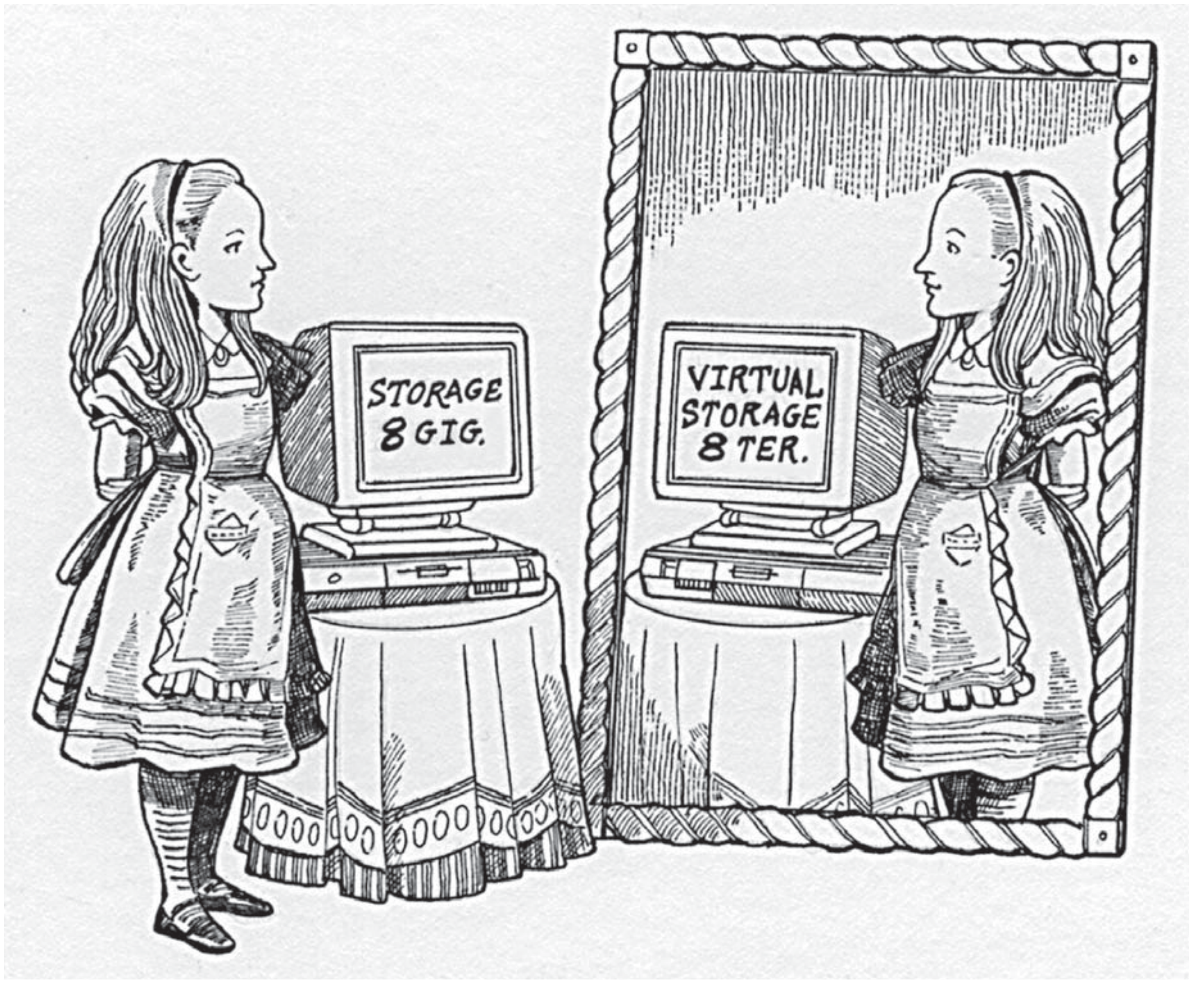
CHAPTER 18

THE INTERNAL OPERATING SYSTEM



Thomas Sperling

18.0 INTRODUCTION

In Chapter 15 we presented an overview of the role of the operating system as a primary computer system component and observed that it is possible to represent the architecture of the operating system as a hierarchy, consisting of several layers of programs that interact with each other to handle the routine tasks of command processing, file management, I/O, resource management, communication, and scheduling. We continued the discussion in Chapter 16 by starting with the most familiar layer, the user interface. Chapter 17 moved inward to the next layer and presented the features and organization of the file management system. The file manager converts the logical representation of files as seen by the user or the user's programs to the physical representation stored and manipulated within the computer.

Now we are ready to examine major features of the remaining inner layers. These layers are designed primarily to manage the hardware and software resources of the computer and its interactions with other computers. In this chapter, we will look at how these internal operations are performed; we will consider how the operating system programs manage processes, memory, I/O, secondary storage, CPU time, and more for the convenience, security, and efficiency of the users.

We will briefly review the concepts from Chapter 15 first. Then we expand our focus to look at the various components, features, and techniques that are characteristic of modern operating systems. We will show you a simple example in which the different pieces have been put together to form a complete system.

A modern system must have the means to decide which programs are to be admitted into memory and when, where programs should reside in memory, how CPU time is to be allocated to the various programs, how to resolve conflicting requirements for I/O services, and how to share programs and yet maintain security and program and data integrity, plus resolve many other questions and issues. It is not uncommon for the operating system to require several hundreds of megabytes of memory just for itself.

In this chapter, we consider the basic operations performed by the operating system. We introduce individually the various tasks that are to be performed by the operating system and consider and compare some of the methods and algorithms used to perform these tasks in an effective manner. We discuss the basic procedure of loading and executing a program, the boot procedure, the management of processes, memory management, process scheduling and CPU dispatch, secondary storage management, and more.

As we have mentioned previously, the modern computer includes additional CPU hardware features that work in coordination with the operating system software to solve some of the more challenging operating system problems. Virtual storage is arguably the most important of these advances. Virtual storage is a powerful technique for solving many of the difficulties of memory management. Section 18.7 is devoted to a detailed introduction to virtual storage. It also serves as a clear example of the integration of hardware and operating system software that is characteristic of modern computer systems.

Other examples include the layering of the instruction set to include certain protected instructions for use only by the operating system, which we presented in Chapter 7, and memory limit checking, which the operating system can use to protect programs from each other.

The subject of operating systems can easily fill a large textbook and an entire course all by itself. There are many interesting questions and problems related to operating systems and many different solutions to the problem of creating a useful and efficient operating system. Obviously, we won't be able to cover this subject in a great amount of detail, but at least you'll get a feeling for some of the more important and interesting aspects of how operating systems work.

The many tasks that a modern operating system is expected to perform also expand the overhead required by the operating system, both in terms of memory and in the time required to perform the different functions. We will also look at some of the measures that are used to determine the effectiveness of an operating system. Finally, you'll have a chance to read about a few of the more interesting problems, especially those that can have a significant effect on the user.

18.1 FUNDAMENTAL OS REQUIREMENTS

Always keep in mind that the fundamental purpose of any operating system is to load and execute programs. This is true regardless of the specific goals, design features, and complexity of the particular operating system that you happen to be looking at.

With this fundamental idea in mind, look again at the various functions that are provided within the operating system. To assist with this task, the hierarchical model is shown again for your convenience in Figure 18.1.

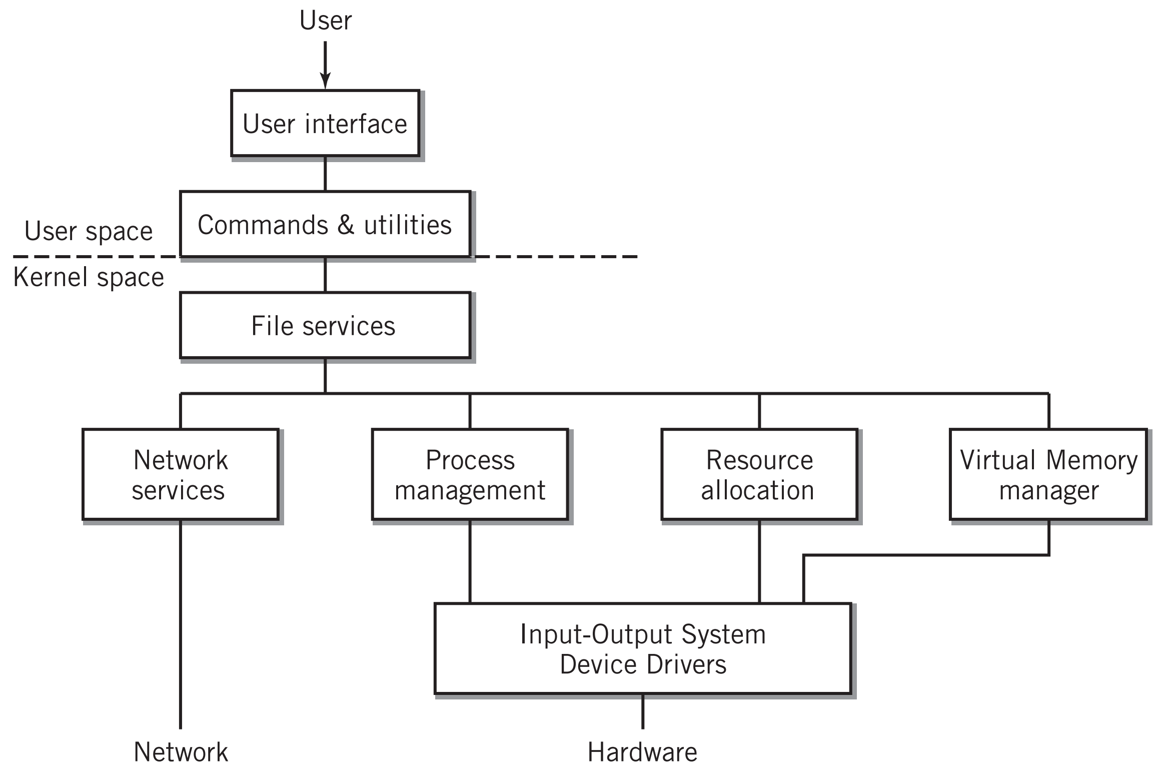
Recall that to load and execute a program, the system must provide a method of getting the program from its storage location on some I/O device, such as disk, into memory; it must provide locations in memory for the program and its data; it must provide CPU time for the program to execute; and it must provide access to the I/O facilities that the program needs during its execution. Since multiple programs are normally sharing the system and its resources, it must do all this in a way that is fair and meets the sometimes conflicting requirements of the different programs.

The lower layers of the model provide programs that fulfill these requirements. The file manager layer translates logical file requests from the command shell or the user's programs into specific physical I/O requests that are then performed by the appropriate I/O device management programs. Resource allocation management is also provided in this layer to resolve conflicts between different programs that may require I/O services at the same time. The I/O device management and resource allocation programs are sometimes known collectively as an I/O control system, or more commonly, IOCS.

The memory management and scheduling operations within the resource allocation function determine if it is possible to load programs and data into memory, and, if so, where in memory the program is to be loaded. Once the program is in memory, the scheduler allocates time for the program to execute. If there are multiple programs in memory, the scheduler attempts to allocate time for each of them in some fair way.

FIGURE 18.1

A Hierarchical Model of an OS



The monitor program, when included, provides overall control of the system. It establishes guidelines for the general management of the system based on goals provided by the human manager of the system. It watches for conflicts, breakdowns, and problems and attempts to take appropriate action to ensure smooth and efficient system operation. Some monitors can even reconfigure and reassign resources dynamically to optimize performance, particularly in clustered systems. These roles are handled by other operating system components in some systems.

To increase security, many operating systems construct these programs as a hierarchy in which each layer of programs in the model requests services from the next innermost layer, using an established calling procedure. Most modern computers provide special protected hardware instructions for this purpose. Recall from Chapter 15 that this is not the only possible architecture for an operating system. At the very least, the critical parts of the operating system will execute in a protected mode while other programs will execute in user mode. A well-designed operating system will repel attempts to penetrate the internal layers of the system by means other than established OS calling procedures. It must isolate and protect each program, yet allow the programs to share data and to communicate, when required.

There are many different ways of performing each of these functions, each with advantages and disadvantages. The trade-offs selected reflect the design goals of the particular system. To give you a simple example, a computer that operates strictly in a batch mode might use a simple CPU scheduling algorithm that allows each program to run without interruption as long as the program does not have to stop processing to wait for I/O. This strategy would not be acceptable on an interactive system that requires fast screen response when a user clicks the mouse or types something into the keyboard. In the latter case, a more sophisticated scheduling algorithm is clearly required.

Before we continue with discussions of the individual resource managers, you should be aware that these managers are not totally independent of each other. For example, if there are more programs in the memory of an interactive system, the scheduler must give each program a shorter period of time if satisfactory user response is to be achieved. Similarly, more programs in memory will increase the workload on a disk manager, making it more likely that there will be several programs waiting for disk I/O at the same time. A well-designed operating system will attempt to balance the various requirements to maximize productive use of the system.

Before proceeding to detailed discussions of each of the major modules in a multitasking operating system, it may provide some insight to introduce you to a simple example of a system, a sort of ''Little Man multitasking operating system'', if you will. The system discussed here does not run on the Little Man Computer, however. It was designed for a real, working computer system. This example illustrates many of the important requirements and operations of a multitasking system.

