such as current state, is set by the operating system (see Section 5-6.3 for further discussion of the JCB).

4-2.2 Policies

The job scheduler must choose from among the "hold" jobs those that will be made "ready" to run. In a small computing center this function may be done by an operator. He may choose arbitrarily, choose his friends, or choose the shortest job. In larger systems, e.g., OS/360, all submitted jobs may be first stored on a secondary storage device whereupon the job scheduler can examine all such jobs. It can then choose which jobs will have system resources committed to them and hence will be run.

We should note the following:

> The key concept is that there are more jobs that wish to run than can be efficiently satisfied by the resources of the system.
>
> In particular:
>
> a most resources are finite (e.g., tapes, memory)
>
> b many resources cannot be shared or easily reassigned to another process.

Scheduling is considered a policy issue because its goals are usually subjective (and sometimes contradictory). For example, how would you rank the following goals?

1 Running as many jobs as possible per day (only run short jobs)
2 Keeping the processor "busy" (only run computation-intensive long jobs)
3 Fairness to all jobs (what does "fair" mean?)

Typical considerations one must deal with in determining a job scheduling policy are:

1 Availability of special limited resources (e.g., tapes)
   a If you give preference, someone can "cheat." For example, if you always first run the job that requests a plotter, then some users will always request a plotter.
   b If you don't give preference, some user suffers an extra delay (e.g., must wait for tape to be mounted or plotter adjusted after having waited to be scheduled).
2 Cost-higher rates for faster service.
3 System commitments–processor time and memory–the more you want, the longer you wait.
4 System balancing–mixing I/O-intensive and CPU-intensive (at M.I.T., for example, you specify A-, B-, or C-type job).
5 Guaranteed service–setting a specific waiting time limit (1 hour) or a general limit (within 24 hours). (At least don't lose the job!)
6 Completing the job by or at a specific time (e.g., 5 PM).

Once the job scheduler has selected a collection of jobs to run, the process scheduler attempts to handle the microscopic scheduling (dynamic assignment of processor to processes). Even ignoring time quanta limits, a process will typically run less than 100 ms before becoming blocked to await I/O completion or a similar event.

The job and process scheduler may interact. The process scheduler may choose to "postpone" ("roll out") a process and require that it go through the macro-level job scheduling again in order to complete. This is especially true in a timesharing system as discussed in Section 4-6. It is impossible to expose all the subtleties of the above considerations. In the following sections we present a few simple examples.

4-2.3 Job Scheduling in Nonmultiprogrammed Environment

In this section let us assume no multiprogramming; that is, once a process has been assigned a processor, it does not release the processor until it is finished. We will first examine a simple example of scheduling jobs using a policy of trying to

JOB	RUN TIME	ARRIVAL	FINISH	TURNAROUND TIME	WEIGHTED TURNAROUND TIME
1	0.3	10.0	10.4	0.4	1.33
2	0.5	10.2	11.1	0.9	1.80
3	0.1	10.4	11.4	1.0	10.00
4	0.4	10.5	11.3	0.8	2.00
5	0.1	10.8	11.3	0.5	5.00
				3.6	20.13

Turnaround time, T = .72
Weighted turnaround time, W = 4.026

Figure 4-19c Multiprogramming with memory and tape needs: summary needs

4-2.5 Job Scheduling Summary

Minimal turnaround time is a rather common objective of a job scheduling algorithm. However, in actual systems we often find it necessary to modify this policy. Reasons might include the following:

1. Some "priority" concept can give certain jobs favored treatment—often for an extra charge.
2. Total throughput must be sufficient to get a day's work done in less than 24 hours. (Recall that some turnaround time techniques would actually let the CPU idle.)

In summary, we have tried to show that the designer must first determine his goals, e.g., low average turnaround, and then experiment with different scheduling algorithms for what he feels are representative job mixes.

Throughout Section 4-2 we tried different scheduling algorithms. The first three were tried with the same job mix. Some conclusions for this job mix would be:

1. First-In First-Out reduces the spread of wait times for each job.
2. Shortest Job First gives a low average turnaround time but clearly favors the short job.
3. Future Knowledge gives the best turnaround time but is difficult to implement.

We showed that in a multiprogramming environment certain job mixes had better turnaround times and other mixes did not. We included a more complicated analysis of scheduling in a multiprogramming environment with I/O overlap. Also taken into consideration were memory and device needs. Other job scheduling algorithms can be found in the case studies in Chapter 9.

4-3 PROCESS SCHEDULING

Section 4-2 discussed job scheduling, which is transition 2 of Figure 4-1. Process scheduling is concerned with transitions 3 and 4.

Once the job scheduler has selected a collection of jobs to run, the process scheduler attempts to handle the microscopic scheduling (i.e., the dynamic assignment of processor to process). The process scheduler is also called the *dispatcher* or *low-level scheduler* in the literature. In a multiprogrammed environment, a process typically uses the processor for only 100 ms or less before becoming blocked to wait for I/O completion or some other event.

In our multiprogramming examples of Section 4-2 we assumed a rather simple round-robin process scheduling algorithm where the processor is shared equally among jobs in memory. However, for reasons discussed in this section, we may wish to use more elaborate process scheduling algorithms.

The job and process scheduler may interact. The process scheduler may choose to postpone, or roll out, a process and require that it go through the macro-level job scheduling again in order to complete. This is especially true in a timesharing system.

In Figure 4-20 we see a single processor multiprogramming system. Processes A and D both request the use of a tape drive during execution. Let us assume there is only one tape drive available, and process A requests it first. The drive is assigned to process A. Later, when process D requests a tape drive, it must be blocked until process A releases the tape drive.

Being blocked is an attribute of the process, not the processor. It means that the processor will switch only among processes A, B, and C, and ignore process D. Usually process D will tie up a portion of memory, but in some systems it can be rolled out or paged out of memory onto secondary storage. In such a case, the blocking of process D has a logical effect that results in a redistribution of resources; the resources need not be wasted. This is an example of processor scheduling considerations that result from process synchronization requirements.

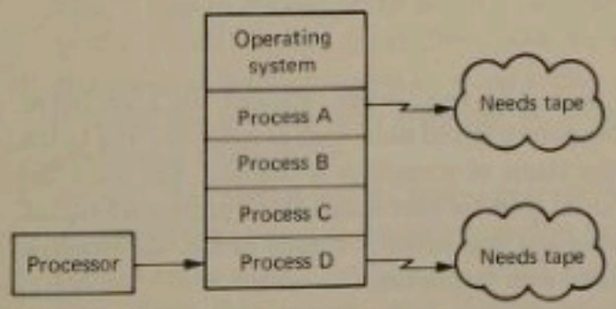


Figure 4-20 Single processor interlock

4-3.1 Functions

The process scheduler must perform the following functions:

1 Keep track of the state of processes.
2 Decide which process gets a processor, when, and for how long.
3 Allocate processors to processes.
4 Deallocate processors from processes.

As in the previous section, we are reluctant to get involved with the very detailed implementation techniques of processor scheduling, as they are highly machine-dependent and are not important to the concepts. The reader wishing to study the details should refer to the processor scheduling technique of the sample operating system contained in Chapter 7 or the case studies in Chapter 9. Here we will focus on policies.

Let us briefly discuss the function of keeping track of the status of the process. The module of the processor scheduler that performs these functions is called the traffic controller. One way of keeping track of the status of a process is to use a database associated with each process in the system called a Process Control Block (PCB). There is a separate PCB for each process; in fact, the PCB is the only really tangible part of a process (see Figure 4-21).

Process identification
Current state
Priority
Copy of active registers
Pointer to list of other processes in same state
etc.

Figure 4-21 Process Control Block

Frequently all PCBs in the same state (e.g., ready, blocked, etc.) are linked together; the resulting list is often given a special name such as *ready list*. The traffic controller is called whenever the status of a resource is changed. The block list may be further subdivided, one chain given for each reason that a process is blocked. Thus if a process requests a device and that device is already in use, the process is linked to the block chain associated with that device.

When the device is released, the traffic controller checks its block chain to see if any processes are waiting for that device. If so, these processes are placed back in ready state to rerequest the device. Alternatively, one of the processes may be assigned the device and placed in ready state, while all the other processes waiting for that device remain blocked.

4-3.2 Policies

The scheduling policy must decide which process is to be assigned a processor and for how long. The length of time a process is assigned a processor may depend on one, or some combination of, the following events:

The process is complete.
The process becomes blocked.
A higher priority process needs the processor.
A time quantum has elapsed.
An error occurs.

The decision as to which process is to run is made by scanning the PCB ready list and applying some policy.

The ready list can be organized in two ways:

1 Each time a process is put on the ready list (from hold, running, or blocked) it is put in its "correct" priority position in order. Then, when the processor becomes free, the top process on the list is run.
2 Processes are arbitrarily placed on the ready list. When it is necessary to find a process to run, the scheduler must scan the entire list of ready processes and pick one to run.

Typical process scheduling policies include the following:

1 *Round robin.* Each process in turn is run to a time quantum limit, such as 100 ms.
2 *Inverse of remainder of quantum.* If the process used its entire quantum last time, it goes to the end of the list. If it used only half (due to I/O wait), it goes to the middle. Besides being "fair," this works nicely for I/O-bound jobs.
3 *Multiple-level feedback variant on round robin.* When a new process is entered, it is run for as many time quantums as all other jobs in the system. Then regular round robin is run.
4 *Priority.* The highest priority ready job is selected. Priority may be purchased or assigned (e.g,. a nuclear reactor process may be run first).
5 *Limited round robin.* Jobs are run round robin until some fixed number of times. Then they are run only if there are no other jobs in the system.
6 *System balance.* In order to keep the I/O devices "busy," processes that do a lot of I/O are given preference.

Since

$$\omega = \frac{\lambda}{\mu+\lambda}$$

we can substitute

$$\left(\frac{\lambda}{\mu}\right) = \left(\frac{\omega}{1-\omega}\right)$$

to get:

$$P_n = \frac{\left(\frac{\omega}{1-\omega}\right)^n}{n! \sum_{i=0}^{n} \left(\frac{\omega}{1-\omega}\right)^i \frac{1}{i!}} \qquad [4\text{-}5]$$

which is the result used originally in Section 3-2.2 and tabulated in Figure 4-14.

The reader must recall all the conditions on this formulation. We assumed that all processes had the same I/O wait percentage, ω, and that all processes were given equal treatment, as in round-robin scheduling. More complex situations can be modeled to determine performance, though the mathematics may become quite complex. (See Arden, 1969; Baskett, 1971; Buzen, 1973; Coffman, 1968; Hellerman, 1970; Kleinrock, 1970; McKinney, 1969).

4-4 MULTIPROCESSOR SYSTEMS

To enhance throughput, reliability, computing power, parallelism, and economies of scale, additional processors can be added to some systems. In early multiprocessor systems the additional processors had specialized functions, e.g., I/O channels. Later multiprocessing systems evolved to include the concept of one large CPU and several peripheral processors (CDC 6600). These processors may perform quite sophisticated tasks, such as running a display. A more common type of multiprocessing is a system having two or more processors, each of equal power, e.g., HIS 6180, IBM 158MP and 168MP, UNIVAC 1108, and Burroughs 6700. There is also the computer network, in which many different computers are connected, often at great distances from one another. Let us focus on the multiprocessor system with many CPUs, each of equal power.

There are various ways to connect and operate a multiprocessor system, such as (1) separate systems, (2) coordinated job scheduling, (3) master/slave scheduling, and (4) homogeneous scheduling. These techniques differ in degree of scheduling sophistication; each will be discussed in turn.

4-4.1 Separate Systems

Some systems, for example, the IBM System/360 Model 67, can be logically subdivided into two or more separate systems, each with one processor, some main memory, and peripheral devices. This is just like having two or more separate computing systems, all in the same room.

The advantage to this organization is that processors, memories, and I/O devices can be easily, though manually, switched. For example, the ensemble can be configured to be two systems, each with one processor, 512K bytes of memory, and 8 disk drives, or one system with one processor, 1024K bytes of memory, and 16 disk drives (second processor is unused); other such alternatives exist.

This configuration flexibility is useful if there are some jobs that require the full complement of memory and/or I/O device resources. Alternatively, if one processor is being repaired, all the other resources can be pooled into one large system rather than be allowed to sit idle.

In this separate system situation, there is no job or process scheduling between the processors except that which is accomplished manually.

4-4.2 Coordinated Job Scheduling

A variation on the separate system multiprocessor technique is the *coordinated job scheduling* approach (this is also called *loosely coupled multiprocessing*). Each processor is associated with a separate system. When a job arrives, it may be assigned to any system. The assignment of a job to a system may be based on a variety of requirements and policies, such as assigning the job to the system with the lightest load.

To accomplish this balancing, all job scheduling must be coordinated. This may be done manually, by a special-purpose computer (as done on the Octopus system at the Lawrence Radiation Laboratory), or by one of the actual processing systems (as done on IBM's OS/VS-2 Job Entry System).

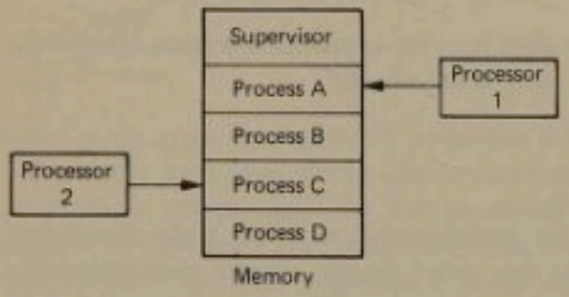
4-4.3 Master/Slave Scheduling

The permanent assignment of a job to a system, as in the coordinated job scheduling technique, cannot handle the short-term balancing of resource demand. In a general multiprocessor system all memory and I/O devices are accessible to all processors. Memory and I/O devices are assigned to the processes, not to the processors. (This is also called *tightly coupled multiprocessing*). The processors are assigned to execute processes as determined by the process scheduler—except that there are now multiple processors available for assignment.

In a master/slave scheduling technique, one processor maintains the status of all processes in the system and schedules the work of all the slave processors.

This is analogous to the relationship between the CPU and I/O processors presented in Chapter 2. For example, the master processor selects a process to be run, finds an available processor, and issues a Start Processor instruction. The slave processor starts execution at the indicated memory location. When the slave encounters an exceptional event, such as an I/O request, it generates an interrupt to the master processor and stops to await further orders. Note that different slave processors may be assigned to a process at different times.

Figure 4-25 illustrates the manner in which multiple processors may be used for multiprogramming. Usually we visualize several separate processes as being in memory. In actuality, a process address space is often paged so that only part of it is in memory at one time; this allows a large number of processes to be active in the system.



Processor 1 working on Process A

Processor 2 working on Process C

Figure 4-25 Multiprogramming with multiple processors

4-4.4 Homogeneous Processor Scheduling

The master slave scheduling approach, besides being conceptually undemocratic, has several performance and operational disadvantages. Under heavy scheduling loads, the master processor may become overloaded and cause a major bottleneck. A more democratic approach is to treat all processors equally, both as masters and slaves.

Under such conditions, the processor scheduling is logically decentralized. A list of processes and their status are stored so as to allow any processor to have access to the list—e.g., a common shared area of memory is used. Whenever a process is stopped due to I/O wait or timer limit, etc., its processor goes to the process state list, updates the process' status, and finds another process to run. Each processor uses the same scheduling algorithm to select the process to run next. If all processors are handled identically, no policy differences need exist with regard to a single processor or a multiple processor operating system.

It is necessary to establish good coordination and communication among processors in a decentralized operating system. The software lockout example presented in Section 4-5.4 illustrates this point.

4-5 PROCESS SYNCHRONIZATION

The problem of process synchronization arises from the need to share resources in a computing system. This sharing requires coordination and cooperation to ensure correct operation.

An example may illustrate some of these points. Figure 4-26 illustrates a typical rapid-service hamburger stand. You submit your order to the order clerk. He places your order on a turntable. The bagger takes orders off the turntable, one at a time, and puts the requested items into a bag. He then delivers the completed order to the cashier. The cashier processes these bags, one at a time, receiving money from you and giving you your order. Furthermore, there is a cook working in the back room, continually replenishing the supply of cooked hamburgers used by the bagger. In this example, each process (person) operates largely independently. The only communication is via certain shared areas: the *order list turntable*, the *cooked hamburger bin*, and the *checkout counter*.

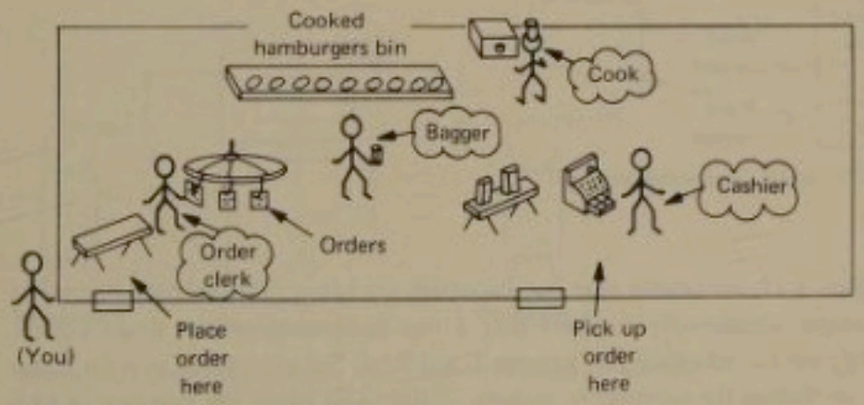


Figure 4-26 Operation of a hamburger stand

This simple example illustrates forms of communication and cooperation that we frequently take for granted. The bagger can fill orders only when there are orders to be processed. What does he do if there is a slack period and the order list is empty? He goes to sleep. How is he awakened when a customer arrives and places an order? The order clerk kicks him! It is a useful exercise to look for other necessary forms of communication and cooperation. For example, what prevents the cook from burying the stand under a pile of hamburgers 100 feet deep during a long slack period?