Example: A Simple Multitasking Operating System

The miniature operating system (hereafter referred to as MINOS) is an extremely small and simple multitasking system with many of the important internal features of larger systems. It is based on a real operating system that was developed by the author in the 1970s for a very early and primitive microcomputer that was used primarily to measure data in remote rural locations. Calculations were performed on the data and the results telecommunicated back to a larger computer for further processing. The original goals of the design were

- First and foremost, simplicity. Memory was very expensive in those days, so we didn't want to use much for the operating system. There was only 8 KB of memory in the machine.
- Real-time support for one very important program that was run frequently and had to operate very fast. This was the data measurement program. The system therefore features a priority scheduling system in choosing which program is to run.

The internal design of MINOS was of more interest and importance to the designers than the user interface or the file system. There was no disk on this computer, only an audio cassette tape recorder, modified to hold computer data, so the file system was simple. (Disks were too expensive, too large, and too fragile for this type of system back then!) There was a keyboard/printer user interface, but no CRT display interface. Security was not a concern.

The features of particular interest to us here are the operation of memory management, process scheduling, and dispatching. Despite their simplicity, the design of these modules is characteristic of the way current operating systems work. These were the important specifications for MINOS:

- Keyboard/printer command line user interface. To keep things simple, there were only a few commands, most of which could be entered by typing a single character. For example, the letter "l" was used to load a program from tape, the letter "s" to save a program to tape.
- Memory was divided into six fixed partitions of different sizes. A memory map is shown in Figure 18.2. One partition was reserved for MINOS, which was entirely memory resident. Partition P-1 was reserved for high-priority programs, most commonly the data retrieval program, since it had to retrieve data in real time. Partitions P-2, P-3, and P-4 were of different sizes, but all shared equal, middle priority. Partition P-5 was a low-priority area, which was used for background tasks, mostly internal system checking, but there was a simple binary editor available that could be loaded into the low-priority partition for debugging and modifying programs.
- The operating system was divided into three levels: the command interface; the I/O subsystem; and the kernel, which contained the memory manager, the communication interface, and the scheduler. The operating system kernel had the highest priority by default, since it had to respond to user commands and provide dispatching services. It could interrupt and preempt other programs. However, routine operations such as program loading were processed at the lowest priority level. A block diagram of MINOS appears in Figure 18.3.

Note again that MINOS did not support a file system or most other user amenities; it was primarily oriented toward program loading and execution. This limitation does not concern us, since the primary focus of this discussion is the internal operation of the system. The two major components of the kernel were the process scheduler/memory manager and the dispatcher.

MINOS was capable of manipulating up to five user programs at a time. The process scheduler handled requests for program loading. The header for a program to be loaded specified a priority level and a memory size requirement. Programs were loaded into the smallest available memory space of the correct priority level that would fit the program. Of course, there was only a single memory area available for each program of the highest and lowest priorities. If space was not available, the process scheduler notified the user; it was up to the user to determine which program, if any, should be unloaded to make room.

For each program in memory, there was an entry in a process control table, shown in Figure 18.4. Recall from Chapter 14 that at any instant in time, one process per CPU is running, while the others are ready to run or waiting for an event, such as I/O completion, to occur. The process control table shows the status of each program and the program counter location where the program will restart when it is next run. In MINOS, it also contained locations for storage and restoration of each of the

FIGURE 18.2

The MINOS Memory Map

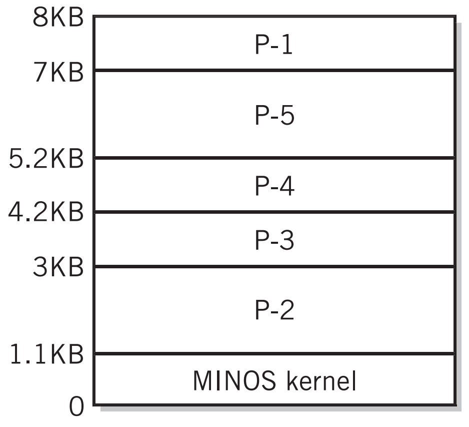
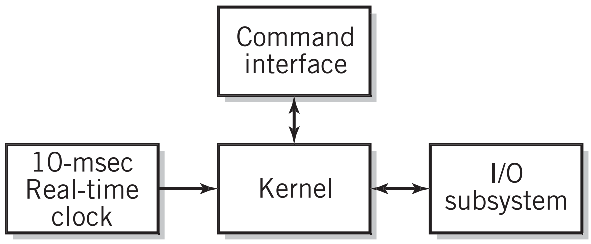


FIGURE 18.3

Block Diagram, MINOS



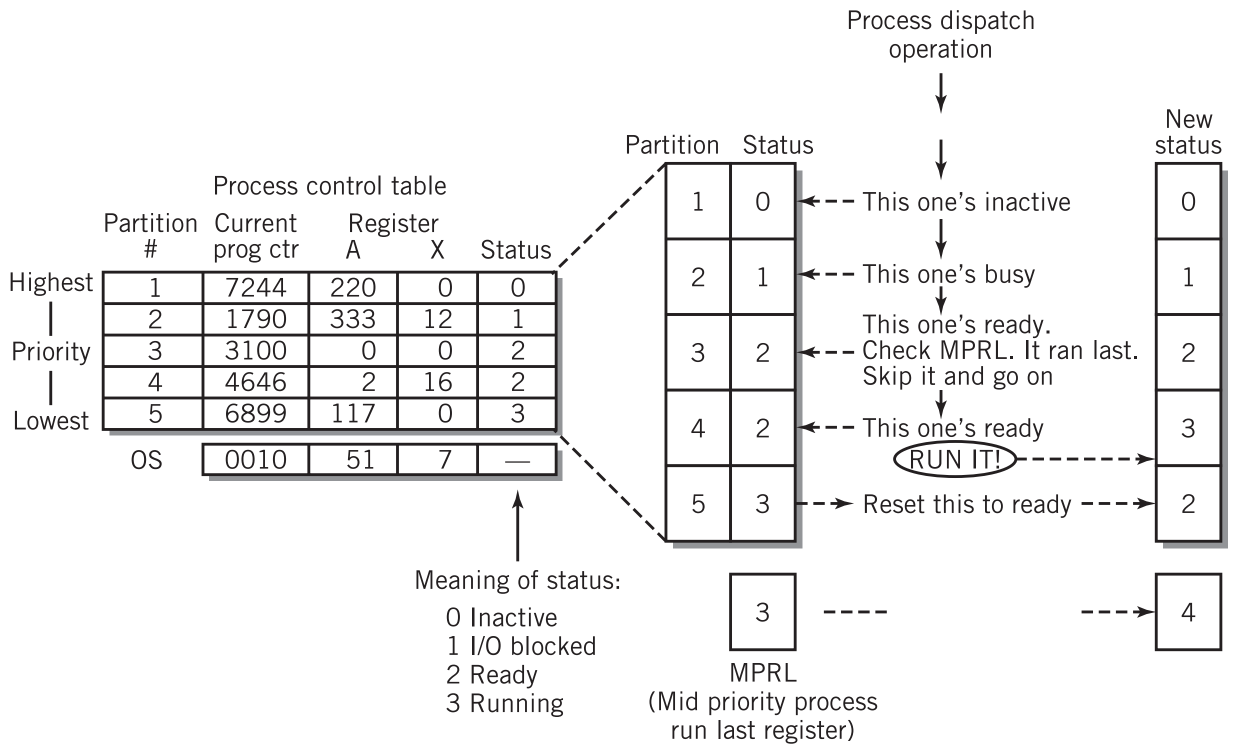
two registers that were present in the microcomputer that was used. There was also one additional register that kept track of which mid-priority process, partition 2, partition 3, or partition 4, was run most recently. We called this register the *mid-priority process run last*, or MPRL, register. Since there was one entry in the process table for each partition, the priority value for each program was already known by the operating system.

The most interesting part of MINOS was the program dispatcher. A real-time clock in the computer interrupted the computer every 1/100th of a second and returned control to the dispatcher. The dispatcher went through the process control table in order of priority and checked the status of each active entry. (An inactive entry is one in which there was no program loaded into the space, or in which the program in the space had completed execution and was not running.) If the entry was blocked because it was waiting for I/O to be completed, it was not available to run and was passed by. The highest-priority ready program was selected and control passed to it. If there were two or three ready programs of the same priority, they were selected in a round-robin fashion (program 2, program 3, program 4, program 2, program 3, . . .), so that each got a turn. The MPRL register was used for this purpose.

The MINOS dispatching algorithm guaranteed that the high-priority real-time program always got first shot at the CPU and that the maximum delay before it could execute

FIGURE 18.4

MINOS Process Dispatch



was 1/100th of a second. The ready bit for this program was actually set by a small interrupt routine controlled by the measuring device. Figure 18.4 illustrates the dispatching process.

The background task represented the lowest priority. By default, this partition contained software routines for testing various aspects of the hardware. Thus, when no other program was selected, MINOS defaulted to the hardware diagnostic routines.

With MINOS as a background, the next nine sections of Chapter 18 consider various aspects of a multitasking operating system in more detail. You may also wish to review Section 15.3 of Chapter 15, which introduces the various services and modules present in a modern multitasking system, before proceeding.

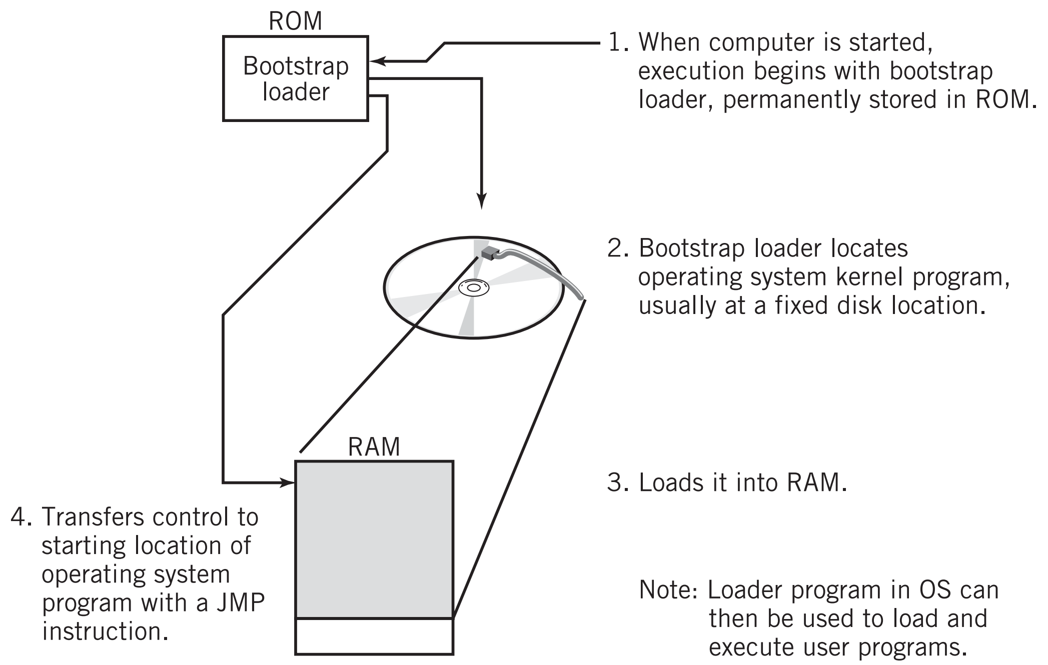
18.2 STARTING THE COMPUTER SYSTEM: THE BOOTSTRAP

As a first step, we need to consider what is required to get the computer started. You will recall that when the computer is first turned on, the contents of RAM are unknown. Furthermore, you know that there must be a program in memory for CPU execution to take place. These two considerations are contradictory; therefore, special means must be included to get the system into an operating state.

Initial program loading and start-up is performed by using a *bootstrap* program that is built permanently into a read-only part of memory for the computer. This bootstrap program begins execution as soon as the machine is powered up. The bootstrap program contains a program loader that automatically loads a selected program from secondary storage into normal memory and transfers control to it. The process is known as bootstrapping, or more simply, as *booting* the computer. IBM calls the process *Initial Program Load*, or *IPL*. Figure 18.5 illustrates the bootstrapping operation.

FIGURE 18.5

Bootstrapping a Computer


ROM
Bootstrap loader
1. When computer is started, execution begins with bootstrap loader, permanently stored in ROM.
2. Bootstrap loader locates operating system kernel program, usually at a fixed disk location.
RAM
3. Loads it into RAM.
4. Transfers control to starting location of operating system program with a JMP instruction.
Note: Loader program in OS can then be used to load and execute user programs.

Since the bootstrap is a read-only program, the program that it loads must be predetermined and must be found in a known secondary storage location, usually at a particular track and sector on a hard disk, although the bootstrap can be tailored to start the computer from another device, or even from another computer if the system is connected to a network. Usually the bootstrap loads a program that is itself capable of loading programs. (This is the reason that the initial program loader is called a bootstrap.) Ultimately, the program loaded contains the operating system kernel. In other words, when the boot procedure is complete, the kernel is loaded, and the computer is ready for normal operation. The resident operating system services are present and ready to go. Commands can be accepted, and other programs loaded and executed. The bootstrap operation is usually performed in two or more stages of loading to increase flexibility in the location of the kernel and to keep the initial bootstrap program small.

EXAMPLE

The PC serves as an appropriate and familiar example of the bootstrap start-up procedure. Although the PC uses a multistep start-up procedure, the method is essentially identical to that we have just described.

The PC bootstrap loader is permanently located in the system BIOS, read-only memory included as part of the computer, and introduced previously in Chapter 15. When the power switch for the computer is turned on, or when the reset button is pushed, control is transferred to the first address of the bootstrap loader program. The PC bootstrap begins by performing a thorough test of the components of the computer. The test verifies that various components of the system are active and working. It checks for the presence of a monitor, of a hard drive if installed, and of a keyboard. It checks the instructions in ROM for errors by calculating an algebraic function of the 1s and 0s, known as a checksum, and comparing that value with a predetermined correct value. It checks RAM by loading known data into every location and reading it back. Finally, it resets the segment registers, the instruction pointer, flags, and various address lines. (The 386, 486, P5, and P6 CPUs set many other registers as well.) The results of these tests appear on the monitor screen.

At the completion of this test, the bootstrap loader determines which disk is the system disk. This location is a setting stored permanently in a special memory, modifiable by the user at startup time. On modern PCs, the system may be booted from a hard disk, a floppy disk, a CD or DVD, or many USB-pluggable devices. The system disk contains a sector known as a boot record, and the boot record is loaded next.

The boot record now takes control. It also contains a loader, which is tailored to the I/O requirements for the particular disk. Assuming that Windows 2000, NT, or XP is to be loaded, the boot record then loads a sequence of files, including the kernel and executive program, NTOSKRNL.EXE; the registry; the hardware interface; various kernel, subsystem, and API libraries; and a number of other components. The items loaded are based on entries in the registry. The user has little control over this process while it is happening. Next, a logon program, WINLOGON.EXE is initiated. Assuming that the user is authorized and that the logon is successful, the kernel sets the user parameters defined in the registry, the Windows GUI is displayed, and control of the system is turned over to the user.

Different BIOSes vary slightly in their testing procedures, and some allow the user to change some PC setup settings when testing takes place. The user can also force the bootstrap to occur one step at a time to remedy serious system problems. Other than that, the user or system administrator controls the PC environment with standard tools provided by the operating system.

As noted, the procedure described here takes place when power is first applied to the computer. This procedure is also known as a cold boot. The PC also provides an alternate procedure known as a warm boot, for use when the system must be restarted for some reason. The warm boot, which is initiated from a selection on the *shutdown* menu, causes an interrupt call that reloads the operating system, but it does not retest the system and it does not reset the various registers to their initial values.

EXAMPLE

It is important to realize that basic computer procedures are not dependent on the size of the computer. The boot procedure for a large IBM mainframe computer is quite similar to that of a PC. IBM mainframe computers are bootstrapped using the Initial Program Load procedure. IPL works very similarly to the PC bootstrap procedure. Whenever power is applied to an IBM mainframe computer, the computer is in one of four operating states: operating, stopped, load, and check stop. The operating and stopped states are already familiar to you. The check stop state is a special state used for diagnosing hardware errors. The load state is the state corresponding to IPL.

The system operator causes the system to enter load state by setting load-unit-address controls and activating the load-clear or load-normal key on the operator's console. The load-unit-address controls establish a particular channel and I/O device that will be used for the IPL. The load normal key performs an initial CPU reset that sets the various registers in the CPU to their initial values and validates proper operation. The load-clear key does the same, but also performs a clear reset, which sets the contents of main storage and many registers to zero.

Following the reset operation, IPL performs the equivalent of a START I/O channel command, as discussed in Chapter 11. The first channel command word is not read from memory, since memory may have been reset to zero. Instead, a built-in READ command is used, which reads the IPL channel program into memory for execution. The IPL channel program then reads in the appropriate operating system code and transfers control to it.

18.3 PROCESSES AND THREADS

When considering a multitasking system, it is easiest to think of each executing task as a program. This representation is not inaccurate, but it is not sufficiently inclusive, precise, or general to explain all the different situations that can occur within a computer system. Instead, we may define each executing task more usefully as a process. A **process** is defined to include a program, together with all the resources that are associated with that program as it is executed. Those resources may include I/O devices that have been assigned to the particular process, keyboard input data, files that have been opened, memory that has been assigned as a buffer for I/O data or as a stack, memory assigned to the program, CPU time, and many other possibilities.

Another way of viewing a process is to consider it as a program in execution. A program is viewed passively: it's a file or a listing, for example. A process is viewed actively: it is being processed or executed.

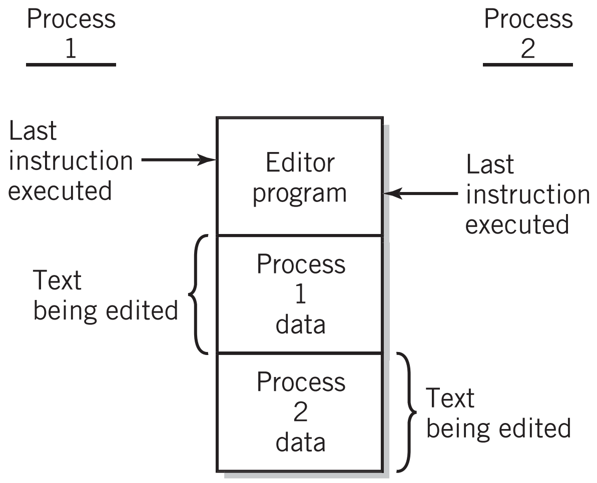
In batch systems, a different terminology is sometimes used. A user submits a **job** to the system for processing; the job is made up of **job steps**, each of which represents a single **task**. It is not difficult to see the relationship among jobs, tasks, and processes. When the job is admitted to the system, a process is created for the job. Each of the tasks within the job also represent processes, specifically, processes that will be created as each step in the job is executed. In this book we tend to use the words job, task, and process interchangeably.

The difference between a program and a process is not usually important in normal conversation, but from the perspective of the operating system the difference may be quite significant and profound. For example, most modern operating systems have the capability of sharing a single copy of a program such as an editor among many processes concurrently. Each process has its own files and data. This practice can save memory space, since only a single copy of the program is required, instead of many; thus, this technique increases system capability. Crucial to this concept, however, is the understanding that each process may be operating in a different part of the program; therefore, each process maintains a different program counter value during its execution time, as well as different data. This concept is illustrated in Figure 18.6. By maintaining a separate process for each user, the operating system can keep track of each user's requirements in a straightforward manner.

Even in a single-user system, multiple processes may share program code. For example, the program code that produces the Windows interface will be shared by all the processes with open windows on a screen. Each process will have its own data: the coordinates of the window, pointers to the menu structure for that window, and so forth.

FIGURE 18.6

Two Processes Sharing a Single Program



To the operating system, the basic unit of work is a process. When a process is admitted to the system, the operating system is responsible for every aspect of its operation. The operating system must allocate initial memory for it and must continue to assure that memory is available to the process as it is needed. It must assign the necessary files and I/O devices and provide stack memory and buffers. It must schedule CPU execution time for the process and perform context switching between the various executing processes. The operating system must maintain the integrity of the process. Finally, when the process is completed, it terminates the process in an orderly way and restores the system facilities and resources to make them available to other processes.

Processes that do not need to interact with any other processes are known as **independent processes**. In modern systems, many processes will work together.

They will share information and files. A large task will often be modularized by splitting it into subtasks, so that each process will only handle one aspect of the task. Processes that work together are known as **cooperating processes**. The operating system provides mechanisms for synchronizing and communicating between processes that are related in some way. (If one process needs the result from another, for example, it must know when the result is available so that it can proceed. This is known as synchronization. It must also be able to receive the result from the other process. This is communication.) The operating system acts as the manager and conduit for these interprocess events.

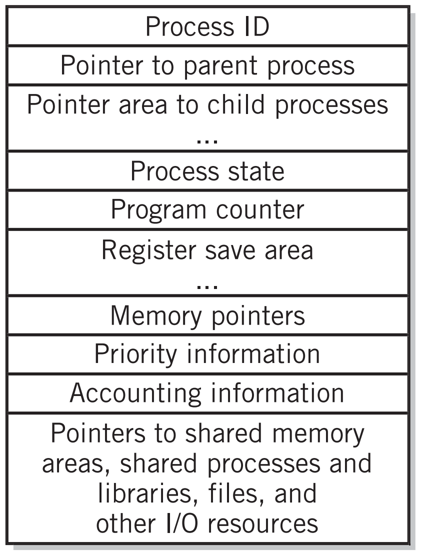
To keep track of each of the different processes that are executing concurrently in memory, the operating system creates and maintains a block of data for each process in the system. This data block is known as a **process control block**, frequently abbreviated as **PCB**. The process control block contains all relevant information about the process. It is the central resource used by the various operating system modules as they perform their process-related functions.

In MINOS, the process control block was simple. It was only necessary to keep track of the program counter and a pair of register values so that processes could be suspended and restarted, plus the status and priority of the program. Since MINOS divided memory into partitions of fixed size, there was exactly one process and therefore one PCB per partition, so it was not even necessary for the operating system to keep track of the memory limits of a process.

In a larger system, process control is considerably more complex. There may be many more processes. Contention for the available memory and for various I/O resources is more likely. There may be requirements for communication between different processes. Scheduling and dispatch are more difficult. The complexity of the system requires the storing of much additional information about the process, as well as more formal control of process operations.

FIGURE 18.7

A Typical Process Control Block


Process ID
Pointer to parent process
Pointer area to child processes
...
Process state
Program counter
Register save area
...
Memory pointers
Priority information
Accounting information
Pointers to shared memory areas, shared processes and libraries, files, and other I/O resources

The contents of a typical process control block are shown in Figure 18.7. Different system PCBs present this information in different order and with some differences in the information stored, but these differences are not important for the purposes of this discussion.

Each process control block in Figure 18.7 contains a process identification name or number that uniquely identifies the block. In Linux, for example, the process identification number is known as a **process identifier**, or more commonly, a **PID**. Active processes are readily observable on the Linux system using the *ps* command.

Next, the PCB contains pointers to other, related processes. This issue is related to the way in which new processes are created. It is discussed in the next section. The presence of this area simplifies communication between related processes. Following the pointer area is an indicator of the process state. In MINOS, four

process states were possible: inactive, ready, blocked, and running. In larger systems, there are other possible states; processor states are discussed later in this section. The program counter and register save areas in the process control block are used to save and restore the exact context of the CPU when the process gives up and regains the CPU.

Memory limits establish the legal areas of memory that the process may access. The presence of this data simplifies the task of security for the operating system. Similarly, priority and accounting information is used by the operating system for scheduling and for billing purposes.

Finally, the process control block often contains pointers to shared program code and data, open files, and other resources that the process uses. This simplifies the tasks of the I/O and file management systems.

Process Creation

A little thought should make it clear to you that a process is created when you issue a command that requests execution of a program, either by double-clicking on an icon or by typing an appropriate command. There are also many other ways in which a process is created. Particularly on interactive systems, process creation is one of the fundamental tasks performed by the operating system. Processes in a computer system are continually being created and destroyed.

Since *any* executing program is a process, almost any command that you enter into a multitasking interactive system normally creates a process. Even logging in creates a process, since logging in requires providing a program that serves as your interface, giving you a prompt or GUI, monitoring your keystrokes, and responding to your requests. In many systems, this is known as a **user process**. In some systems, all processes that are not modules of the operating system are known as user processes.

It should also be remembered that the operating system itself is made up of program modules. These modules, too, must share the use of the CPU to perform their duties. Thus, the active parts of the operating system are, themselves, processes. When a process requests I/O or operating system services, for example, processes are created for the various operating system program modules that will service the request, as well as for any additional processes resulting from the request. These processes are sometimes known as **system processes**.

In batch systems, jobs are submitted to the system for processing. These jobs are copied, or spooled, to a disk and placed in a queue to await admission to the system. A long-term scheduler in the operating system, discussed in Section 18.5, selects jobs as resources become available and loads them into memory for execution. A process is created when the long-term scheduler determines that it is able to accept a batch job and admits it to the system.

For convenience, operating systems generally associate processes with the process that created them. Creating a new process from an older one is commonly called **forking** or **spawning**. The spawning process is called a **parent**. The spawned process is known as a **child**. Many systems simply assign priorities, resources, and other characteristics to the child process by **cloning** the parent process. This means creating a process control block that is a duplicate of itself. Once the child process begins to execute, it goes by way of its own path. It can request its own resources and change whatever characteristics it needs to.

As an example of process creation, a C++ program compiler might create child processes that perform the different stages of compilation, editing, and debugging. Each child process is created when the specific task is needed and killed when the task is complete. Incidentally, note the synchronization between processes that is suggested by this example. If the compile process encounters an error, for example, the parent is notified so that it can activate an editor process. A successful compile will result in a load process that will load the new program for execution. And so on.

Removing a parent process usually kills all the child processes associated with it. Since a child process can itself have children, the actual process structure may be several generations deep. Pointers are used within the process control block to help keep track of the relationships between different processes.

When the process is created, the operating system gives it a unique name or identification number, creates a process control block for it, allocates the memory and other initial resources that the process needs, and performs other operating system bookkeeping functions. When the process exits, its resources are returned to the system pool, and its PCB is removed from the process table.