Analogous situations exist in a computing system among the processes. In some cases the coordination is forced upon them by the operating system because of the scarcity of resources, for example, the need to wait for access to an I/O channel. In other cases, a single job may consist of several processes that all work together. An information retrieval system, such as an airline reservation system, may have several processes performing separate tasks, as in our hamburger stand example. One task may accept requests, check for input correctness, and pass the request along to the "worker" process. The worker process accesses the data to find the necessary information and sends it to the "output" process. The output process routes the response back to the original requester.

Associated with processor allocation and interprocess communication are two synchronization problems, race condition and deadly embrace.

4-5.1 Race Condition

A *race condition* occurs when the scheduling of two processes is so critical that the various orders of scheduling them result in different computations. Race conditions result from the explicit or implicit sharing of data or resources among two or more processes. Figure 4-27 illustrates a simple form of race.

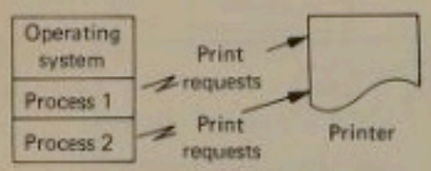


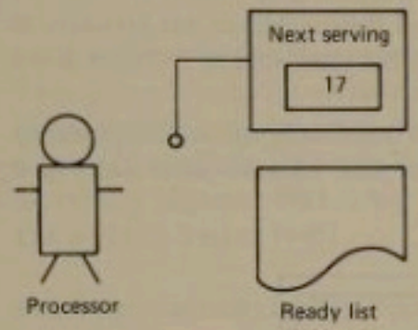
Figure 4-27 Simple race condition

In Figure 4-27 we assume that two processes are being run (multiprogrammed). Each process occasionally requests that a line be printed on the single printer. Depending on the scheduling of process 1 and 2, all the printout of process 1 may precede or follow the printout of process 2. But most likely the printout of each process will be interspersed on the printed paper.

One solution to this predicament is to require that a process explicitly *request* use of the shared resource (the printer in this case) prior to using it. When all use of the resource is completed (e.g., all printout completed), the resource may then be released by the process. The operations *request* and *release* are usually part of the operating system facilities handled by the traffic controller. If a process requests a resource that is already in use, the process automatically becomes blocked. When the resource becomes available as a result of a later release, the blocked process may be assigned the resource and made ready. Alternate solutions (e.g., SPOOLing) to the printer problem are discussed in Chapter 5.

In addition to physical devices, there are other shared resources, such as common tables (analogous to the order-list turntable of Figure 4-26), that require the same type of process synchronization.

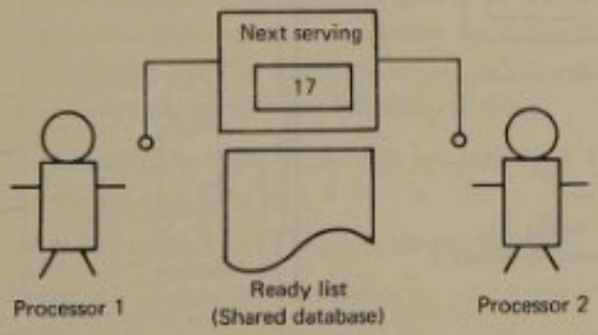
For an example of such a race condition, consider Figure 4-28, which depicts a procedure for selecting a process from the ready list and placing it in the running state. This procedure is executed by the process scheduler. This algorithm works well for a single processor, but runs into trouble for multiple processors, as shown in Figure 4-29.



When current process is completed or postponed:

1 Find current process in ready list
2 Mark it "complete" or "postponed"
3 Note "next serving" number (17)
4 Find process 17 in ready list
5 Make note of process to be done
6 Adjust ready list for "next serving" number (18)

Figure 4-28 Scheduling (single processor)



Assume both processors complete their current process at same time:

1 Processor 1 finds its current process in ready list
2 Processor 2 finds its current process in ready list
3 Processor 1 marks its current process
4 Processor 2 marks its current process
5 Processor 1 notes "next serving" (17)
6 Processor 2 notes "next serving" (17)
7 Processor 1 finds process 17 in ready list and notes it
8 Processor 2 finds process 17 in ready list and notes it
9 Processor 1 adjusts ready list "next serving" (18)
10 Processor 2 adjusts ready list "next serving" (19)

Now both processors are simultaneously working on the same process and process 18 has been skipped.

Figure 4-29 Scheduling (multiple processors)

PRODUCER PROCESS, P_p	CONSUMER PROCESS, P_c
PRODUCE:	CONSUME:
⋮ (produce item)	P (S_2)
P (S_1)	⋮ (remove item from buffer)
⋮ (put item in buffer)	V (S_1)
V (S_2)	⋮ (consume item)
GO TO PRODUCE	GO TO CONSUME

Once again, the reader should verify for himself that the correct synchronization does, in fact, occur. A simulated execution is a useful way to study semaphore coordination as shown in Figure 4-31 where n = 2. Notice that the producer is not allowed to add an item to a full buffer, nor is the consumer to remove an item from an empty buffer.

This same algorithm can be used by multiple producers and consumers. Other uses of the semaphore synchronization are illustrated in the sample operating system.

	ACTIONS		SEMAPHORE VALUES	
Event	P_p (*Producer*)	P_c (*Consumer*)	S_1	S_2
0	...	...	2	0
1	–	P(S_2) blocked	2	-1
2	P (S_1)	– (buffer	1	-1
3	V (S_2) unblocks P_c	– "empty")	1	0
4	P (S_1)	–	0	0
5	–	V (S_1)	1	0
6	V (S_2)	–	1	1
7	P (S_1)	–	0	1
8	V (S_2)	–	0	2
9	P (S_1) blocked	–	-1	2
10	– (buffer	P (S_2)	-1	1
11	– "full")	V (S_1) unblocks P_p	0	1
etc.				

Figure 4-31 Simulated execution of two synchronized processes

4-5.2.4 MESSAGE COMMUNICATION

The synchronization mechanisms we have discussed provide communication between processes indirectly by means of a shared buffer area and a shared lock byte or semaphore. Situations such as the producer/consumer problem can also be solved by providing direct process-to-process communication by means of the primitives SEND (P_r,M) and RECEIVE (P_s,M) where P_r and P_s are the names of processes and M is a message (i.e., a k-byte character string).

SEND (P_r,M) saves the message M for the receiver process P_r. RECEIVE (P_s,M) returns a message M, if there is one, to the requestor process; it also returns the name of the process P_s that sent the message. If there are no messages for the RECEIVE requestor, it becomes blocked until a message is sent to it. The buffering of messages and synchronization are handled automatically by the SEND and RECEIVE primitives.

Our sample operating system presents a detailed implementation of a message communication facility and numerous examples of its usage.

4-5.3 Deadly Embrace

A *deadly embrace* is a situation in which two processes are unknowingly waiting for resources that are held by each other and thus unavailable. Consider the two processes depicted in Figure 4-32.

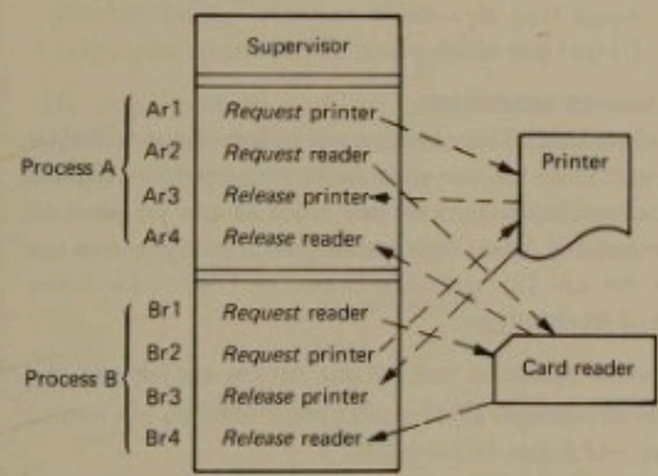


Figure 4-32 Deadly embrace situation

Processes A and B are sharing use of the printer and card reader by means of the request and release operations. Due to independent scheduling of the processes, the request and release operations may be interspersed in several different orders such as:

(1) Ar1, Ar2, Ar3, Ar4, Br1, Br2, Br3, Br4

(2) Br1, Br2, Br3, Br4, Ar1, Ar2, Ar3, Ar4

(3) Ar1, Ar2, Br1, Ar3, Ar4, Br2, Br3, Br4

The reader should assure himself that these sequences are reasonable and perform correctly. Note that in number (3) process B is temporarily blocked by Br1 until process A issues Ar4.

Device Management

In this chapter we group together the management of all devices other than the processor and memory for three reasons: (1) a separate chapter for each would not be warranted; (2) we can state some general concepts that are applicable to all; and (3) many contemporary systems consolidate all the modules that manage these devices.

This chapter focuses on the management of I/O devices. These include actual I/O devices, such as printers, card readers, tapes, disks, and drums, and supporting devices, such as control units or control channels. A brief discussion of I/O device characteristics is presented in Sections 5-2 and 5-3. Some readers may already be familiar with some of these devices; our purpose in these sections is to merely give a feel for the relative speeds and characteristics of the important devices. A large installation may devote over half the system cost to I/O devices. Therefore, it is desirable to use these devices in an efficient manner.

The basic functions of device management are:

1 Keeping track of the status of all devices, which requires special mechanisms. One commonly used mechanism is to have a database such as a Unit Control Block (UCB) associated with each device. (See Section 5-5.1 for a sample format of a UCB.)

2 Deciding on policy to determine who gets a device, for how long, and when. A wide range of techniques is available for implementing these policies. For example, a policy of high device utilization attempts to match the nonuniform requests by processes to the uniform speed of many I/O devices. There are three basic techniques for implementing the policies of device management:

   a Dedicated—a technique whereby a device is assigned to a single process.

   b Shared—a technique whereby a device is shared by many processes.

   c Virtual—a technique whereby one physical device is simulated on another physical device.

3 Allocation—physically assigning a device to process. Likewise the corresponding control units and channels must be assigned.

4 Deallocation policy and techniques. Deallocation may be done on either a process or a job level. On a job level, a device is assigned for as long as the job exists in the system. On a process level, a device may be assigned for as long as the process needs it.

This chapter will focus mainly on the policies of device management, since the techniques are heavily device-dependent and are changing rapidly with hardware advances. However, Chapter 7 gives an example of the implementation of a dedicated policy where each device is assigned solely to one job.

The module that keeps track of the status of devices is called the *I/O traffic controller*. We have called all modules associated with the operation of a single device the *I/O device handlers*. In our design of Figure 1-9, each I/O device has a device handler that resides in a separate process associated with that device. The I/O device handler's function is to create the channel program for performing the desired function, initiate I/O to that device, handle the interrupts from it, and optimize its performance. In short, the device handler performs the physical I/O. The *I/O scheduler* decides when an I/O processor is assigned to a request and sets up the path to the device.

5-1 TECHNIQUES FOR DEVICE MANAGEMENT

Three major techniques are used for managing and allocating devices: (1) dedicated, (2) shared, and (3) virtual.

5-1.1 Dedicated Devices

A *dedicated* device is allocated to a job for the job's entire duration. Some devices lend themselves to this form of allocation. It is difficult, for example, to share a card reader, tape, or printer. If several users were to use the printer at the same time, would they cut up their appropriate output and paste it together? Unfortunately, dedicated assignment may be inefficient if the job does not fully and continually utilize the device. The other techniques, shared and virtual, are usually preferred whenever they are applicable.

5-1.2 Shared Devices

Some devices such as disks, drums, and most other Direct Access Storage Devices (DASD) may be shared concurrently by several processes. Several processes can read from a single disk at essentially the same time. The management of a shared device can become quite complicated, particularly if utmost efficiency is desired. For example, if two processes simultaneously request a Read from Disk A, some mechanism must be employed to determine which request should be handled first. This may be done partially by software (the I/O scheduler and traffic controller) or entirely by hardware (as in some computers with very sophisticated channels and control units).

Policy for establishing which process' request is to be satisfied first might be based on (1) a priority list or (2) the objective of achieving improved system output (for example, by choosing whichever request is nearest to the current position of the read heads of the disk).

Shared devices are further discussed in Sections 5-2.2.3 and 5-5.

5-1.3 Virtual Devices

Some devices that would normally have to be dedicated (e.g., card readers) may be converted into shared devices through techniques such as SPOOLing. For example, a SPOOLing program can read and copy all card input onto a disk at high speed. Later, when a process tries to read a card, the SPOOLing program intercepts the request and converts it to a read from the disk. Since a disk may be easily shared by several users, we have converted a dedicated device to a shared device, changing one card reader into many "virtual" card readers. This technique is equally applicable to a large number of peripheral devices, such as teletypes, printers, and most dedicated slow input/output devices.

In Section 5-6 we discuss the techniques for virtualizing some devices and the problems associated with implementing these algorithms on particular systems that have memory limitations.

5-1.4 Generalized Strategies

Some professionals have attempted to generalize device management even more than is done here. For example, the call side of SPOOLing is similar to the file system and buffering bears striking similarity to SPOOLing. We could generalize even more and view memory, processors, and I/O devices as simple devices to which the same theory should apply. At present we realize that there are practical limitations to these generalizations, although at some time in the future more general theories may emerge that are applicable to all devices and that have direct practical applications. Therefore, while continuing our research to produce such general theories, we recognize that they will not emerge without thorough understanding of the issues presented in this book.

5-2 DEVICE CHARACTERISTICS—HARDWARE CONSIDERATIONS

Almost everything imaginable can be, and probably has been, used as a peripheral device to a computer, ranging from steel mills to laser beams, from radar to mechanical potato pickers, from thermometers to space ships. Fortunately, most computer

installations utilize only a relatively small set of peripheral devices. Peripheral devices can be generally categorized into two major groups: (1) input or output devices and (2) storage devices.

5-2.1 Input or Output Devices

An *input device* is one by which the computer "senses" or "feels" the outside world. These devices may be mechanisms such as thermometers or radar devices, but, more conventionally, they are devices to read punched cards, punched paper tape, or messages typed on typewriterlike terminals. An *output device* is one by which the computer "affects" or "controls" the outside world. It may be a mechanism such as a temperature control knob or a radar direction control, but more commonly it is a device to punch holes in cards or paper tape, print letters and numbers on paper, or control the typing of typewriterlike terminals.

In previous chapters we have described some common I/O devices, such as printers and card readers (see the buffering examples of Chapter 2). In this chapter we will focus on the *storage devices*.

5-2.2 Storage Devices

A storage device is a mechanism by which the computer may store information (a procedure commonly called *writing*) in such a way that this information may be retrieved at a later time (*reading*). Conceptually, storage devices are analogous to human storage mechanisms that use pieces of paper, pencils, and erasers.

It is helpful to differentiate among three types of storage devices. Our differentiation is based on the variation of access times (T_{ij}) where:

T_{ij} = time to access item j, given last access was to item i (or current position is item i)

We differentiate the following types of storage devices:

1 *Serial Access* where T_{ij} has a large variance (e.g., tape)
2 *Completely Direct Access* where T_{ij} is constant (e.g., core)
3 *Direct Access* where T_{ij} has only a small variance (e.g., drum)

5-2.2.1 SERIAL ACCESS DEVICES

A *serial access* storage device can be characterized as one that relies on strictly physical sequential positioning and accessing of information. Access to an arbitrary, stored item requires a "linear search" so that an average access takes the time required to read half the information stored. A Magnetic Tape Unit (MTU) is the most common example of a serial access storage device. The MTU is based upon the same principles as the audio tape deck or tape cassette, but instead of music

or voices, binary information is stored. A typical serial access device is depicted in Figure 5-1.

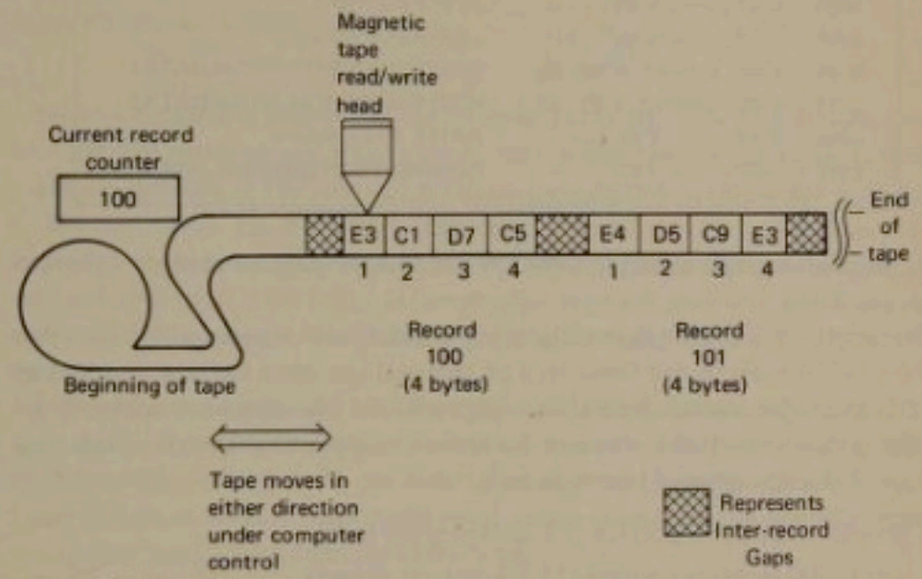


Figure 5-1 Magnetic tape unit

Information is usually stored as groups of bytes, called *records*, rather than single bytes. In general, a record can be of arbitrary length, e.g., from 1 to 32,768 bytes long. In Figure 5-1 all the records are four bytes long. Records may be viewed as being analogous to spoken words on a standard audio tape deck. Each record can be identified by its physical position on the tape; the first record is 1, the second, 2, and so on. The current record counter of Figure 5-1 serves the same function as the "time" or "number-of-feet" meter of audio tape decks. If the tape is positioned at its beginning (i.e., completely rewound), and we wish to read record number 101, it is necessary to skip over all intervening records (e.g., 1, 2, . . ., 99, 100) in order to reach record number 101.

In addition to obvious I/O commands, such as Read the Next Record or Write the Next Record, a tape unit usually has commands such as Read the Last Record (read backward), Rewind to Beginning of Tape at High Speed, Skip Forward (or Backward) One Record without Actually Reading or Writing Data. In order to help establish groupings of records on a tape, there is a special record type called a *tape mark*. This record type can be written only by using the Write Tape Mark command. If the tape unit encounters a tape mark while reading or skipping, a special interrupt is generated. Tape marks are somewhat analogous to putting bookmarks in a book to help find your place.

A 370 channel program using a magnetic tape unit might look like this:

CCW	X'07', - - -, X'40', - - -	REWIND TO BEGINNING OF TAPE
CCW	X'37', - - -, X'40', - - -	SKIP RECORD 1
CCW	X'37', - - -, X'40', - - -	SKIP RECORD 2
CCW	X'01', DATA1, X'40', 80	WRITE RECORD 3 FROM DATA1
CCW	X'01', DATA2, X'40', 80	WRITE RECORD 4 FROM DATA2
CCW	X'1F', - - -, X'40', - - -	WRITE TAPE MARK
CCW	X'07', - - -, X'00', - - -	REWIND TO BEGINNING AGAIN

A program can use a simple one-CCW channel program to read or write one record at a time.

Contemporary magnetic tape units, using half-inch-wide tape, usually store either 1600 or 6250 bytes in one linear inch of tape and can move the tape at speeds up to 200 inches per second. Even at this speed it would take about one minute to get to the middle record of a standard 2,400-foot tape starting from the beginning.

Typical characteristics of tape units are:

density—1600 bytes per inch (19,200 bytes per foot)

speed—200 inches per second (16 2/3 feet per second)

length—2400 feet long (i.e., capacity is about 46,000,000 bytes)

maximum access = 2 minutes

random (average) access = 1 minute

serial access (at one record read the next) ≈ 4 ms

note T_{ij} wide variance

Serial access storage devices are normally used for applications that require mainly sequential accessing, such as the copies of punched card decks, or the intermediate file copies produced by pass 1 of an assembler or loader. In current data processing, a large percentage of the applications are primarily sequential.

The Inter-record Gap (IRG) shown in Figure 5-1 is necessary due to the physical limitations of starting and stopping a tape. In order to read the next record in a stopped tape, the tape drive takes a finite amount of time to get up to speed. Similarly, after a record has been read, some tape goes by before it can be stopped.

The IRG is usually from 1/4 to 3/4 of an inch (approximately equivalent to 400 to 1200 bytes). The gap length for a specific device is a constant. Thus it is common to store records of greater than 1000 bytes; otherwise, 50 percent or more of the tape is wasted. For example, if we stored 80-byte records with a 3/4-inch IRG, only about 6.3 percent of the tape would be utilized (or 2.7 million out of a possible 47 million bytes).

To minimize gap waste, we frequently *block* records. *Blocking* is placing multiple logical records into one *physical record* (a block). Figure 5-2 depicts four 80-byte logical records (cards) in one 320-byte physical record (block).

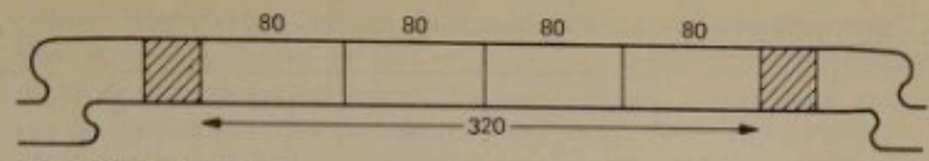


Figure 5-2 Blocked records

Blocks are typically 800 to 8000 bytes long. If they were any longer, there would be reliability problems and when read in, they would take up too much memory.

The mechanisms of blocking and deblocking will be elaborated further in Chapter 6. Basically, when the file system receives a request to read a logical record, this request is converted into a read of the appropriate blocked physical record that contains the logical record. The entire physical record is read into a buffer area in core, and the request is satisfied by extracting the logical record from the buffer area.

Substantial speed gains can be achieved if a table is kept aware of the contents of these buffer areas. For example, the next read request is usually for the next logical record, which probably is in the same physical record (now in the buffer area). There is thus no need to read the physical record again from the tape—the logical record may be extracted from the buffer area.

There are three major advantages to blocking:

1. Fewer I/O operations are needed since each I/O operation reads or writes multiple logical records at a time.
2. There is less wasted space since record length is larger than the Inter-record Gap length.
3. Smaller tape space is covered when reading many records (less distance to travel) since there is less wasted space.

There are also three major disadvantages:

1. Software overhead and routines are required to do the blocking, deblocking, and record keeping.
2. Buffer space is wasted (e.g., you might have wanted only 80 bytes but had to read 8000 bytes). On small machines this may be 10 percent or more of all memory.
3. There is more likelihood of tape errors since long records are being read.

5-2.2.2 COMPLETELY DIRECT ACCESS DEVICES

A completely direct access device is one in which the access time T_{ij} is a constant. Magnetic core memory, semiconductor memory, read-only wired memories, and diode matrix memories are all examples of completely direct access memories.

Figure 5-3 depicts the workings of a core memory.

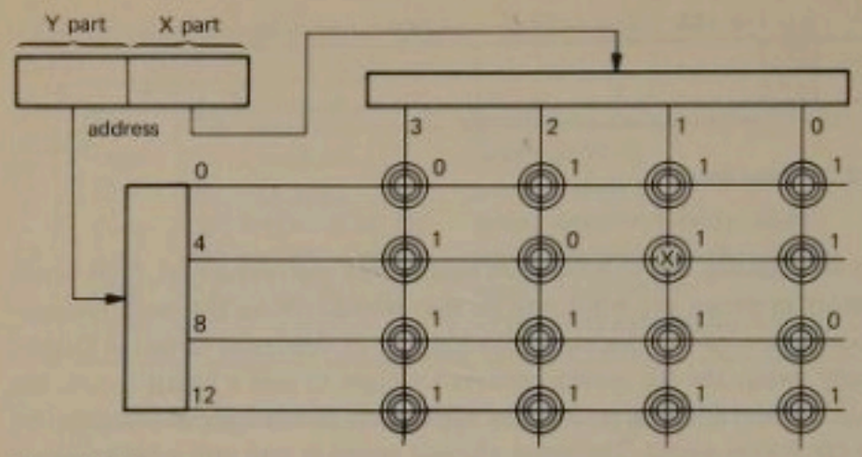


Figure 5-3 Core memory

In Figure 5-3 the 16 circles represent magnetic ferrite cores. All cores are connected by wires, each core having two selection wires through it. When the hardware receives a read request for some address, the address is decomposed into an x and y portion. The hardware then selects the appropriate core. This is done by passing current through the two selection wires. Only the core at the specified (x,y) coordinate receives a "double dose" of current; all other cores receive half the amount of current or no current at all. The double dose of current is sufficient to "flip" the magnetic field from 1 to 0.

One property of a magnetic core is that when it changes magnetic state it induces a current. A third sensing wire is passed through all the cores. If the appropriate core was 1 and it was switched to 0, a current would be induced in this wire. If it was already 0, there would be no induced current.

For example, in Figure 5-3, if a request to read bit 5 (binary 0101) were received, the hardware would decompose the address:

01,01		y = 4
y x	or	and
		x = 1

Thus, the sense output would be a 1 (the x core of Figure 5-3). Note that the process of reading a core destroys its information content (i.e., sets it to 0 always); thus an additional cycle is necessary to write the information back.

For core memory T_{ij} is constant for all i and j and is typically about 500 nanoseconds to 2 microseconds. Usually, this type of memory is used as *main memory*. Occasionally, if it is sufficiently inexpensive (and hence usually slower—greater than 2 microseconds), it is used as a storage device (e.g., IBM's Large Core Storage or CDC's Extended Core Storage). This use of completely direct access device is rare at present, but we may see more of it in the future (Risko, 1968).

Most contemporary computers use semiconductor (transistorlike) memories. The basic ideas are similar to the core memories described above, but semiconductor memories offer advantages in cost and speed. Typical semiconductor memories operate between 50 to 500 nanoseconds access time.

Although we say this type of storage device may be shared on the nanosecond level, it is limited in the sense that only one processor (CPU or I/O channel) can read a specific location (or block of locations) at a given time. Whenever more than one processor is accessing memory, especially if there is one CPU and several I/O channels, we frequently say that the channels are *memory cycle stealing*. Memory cycle stealing can degrade the performance of a system and is a factor with which the I/O traffic controller must contend.

5-2.2.3 DIRECT ACCESS STORAGE DEVICES (DASD)

A direct access device is one that is characterized by small variances in the access time T_{ij}. These have been called *Direct Access Storage Devices*. Although they do not offer completely direct access, the name persists in the field.

In this section we give two examples of DASD, fixed-head and moving-head devices.

5-2.2.3.1 *Fixed-Head Drums and Disks*

Figure 5-4 depicts a DASD storage device similar to a magnetic drum.

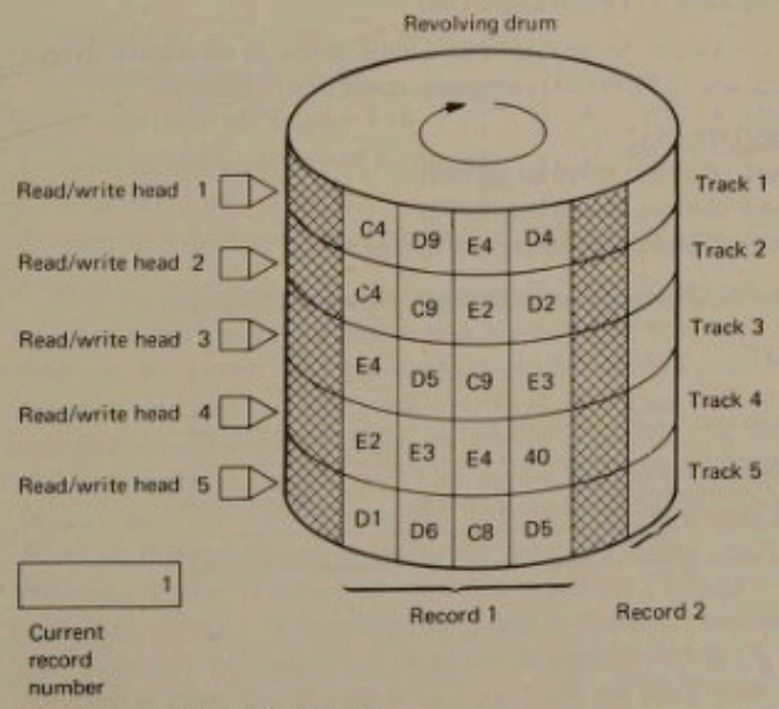
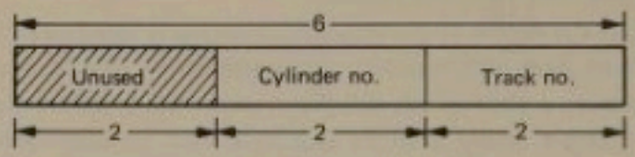


Figure 5-4 Fixed-head drum DASD

On the 370 there is a separate I/O command, called a *Seek* operation, to position the arm. Once the arm is positioned, the device is created exactly like the magnetic drum described in Section 5-2.2.3.1. A Seek position is specified using six bytes, as follows:



Unlike the record search done by the Search ID command, once the arm has been positioned at a cylinder, it stays at that position until another Seek command causes it to move. Thus, the seek command may be issued separately, which is called Preseeking, or it may be part of a long channel program.

Figure 5-6 shows the commands that can be used to read the record at Cylinder 3, Track 1, Record 2 assuming the area is not currently positioned at Cylinder 3.

```
         CCW  X'0B',ADDR,X'40',6          Seek cylinder.
SEARCH   CCW  X'31',ADDR+2,X'40',5        Search for desired record.
         CCW  X'08',SEARCH,X'40',- -      Keep trying to find.
         CCW  X'06',DATA,X'00',80         Read 80 bytes of data.
ADDR     DC   X'00 00 00 03 00 01 02'     Full address of
                     Cylinder  Track Record  Cylinder 3, Track 1,
                                             Record 2.
DATA     DS   CL80
```

Figure 5-6 CCW for reading record

Typical moving-arm disk DASDs have 10 to 50 tracks and 200 to 400 arm positions with characteristics such as:

maximum access = 75 ms
(55 ms maximum arm movement,
20 ms maximum rotation)
average (random) access = 30 ms
serial access < 1 ms

T_{ij} – minor variance (less than two orders of magnitude)

capacity = 100 million bytes (400 cylinders * 20 heads * 16,000 bytes per track)

5-2.2.3.3 *General Characteristics*

Some DASD devices allow keys to be associated with each record. The *keys* are assigned by the user and can denote the contents of a record. The controller for such devices allows commands to read or write a record addressed not by a track, arm, or cylinder, but rather by its key. For example,

Search key = HARRY instead of Search ID = 0201

There are often some constraints on DASDs that allow this form of addressing, e.g., the read head must already be positioned on the right cylinder.

Given the present trend toward reductions in the price of CPUs, it is probably better to keep a table in memory and not to use such searching on the disks, thus keeping the cost of the channels down.

The preceding examples are representative of storage devices, since in spite of numerous variations, all have similar characteristics.

5-3 CHANNELS AND CONTROL UNITS

Device management must also take responsibility for the channels and control units. The *channels*, as explained in Chapter 2, are the special processors that execute the channel programs (i.e., the CCWs). Due to the high cost of channels, there are usually fewer channels than devices. The channels must, therefore, be switched from one device to another. Further cost savings can be obtained by taking advantage of the fact that relatively few I/O devices are in use at one time. Thus, it is wise to separate the device's electronics from its mechanical components. The electronics for several devices can be replaced by a single control unit, which can be switched from one device to another. Figure 5-7 illustrates the relationship between channels, control units, and devices. Typically, there are up to eight control units per channel and up to eight devices of the same type per control unit.

In order to initiate an I/O operation, a path is required between memory and the device. For example, to access drum J requires the use of channel A and drum control unit D. If an I/O operation to drum G were in progress, it would not be possible to initiate I/O on drum J because channel A would not be available (the I/O on drum G requires use of channel A and control unit C).

The device management routines must keep track of the status of (1) channels, (2) control units, and (3) devices. Since the channels and control units are usually scarce, they frequently create a serious bottleneck. Several hardware techniques can be used to lessen this problem: independent device operation, device buffering, multiple paths, and block multiplexing.

5-3.1 Independent Device Operation

In many cases there is sufficient capability in the device to complete an I/O operation without further assistance from the channel or control unit. A disk seek operation, which might require 50 ms, only needs the channel long enough to transmit the seek address from memory to the disk–10 or 20 μs. The seek can be completed

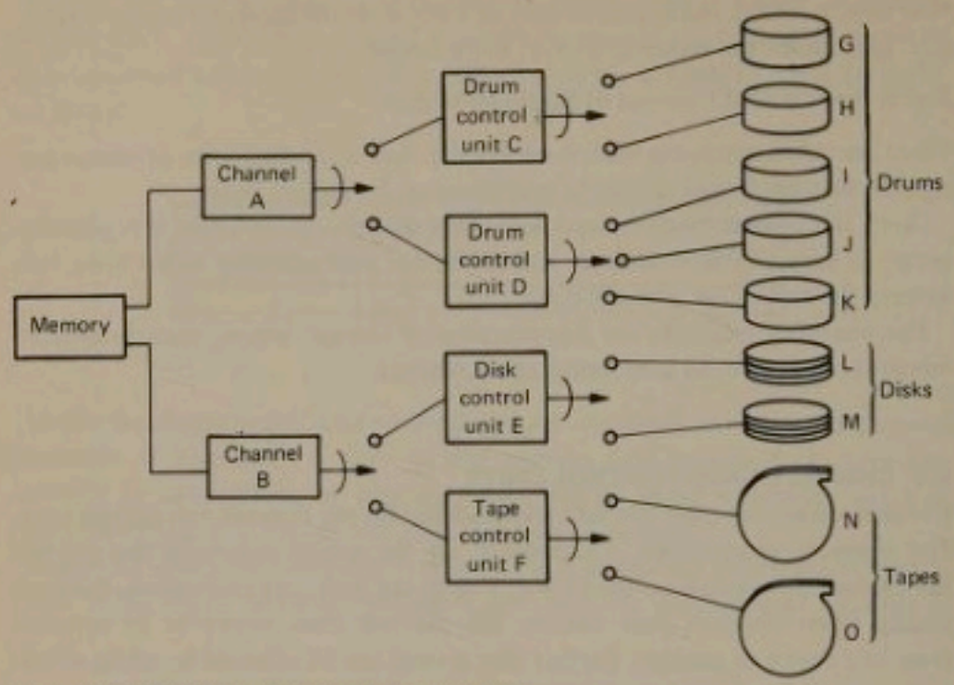


Figure 5-7 I/O configuration of a computer system

by the device alone. Likewise, there are operations that may require both the device and the control unit but not the channel for completion.