Process States

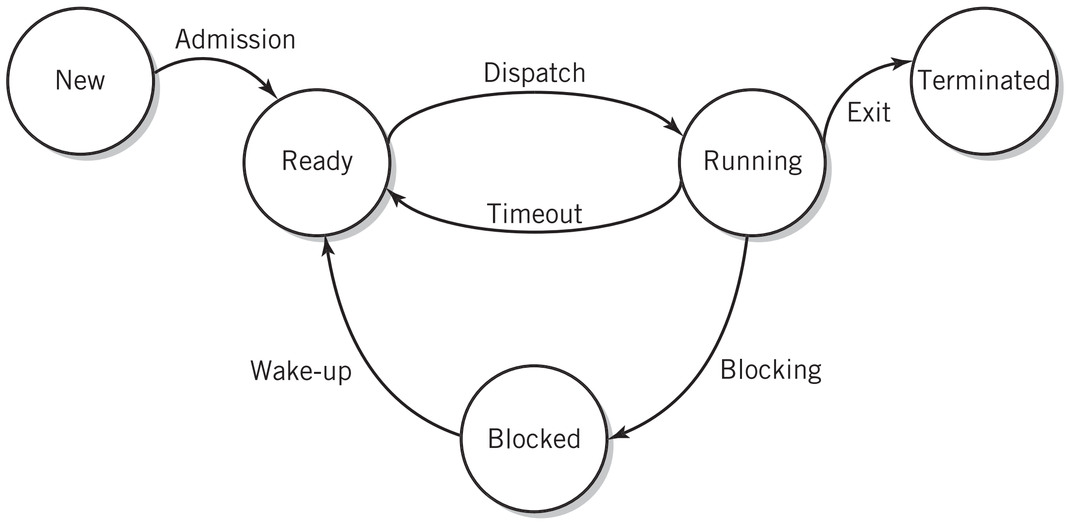
Most operating systems define three primary operating states for a process. These are known as the **ready state**, the **running state**, and the **blocked state**. The relationship between the different process states is shown in Figure 18.8.

Once a process has been created and admitted to the system for execution, it is brought into the *ready* state, where it must compete with all other processes in the ready state for CPU execution time. Being in the ready state simply means that a process is capable of execution if given access to the CPU.

At some point in time, presumably, the process will be given time for execution. The process is moved from the ready state to the *running* state. Moving from the ready state to the running state is called **dispatching** the process. During the time that the process is in the running state, the program has control of the CPU and is able to execute instructions.

FIGURE 18.8

The Major Process States



Of course, only one process can be in the running state at a time for a uniprocessor system. If there are multiple processors or a cluster under the operating system's control, the OS is responsible for dispatching a process to run in each available CPU. In a typical multitasking system, there may be many processes in *blocked* or ready states at any given time.

When I/O or other services are required for the continuation of program execution, the running process can no longer do any further useful work until its requirement is satisfied. Some operating systems will suspend the program when this occurs; others will allow the program to remain in the running state, even though the program is unable to proceed. In the latter case, most well designed programs will suspend themselves, unless the interruption is expected to be extremely brief. This state transition is known as **blocking**, and the process remains in a blocked state until its I/O requirement is complete. When the I/O operation is complete, the operating system moves the process from the blocked state back to the ready state. This state transition is frequently called **wake-up**. Blocking can also occur when a process is waiting for some event other than I/O to occur, for example, a completion signal or a data result from another process.

Nonpreemptive systems will allow a running process to continue running until it is completed or blocked. **Preemptive systems** will limit the time that the program remains in the running state to a fixed length of time corresponding to one or more quanta. If the process remains in the running state when its time limit has occurred, the operating system will return the process to the ready state to await further time for processing. The transition from the running state to the ready state is known as **time-out**.

When the process completes execution, control returns to the operating system, and the process is *destroyed* or *killed* or *terminated*.

Some operating systems provide one or more additional states, which are used to improve the efficiency of the computer system. Some processes make heavy demands on particular resources, say, a disk drive or a printer, or even the CPU, in such a way that other processes are unable to complete their work in an efficient manner. In this case the operating system may place a process in a **suspended state** until the required resources can be made available. When this occurs, the process is returned to a ready state. The transition from the *suspended* state to the ready state is known as **resumption**. Some operating systems also allow a user to suspend a process. On UNIX systems, for example, typing Control-z is one way in which to suspend a process. The process may be resumed by issuing the command *fg*, together with the process identification number of the process. Some operating systems will also swap out a suspended process from memory to secondary storage when the system becomes overloaded and will swap it back in when the load is lighter. Particularly in small systems, the use of **swap files** for this purpose is common. Even in large computer systems, transaction processing software often contains interactive processes that are used infrequently. These processes are often swapped out when they are not being used and returned to memory when they are activated by a user request. This technique is called **roll-out, roll-in**. The *suspend, resume*, and *swap* states have been left off the diagram for clarity.

Threads

It is common in modern systems to provide capability for a sort of miniprocess, known as a **thread**. A thread represents a piece of a process that can be executed independently of other parts of the process. (Think of the spell-checker in a word processor that checks words as you type, for example.) Each thread has its own context, consisting of a program counter value, register set, and stack space, but shares program code, and data, and other system resources such as open files with the other member threads in the process. Threads can operate concurrently. Like processes, threads can be created and destroyed and can be in ready, running, and blocked states. Context switching among threads is easier for the operating system to manage because there is no need to manage memory, files, and other resources and no need for synchronization or communication within the process, since this is handled within the process itself. This advantage suggests, however, that more care needs to be taken when the program is written, to assure that threads do not interact with each other in subtle ways that can create conditions that cause the program to fail. Note that there is no protection among the threads of a process, since all the threads are using the same program code and data space.

Some systems even provide a mechanism for context switching of threads independent of the process switching mechanism. This means that in these systems threads can be switched without the involvement of the operating system kernel. If a process becomes I/O blocked, it cannot proceed until the block is resolved. On the other hand, if a thread becomes blocked, other threads in the process may be able to continue execution within the process's allotted time, resulting in more rapid execution. Because the inner layers of the operating system are not even aware of thread context switching in these systems, thread switching is extremely rapid and efficient. Threads in these systems are commonly known as **user-level threads**.

Threads came about as a result of the advent of **event-driven programs**. In older programs with traditional text-based displays and keyboard input, there was a single flow of control. Event-driven programs differ in that the flow of control depends in a much more dramatic way on user input. With a modern graphical user interface, a user can pull down a menu and select an action to be performed at almost any time. Selecting an item from a menu or clicking a mouse in a particular place in a particular way is known as an **event**. The program must be able to respond to a variety of different events, at unknown times, and in unknown order of request.

Most such events are too small to justify creation of a new process. Instead, the action for each event is treated as a thread. The thread can be executed independently, but without the overhead of a process. There is no control block, no separate memory, no separate resources. The primary requirement for a thread is a context storage area to store the program counter and registers when context switching takes place. A very simple thread control block is adequate for this purpose. Threads are processed in much the same way as processes.

18.4 BASIC LOADING AND EXECUTION OPERATIONS

Since the CPU's capability is limited to the execution of instructions, every operation in a computer system ultimately arises from the fundamental ability to load and execute programs. Application programs do the users' work. Operating system programs and utilities manage files, control I/O operations, process interrupts, provide system security, manage the user interface, log operations for the system administrator to analyze, and much more. Except for programs that permanently reside in ROM, every one of these programs must be loaded into memory before it can be executed.

In general-purpose computer systems, the only programs permanently resident in ROM are usually just the few that are needed to boot the system. All other programs are loaded after the system is operational. Many of these programs are loaded into memory at start-up time and remain resident as long as the computer is on; others are loaded as they are requested or needed, but in either case, the program loading operation is central to system operation. Observing the steps in the loading process exposes the workings and interactions of many of the basic operating system components.

Incidentally, the program loader itself is a program that generally must be loaded; as we already noted in Section 18.2, this initial load occurs during the boot process. After that, the loader process remains resident in memory, ready for use.

In the previous section, you saw how processes are created from programs administratively by the process management component of the operating system. You are already aware, therefore, that requests for program loads are spawned from application or system programs that are already running. Now we take a brief look at the next step: what happens after the process is created but before it is loaded into memory and executed. This will prepare you for more detailed discussions of the memory management and scheduling issues that follow.

Figure 18.9 shows the basic steps required to load the program and ready it for execution.

18.5 CPU SCHEDULING AND DISPATCHING

CPU scheduling provides mechanisms for the acceptance of processes into the system and for the actual allocation of CPU time to execute those processes. A fundamental objective of multitasking is to optimize use of the computer system resources, both CPU and I/O, by allowing multiple processes to execute concurrently. CPU scheduling is the means for meeting this objective. There are many different algorithms that can be used for CPU scheduling. The selection of a CPU scheduling algorithm can have a major effect on the performance of the system.

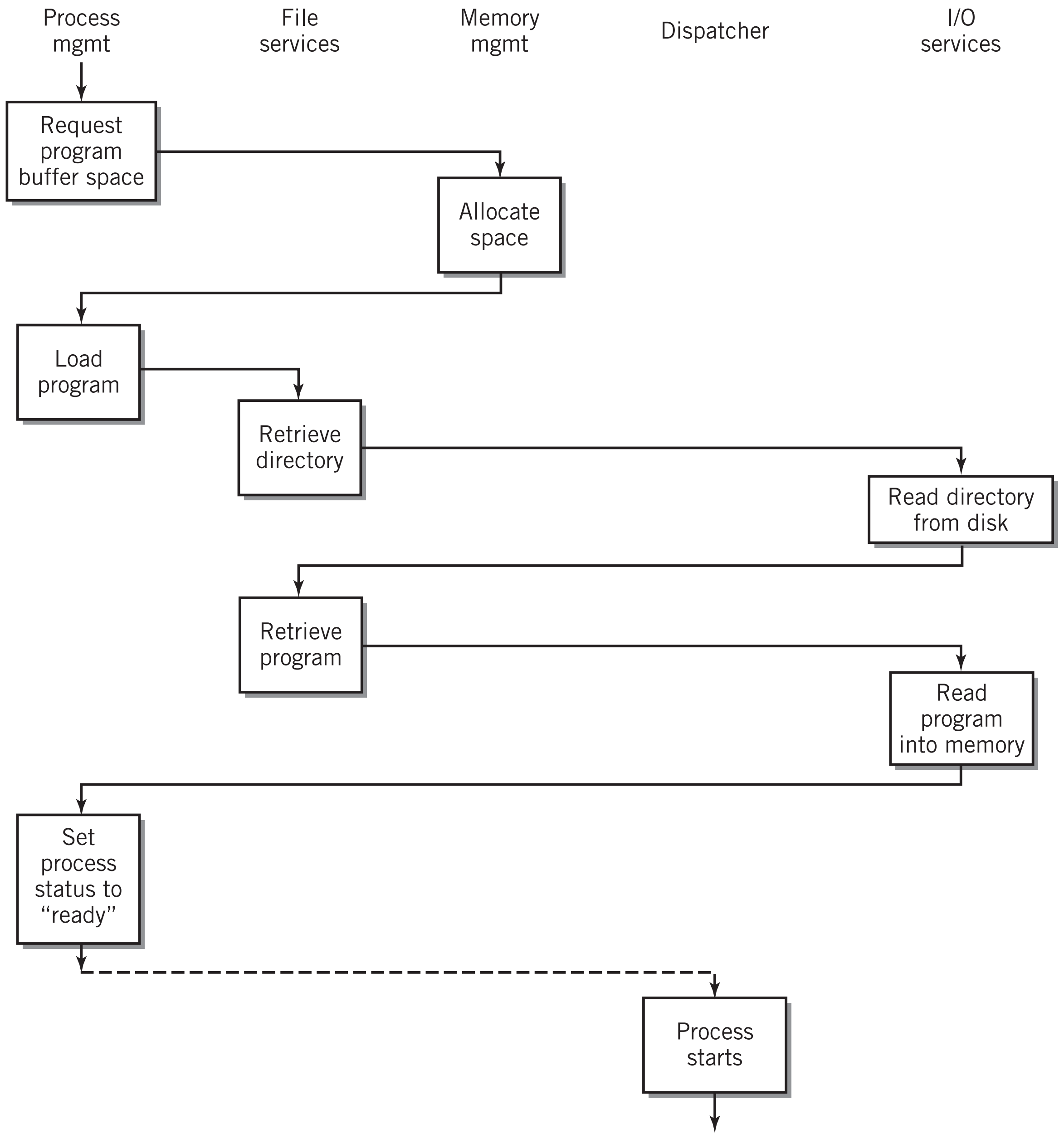
As a way to optimize system performance, the CPU scheduling task is separated into two different phases. The **high-level**, or **long-term, scheduler** is responsible for admitting processes to the system. The **dispatcher** provides short-term scheduling, specifically, the instant-by-instant decision as to which one of the processes that are ready should be given CPU execution time. The dispatcher also performs context switching. Some systems also include a third, middle-level scheduler, which monitors system performance. When present, the middle-level scheduler can suspend, or **swap out**, a process by removing it from memory temporarily and replace it with another waiting process. This operation is known as **swapping**. Swapping is done to improve overall system performance. It would be used if a particular process were hogging a resource in such a way as to prevent other processes from executing.

High-Level Scheduler

The high-level scheduler determines which processes are to be admitted to the system. The role of the high-level scheduler is minimal for processes created in an interactive environment. Such processes are usually admitted to the system automatically. If a user

FIGURE 18.9

Loading and Executing a Process



requests a service that requires the creation of a new process, the high-level scheduler *will* attempt to do so unless the system is seriously overloaded. To refuse the user in the middle of her or his work would be undesirable. The high-level scheduler will refuse a login process, however, if it appears that doing so would overload the system. The high-level scheduler will refuse admission to the system if there is no place to put the program in memory or if other resources are unattainable. If the request is a user login request, the user will have to wait until later to try again. Otherwise, requests are usually accepted, even though it may slow down the system. You may have experienced such slowdowns when working with Windows. You may have even gotten an ''out-of-memory'' message if you tried to do too many things at once!

For batch processes, the high-level scheduler has a much more important role. Since most modern systems are predominately interactive, the use of batch processes is generally limited to processes with demanding resource requirements, for example, a monthly billing program for a large utility or department store chain, or an economics problem with huge amounts of data and complex calculations to be performed on the data. Processes of this type can make it difficult for regular users to get their work done if the process is executed during a busy time of day.

With batch processes, a delay in processing is usually acceptable to the user; therefore, the high-level scheduler has more flexibility in deciding when to admit the process to the system. The high-level scheduler can use its power to balance system resource use as an attempt to maximize the efficiency of the system and minimize disruption to the regular users.

Dispatching

Conceptually, the dispatching process is simple. Whenever a process or thread gives up the CPU, the dispatcher selects another candidate that is ready to run, performs a context switch, and sets the program counter to the program counter value stored in the process control block to start execution. In reality, dispatching is much more complex than it first appears. There are a number of different conditions that might cause a process to give up the CPU, some voluntary and some involuntary, as established by the operating system. Presumably, the goal of the dispatcher is to select the next candidate in such a way as to optimize system use. But, in fact, there are a number of different measurement criteria that can be used to define ''optimum'' system performance. Frequently, these criteria are in conflict with each other, and the characteristics of the candidates in contention as well as different conditions within the system can also affect the selection of a particular candidate for CPU execution at any given time.

Similarly, processes vary in their requirements. Processes can be long or short in their requirement for CPU execution time, they can require many resources, or just a few, and they can vary in their ratio of CPU to I/O execution time. Different scheduling algorithms favor different types of processes or threads and meet different optimization criteria. For example, an algorithm that maximizes throughput by consistently placing short jobs at the front of the queue is clearly not fair to a longer job that keeps getting delayed.

As a result, there are a number of different scheduling algorithms that can be used. The choice of scheduling algorithm then depends on the optimization objective(s) chosen, along with the expected mix of process types. Analysis requires consideration of a wide variety of process mix possibilities and dynamic situations. Some of the objectives considered are shown in the table in Figure 18.10. Of the various objectives in the table, the prevention of **starvation** is particularly noticeable. Some algorithms with otherwise desirable properties have a potential to cause starvation under certain conditions. It is particularly important that the algorithm selected not permit starvation to occur.

With operating systems that support threads, dispatching normally takes place at the thread level. As an additional criterion, the candidate selection decision can be made at

FIGURE 18.10

System Dispatching Objectives

Ensure fairness	The scheduler should treat every process equally. This means that every process should get a fair share of the CPU time.
Maximize throughout	The scheduler should attempt to maximize the number of jobs completed in any given time period.
Minimize turnaround time	The scheduler should minimize the time between submission of a job and its completion.
Maximize CPU utilization	The scheduler should attempt to keep the CPU busy as close to 100% of the time as possible.
Maximize resource allocation	The scheduler should attempt to maximize the use of all resources by balancing processes that require heavy CPU time with those emphasizing I/O.
Promote graceful degradation	This objective states that as the system load becomes heavy, it should degrade gradually in performance. This objective is based on the assumption that users expect a heavily loaded system to respond more slowly, but not radically or suddenly so.
Minimize response time	This objective is particularly important in interactive systems. Processes should complete as quickly as possible.
Provide consistent response time	Users expect long jobs to require more actual time than short jobs. They also expect a job to take about the same amount of time each time it is executed. An algorithm that allows a large variation in the response time may not be considered acceptable to users.
Prevent starvation	Processes should not be allowed to starve. *Starvation* is a situation that occurs when a process is never given the CPU time that it needs to execute. Starvation is also called *indefinite postponement*.

either the process or thread level. Some systems will select a candidate that meets criteria measured at the process level. A process is selected, then a thread within that process is dispatched. Other systems will select a thread for dispatch based on thread performance criteria without regard to the process to which they belong.

Some systems implement only a single algorithm, selected by the original system designers. Others provide options that can be selected by the administrator of the particular system installation. Other than preventing starvation, the most important consideration in selecting a scheduling algorithm is to determine the conditions under which dispatching is to be performed preemptively or nonpreemptively.

Early batch systems were predominately nonpreemptive. In a nonpreemptive system, the process assigned to the CPU by the dispatcher is allowed to run to completion, or until it voluntarily gives up the CPU. Nonpreemptive dispatching is efficient. The overhead required for the dispatcher to select a candidate and perform context switching in a preemptive system, particularly if the quantum time is short, becomes a substantial percentage of the overall CPU time available.

Nonpreemptive dispatching does not quite work in modern interactive systems. Some interrupts, particularly user keystrokes and mouse movements, demand immediate attention. **Response time** is an important criterion to a user sitting at a terminal waiting for a result. A long process executing nonpreemptively can cause the system to "hang" for a while. An additional disadvantage of nonpreemptive processing is that a buggy program with an infinite loop can hang the system indefinitely. Most nonpreemptive systems actually have a time-out built in for this purpose. A compromise position uses nonpreemptive processing for executing processes that do not require immediate responses, but allows critical processes to interrupt temporarily, always returning control to the nonpreemptive process. Earlier versions of Windows, through Version 3.1, presented another compromise that was dependent on the cooperation of the processes themselves. This position assumed that processes would voluntarily relinquish control on a regular basis, to allow other processes a chance to execute. To a large measure, this approach worked, although less well than true preemptive multitasking; however, it is subject to errors that may occur in individual processes that can prevent the execution of other processes.

Linux presents another compromise approach: user processes (i.e., regular programs) run preemptively, but operating system programs run nonpreemptively. An important requirement to this approach is that operating system processes run quickly and very reliably. The advantage to this approach is that critical operating system processes can get their work done efficiently without interruption from user processes.

The next section introduces a few typical examples of dispatching algorithms. There are many other possibilities, including algorithms that use combinations of these examples.

Nonpreemptive Dispatch Algorithms

FIRST-IN, FIRST-OUT Probably the simplest possible dispatch algorithm, **first-in, first-out (FIFO)** simply assumes that processes will be executed as they arrive, in order. Starvation cannot occur with this method, and the method is certainly fair in a general sense; however, it fails to meet other objectives. In particular, FIFO penalizes short jobs and I/O-bound jobs, and often results in underutilized resources. As an illustration of the subtle difficulties presented when analyzing the behavior of an algorithm, consider what happens when one or more short, primarily I/O-based jobs are next in line behind a very long CPU-bound job in a FIFO queue. We assume that the scheduler is nonpreemptive but that it will allow another job to have the CPU when the executing job blocks for I/O. This assumption is essential to the full utilization of the CPU.

At the start of our observation, the long job is executing. While this happens, the short job(s) must sit and wait, unable to do anything. Eventually, the long job requires I/O and blocks. This finally allows the short jobs access to the CPU. Because they are predominately I/O-based jobs, they execute quickly and block, waiting to do I/O. Now, the short jobs must wait again, because the long job is using the I/O resources. Meanwhile, the CPU is idle, because the long job is doing I/O, and the short jobs are also idle, waiting to do I/O. Thus, FIFO can result in long waits and poorly balanced use of resources, both CPU and I/O.

SHORTEST JOB FIRST The **shortest job first (SJF)** method will maximize throughput by selecting jobs that require only a small amount of CPU time. The dispatcher uses as its basis time estimates provided with the jobs when they are submitted. To prevent the user from lying, systems that use this algorithm generally inflict a severe penalty on jobs that run more than a small percentage over their estimate. Since short jobs will be pushed ahead of longer jobs, starvation is possible. When SJF is implemented, it generally includes a dynamic priority factor that raises the priority of jobs as they wait, until they reach a priority where they will be processed next regardless of length. Although SJF maximizes throughput, you might note that its **turnaround time** is particularly inconsistent, since the time required to complete a job depends entirely on the mix of the jobs submitted both before it, and possibly after it.

PRIORITY SCHEDULING **Priority scheduling** assumes that each job has a priority assigned to it. The dispatcher will assign the CPU to the job with the highest priority. If there are multiple jobs with the same priority, the dispatcher will select among them on a FIFO basis.

Priorities can be assigned in different ways. On some systems that charge their users for CPU time, users select the priority. The fee is scaled to the priority, so that higher priorities cost more. In other systems, the priority is assigned by the system. Many factors can be used to affect performance, and the priorities may be assigned statically or dynamically. For example, a system may assign priority on the basis of the resources that the process is requesting. If the system is presently CPU-bound, it can assign an I/O-bound process a high priority to equalize the system.

Another variation on priority scheduling is basically nonpreemptive, but adds a preemptive element. As the process executes, it is periodically interrupted by the dispatcher, which reduces its priority, a little at a time, based on its CPU time used. If its priority falls below that of a waiting process, it is replaced by the higher-priority process.

Preemptive Dispatch Algorithms

ROUND ROBIN The simplest preemptive algorithm, **round robin** gives each process a quantum of CPU time. If the process is not completed within its quantum, it is returned to the back of the ready queue to await another turn. The round-robin algorithm is simple and inherently fair. Since shorter jobs get processed quickly, it is reasonably good on maximizing throughput. Round robin does not attempt to balance the system resources and, in fact, penalizes processes when they use I/O resources, by forcing them to reenter the ready queue. A variation on round robin that is used by some UNIX systems calculates a dynamic priority based on the ratio of CPU time to total time that the process has been in the system. The smallest ratio is treated as the highest priority and is assigned the CPU next. If no process is using I/O, this algorithm reduces back to round robin, since the process that had the CPU most recently will have the lowest priority, and the priority will climb as it waits. The round-robin technique is illustrated in Figure 18.11.

MULTILEVEL FEEDBACK QUEUES The **multilevel feedback queue algorithm** attempts to combine some of the best features of several different algorithms. This algorithm favors short jobs by providing jobs brief, but almost immediate, access to the system. It favors I/O-bound jobs, resulting in good resource utilization. It provides high throughput, with reasonably consistent response time. The technique is shown in Figure 18.12. The dispatcher provides a number of queues. The illustration shows three. A process initially enters the queue at the top level. The queue at the top level has top priority, so a new process will quickly receive a quantum of CPU time. Short processes will complete at this point. Since I/O-bound processes often require just a short amount

FIGURE 18.11

Round-Robin Scheduling

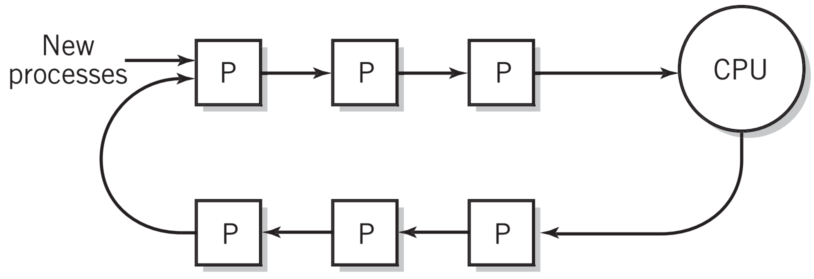
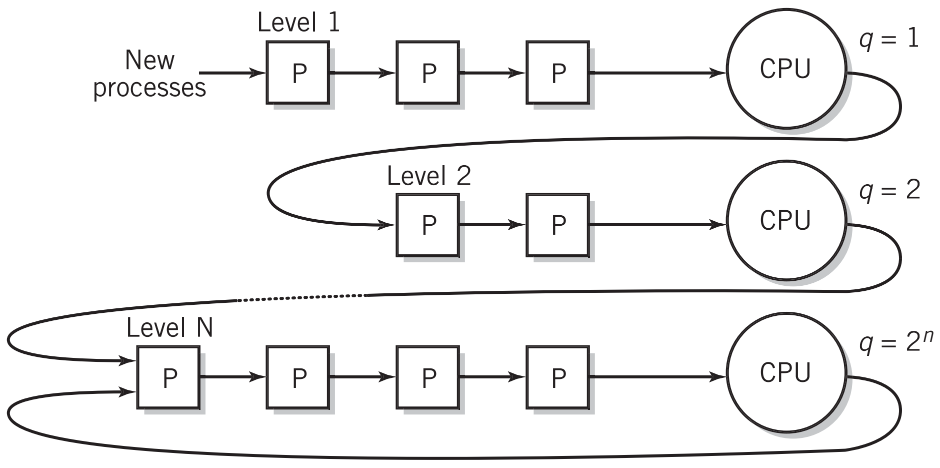


FIGURE 18.12

Multilevel Feedback Queue



of initialization to establish their I/O needs, many I/O-bound processes will be quickly initialized and sent off for I/O.

Processes that are not completed are sent to a second-level queue. Processes in the second-level queue receive time only when the first-level queue is empty. Although starvation is possible, it is unlikely, because new processes pass through the first queue so quickly. When processes in the second level reach the CPU, they generally receive more time. A rule of thumb doubles the number of quanta issued at each succeeding level. Thus, CPU-bound processes eventually receive longer time slots in which to complete execution. This method continues for as many levels as the system provides.

The final level is a round robin, which will continue to provide time until the process is complete. Some multilevel feedback queues provide a good behavior upgrade to processes that meet certain criteria.

DYNAMIC PRIORITY SCHEDULING As noted above, the technique of **dynamic priority recalculation** can also be used as a preemptive dispatching technique. Both Windows 2000 and Linux use a dynamic priority algorithm as their primary criterion for dispatch selection. The algorithms on both systems adjust priority based on their use of resources. Details of the Windows and Linux dispatch algorithms are presented in Supplemental Chapter 2.

18.6 MEMORY MANAGEMENT

Memory management is the planned organization of programs and data into memory. The goal of memory management is to make it as simple as possible for programs to find space, so that they may be loaded and executed, together with the additional space that may be required for various buffers. A secondary and related goal is to maximize the use of memory, that is, to waste as little memory as possible.

Today, nearly all memory management is performed using virtual storage, a methodology that makes it appear that a system has a much larger amount of memory than actually exists physically. Virtual storage is discussed in Section 18.7.