When an I/O operation is initiated, the entire path must be available. In order to indicate when parts of the path become available, there may be separate I/O completion interrupts: channel end, control unit end, and device end. The type of I/O completion interrupt is designated by the status bits in the Channel Status Word. If more than one part of the path becomes available at the same time, there will be only one interrupt but the status bits will indicate all available parts (e.g., both the channel end and control unit end bits will be on in the CSW).

If a *channel end* occurs, the channel is no longer busy, although the device may still be busy. An I/O operation can then be initiated on some other device using that channel.

5-3.2 Buffering

By providing a data buffer in the device or control unit, devices that would ordinarily require the channel for a long period can be made to perform independently of the channel. For example, a card reader may physically require about 60 ms to read a card and transfer the data to memory through the channel. However, a *buffered card reader* always reads one card before it is needed and saves the 80 bytes in

a buffer in the card reader or control unit. When the channel requests that a card be read, the contents of the 80-byte buffer are transferred to memory at high speed (e.g., 100 μs) and the channel is released. The device then proceeds to read the next card and buffer it independently. Card readers, card punches, and printers are frequently buffered.

5-3.3 Multiple Paths

We could reduce the channel and control unit bottlenecks by buying more channels and control units. A more economical alternative, however, is to use the channels and control units in a more flexible manner and allow multiple paths to each device. Figure 5-8 illustrates such an I/O configuration. In this example there are four different paths to device E:

1 Channel A – Control Unit C – Device E
2 Channel A – Control Unit D – Device E
3 Channel B – Control Unit C – Device E
4 Channel B – Control Unit D – Device E

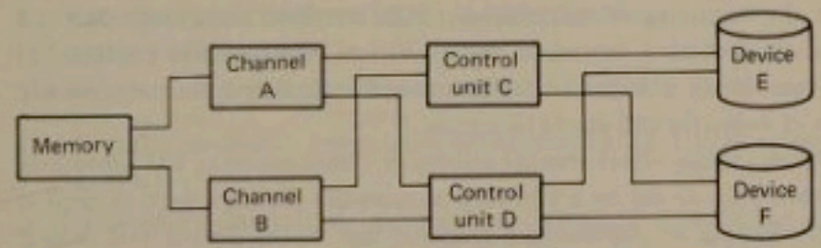


Figure 5-8 I/O configuration with multiple paths

If channel A and control unit C were both busy due to I/O for device F, a path could be found through channel B and control D to service device E. An actual I/O configuration may have eight control units per channel and eight devices per control unit rather than the sparse configuration of Figure 5-8. A completely flexible configuration would make all control units accessible to all channels and all devices accessible to all control units, thus providing possibly 256 alternate paths between memory and each device. For economic reasons, this extreme case, though feasible, is seldom found.

In addition to providing flexibility, a multiple path I/O configuration is desirable for reliability. For example, if channel A malfunctions and must be repaired, all the control units and devices can still be accessed through channel B. It is the responsibility of the device management routines to maintain status information on all paths in the I/O configuration and find an available route to access the desired device.

5-3.4 Block Multiplexing

On conventional 370 selector channels, the techniques of independent device operation and buffering are applicable only to the last CCW of a channel program. The channel and control unit remain connected to the device for the duration of the channel program. A 370 *block multiplexor channel* can be servicing multiple channel programs at the same time (e.g., eight channel programs). When the channel encounters an I/O operation such as a buffered or independent device operation that does not need the channel for awhile, it automatically switches to another channel program that needs immediate servicing. In essence, a block multiplexor channel represents a hardware implementation of multiprogramming for channel programs.

This type of channel multiprogramming could be simulated on a selector channel by making all channel programs only one command long. The device management software routines could then reassign the channel as necessary. However, this approach is not very attractive unless the central processor is extremely fast, since the channel switching must be done very frequently and very quickly.

5-4 DEVICE ALLOCATION CONSIDERATIONS

As noted at the beginning of this chapter, there are three major techniques of device allocation: dedicated, shared, or virtual. Virtual, which will be explained in Section 5-6, transforms a dedicated device into a shared device; thus, it is merely a special case of dedicated and shared allocation.

Certain devices require either manual actions or time-consuming I/O operations to be performed prior to use by a job, such as manually placing a deck of cards in a card reader's hopper or positioning a magnetic tape to the desired data record. To switch use of such a device among several jobs would require an excessive amount of work. Thus, these devices are dedicated to one job at a time and are not reassigned until that job releases the device—usually at the termination of the job. The assignment of a device to a job is usually performed in conjunction with the job-scheduling routines of processor management.

Due to the relative ease and speed with which a direct access device such as a magnetic drum or disk can be repositioned, it is possible to share it among several jobs running concurrently. A shared device is logically treated as if it were a group of dedicated devices. Each job using a shared device logically has a separate area to itself; the direct access property of the device makes it economical to switch back and forth among these areas. The device management routines may decide which shared device will be assigned to a job and what area of the device will be assigned. These decisions are also frequently made in conjunction with the job-scheduling routines.

For both dedicated and shared devices, static and dynamic allocation decisions must be made. The static decisions largely involve determining which dedicated

and shared devices are to be assigned to the job. In general, device management tries to spread the allocation among devices so as to minimize channel and control unit bottlenecks. Often, the user can provide information on the job control cards indicating which of the devices will be used concurrently (these should be on separate channels) and which will be used at different times (these can be safely assigned to the same channel). The OS/360 Affinity (AFF) and Separation (SEP) job control language options serve this purpose (Brown, 1970; Cadow, 1970).

Since channels and control units are so scarce, they are seldom permanently assigned to a job. Instead, an assignment is made when a job makes an I/O request to device management. The assigned channel and control unit are deallocated and available for reassignment as soon as the I/O request is completed.

The static allocation decisions of device management are analogous to the static decisions of the job scheduler in processor management. The dynamic allocation is similar in purpose to the process scheduler of processor management. In the remainder of this chapter we will focus primarily on the dynamic allocation functions of device management.

5-5 I/O TRAFFIC CONTROLLER, I/O SCHEDULER, I/O DEVICE HANDLERS

The functions of device management can be conveniently divided into three parts:

1. *I/O traffic controller*—keeps track of the status of all devices, control units, and channels.
2. *I/O scheduler*—implements the policy algorithms used to allocate channel, control unit, and device access to I/O requests from jobs.
3. *I/O device handlers*—perform the actual dynamic allocations, once the I/O scheduler has made the decision, by constructing the channel program, issuing the start I/O instruction, and processing the device interrupts. There is usually a separate device handler for each type of device.

5-5.1 I/O Traffic Controller

Whereas the I/O scheduler is mainly concerned with *policies* (e.g., who gets the device and when), the I/O traffic controller is primarily concerned with *mechanics* (e.g., can the device be assigned?). The traffic controller maintains all status information.

The I/O traffic controller becomes complicated due to the interdependencies introduced by the scarcity of channels and control units, and the multiple paths illustrated earlier in Figures 5-7 and 5-8. The traffic controller attempts to answer at least three key questions:

1 Is there a path available to service an I/O request?
2 Is more than one path available?
3 If no path is currently available, when will one be free?

In order to answer these questions, the traffic controller maintains a database reflecting status and connections. This can be accomplished by means of Unit Control Blocks (UCB), Control Unit Control Blocks (CUCB), and Channel Control Blocks (CCB) as depicted in Figure 5-9. The first question can be answered by

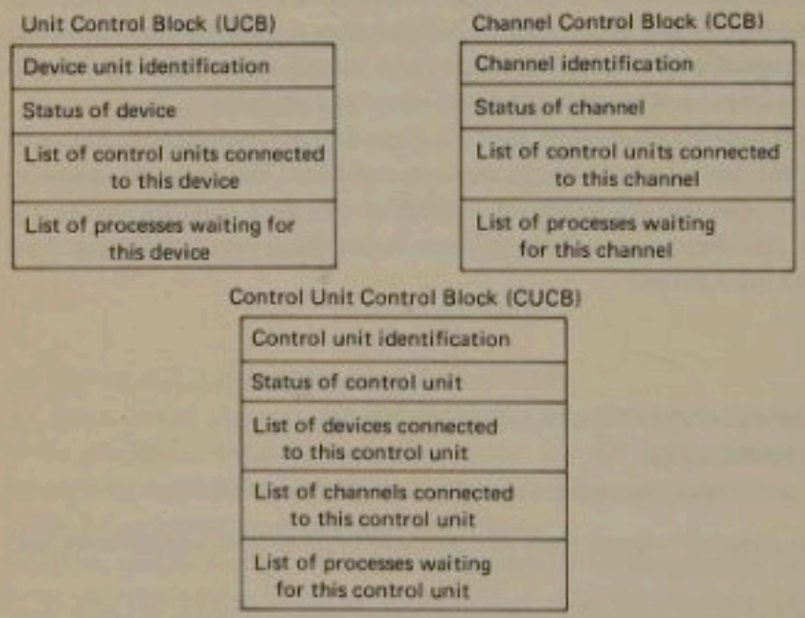


Figure 5-9 Control blocks

starting at the desired UCB and working back through the connected control units and channels trying to find a combination that is available. In a similar manner it is possible to determine all available paths. This second question may be important, especially when the I/O configuration is not symmetric, as shown in Figure 5-10, since choosing one path may block out other I/O requests. Thus, one path may conflict with fewer I/O requests than another path. In Figure 5-10, there may be up to five different paths to device H, yet only one possible path for either device G or I.

Under heavy I/O loads, it is quite likely that there will not be a path available at the time an I/O request is issued by a process. When an I/O completion interrupt occurs, one or more components (i.e., device, control units, and/or channel) become available again. We could try all I/O requests that are waiting to see if any can now be satisfied; if, however, hundreds of I/O requests were waiting, this could be a time-consuming approach. A more efficient approach would be to list all I/O requests that are "interested" in the component in the control block, as suggested in Figure 5-9. Only those particular I/O requests would then have to be examined.

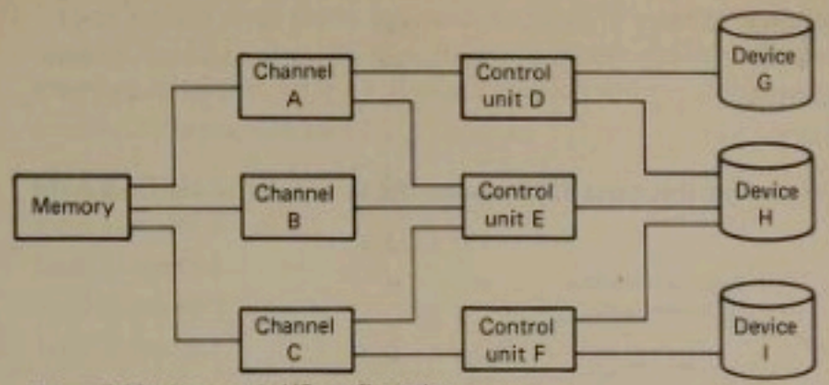


Figure 5-10 Assymetric I/O configuration

5-5.2 I/O Scheduler

If there are more I/O requests pending than available paths, it is necessary to choose which I/O requests to satisfy first. Many of the same concerns discussed in process scheduling are applicable here. One important difference is that I/O requests are not normally timesliced, that is, once a channel program has been started it is not usually interrupted until it has been completed. Most channel programs are quite short and terminate within 50–100 ms; it is, however, possible to encounter channel programs that take seconds or even minutes before termination.

Many different policies may be incorporated into the I/O scheduler. For example, if a process has a high priority assigned by the process scheduler, it is reasonable also to assign a high priority to its I/O requests. This would help complete that job as fast as possible.

Once the I/O scheduler has determined the relative orderings of the I/O requests, the I/O traffic controller must determine which, if any, of them can be satisfied. The I/O device handlers then provide another level of device-dependent I/O scheduling and optimization.

5-5.3 I/O Device Handlers

In addition to setting up the channel command words, handling error conditions, and processing I/O interrupts, the I/O device handlers provide detailed scheduling algorithms that are dependent upon the peculiarities of the device type. There is usually a different device handler algorithm for each type of I/O device.

5-5.3.1 ROTATIONAL ORDERING

Under heavy I/O loads, several I/O requests may be waiting for the same device. As noted earlier, there may be significant variations in the T_{ij} access time for a device. Certain orderings of I/O requests may reduce the total time required to service the I/O requests. Consider the drumlike device that has only four records stored in it (see Figure 5-11). The following four I/O requests have been received and a path to the device is now available:

5-5.3.3 SEEK ORDERING

Moving-head storage devices, as described in Section 5-2.2.3.2, have the problem of seek position in addition to rotational position. I/O requests require a three-part address: cylinder number, track number, and record number. If the requests

cylinder 1, track 2, record 1
cylinder 40, track 3, record 3
cylinder 5, track 6, record 5
cylinder 1, track 5, record 7

were processed in the order shown, a considerable amount of time would be spent to move the seek arm back and forth. A more efficient ordering would be:

cylinder 1, track 2, record 1
cylinder 1, track 5, record 7
cylinder 5, track 6, record 5
cylinder 40, track 3, record 3

These requests are in *Minimum seek time order*, that is, the distances between seeks have been minimized.

There are many other possible seek orderings (Teorey, 1971, 1972). When rotational position is important, it is sometimes better to seek a further distance to get a record that will be at a closer rotational position. For an extreme example, if seek times were very small in comparison to the rotation time, the first ordering shown might be best since it accesses records in the order 1, 3, 5, 7. Determining this *minimum service time order* can be quite complex. It is further complicated by the fact that seek times, though monotonic, are not usually linear as a function of seek distance, i.e., the time to move the seek arm 20 cylinders is usually less than double the time to move it 10 cylinders.

In both the minimum seek time and minimum service time cases, the ordering was determined by the current I/O requests for that device. New I/O requests are likely to occur while the original ones are being satisfied. In retrospect, the ordering chosen on the basis of the original I/O requests may be poor when the new I/O requests are considered. Unless the system has a "crystal ball" feature, it is impossible to know what future requests will be. Several schemes have been used to minimize the negative effects of future I/O requests. Usually the ordering is reevaluated after each individual request has been processed so as to allow consideration of any new requests. *Unidirectional* or *linear sweep service time* ordering is a technique that moves the arm monotonically in one direction. When the last cylinder is reached or there are no further requests in that direction, the arm is reset back to the beginning and another sweep is started. This technique attempts to decrease the amount of back and forward arm thrashing caused by the continued arrival of new I/O requests.

5-5.4 Device Management Softward Overhead

Each of the device management algorithms discussed in the preceding sections provides additional device efficiency but at the cost of increased processor time (unless the algorithms are incorporated in the hardware, in which case there is increased hardware expense). The value of these algorithms depends upon the speed of the processor and the amount of I/O requests. If there are few I/O requests, multiprogramming of the processor alone may be sufficient to attain high throughput. If the processor is quite slow, the processor overhead may swamp the I/O advantages.

For example, a System/360 Model 30, with an average instruction time of 30 μs would take 30 ms to execute a 1,000-instruction reordering algorithm. If the I/O requests take less than 30 ms to service without reordering, there is no savings attained. However, reordering techniques do become very important on large-scale computers.

Virtual storage memory management increases the load on device management in two ways. First, the page swapping often introduces a large amount of I/O requests. Second, the device handler must handle translation from virtual memory addresses to physical memory addresses in the construction of CCWs, since most channels require physical memory addresses rather than virtual memory addresses.

Finally, device management must ensure the protection of each job's areas in main memory and secondary storage. In the extreme, this would require very careful analysis of all I/O requests and addresses referenced. This can be done because all I/O instructions are privileged and, thus, can be initiated only by the operating system. On the 360 it is possible to use the memory lock hardware to safeguard memory areas. Similar hardware features are often provided to protect secondary storage areas.

In total, a substantial amount of processor capacity is consumed by device management. This overhead is usually worthwhile since device management can significantly increase system throughput by decreasing system I/O wait time.

5-6 VIRTUAL DEVICES

5-6.1 Motivation

Devices such as card readers, punches, and printers present two serious problems that hamper effective device utilization.

First, each of these devices performs best when there is a steady, continuous stream of requests at a specified rate that matches its physical characteristics (e.g., 1,000 cards per minute read rate, 1,000 lines per minute print rate, etc.). If a job attempts to generate requests faster than the device's performance rate, the job must wait a significant amount of time. On the other hand, if a job generates requests at a much lower rate, the device is idle much of the time and is substantially underutilized.