Until the advent of virtual storage, however, effective memory management was a difficult problem. There may be more programs than can possibly fit into the given amount of physical memory space. Even a single program may be too large to fit the amount of memory provided. Compounding the difficulty, recall that most programs are written to be loaded contiguously into a single space, so that each of the spaces must be large enough to hold its respective program. Fitting multiple programs into the available physical memory would require considerable juggling by the memory management module.

In passing, we point out to you that there is also a potential relationship between scheduling and memory management. The amount of memory limits the number of programs that can be scheduled and dispatched. As an extreme example, if the memory is only large enough to hold a single program, then the dispatch algorithm is reduced to single tasking, simply because there is no other program available in memory to run. As more programs can be fit into memory, the system efficiency increases. More programs get executed, concurrently, in the same period of time, since the time that would be wasted when programs are blocked is now used productively. As the number of programs increases still further, beyond a certain point the resident time of each program starts to increase, because the available CPU time is being divided among programs that can all use it, and new programs are continually being added that demand CPU time.

Nonetheless within reason, it is considered desirable to be able to load new processes as they occur, particularly in interactive systems. A slight slowdown is usually considered preferable to a user being told that no resources are available to continue his or her work. As we have hinted a number of times, virtual storage provides an effective and worthwhile solution to the problem of memory management, albeit at the cost of additional hardware, program execution speed, disk usage, and operating system complexity. Before we explain the process of memory management using virtual storage, however, it is useful to offer a brief introduction to traditional memory management techniques to set the issues of memory management in perspective.

Memory Partitioning

The simplest form of memory management divides the memory space into a number of separate partitions. This was the method used prior to the introduction of virtual storage.

Today, it is used only in small embedded systems, where the number of programs running at a given time is small and well controlled. Each partition is used for a separate program.

Two different forms of memory partitioning can be used. **Fixed partitioning** divides memory into fixed spaces. The MINOS memory was managed using fixed partitioning. **Variable partitioning** loads programs wherever enough memory space is available, using a **best-fit, first-fit**, or **largest-fit algorithm**. The best-fit algorithm uses the smallest space that will fit the program. The first-fit algorithm simply grabs the first space available that fits the program. The largest-fit algorithm, sometimes called **worst-fit**, uses the largest space available, on the theory that this will leave the maximum possible space for another program. Figure 18.13 shows variable partitioning at work. Note that the starting positions of programs shift as space becomes available for new programs.

Realistically, partitionining is not suitable for modern general-purpose computing systems. There are two reasons for this:

- First, no matter which method is used, memory partitioning results in **fragmentation** of memory. This is seen in Figure 18.13. Fragmentation means that memory is being used in such a way that there are small pieces of memory available that, if pushed together, would be sufficient to load one or more additional programs. **Internal fragmentation** means that there is memory that has been assigned to a program that does not need it, but can't be used elsewhere. Fixed partitioning results in internal fragmentation. **External fragmentation** means that there is memory that is not assigned, but is too small to use. Variable partitioning will, after a while, result in external fragmentation, since the replacement of one program in an available space with another will almost always result in a bit of space left over. Eventually, it may be necessary to have the memory manager move programs around to reclaim the space. Internal and external fragmentation are shown in Figure 18.14.

FIGURE 18.13

Variable Partitioning of Memory at Three Different Times

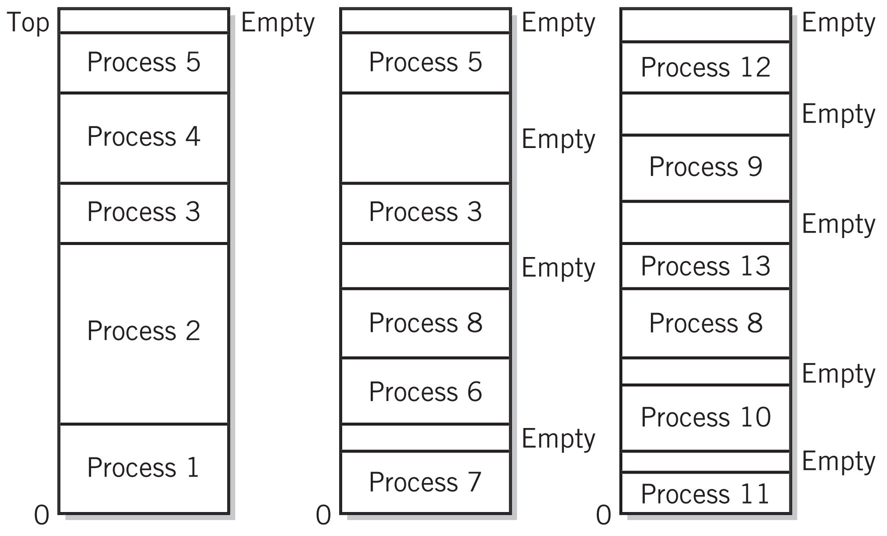
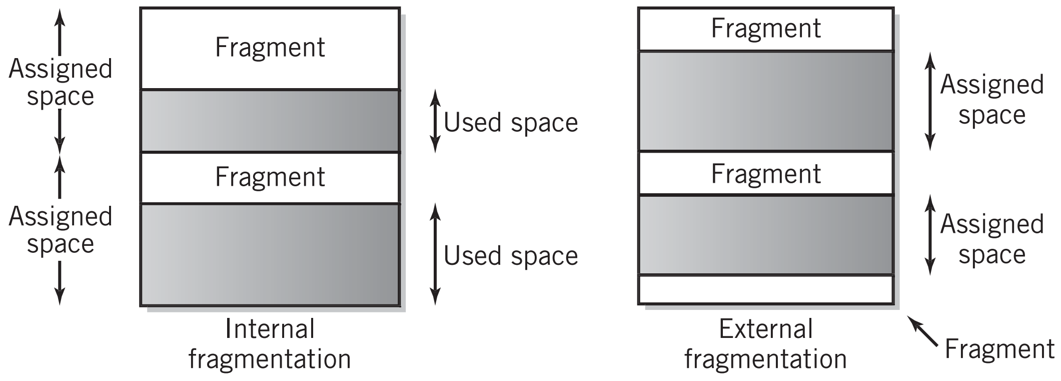


FIGURE 18.14

Internal and External Fragmentation



Although fragmentation is manageable when the number of programs to be loaded is small, and the size of each program known in advance, this is not the case for any general-purpose system.

- Second, the size of most modern programs is sufficiently large that partitioning memory will make it even more difficult to find memory spaces large enough to accommodate all of the programs and data that the average user routinely expects to run concurrently. (You have already seen, in Chapter 17, similar fragmentation and partitioning problems that occur in the storage of files on disk.)

18.7 VIRTUAL STORAGE

Overview

There are three major problems inherent in the traditional (now outdated) memory management schemes described in the previous section:

1. As the system runs, fragmentation makes it harder and harder to find open spaces large enough to fit new programs as they are introduced into the system.
2. From Chapters 6 and 7, you should recall that Little Man programs, and indeed, all, programs, are coded on the assumption that they will be loaded into memory and executed starting from memory location 0. The address field in many, but not all, instructions points to an address where data is found or to the target address of a branch. Of course in reality, it is only possible to load one program at that location in memory. All other programs must be loaded into memory starting from some other address. That means that the operating system's program loader must carefully adjust the address field of all affected instructions to compensate for the actual addresses where the data or the branch target will actually be found.
3. There is often not enough memory to load all of the programs and their resources that we wish to execute at once.

Virtual storage (or **virtual memory—**the words are synonymous), is the near-universally accepted solution to the problems inherent in memory management. Virtual storage uses a combination of operating system software and special purpose hardware to simulate a memory that meets the management needs of a modern system. The primary method of implementing virtual storage is called **paging**.

Pages and Frames

To begin, assume that memory is divided into blocks. These blocks are called **frames**. Usually, all the frames are of equal size, typically 1 KB–4 KB. The exception, an alternative method called **segmentation** is used more rarely, and will be described later. The size of the blocks is permanently set as a design parameter of the particular hardware architecture, based on a number of factors. The most important criterion for the block size is that it must correspond exactly to a particular number of address bits. This guarantees that every address within the block is expressed by the same number of digits. In the Little Man Computer, for example, a block size of 10 would be the only reasonable choice, since every address within the block would be expressed with one digit (0–9). Similarly, in a real, binary-based computer, a 12-bit address can access an address space of exactly 4 KB.

The number of blocks depends on the amount of memory installed in the machine, but, of course, can't exceed the largest memory address possible, as determined by the architecture of the instruction set. We could install 60 mailboxes in the Little Man Computer, for example; this would give us six frames within the constraint that the address field of the LMC instructions limits us to a maximum of 100 mailboxes, or ten frames.

The blocks are numbered, starting from 0. Because the block size was selected to use a specific, fixed number of bits (or decimal digits for the Little Man Computer), an actual memory address consists simply of the block number concatenated with the address within the block. By selecting a frame size that corresponds exactly to a given number of digits, we can simply concatenate to get the whole address.

EXAMPLE

Suppose that a Little Man memory consisted of 60_{10} mailboxes, divided into 6 frames. Each frame is a one-digit block of size 10. The frames would be numbered from 0 through 5, and the address of a particular location within the frame would be a number from 0 to 9. Then, location number 6 in frame 3 would correspond to location 36 in memory. Similarly, memory address 49 would be in frame 4; the address would correspond to location 9 in that frame. Figure 18.15(a) illustrates this example.

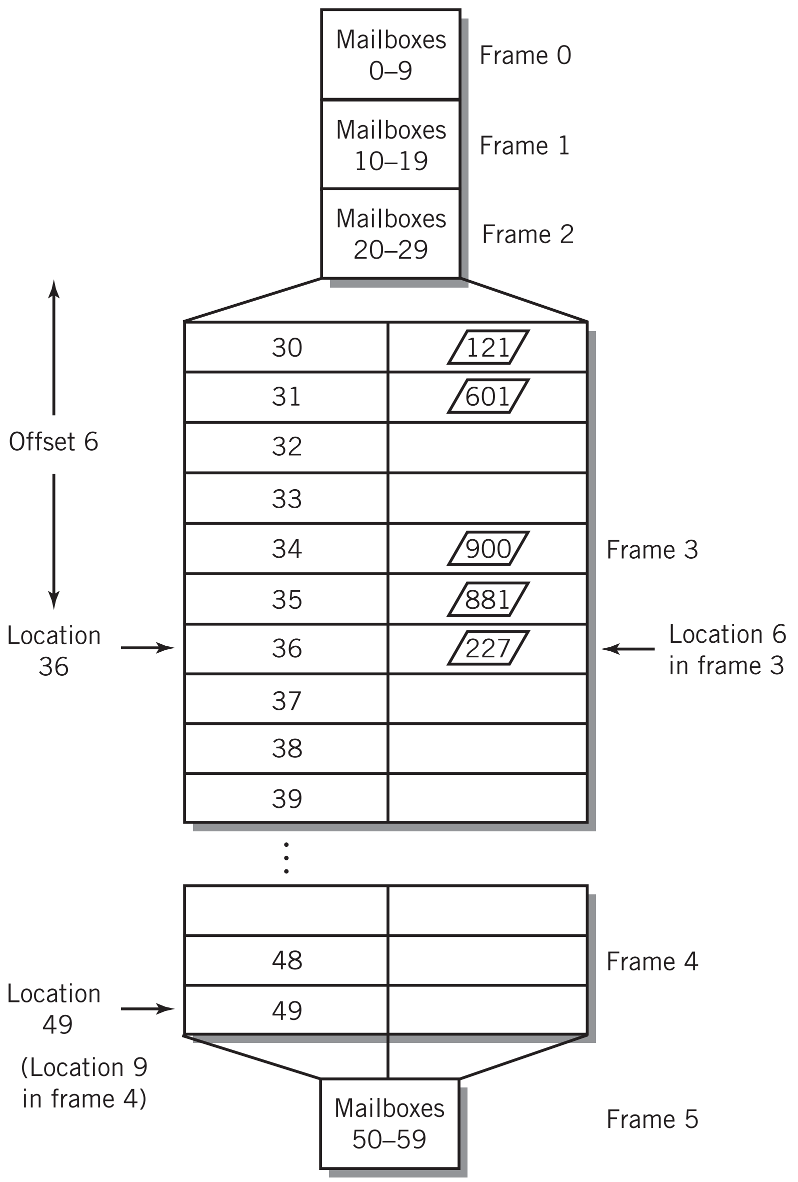
EXAMPLE

Now consider a binary computer with 1 GB of memory divided into 4 KB frames. There will be 256 K, or approximately a quarter of a million, frames. (We divided 1 G by 4 K to get 256 K.) Another way to look at this is to realize that to address 1 GB of memory requires a 30-bit address. 4 KB frames will require 12 bits for addresses; therefore the number of frames will correspond to 18 bits, or 256 K frames.

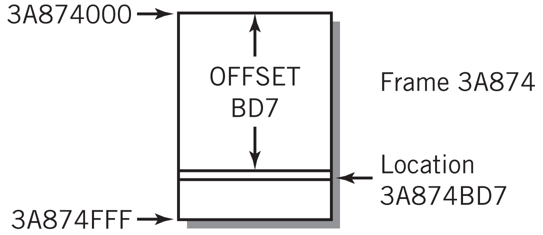
For convenience, we will illustrate the example in hexadecimal. Remember that each hexadecimal digit represents 4 bits. Memory location $3A874BD7_{16}$ would then be located in

FIGURE 18.15

Identifying Frames and Offsets



a. Little man computer frames and offsets



b. Binary computer frames and offsets

the frame block numbered $3A874_{16}$, and specifically found at location $BD7_{16}$ of that frame. Notice that the frame block number requires a maximum of 18 bits and that the location within the frame uses 12 bits. See Figure 18.15(b) for clarification. Similarly, location number 020_{16} within frame number $15A3_{16}$ corresponds to memory location $15A3020_{16}$.