Second, these devices must be dedicated to a single job at a time. Thus, since most jobs perform some input and output, we would require as many card readers and printers as we have jobs being multiprogrammed. This would normally be uneconomical because most jobs use only a fraction of the device's capacity.

To summarize, these devices should have a steady request rate, whereas actual programs generate highly irregular requests. A small input/output-intensive program (e.g., a check-printing program) might generate thousands of print requests in a few seconds of CPU time. On the other hand, a compute-intensive program (e.g., determine the first 1,000 prime numbers) may print only a single line of output for every hour of CPU time. Buffering and multiprogramming, as illustrated in Chapter 2, help reduce these problems but are not capable of solving them completely. Furthermore, many other devices, such as plotters, graphic displays, and typewriter terminals, have similar problems.

5-6.2 Historical Solutions

These problems would disappear, or at least substantially decrease, if it were possible to use Direct Access Storage Devices for all input and output. A single DASD can be efficiently shared and simultaneously used for reading and/or writing data by many jobs, as shown in Figure 5-12. Furthermore, DASDs provide very high performance rates, especially if the data are blocked, thereby decreasing the amount of wait time for jobs that require substantial amounts of input/output.

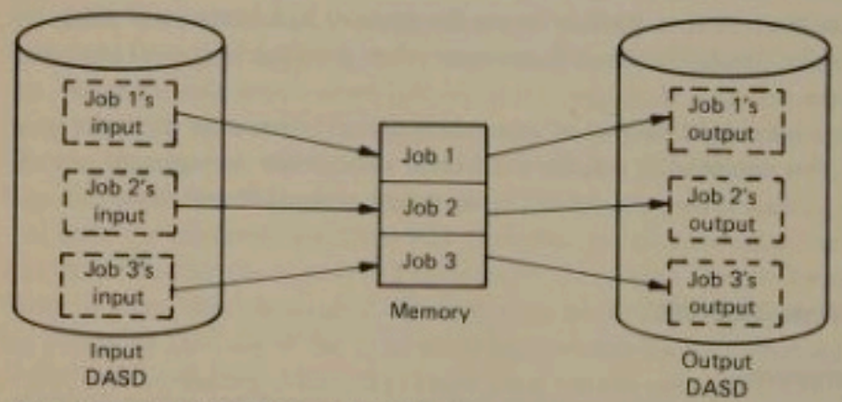


Figure 5-12 Use of DASDs for input and output

Although theoretically appealing, the use of DASDs for all input and output appears impractical. How do we get our input data onto the DASD in the first place? What do we do with the output data recorded on the output DASD? Consider for a moment walking into a drugstore with a disk pack and trying to convince the clerk that recorded on a specific part of the disk is your payroll check—and you wish to cash it!

5-6.2.1 OFFLINE PERIPHERAL OPERATIONS

Fortunately, a solution to this dilemma can be found in the three-step process illustrated in Figure 5-13. At step 1 we use a separate computer whose sole function is to read cards at maximum speed and record the corresponding information on a DASD. Two or more card readers may be used by computer 1, depending upon the amount of input that must be transcribed.

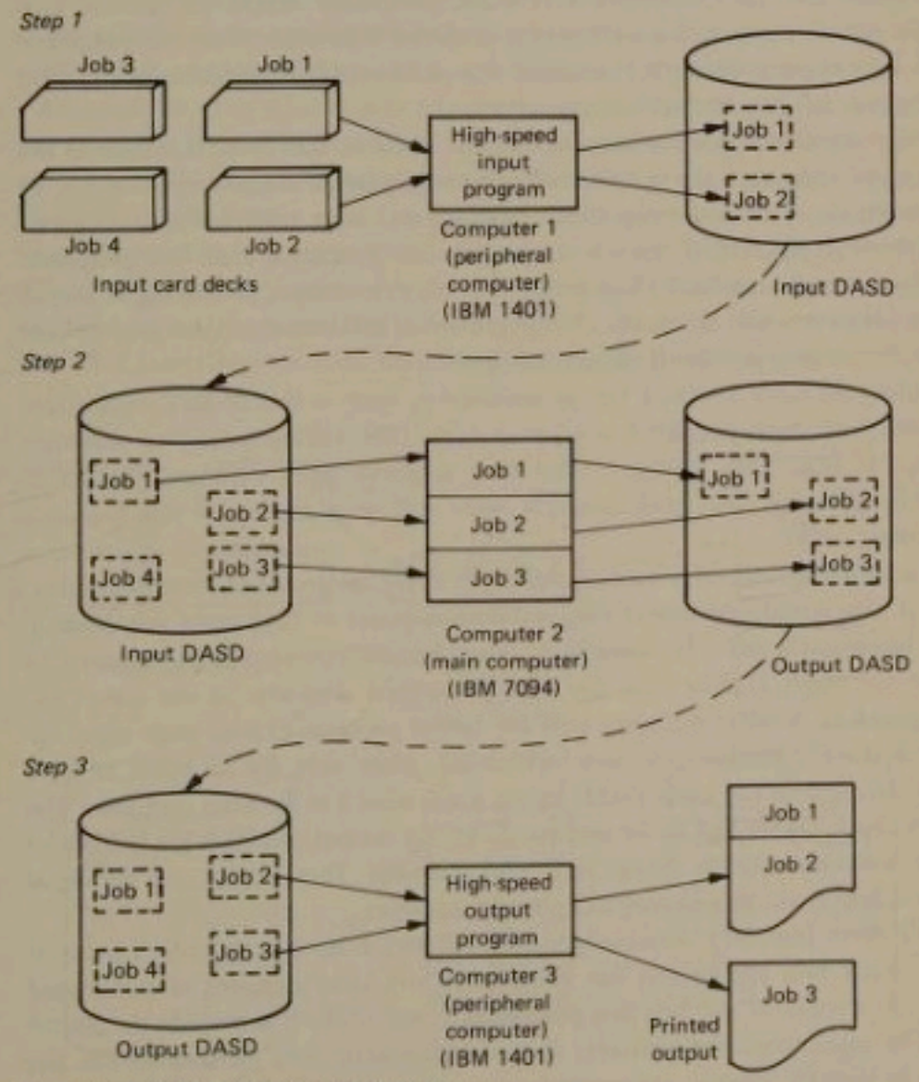
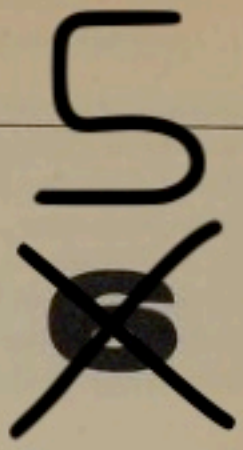


Figure 5-13 Offline peripheral operations

Information Management



6-1 INTRODUCTION

Information management is concerned with the storage and retrieval of information entrusted to the system in much the same manner as a library. The following are the basic functions of information management.

1. Keeping track of all information in the system through various tables, the major one being the file directory—sometimes called the Volume Table of Contents (VTOC). These tables contain the name, location, and accessing rights of all information within the system.
2. Deciding policy for determining where and how information is stored and who gets access to the information. Factors influencing this policy are efficient utilization of secondary storage, efficient access, flexibility to users, and protection of access rights to the information requested.
3. Allocating the information resource. Once the decision is made to let a process have access to information, the allocation modules must find the desired information, make the desired information accessible to the process, and establish the appropriate access rights.
4. Deallocating of the resource. Once the information is no longer needed, temporary table entries and other such resources may be released. If the user has updated the information, the original copy of that information may be updated for possible use by other processes.

The modules of information management are sometimes collectively referred to as the *file system*. Information management is conceptually simple. Yet information is one of the most important resources of an operating system and perhaps the one most poorly managed in contemporary systems.

In Figure 1-9 we placed the modules of information management in a single outer ring of the kernel of the operating system. However, a full implementation

of this manager can be quite complicated. Thus our approach is to further delineate the modules of the information manager into levels, as shown in Figure 6-1. Such a delineation was first presented by Madnick (1969) and is a framework for implementing a file system.

Each level of the file system follows structured programming techniques. Each level depends only on levels below it and only calls downward. This hierarchical approach for the design of the file system does not require that the rest of the operating system be implemented in the hierarchical manner implied in Figure 6-1.

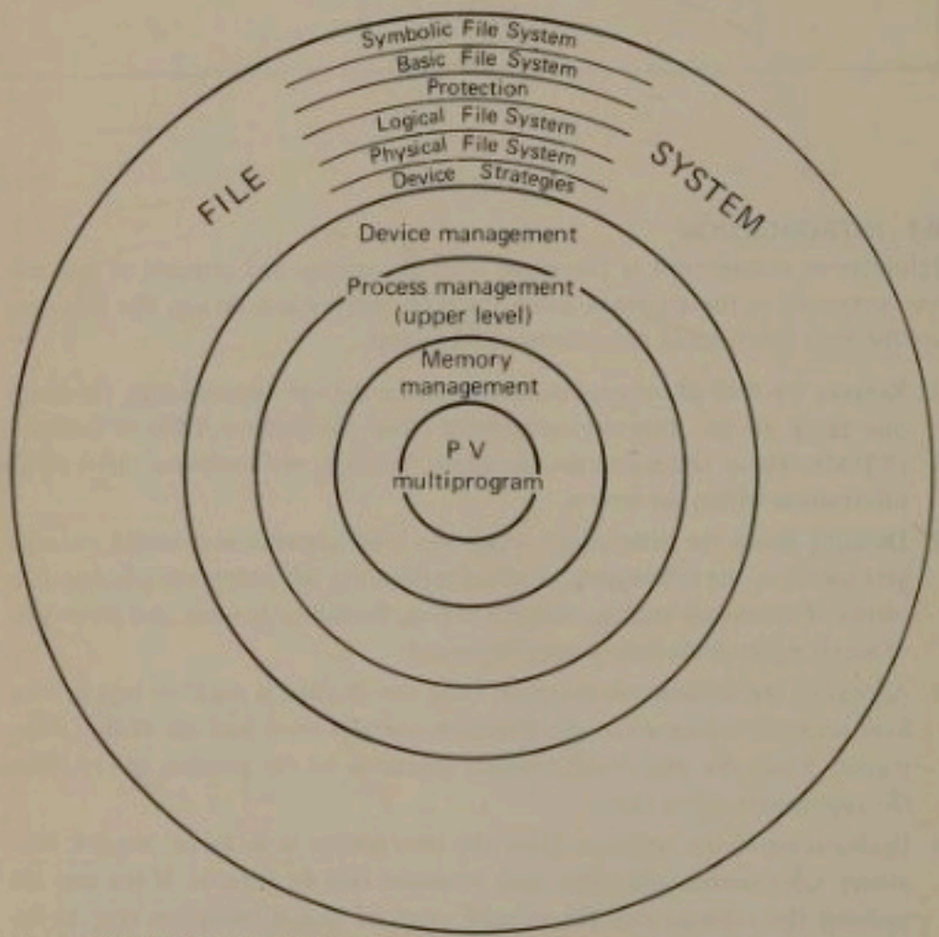


Figure 6-1 Kernel of an operating system

The file system is intended to provide convenient management of both permanent (long-term) and temporary (short-term) information. The programmer is freed from problems such as the allocation of space for his information, physical storage formats, and I/O accessing. One goal of a file system is to allow the programmer to concern himself only with the logical structure and operations performed in processing his information. File systems may also make it possible to share information among users and protect information from unauthorized access.

We will present only basic file system concepts in this chapter. It is useful to distinguish among file systems (i.e., basic information management mechanisms), data management systems, and database systems.

A *file system* is concerned only with the simple logical organization of information. It deals with collections of unstructured and uninterpreted information at the operating system level. Each separately identified collection of information is called a *file* or *data set*.

A *data management system* performs some structuring but no interpretation of the information, e.g., the system may know that each file is divided into entries of a certain size and although it allows the programmer to manipulate these entries it doesn't know what they mean (IBM's IMS-II is a data management system).

A *database system* is concerned with both the structure and interpretations of data items. An example is an airline reservations system, which allows a user to ask "How many people are there on flight 904?"

This chapter focuses mainly on file systems. Data management and database systems are discussed only briefly in Section 6-7. A *file* is a collection of related information units (*records*). For example, in an inventory control application, one line of an invoice is a data item, a complete invoice is a record, and a complete set of such records is a file.

File systems have evolved from simple mechanisms for keeping track of system programs to more complex ones that provide users and the operating system with the means for storing, retrieving, and sharing information. Thus the file system is needed by other components of the operating system as well as by user jobs.

Let us first take a simple user request for information and analyze the steps necessary to fulfill that request. From this simple example we will construct a general model of a file system. We then apply that model to design more flexible file systems.

6-2 A SIMPLE FILE SYSTEM

Let us consider the steps needed to process the following PL/I-like statement:

READ FILE (ETHEL) RECORD (4) INTO LOCATION (12000)

This is a request to read the fourth record of the file named ETHEL into a memory location 12000. Such a request invokes elements of the file system.

For the purpose of this example, let us assume that the file ETHEL is stored on a disk, as shown in Figure 6-2 (see Chapter 5 for details on storage devices). The file ETHEL is the shaded portion and consists of seven logical records. Each logical record is 500 bytes long. The disk consists of 16 physical blocks of 1000 bytes

each. (We call the physical units of the storage device *blocks* to differentiate from the logical units of interest to the programmer which are called *records*—these are analogous to PL/I data structures.) We can store two logical records of file ETHEL into each physical block. Instead of using the physical two-component (track number, record number) address, we will assign each physical block a unique number, as indicated in Figure 6-2.

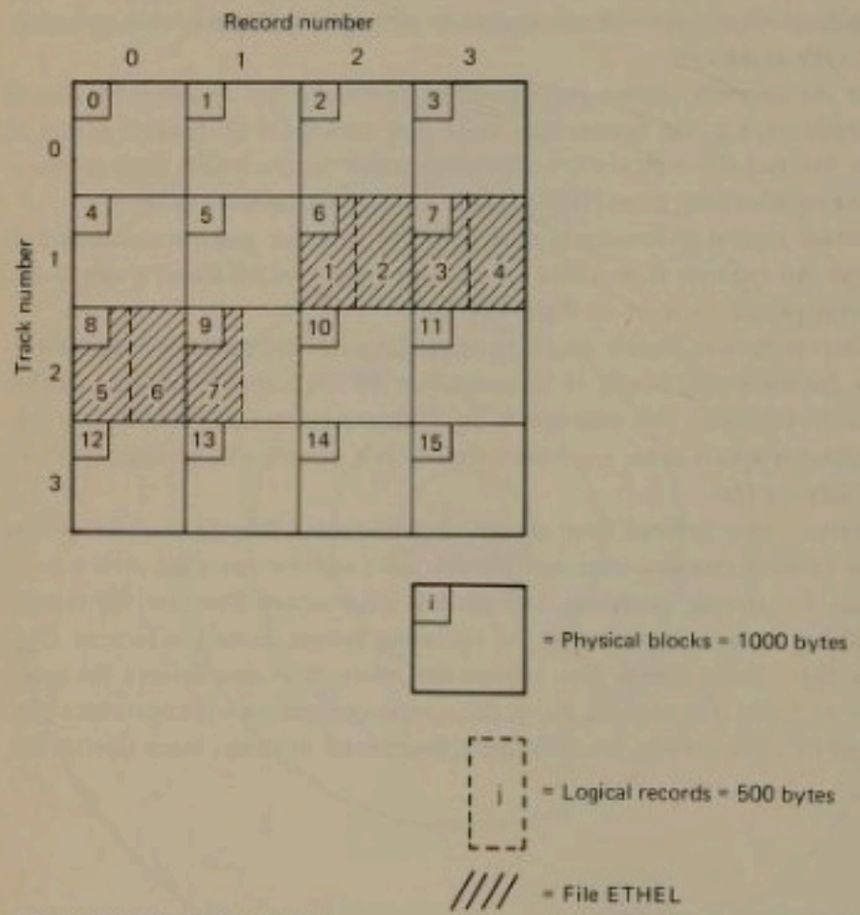


Figure 6-2 Physical file storage

6-2.1 File Directory Database

Information on each existing file's status, for example, name and location, must be maintained in a table. As noted above, this table is often called a file directory or a Volume Table of Contents in many systems, most notably IBM's OS/360.

The name Volume Table of Contents comes from viewing the storage device medium as a "container of information" and thus a *volume*. The distinction between a storage device and a volume becomes important when removable volumes (e.g., disk pack, tape reel) are used. These volumes can be physically removed from

their original storage device and placed on another storage device of the same type, sometimes even on a different computer system.

Figure 6-3 depicts a sample file directory. Each entry contains information necessary to find and access a file. The logical organization, i.e., programmer's view, is described in terms of the logical record length and the number of records in the file. (At this time we assume that all records within a file are the same length; other file organizations are presented later in Section 6-7.)

The physical location of the file on the storage device can be determined by means of the starting block number information in conjunction with the logical record length and number of records. File ETHEL, for example, consists of seven 500-byte records that require 3500 bytes of storage. Since the physical blocks are 1000 bytes long, four blocks are needed to hold file ETHEL. The first of these blocks, noted in the file directory, is 6. Thus, ETHEL occupies physical storage blocks 6, 7, 8, and 9, as previously depicted in Figure 6-2.

An important point here is that we distinguish between logical structuring and physical structuring. The file ETHEL is logically structured from the user's point of view as a sequential file, i.e., it consists of seven records numbered 1 through 7. It is also physically stored as a sequential file, i.e., its records are stored adjacent to one another in consecutive physical blocks. However, the physical structure of ETHEL need not be sequential. The blocks of ETHEL could be scattered all over secondary storage, and a mapping between the logical records and the physical blocks might then become quite complicated. Examples of other such structurings are presented later in Section 6-8.

There are six entries listed in the file directory of Figure 6-3. Three of these

ENTRY NUMBER	NAME	LOGICAL RECORD SIZE	NUMBER OF LOGICAL RECORDS	ADDRESS OF FIRST PHYSICAL BLOCK	PROTECTION AND ACCESS CONTROL
1	MARILYN	80	10	2	Access by everyone
2	Free	1000	3	3	(Free)
3	ETHEL	500	7	6	Read only
4	JOHN	100	30	12	Read for MADNICK Read/Write for DONOVAN
5	Free	1000	2	10	(Free)
6	Free	1000	1	15	(Free)

Figure 6-3 Sample file directory

(MARILYN, ETHEL, and JOHN) correspond to real user files. The other three entries are needed merely to account for presently unused storage space. Thus the file directory indicates the usage of all the physical storage blocks. (*Note:* blocks 0 and 1 are reserved for special purposes to be explained later.)

6-2.2 Steps to Be Performed

Figure 6-4 presents the steps that must be performed to satisfy a request, such as:

READ FILE (ETHEL) RECORD (4) INTO LOCATION (12000)

These steps utilize the sample file directory depicted in Figure 6-3. The reader should follow through the steps, one at a time, to confirm the correctness of the algorithm.

STEP NUMBER	ACTION
Step 1	The file directory is searched to find the entry for ETHEL (entry number 3).
Step 2	Information about ETHEL is retrieved and extracted from the file directory entry, namely: logical record size, address of first physical block, and protection and access control information.
Step 3	Based on the protection information, a decision is made whether to allow the requesting process to access ETHEL.
Step 4	The Logical Record Address specified (record 4 of ETHEL) is transformed into a Logical Byte Address Logical Byte Address = (record number −1) × (record size) which is (4−1) × (500) = 1500.
Step 5	The Logical Byte Address is then transformed into a Physical Byte Address. Logical Byte Address 1500 corresponds to byte 500 within the second 1000-byte physical block of ETHEL, which corresponds to physical block 7.
Step 6	Physical block 7 is read into a 1000-byte buffer area in main memory. This is accomplished by an I/O program similar to those described in Chapters 2 and 5.
Step 7	Record 4 is extracted from the buffer area and moved into the user's area at location 12000. That is, locations 500 through 999 of BUFFER are moved to locations 12000 through 12499 in main memory.

Figure 6-4 Steps involved in processing READ request

6-3 GENERAL MODEL OF A FILE SYSTEM

In this section a general model of a file system is introduced (Madnick, 1969). Figure 6-5 indicates the key components of a file system. Most of these components correspond to generalizations of specific steps of Figure 6-4.

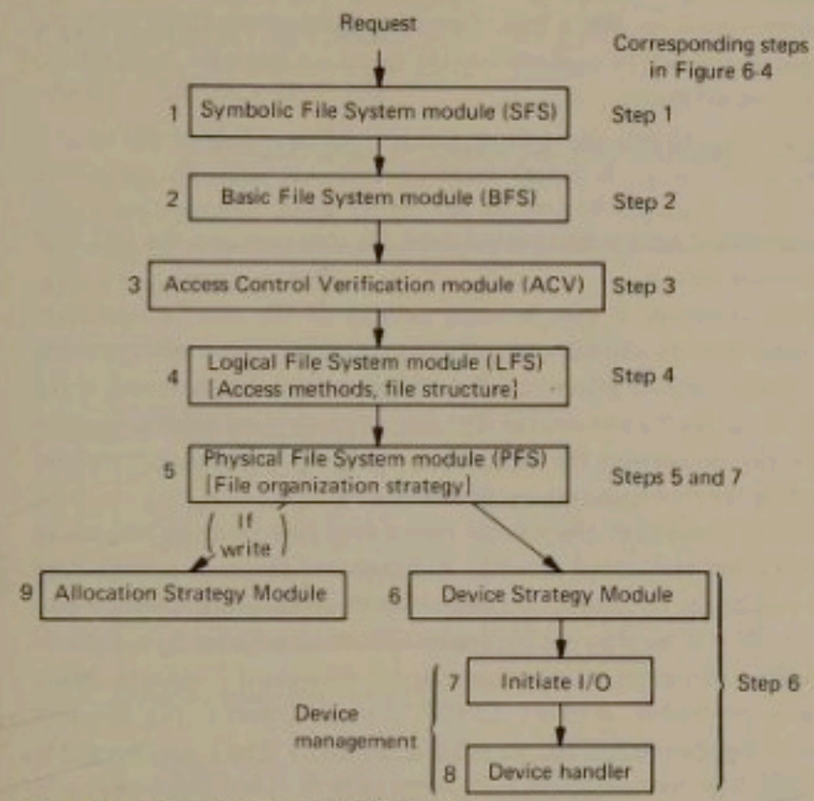


Figure 6-5 Hierarchical model of a file system

Reflecting the ideas introduced in Chapter 1, the components in this file system are organized as a structured hierarchy. Each level in the hierarchy depends only on levels below it. This concept improves the possibility of systematic debugging procedures. For example, once the lowest level is debugged, changes at higher levels have no effect. Debugging can proceed from lowest to highest levels.

Although the particular details presented in this chapter (e.g., format of file directory, file maps) may vary significantly from system to system, the basic structure is common to most contemporary file systems. For example, depending on the structure of a specific file system, certain of the modules in Figure 6-5 may be merged together, further subdivided, or even eliminated—but the underlying structure should still be the same.

6-3.1 File Directory Maintenance

Before discussing the functions of the file system components, it is important to answer certain questions about the file directory, such as: How are the file directory entries created and filled in? Where is the file directory stored? Is it necessary to search the entire directory for every request?

In a basic system such as IBM's Basic Operating System/360 (BOS/360), the programmer must keep track of available storage space and maintain the file directory by control cards similar to:

```
CREATE ETHEL, RECSIZE=500, NUMRECS=7, LOCATION=6
DELETE JOHN
```

The CREATE command adds a new entry to the file directory and the DELETE command removes an old one.

If the entire file directory is kept in main memory all the time, a significant amount of memory may be needed to store it. A more general approach is to treat the file directory as a file and place it on the storage volume. Furthermore, if the file directory is stored on the volume, the files may be easily transferred to another system by physically transferring the volume (tape reel, disk pack, etc.) containing these files as well as the corresponding file directory.

If the file directory is stored on the volume, then it may take a substantial amount of time to search it for each access to a file. Although the directory may be quite large, possibly containing thousands of entries, usually only a few files are ever used at one time. Thus if we copy the entries for files that are currently in use into main memory, the subsequent search times can be substantially reduced. Many file systems have two special requests, OPEN, to copy a specific file directory entry into main memory, and CLOSE, to indicate the entry is no longer needed in main memory. Likewise, we talk about a file being *open* or *closed* depending upon the current location of its file directory entry.

6-3.2 Symbolic File System

The first processing step is called the Symbolic File System (SFS). A typical call would be:

CALL SFS (function, file name, record number, location),

such as:

CALL SFS (READ, "ETHEL", 4, 12000)

The Symbolic File System uses the file name to locate that file's unique file directory entry. This function corresponds to step 1 in Figure 6-4.

In our simple system we had assumed a one-to-one correspondence between file names and files. In more advanced file systems, the facility must exist to have several files with the same name and be able to call the same file by different names. To implement such facilities, we divide the file directory of Figure 6-3 into two separate

directories, a Symbolic File Directory (SFD) and a Basic File Directory (BFD) as shown in Figure 6-6. The symbolic file system must search the symbolic file directory to determine a unique identifier (ID) for the file requested (see Figure 6-6a). This ID is used instead of the file's name to locate entries in the Basic File Directory. The Basic File System allows operations on files using the ID such as:

CALL BFS (READ,5,4,12000)

where 5 is ETHEL's ID.

Since the SFDs are usually kept on the storage device (see ID 2 in Figure 6-6b), entries for files that are currently in use (called *active* or *opened* files) are copied into main memory. These copies can be used to eliminate repeated I/O access to the same entry on the storage device. We will call this table of active file entries the Active Name Table (ANT).

	NAME	UNIQUE IDENTIFIER (ID)
1	MARILYN	3
2	ETHEL	5
3	JOHN	6

16 bytes — 4 bytes
20 bytes per entry

(a) Sample Symbolic File Directory

ID	LOGICAL RECORD SIZE	NUMBER OF LOGICAL RECORDS	ADDRESS OF FIRST PHYSICAL BLOCK	PROTECTION AND ACCESS CONTROL
1	–	–	0	(Basic File Directory)
2	20	3	1	(Symbolic File Directory)
3	80	10	2	Access by everyone
4	1000	3	3	(Free)
5	500	7	6	Read only
6	100	30	12	Read for MADNICK Read/write for DONOVAN
7	1000	2	10	(Free)
8	1000	1	15	(Free)

(b) Sample Basic File Directory

Figure 6-6 Sample Symbolic and Basic File Directories

In summary the Symbolic File System is responsible for:

1 The Symbolic File Directory
2 The Active Name Table
3 Communication with the Basic File System.

6-3.3 Basic File System

The second processing step is called the Basic File System (BFS). A typical call would be:

CALL BFS (function, file ID, record number, location),

such as:

CALL BFS (READ, 5, 4, 12000)

which was shown in Section 6-3.2. The Basic File System uses the file ID to locate that file's entry in the Basic File Directory (see Figure 6-6b) and copies the entry into main memory. This function corresponds to step 2 in Figure 6-4.

By permanently assigning the Symbolic File Directory a specific ID (e.g., ID 2 in Figure 6-6b), the SFS can use the facilities of the BFS to search the directory for the desired name and ID. For example, the PL/I program segment

```
        DECLARE 1   SFD_ENTRY,
                    2 NAME  CHARACTER (16),
                    2 ID    FIXED BINARY (31);
        DO    I = 1 TO 3;
              CALL  BFS (READ, 2, I, SFD_ENTRY);
              IF  SFD_ENTRY.NAME = DESIRED_NAME
              THEN GO TO FOUND;
              END;
FOUND:        DESIRED_ID = SFD_ENTRY.ID;
```

could be used to search the symbolic file directory.

The BFD entries for active files are saved in an Active File Table (AFT). This eliminates repeated access to the BFD for the same entry.

The AFT entry is used instead of the file ID for the next step, the Access Control Verification (ACV), which processes calls such as

CALL ACV (READ, 2, 4, 1200)

where AFT entry 2 contains the information for file ID 5 (file ETHEL).

In summary, the Basic File System is responsible for:

1 The Basic File Directory
2 The Active File Table
3 Communication with the Access Control Verification module

For the simple file system of Section 6-2, it is quite easy to combine the SFS and BFS functions. In later sections we will introduce additional functionality where this logical separation is much more significant.

6-3.4 Access Control Verification

The third processing step is called Access Control Verification (ACV). A typical call would be

CALL ACV (function, AFT entry, record number, location)

such as

CALL ACV (READ, 2, 4, 12000)

which was shown in Section 6-3.3. The Access Control Verification acts as checkpoint between the Basic File System and the Logical File System. It compares the desired function (e.g., READ, WRITE) with the allowed accesses indicated in the AFT entry. If the access is not allowed, an error condition exists and the file system request is aborted. If the access is allowed, control passes directly to the Logical File System. This function corresponds to step 3 in Figure 6-4.

6-3.5 Logical File System

The fourth processing step is called the Logical File System (LFS). A typical call would be

CALL LFS (function, AFT entry, record number, location)

such as

CALL LFS (READ, 2, 4, 12000)

which is identical, in form, to the ACV call shown in Sections 6-3.3 and 6-3.4. The Logical File System converts the request for a logical record into a request for a logical byte string. That is, a file is treated as a sequential byte string without any explicit record format by the Physical File System. In this simple case of fixed-length records, the necessary conversion can be accomplished by using the record length information from the AFT entry. That is, the

Logical Byte Address = (Record Number − 1) × Record Length

and

Logical Byte Length = Record Length

This function corresponds to step 4 in Figure 6-4.

The Logical Byte String Address and Length are used instead of the record number for the next step, the Physical File System, which processes calls such as

CALL PFS (READ, 2, 1500, 500, 12000)

Note that by permanently assigning the BFD file a specific ID (such as ID 1 in Figure 6-6b) and a specific AFT entry number (such as AFT entry 1), the Basic File System can call upon the Logical File System to read and write entries in the Basic File Directory. (It is necessary to have a special procedure for fetching BFD entry 1 into AFT entry 1 when the system is initially started or restarted.)

In summary, the Logical File System is responsible for:

1 Conversion of record request to byte string request
2 Communication with the Physical File System

Examples of more complex logical record organizations, such as variable-length records, are presented in later sections.

6-3.6 Physical File System

The fifth processing step is called the Physical File System (PFS). A typical call would be

CALL PFS (function, AFT entry, byte address, byte length, location)

such as

CALL PFS (READ, 2, 1500, 500, 12000)

which was shown in Section 6-3.5. The Physical File System uses the byte address plus the AFT entry to determine the physical block that contains the desired byte string. This block is read into an assigned buffer in main memory, using the facilities of the Device Strategy Module, and the byte string is extracted and moved to the user's buffer area. This function corresponds to steps 5 and 7 in Figure 6-4.

The mapping from Logical Byte Address to Physical Block Number, for the simple contiguous organization of Figure 6-2, can be accomplished by:

$$\text{Physical Block Number} = \frac{\text{Logical Byte Address}}{\text{physical block size}} + \text{address of first physical block}$$

For the request for the byte string starting at Logical Byte Address 1500, the Physical Block Number is:

$$\text{Physical Block Number} = \frac{1500}{1000} + 6 = 1 + 6 = 7$$

which is the block containing record 4 of file ETHEL (see Figure 6-2).

The location of the byte string within the physical block can be determined by:

$$\text{Physical Block Offset} = \text{remainder}\left[\frac{\text{Logical Block Address}}{\text{physical block size}}\right]$$

such as:

$$\text{Physical Block Offset} = \text{remainder}\left[\frac{1500}{1000}\right] = 500.$$

Thus, the byte string starts at the offset 500 within the physical block, as expected from Figure 6-2.

In order to perform these calculations, the Physical File System must know the mapping functions and physical block size used for each storage device. If these were the same for all devices, the information could be built into the PFS routines. Usually there are variations depending upon the device type (e.g., a large disk may be handled differently from a small drum). Thus, this information is usually kept on the storage volume itself and read into the Physical Organization Table (POT) when the system is first started.

If a WRITE request were being handled and the corresponding physical block had not been assigned, the Physical File System would call upon the Allocation Strategy Module (ASM) for the address of a free block of secondary storage. In the simple file system of Section 6-2, all the blocks are allocated in advance by a special CREATE facility, as discussed in Section 6-3.1.

In summary, the Physical File System is responsible for:

1 Conversion of logical byte string request into Physical Block Number and Offset
2 The Physical Organization Table
3 Communication with the Allocation Strategy Module and the Device Strategy Module

Later sections will introduce many more physical system organizations and performance considerations that have an impact on the physical file system.

6-3.7 Allocation Strategy Module

The Allocation Strategy Module (ASM) is responsible for keeping track of unused blocks on each storage device. A typical call would be:

CALL ASM (POT entry, number of blocks, first block)

such as:

CALL ASM (6, 4, FIRSTBLOCK)

where POT entry 6 corresponds to the storage device on which file ETHEL is to

reside and FIRSTBLOCK is a variable in which the address of the first of the four blocks requested is returned.

The ASM may be invoked by a special setup procedure, such as the CREATE command of Section 6-3.1, or automatically by the Physical File System as indicated in Figure 6-5.

Figure 6-6b indicates how the location of groups of available blocks can be recorded in the Basic File Directory by treating them as special files. Other techniques for maintaining this information are presented in subsequent sections.

6-3.8 Device Strategy Module

The Device Strategy Module (DSM) converts the Physical Block Number to the address format needed by the device (e.g., physical block 7 = track 1, record 3, as shown in Figure 6-2). It sets up appropriate I/O commands for that device type. This function corresponds to step 6 in Figure 6-4. All previous modules have been device-independent. This one is device-dependent. Control then passes to the I/O scheduler.

6-3.9 I/O Scheduler and Device Handler

These modules correspond to the device management routines that were discussed in Chapter 5. They schedule and perform the reading of the physical block containing the requested information.

6-3.10 Calls and Returns Between File System Modules

After the physical I/O is completed, control is returned to the Device Strategy Module. The DSM checks for correct completion and returns control to the PFS, which extracts the desired information from the buffer and places it into the desired location. A "success" code is then returned back through all other modules of the file system.

Why isn't an immediate return made to the uppermost level? There are two reasons: (1) An error may be detected at any level, and that level must return the appropriate error code (for example, the Logical File System may detect the attempted access of an address beyond the file); and (2) any level may generate several calls to lower levels (for example, the Symbolic File System may call the Basic File System several times to read entries from the Symbolic File Directory).