Effectively, we are dividing each memory address into two parts: a frame number and the specific address within the particular frame. The address within the frame is called an **offset**, because it represents the offset from the beginning of the frame. (It should be clear to you that the first address in a frame, which is the beginning of the frame, is 0, with the correct number of digits of course, so address 1 is offset by 1 from the beginning, and so on.)

It is not immediately obvious why we are dividing memory into frame blocks, but the reason will become clear shortly. Here's a hint: note the similarity between the frame blocks that make up the memory space and the blocks that make up the space on a hard disk. Then, recall that we can find data within a file even if we store the files on a hard disk noncontiguously.

Suppose we also divide a program into blocks, where each block in the program is the same size as a frame. The blocks in a program are called **pages**. See Figure 18.16. The number of pages in a program obviously depends on the size of the program. We will refer to the instruction and data memory address references in a program as "logical" or "virtual" memory references, as opposed to the physical memory references that actually go out to memory and store and retrieve instructions and data. The words *logical* and *virtual* are used interchangeably.

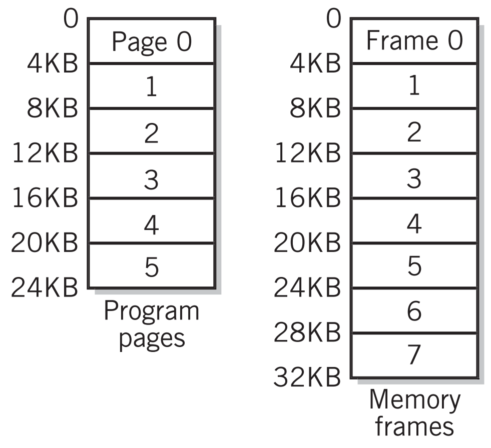
Like frames, the number of pages is also constrained by the instruction set architecture, but, as we will show you later, it is not limited to the size of installed memory. Stated differently, a program can be larger than the amount of memory installed in a computer, and still execute successfully, although possibly slowly.

The key to this magical sleight-of-hand is a technique called **dynamic address translation (DAT)**. Dynamic address translation is built into the CPU hardware of every modern computer. The hardware automatically and invisibly translates every individual address in a program (the *virtual* addresses) to a different corresponding physical location (the *physical* addresses). This allows the operating system's program loader to place the pages of a program into any available frames of physical memory, page by page, noncontiguously, so that it is not necessary to find a contiguous space large enough to fit the entire program. Any page of any program can be placed into any available frame of physical memory. Since every frame is essentially independent, the only fragmentation will be the small amount of space left over at the end of the last page of each individual program.

For each program, the operating system creates a **page table**, which keeps track of the corresponding frame location in physical memory where each page is stored. There is one entry in the table for each page of the program. The entry contains the page number and its corresponding frame number.

FIGURE 18.16

Frames and Pages



Since each page fits exactly into a frame, the offset of a particular address from the beginning of a page is also exactly the same as the offset from the beginning of the frame where the page is physically loaded. To translate a virtual address to a physical address, the virtual address is separated into its page number and an offset; a look-up in the program's page table locates the entry in the table for the page number, then translates, or *maps*, the virtual memory reference into a physical memory location consisting of the corresponding frame number and the same offset. We remind you again: this operation is implemented in hardware. Every memory reference in a fetch-execute cycle goes through the same translation process. The address that would normally be sent to the memory address register (MAR) is mapped through the page table and *then* sent to the MAR. It is also important to remember that the translation process is entirely invisible to the program. As far as the program can tell, every memory reference is exactly where the program says it is.

A simple example illustrates the translation process.

EXAMPLE

Consider a program that fits within one page of virtual memory. Placement of the program is shown in Figure 18.17a. The page for the program would be numbered 0, of course. If we assume a page size of 4 KB, then any logical memory location in this program would be between 0000 and 0FFF. Suppose frame 3 is available in which to place this program in physical memory. Then the physical addresses for this program to execute properly must all be between 3000 and 3FFF. We obtain the correct address in each case by changing the page number (0) to the frame number (3), keeping the offset the same as it was. A LOAD instruction, LOAD 028A, for example, would be translated to find its data in physical memory at 328A.

FIGURE 18.17(a)

A Simple Page Table Translation

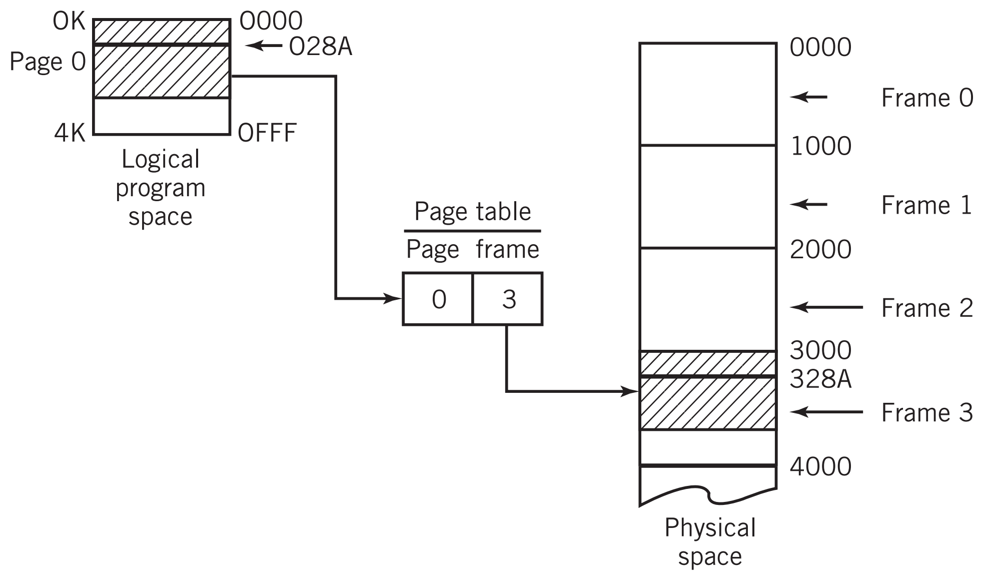
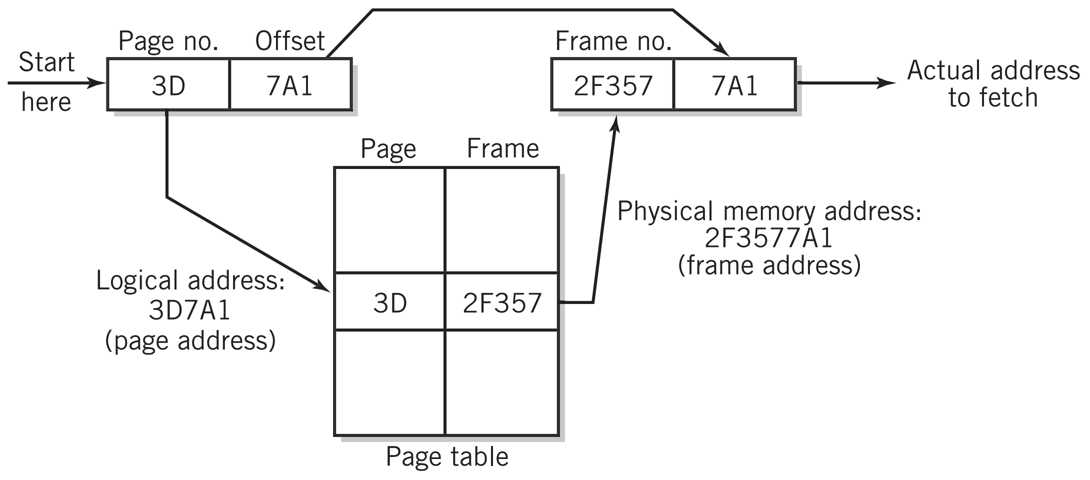


FIGURE 18.17(b)

The Page Translation Process



Another example, drawn differently to show the translation process more clearly, is presented in Figure 18.17b.

With virtual storage, each process in a multitasking system has its own virtual memory, and its own page table. Physical memory is shared among the different processes. Since all the pages are the same size, any frame may be placed anywhere in memory. The pages selected do not have to be contiguous. The ability to load any page into any frame solves the problem of finding enough contiguous memory space in which to load programs of different sizes.

Figure 18.18 shows a mapping for three programs located in memory. Note that each program is written as though it will load starting from address 0, eliminating the need for the loader to adjust a program's memory addresses depending on where the program is loaded. Since each program's page table points to a different area of physical memory, there is no conflict between different programs that use the same virtual addresses.

To complete this part of the discussion, let us answer two questions that may have occurred to you:

- Where do the page tables reside and how are they accessed by the hardware for address translation?
- How are memory frames managed and assigned to pages?

The simple answer to the first question is that page tables reside in memory, just like any other program or data. A page table address register in the CPU holds a pointer to the address in memory where the page table is located. The pointer is stored as part of the process control block; the address of the page table for the current process is loaded into the register as part of the context switching mechanism.

Although this answer is accurate, it is not quite complete. There are a few bells and whistles that improve performance, to be discussed later in this section under the paragraph title *Page Table Implementation*.

FIGURE 18.18

Mapping for Three Processes

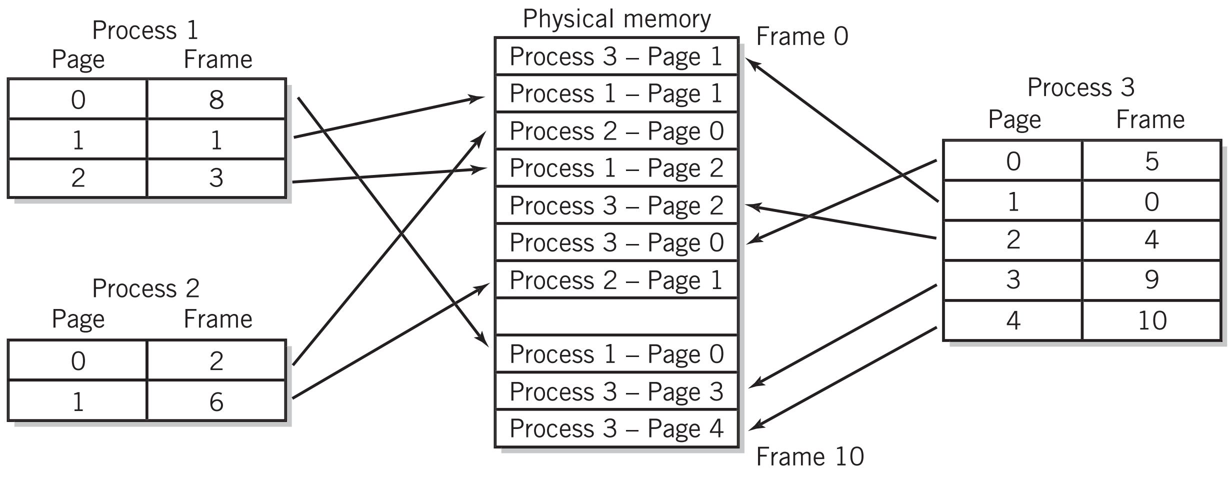


FIGURE 18.19

Inverted Page Table for Process Page Tables Shown in Figure 18.18

Frame	Process #	Page	
0	3	1	
1	1	1	
2	2	0	
3	1	2	
4	3	2	
5	3	0	
6	2	1	
7			Free page frame
8	1	0	
9	3	3	
10	3	4	

The answer to the second question is that physical memory is shared among all of the active processes in a system. Since each process has its own page table, it is not practical to identify available memory frames by accumulating data from all of the tables. Rather, there must be a single resource that identifies the entire pool of available memory frames from which the memory manager may draw, when required. There are two common approaches in use. One is to provide an *inverted page table*, which lists every memory frame with its associated process and page. This table shows the actual use of physical memory at every instant. Any frame without an associated page entry is available for allocation. Figure 18.19 illustrates an inverted page table. We'll leave it as a simple exercise for you to identify the available frames.

A second method maintains a list of available frames, usually as a simple linked list. When a process needs frames, it takes them from the top of the list. When a process exits, its frames are added to the end of the list. Since frame contiguity is unimportant, this is an effective way to manage the free frame pool.

The Concept of Virtual Storage

The first two issues of memory management that we raised initially are thus solved. But, as the TV infomercials say, "Wait—there's still more!"

As we noted before, the third major challenge for memory management is the limited total quantity of physical memory available. Even hundreds of megabytes of memory can only hold a few modern programs. Up to this point, we have assumed that there is a frame available for every page that needs one. While page-to-frame translation has eliminated the question of how to fit programs into existing memory space, the next step is more important and useful: we will show you how the concept of virtual storage allows the system to extend the address space far beyond the actual physical memory that exists. As you will see, the additional address space required to store a large number of programs is actually provided in an auxiliary form of storage, usually disk, although some systems now make it possible to use flash memory for this purpose.

So far we have assumed that all the pages of an executing program are located in frames somewhere in physical memory. Suppose that this were not the case—that there are not enough frames available to populate the page table when the program is loaded. Instead, only some pages of the program are present in physical memory. Page table entries without a corresponding frame are simply left empty. Can the program execute?

The answer depends on which pages are actually present in corresponding frames of physical memory. To execute a program instruction or access data, two requirements must be met.