A function of all levels of Figure 6-5 is verification, which is substantiating the validity of the request. The Symbolic File System checks to see whether or not a file exists. The Access Control Verification checks to see whether or not access is permitted. The Logical File System checks to see whether the requested logical address is within the file. The Device Strategy Module checks to see whether or not such a device exists.

6-3.11 Shortcomings of Simple File System Design

Let us summarize some of the assumptions, inefficiencies, and inadequacies of our simple file-system system:

1 *Symbolic File System*
We assumed that each file had a single unique name.

2 *Access Control Verification*
How are files shared? How can the same file have different protection rights for different uses?

3 *Logical File System*
We assumed that each file consisted of fixed-length records, and that these records were viewed as sequential. How are accesses to variable-length records handled? Are there other file structuring methods? Are there more flexible methods for the user?

4 *Physical File System*
Are there physical structurings other than sequential? Are these more efficient in space? In accesses? We assumed that for each request this module would activate an access to the disk. What if the request following READ ETHEL RECORD (4) were READ ETHEL RECORD (3)? Isn't record 3 in the system buffer area already?

5 *Allocation Strategy*
How are files stored on secondary storage? Are there fragmentation problems when files are deleted? In other words, is secondary storage being used efficiently?

6 *Device Strategy Module*
Can this module restructure or reorder requests to match more efficiently the characteristics of the device on which the information is stored?

In the sections that follow we will explore more general techniques for each of these levels, techniques that give the user more flexibility and may thus increase the system's efficiency.

6-4 SYMBOLIC FILE SYSTEM

The SFS's function is to map the user's symbolic reference to an ID. Let us develop a scheme that will offer additional features beyond those of Section 6-3.

1 Multiple Names Given to the Same File

This feature, sometimes called *aliases*, could be useful if several different programs assumed different names for the same file. The user may also find certain names to be more meaningful under different circumstances.

2 Sharing Files

This feature, sometimes called *linked* files, gives multiple users access to the same physical file, offering them more flexibility and, frequently, greater system efficiency. This is because buffers in main memory as well as secondary storage space can be shared. If, for example, several users are accessing the same file, only one master copy of that file need exist. An early version of M.I.T.'s CTSS timesharing system (see Chapter 9) provides an example of the inefficiency that can result from a system that does not allow the sharing of files. There was a very popular program called DOCTOR that simulated a psychiatrist. Over half the users on the system had a separate copy of this program. Due to the large size of this program, there was a time when over half the secondary storage consisted of copies of DOCTOR.

3 The Same Name Given to Different Files

Since many users create and use files, it is possible that two different programmers may accidently choose the same name for a file (e.g., TEST). Thus when one of them uses the control cards as follows:

```
DELETE TEST /* delete my old copy*/
CREATE TEST /* create a new copy*/
```

he may be deleting the other programmer's file TEST. Likewise, the READ and WRITE requests will use the same file for both programmers. Maintaining a manual file name registry to avoid duplicate names would be a time-consuming and awkward solution.

4 Subdividing Files into Groups

A given user (or group of users) may be responsible for a large number of files. These files may logically belong to separate projects or tasks. To help keep track of these files in an organized way, it is desirable to group related files together. Such a group is often called a *library* (or *partitioned data set* in IBM terminology).

The symbolic file name is a very convenient way for a programmer to remember and refer to this file, but we have noted several shortcomings to the simple file directory scheme presented. We therefore suggest a more flexible file-naming scheme that incorporates all the additional features we have listed above.

6-4.1 Directory Files

Notice that in Figure 6-6 we separated the file directory into two separate components: the Symbolic File Directory and the Basic File Directory. The SFD is

concerned with file naming whereas the BFD is concerned with the physical file. We can generalize this mechanism in two ways: (1) have multiple symbolic directories, and (2) treat symbolic directories as files.

To eliminate or at least minimize naming conflicts, each programmer or programmer group can be assigned a separate SFD. Thus there is a need for a Master Directory that specifies the BFD entry for each programmer's private SFD.

Note that in Figure 6-6b the Master Directory has a unique ID, i.e., it is always entry 2 in the Basic File Directory. This allows the Symbolic File System to access the Master Directory via the Basic File System.

6-4.2 Sample Hierarchical File Structure

Figure 6-7 illustrates the details of a simple hierarchical file structure consisting of seven data files and four SFD files in addition to the Basic File Directory. The

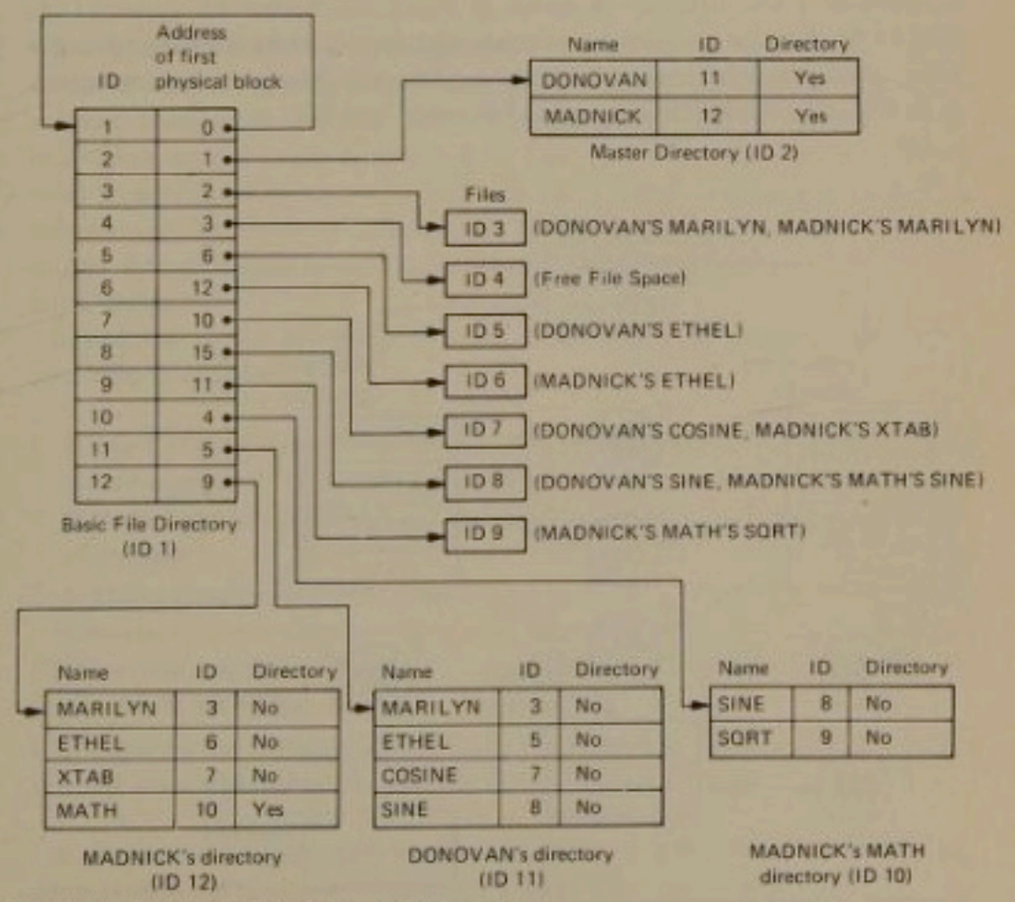


Figure 6-7 Detailed hierarchical file structure

first BFD entry is for the BFD itself; the second entry is always assigned to the Master Directory file. Remember that for all practical purposes it is just like every other file with a file directory entry, containing record length, etc.

The Master Directory file has an entry for each programmer's Symbolic File Directory. In this case, there are two programmers, DONOVAN and MADNICK. Each programmer's directory has entries for all files accessible to him. DONOVAN has four files, MARILYN, ETHEL, COSINE, and SINE, whereas MADNICK has four files, MARILYN, ETHEL, XTAB, and MATH where MATH itself is a Symbolic File Directory with entries for two files, SINE and SQRT.

Notice that the file MARILYN is really the same file (i.e., same ID) for both programmers; this is often called a *linked file*. On the other hand, DONOVAN's file ETHEL is not the same as MADNICK's file ETHEL (i.e., different IDs). There are several other flexible features illustrated in this example. (Note that DONOVAN's file COSINE and MADNICK's file XTAB are really the same file.) The reader should confirm these observations by studying Figure 6-7.

A *hierarchical file organization* such as the one described above is often diagrammed as a tree structure as shown in Figure 6-8. Several file systems (e.g., MULTICS) allow the programmer to create multilevel subsidiary directories, sometimes called private libraries, so that the structure may become several levels deep, as illustrated by the MATH directory in Figures 6-7 and 6-8.

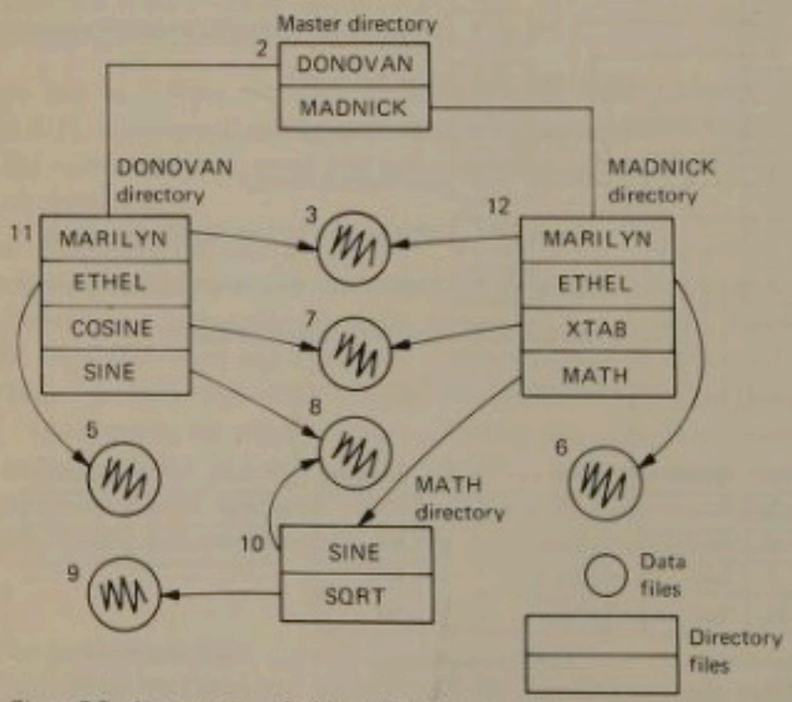


Figure 6-8 Abstract view of a hierarchical file structure

6-5 BASIC FILE SYSTEM

When the Basic File System is called, it is given the unique ID of the file requested. The BFS first searches the AFT to determine whether the file is already "opened." If not, the Basic File System searches the Basic File Directory for the entry of the requested file. It then places the information from the entry into the AFT.

If the unique ID assigned to a file is the same as the file's entry number in the Basic File Directory, there is no "searching" required to find the entry in the BFD.

Under certain circumstances, such as when a user wishes to delete a file (e.g., DELETE DONOVAN'S ETHEL), it is desirable to know if this file appears in more than one Symbolic File Directory entry. Frequently, this is accomplished by keeping a count in the BFD entry of the number of SFD entries that point to it. Whenever a link or alias is made, this count is increased. Whenever a link or alias is removed, this count is decreased. (When the file itself is deleted, the Symbolic File System must find and delete all corresponding Symbolic File Directory entries.)

6-6 ACCESS CONTROL VERIFICATION

Access control and sharing must be considered in conjunction with each other. For example, a user could protect his information by placing the entire computer system in a locked vault accessible only to him. In some of its installations the government takes an approach somewhat similar to this one by having three separate computer systems—one for top secret information, another for secret information, and yet another for confidential material. Usually, however, this extreme form of protection is not desirable.

In addition to user flexibility, there are several system advantages in allowing a controlled sharing facility:

1. It saves duplication of files. On one system, for example, there was a single shared copy of a certain 128K program that resided on disk. When the shared copy was removed, about 50 private copies appeared equal to about 6.4M bytes of disk space.
2. It allows synchronization of data operations. If two programs are updating bank balances, it is desirable that they both refer to the exact same file at all times. (Note that appropriate interlocks, as discussed in Chapter 4, are needed to coordinate the update actions.)
3. It improves the efficiency of system performance, e.g., if information is shared in memory, a possible reduction in I/O operations may occur.

In this section we will suggest three techniques for keeping track of and enforcing access control rights—access control matrix, passwords, and cryptography.

the code key. The user may change that argument whenever he wishes. There are many techniques for encoding information (Skatrud, 1969). One is simply to use the key to start a random number generation. The encoder adds successive random numbers to the bytes of the file to be encoded. Decoding subtracts these numbers to obtain the original data back. An authorized user knows how to start the random number generator with the same code key for decoding as was used for encoding.

In this scheme the code key does not have to be stored in the system. The user needs to enter it only while he is encoding or decoding the file. Thus, there is no table that a dishonest systems programmer can read to find all code keys or allow himself access. Cryptography has the advantage of not having the key stored in the system, but it does incur the cost of encoding and decoding of files.

6-6.4 Summary

The access control mechanisms described in this section may be used independently or in combination. The issues of privacy, security, protection, sharing, and access control have received considerable attention by researchers. Additional information can be found in the bibliography: Comber (1969), Conway (1972), Graham (1967), Graham and Denning (1972), Hoffman (1969), Lampson (1969, 1971), Molho (1970), Needham (1972), Scherf (1973), Schroeder (1969), Skatrud (1969), and Weissman (1969).

6-7 LOGICAL FILE SYSTEM

The Logical File System is concerned with mapping the structure of the logical records onto the linear byte-string view of a file provided by the Physical File System. In particular, it must convert a request for a record into a request for a byte string.

In the sample file system of Section 6-2, the Logical File System provides facilities for direct access, based on record number, to any of the records, all of which are the same length. In conventional file systems and data management systems many additional record structures are supported. These are often called *access methods*, such as:

1 Sequentially structured fixed-length records (sequential and direct access)
2 Sequentially structured variable-length records (sequential and direct access)
3 Sequentially structured keyed records
4 Multiple-keyed records
5 Chained structured records
6 Relational or triple-structured records

6-7.1 Sequentially Structured Fixed-length Records

For this record organization, the user views his file as a sequence of fixed-length records (i.e., all records are the same length), as shown in Figure 6-10. This structure is useful for storing "card-image" and "print-image" files as well as SPOOL files.

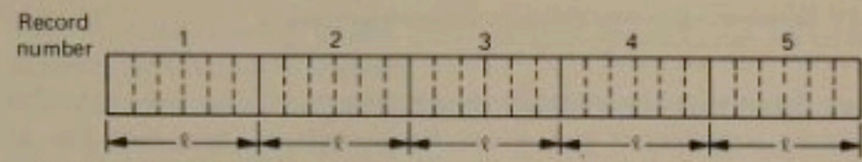


Figure 6-10 Sequential fixed-length records (all records ℓ bytes long)

6-7.1.1 SEQUENTIAL ACCESS

For sequential access, on each request the user wishes to have the "next" record—the same as reading cards from a card reader. This request would look similar to

READ FILE (ETHEL) NEXT INTO LOCATION (BUFFER)

where the explicit record number is omitted.

In order to implement this facility, the Logical File System must maintain a Current Logical Byte Address (CLBA) in the AFT entry for the file. When the file is initially opened, the CLBA is set to 0. Subsequently, after processing each request, the CLBA is updated by

$$CLBA = CLBA + RL$$

where RL is the record length.

6-7.1.2 DIRECT ACCESS

For direct access the user explicitly specifies the record desired. This request is the same as our earlier example:

READ FILE (ETHEL) RECORD (4) INTO LOCATION (BUFFER)

The CLBA can be computed by the simple formula:

$$CLBA = (RN - 1) \times RL$$

where RN is the designated record number.

6-7.2 Sequentially Structured Variable-length Records

For this record organization, the user views his file as a sequence of records of possibly differing lengths. This structure is useful for storing names, employment histories, and printer output (with trailing blanks omitted), as shown in Figure 6-11.

Such a structuring allows a *top down* access, e.g., how many screws are there in a TV; or a *bottom up* access, e.g., what assemblies use screws? We will leave the discussion of the implementation of such systems to the references, such as Chapin (1969) and Senko (1973).

6-7.6 Relational or Triple-structured Records

Another database organization that may be built on top of a basic file system is a relational or triple structure (Codd, 1970). Logical records may bear a relation to other records, e.g., Frank is John's father. Figure 6-17 depicts three relations.

RECORD 1	– RELATION	– RECORD 2
FRANK	– FATHER	– JOHN
JOHN	– BROTHER	– PAUL
JOHN	– FATHER	– JAMES

Figure 6-17 A relational database

Questions that a user may ask of such a system are:

Who is JAMES' FATHER (i.e., ? – FATHER – JAMES)
Name all the FATHERS (i.e., ? – FATHER – ?)

Such systems are discussed in Codd (1970) and Senko (1973).

6-8 PHYSICAL FILE SYSTEM

The primary function of the PFS is to perform the mapping of Logical Byte Addresses into Physical Block Addresses. In the organization of Figure 6-5, the Logical File System calls the Physical File System, passing to it the logical byte address and length of the information requested. The PFS may first call the Allocation Strategy Module (if a write request) and then the Device Strategy Module, or it may call the DSM directly.

In our division of responsibility, the PFS keeps track of the mapping from Logical Byte Address to the blocks of physical storage. In previous sections the simple mapping function (Physical Block Address = first physical block of file $+ \frac{\text{Logical Byte Address}}{\text{block size}}$) was a consequence of the simple physical storage structure of the file. All blocks were stored as in Figure 6-2, i.e., contiguous on secondary storage.

In this section let us briefly discuss three additional considerations:

1 Minimizing I/O operations
2 Allowing logical record size independent of physical block size
3 Allowing noncontiguous allocation of file space

6-8.1 Minimizing I/O Operations

If all the records of file ETHEL (see Figure 6-2) were to be read sequentially, it would require seven I/O operations (each operation reading one physical block) to read the seven logical records using the algorithm presented in Figure 6-4. Since the entire file occupies only four physical blocks, we might suspect that we could reduce the number of I/O operations by eliminating redundant operations.

The number of I/O operations can be reduced by keeping track of which physical blocks are in core. Quite sophisticated algorithms can be used to determine which blocks to keep in core. For each I/O read operation a physical block is copied into a buffer in main memory. If we keep track of which physical block is presently in the buffer, we can modify step 5 of the algorithm (Figure 6-4) to test whether or not the desired block has already been read. If the block is already in the buffer, the I/O read operation (step 6) can be skipped.

In the case of file ETHEL, the request for record 1 will cause physical block 6 to be copied into the buffer. The subsequent request for record 2 will operate on the buffer without the need for any additional I/O operation. The request for record 3 will then cause physical block 7 to be copied into the buffer, and so on. In this way the entire file ETHEL can be processed sequentially with only four I/O operations instead of seven.

This technique, often called *file buffering*, has a number of variations. For example, a separate buffer may be assigned to each opened file, or a pool of buffers may be shared by all files. Furthermore, to minimize the detrimental effects of random accessing, as opposed to sequential accessing, and to allow certain I/O overlapping, it may be desirable to assign multiple buffers to a single file (whence comes the term "double buffering"). The considerations involved with effective file buffering strategies are very similar to those relating to the page replacement algorithms discussed in Section 3-6.2 and the device management buffering discussed in Section 5-3.2.

6-8.2 Allowing Logical Record Size Independent of Physical Block Size

In the simple system of Section 6-2, a physical record must hold some integral number of logical records, with no space left over. There are, unfortunately, situations in practice in which the programmer desires to process logical records of lengths that are not an even factor of block length. There are various ways to solve this problem. For example, we can effectively store fifty 13-byte logical records in the 1000-byte blocks by using 20-byte logical records, which are really 13-byte records each with 7 bytes of wasted space. This strategy, though effective, is not efficient in the use of secondary storage space.

Many storage devices allow a certain amount of flexibility in the format of the individual tracks. For example, some, such as the IBM 2314 and 3330, permit the

programmer to change the track format from sixteen 1000-byte blocks (see Figure 6-1) to thirty 500-byte blocks. This format change is not usually linear since a certain amount of extra space must be used for the Inter-record Gap (IRG) of every block no matter how long it is (see Chapter 5).

In general, the strategy of changing track formats is not very convenient. A more general approach is to modify steps 5 and 7 of Figure 6-4 (as shown in Figure 6-18) and thus allow logical records to straddle physical blocks. This approach also has the benefit of allowing logical records that are larger than the maximum size physical block.

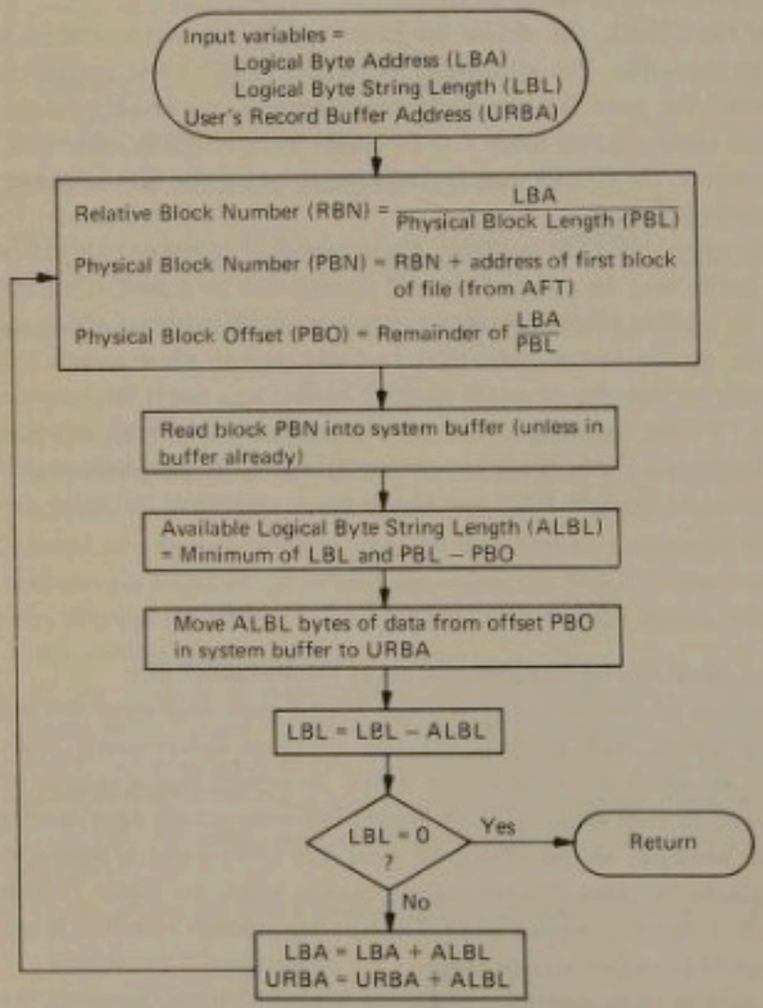


Figure 6-18 Revised version of steps 5, 6, and 7 of Figure 6-4

6-8.3 Allowing Noncontiguous Allocation of File Space

Contiguous allocation of file space, as assumed in the simple system of Section 6-2, is not always practical or possible. Various reasons are presented in Section 6-9 (Allocation Strategy Module) that follows. If the physical blocks of a file are not contiguous, a different algorithm is needed to map from logical byte address to physical block number. Two popular techniques are (1) chained blocks and (2) file maps. There are also many variations on these basic techniques.

6-8.3.1 CHAINED BLOCKS

In a chained blocks mapping, each physical block contains the address of the next "logically contiguous" block. Figure 6-19 shows file ETHEL which has been assigned blocks 6, 9, 10, and 14 (see Figure 6-2 for comparison). The Basic File Directory entries contain the address of the first physical blocks, as before, but subsequent block addresses are found by means of the address pointers (often called *chains*).

The address pointer may be stored within the physical block, thereby decreasing the data capacity of the block slightly (i.e., holds only 998 bytes per block instead of 1000 bytes), or as part of the header field (e.g., key field) of the data block which is possible on many devices such as the IBM 3330.

The chained blocks approach is efficient for sequential access to the blocks (corresponding to sequential access to the logical records) but quite inefficient for direct access for reasons similar to those discussed in Section 6-7.2.2. The chained block technique is used in the CTSS system described in Chapter 9.

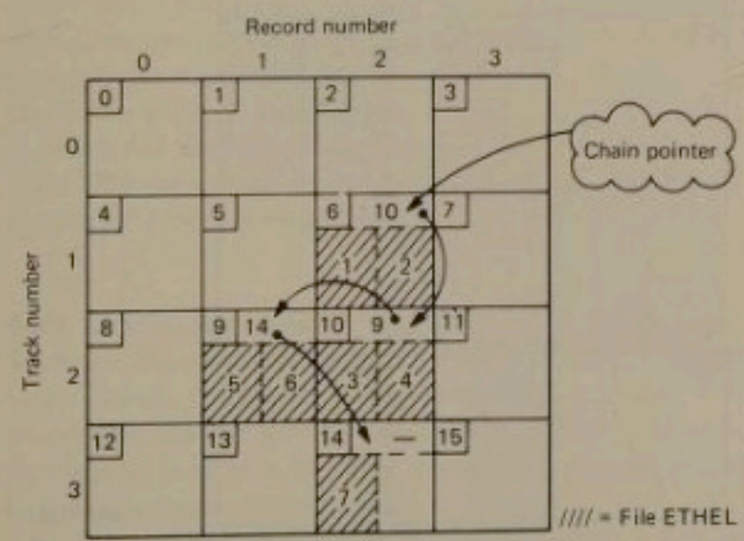


Figure 6-19 Chained blocks

6-8.3.2 FILE MAPS

Another approach to noncontiguous allocation is to use a File Map Table to map each Logical Block Address to its physical block address (similar to the file maps used for paging discussed in Section 3-6.2.1). This file map may be stored as part of the entry in the BFD or in a separate block. Figure 6-20 illustrates the latter approach. Block 6 is used to hold the file map whereas blocks 7, 9, 10, and 14 are used to hold the data records.

When the file is opened, all or part (if the file is especially large) of the file map is copied into memory as part of the Active File Table information. In this way the mapping can be done as efficiently as the contiguous allocation mapping. The mapping function becomes

$$\text{Physical Block Address} = \text{File Map}\left(\frac{\text{Logical Byte Address}}{\text{block size}}\right)$$

which is just an indexed table access.

If the file map is very large, it may require several physical blocks. In this case, it may be necessary to have file maps for the file maps. This is the case with the VM/370 system described in Chapter 9.

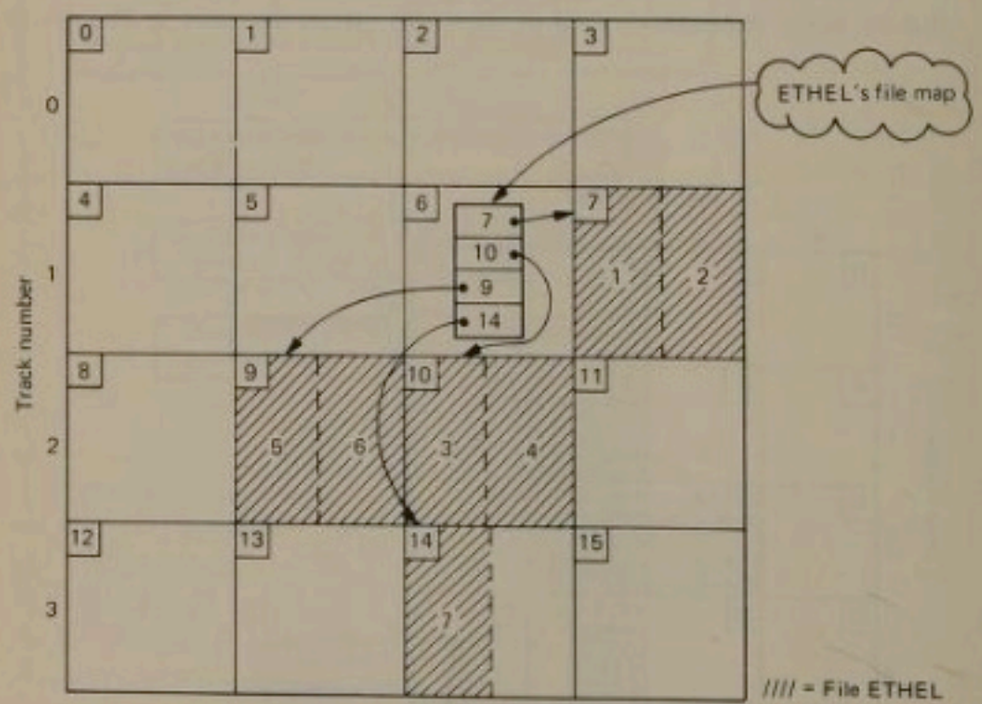


Figure 6-20 File map

6-9 ALLOCATION STRATEGY MODULE

Some systems require manual allocation of file space by using the LOCATION field of a CREATE control card as discussed in Section 6-3.1. This forces the programmer to keep track manually of space, which may be a very cumbersome task when using a typical storage device with 10,000 to 100,000 blocks or more. Also, since there are usually many programmers using the same storage device, it is necessary for a programmer to keep track of everyone else's files if he wishes to know what spaces are available. Assuming contiguous allocation, the following control cards:

```
CREATE ETHEL, RECSIZE = 500, NUMRECS = 7, LOCATION = 6
CREATE JOHN, RECSIZE = 100, NUMRECS = 7, LOCATION = 6
```

which cause files ETHEL and JOHN to overlap, would not be recognized as errors in many simple file systems. (This overlap may even be desirable under some strange circumstances, but it is usually indicative of an error in manual allocation.)

For the reasons discussed above, it is desirable for the programmer merely to specify

```
CREATE ETHEL, RECSIZE = 500, NUMRECS = 7
CREATE JOHN, RECSIZE = 100, NUMRECS = 30
```

and let the file system allocate the necessary space. In general, the programmer has no interest in the specific location anyway.

6-9.1 Automatic Allocation

There are numerous techniques by which the system can keep track of space. One simple way is to maintain "dummy" or "free" file directory entries, one for each contiguous area of blocks not assigned as shown in Figures 6-3 and 6-6b. Then, to satisfy the request for four blocks for ETHEL, the file system looks for a free file directory entry that specifies an area of four blocks or longer. Figure 6-21 illustrates this process for the control cards referred to above.

The IBM OS/360, for example, uses a similar automatic allocation technique (see Chapter 9 for more details).

6-9.2 Dynamic Allocation

There are two problems connected with the automatic allocation technique described above: (1) it leads to fragmentation of secondary storage; and (2) users often don't know in advance how big their files are going to be (e.g., the size of the output object deck file from a compiler is a function of the size and characteristic of the input source program).

As in the case of fragmentation of main memory, two techniques may be used to utilize secondary storage more efficiently: (1) relocation, e.g., some systems, such

ENTRY NO.	NAME	LOGICAL RECORD SIZE	NUMBER OF LOGICAL RECORDS	ADDRESS OF FIRST PHYSICAL BLOCK	PHYSICAL BLOCKS
1	MARILYN	80	40	2	(2, 3, 4, 5)
2	Free	1000	4	6	(6, 7, 8, 9)
3	STU	132	5	10	(10)
4	Free	1000	5	11	(11, 12, 13, 14, 15)
5					
6					

(a) Before CREATES are processed

ENTRY NO.	NAME	LOGICAL RECORD SIZE	NUMBER OF LOGICAL RECORDS	ADDRESS OF FIRST PHYSICAL BLOCK	PHYSICAL BLOCKS
1	MARILYN	80	40	2	(2, 3, 4, 5)
2	ETHEL	500	7	6	(6, 7, 8, 9)
3	STU	132	5	10	(10)
4	JOHN	100	30	11	(11, 12, 13)
5	Free	1000	2	14	(14, 15)
6					

(b) After CREATES are processed

Figure 6-21 Automatic file space allocation

as IBM's 1130 Disk Monitor System, do compact the files on the disks periodically; (2) noncontiguous file space allocation. By using noncontiguous allocation not only is fragmentation avoided but also it is easy to dynamically increase the size of a file during program execution. These are the motivations for use of file maps or chained blocks as introduced in Section 6-8.3.

6-9.2.1 FREE FILE MAPS

The allocation technique depicted by Figure 6-21 works well when there are only a few large free areas. If there are a large number of small free areas, the file directory can become very large and inefficient. One solution to this problem is to use a special file map (similar to those of Section 6-8.3.2) that contains a list of all available blocks. Whenever a block must be allocated, it is removed from this list. Whenever one or more blocks are deallocated (e.g., a file is deleted), these blocks are added to the free file map.

6-9.2.2 FREE CHAIN

Another allocation approach is to chain together all free blocks (in a manner similar to Section 6-8.3.1). In this case, there is no searching required. Whenever a block is needed, it is removed from the front of the chain (the first block in the chain). Conversely, when a block is deallocated, it is placed on the beginning of the chain.

6-9.2.3 BIT MAP

Another popular approach uses a sequence of bits, called a *bit map*, to indicate the status of each of the blocks. The status of the sixteen blocks of Figure 6-21a could be represented by the 16-bit string:

1111110000100000

where a 1 means the corresponding block is in use and a 0 means that block is free. This technique is used in the VM/370 system described in Chapter 9.

The principal advantage of the bit map scheme is that it can be kept in core, because it can be quite small. Thus the allocation can be performed at great speed.

6-10 DEVICE STRATEGY MODULE, I/O INITIATOR, DEVICE HANDLER

The basic functions of these routines are as follows:

1. Map the physical address into a device address (e.g., the physical address of file ETHEL record 4 is physical block 7, which is device address track 1, record 3).
2. Create the channel program, e.g., a program to do the following:
   a. Seek to track 1
   b. Search track 1 for record 3
   c. Wait until disk reaches it
   d. Read
3. Request the I/O, that is, call the I/O scheduler to schedule this I/O.
4. Handle the I/O interrupts and error conditions that may occur.

The algorithms for these routines have been covered previously in Chapter 5 (Device Management).

6-11 TRENDS IN INFORMATION MANAGEMENT

One may note the following similarities between the file system and the segmentation and paging concepts discussed in Chapter 3.

1. The notions of file and segment both involve the logical organization of information (i.e., files correspond to segments).
2. Both files and segments may be of arbitrary size and may grow in size dynamically.