- The instruction or data must be in physical memory.
- The page table for that program must contain an entry that maps the virtual address being accessed to the physical location containing the instruction or data.

These two requirements are related. The existence of a page listing in the page table implies that the required value is in memory and vice versa. If these two conditions are met, then the instruction can execute as usual. This suggests, correctly, that instructions and data that are *not* being accessed do not have to be in memory. At any given time in the execution of a program, there are active pages and inactive pages. Only the active pages require corresponding frames in the page table and in physical memory. Thus, it is possible to load only a small part of a program and have it execute.

Page Faults

The real question is what happens when an instruction or data reference is on a page that does not have a corresponding frame in memory. The memory management software maintains the page tables for each program. If a page table entry is missing when the memory management hardware attempts to access it, the fetch-execute cycle will not be able to complete.

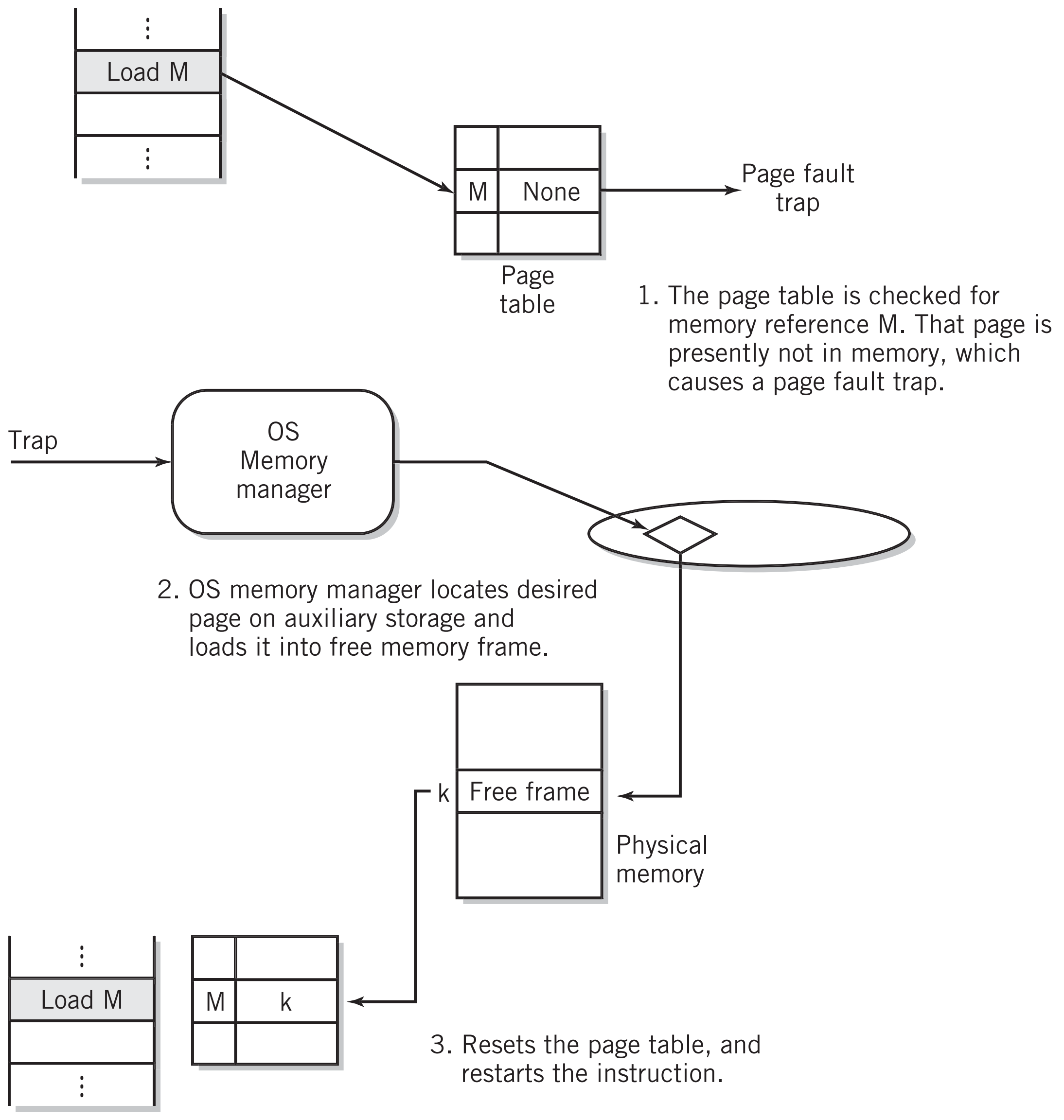
In this case, the CPU hardware causes a special type of interrupt called a **page fault** or a **page fault trap.** This situation sounds like an error, but actually it isn't. The page fault concept is part of the overall design of virtual storage.

When the program is loaded, an exact, page-by-page image of the program is also stored in a known auxiliary storage location. The auxiliary storage area is known as a **backing store** or, sometimes, as a **swap space** or a **swap file.** It is usually found on disk, but some recent systems use flash memory for this purpose. Also assume that the page size and the size of the physical blocks on the auxiliary storage device are integrally related, so that a page within the image can be rapidly identified, located, and transferred between the auxiliary storage and a frame in memory.

When a page fault interrupt occurs, the operating system memory manager answers the interrupt. And now the important relationship between the hardware and the operating system software becomes clearer. In response to the interrupt, the memory management software selects a memory frame in which to place the required page. It then loads the page from its program image in auxiliary storage. If every memory frame is already in use, the software must pick a page in memory to be replaced. If the page being replaced has been altered, it must first be stored back into its own image, before the new page can be loaded. Page replacement algorithms are discussed later in this section. The process of page replacement is also known as **page swapping.** The steps involved in handling a page fault are shown in Figure 18.20.

FIGURE 18.20

Steps in Handling a Page Fault


Load M
M
None
Page fault trap
Page table
1. The page table is checked for memory reference M. That page is presently not in memory, which causes a page fault trap.
Trap
OS Memory manager
2. OS memory manager locates desired page on auxiliary storage and loads it into free memory frame.
k
Free frame
Physical memory
Load M
M
k
3. Resets the page table, and restarts the instruction.

Most systems perform page swapping only when it is required as a result of a page fault. This procedure is called **demand paging.** A few systems attempt to anticipate page needs before they occur, so that a page is swapped in before it is needed. This technique is called **prepaging.** To date, prepaging algorithms have not been very successful at predicting accurately the future page needs of programs.

When the page swap is complete, the process may be started again where it left off. Most systems return to the beginning of the fetch-execute cycle where the page fault occurred, but a few systems restart the instruction in the middle of its cycle. Regardless of which way is used, the required page is now present, and the instruction can be completed. The importance of page swapping is that it means that a program does not have to be loaded into memory in its entirety to execute. In fact, the number of pages that must be loaded into memory to execute a process is quite small. This issue is discussed further in the next section.

Therefore, virtual storage can be used to store a large number of programs in a small amount of physical memory and makes it appear that the computer has more memory than is physically present. Parts of each program are loaded into memory. Page swapping handles the situations when required pages are not physically present. Furthermore, since the virtual memory mapping assures that any program page can be loaded anywhere into memory, there is no need to be concerned about allocating particular locations in memory. Any free frame will do.

Working Sets and the Concept of Locality

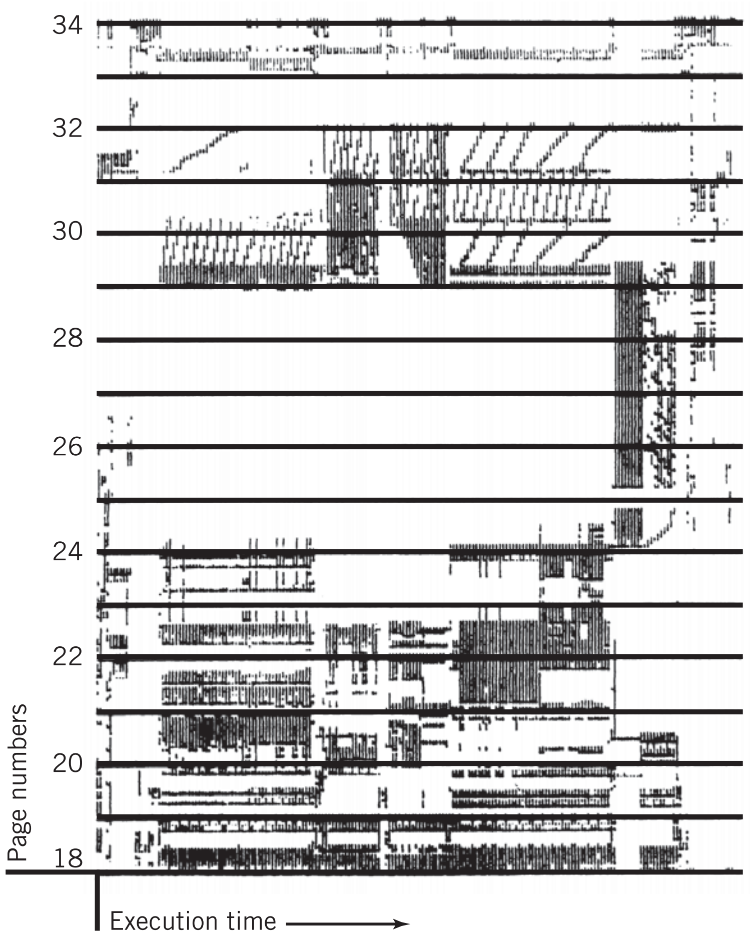
How many pages should be assigned to a new process just entering the system? It would seem that the more pages that are initially assigned to a process, the less likely it would be that a page fault would occur during execution of the process. Conversely, the more pages assigned to a process, the fewer the number of processes that will fit into memory. There is a lower limit on the number of pages to be assigned, which is determined by the instruction addressing modes used by the particular computer. Executing a single instruction in an indirect addressing machine, for example, requires at least three pages, the page where the instruction resides, the page where the indirect address is stored, and the page where the data is found. This assumes that each item is on a different page, but it is necessary to make the worst-case assumption to prevent instructions from failing in this way. Other instruction sets can be analyzed similarly.

More practically, experimentation performed in the early 1970s showed that during execution programs exhibit a tendency to stay within small areas of memory during any given period of time. Although the areas themselves change over time, the property continues to hold throughout the execution of the program. This property is called the **concept of locality.** An illustration of the concept at work is shown in Figure 18.21. The concept of locality makes sense intuitively. Most well-written programs are written modularly. During the initial phase of execution of a program, a small part of the program initializes variables and generally gets the program going. During the main body of the program, the likely operations consist of small loops and subroutine calls. These represent the different area of memory being executed at different times.

An effective compromise would allocate a sufficient number of pages to satisfy the locality of a particular program. This number of pages would be sufficient to run the

FIGURE 18.21

Memory Use with Time, Exhibiting Locality



Source: Operating Systems 2/e by Stallings, W. © 1995. Reprinted by permission of Prentice-Hall, Upper Saddle River, NJ.

program normally. Page faults would only occur when the local area being used by the program moves to a different part of the program. The number of pages that meets the requirement of locality is called a **working set.** It differs somewhat from program to program, but it is possible to establish a reasonable page quantity that meets the needs of most programs without an undue number of page faults. Some systems go further and monitor the number of page faults that actually occur for each process. They then dynamically adjust the size of the working set for each process to try to meet its needs.

Page Sharing

An additional feature of virtual storage is the ability to share pages among different processes that are executing the same program. As long as the code is not modified, that is, the code is pure, there is no need to have duplicate program code stored in memory. Instead, each process shares the same program code page frames and provides its own work space for data. The page tables for each process will simply point to the same physical memory frames. This simplifies the management of multiple processes executing the same program.

Page Replacement Algorithms

There will be times on a heavily loaded system when every available page in memory is in use. When a page fault occurs, the memory manager must pick a page to be eliminated from memory to make room for the new page that is needed. The goal, of course, is to replace a page that will not be needed in the near future. There are a number of different algorithms that are used. As usual with operating system algorithms, each has advantages and disadvantages, so selecting an algorithm is a matter of trade-offs. Some systems select pages to be replaced from the same process. Others allow replacement from any process in the system. The former is known as **local page replacement**; the latter is called **global page replacement.** Global page replacement is more flexible, since there are a much larger number of pages to choose from. However, global page replacement affects the working set size of different processes and must be managed carefully.

As an additional consideration, some pages must never be removed from memory because doing so could eventually make the system inoperable. For example, removing the disk driver would make it impossible to swap in any new pages, *including the disk driver!* To prevent this situation, the frames corresponding to critical pages are locked into memory. These frames are called **locked frames.** An additional bit in each row of the page table is set to indicate that a frame is locked. Locked frames are never eligible for replacement.

FIRST-IN, FIRST-OUT PAGE REPLACEMENT The simplest **page replacement algorithm** is a first-in, first-out algorithm. The oldest page remaining in the page table is selected for replacement. FIFO does not take into account usage of the page. Logically, a page that has been in memory for a long period of time is probably there because it is heavily used. The page being removed may be in current use, which would result in a second page fault and force the system to reload the page almost immediately. FIFO has a second, interesting deficiency. You would assume that increasing the number of pages available to a process would reduce the number of page faults for that process. However, it has been shown that under certain conditions, use of the FIFO page replacement algorithm results in more page faults with an increased number of pages, instead of fewer. This condition is known as **Belady's anomaly.** If you are interested, examples of Belady's anomaly can be found in the references by Deitel [DEIT03] and Silberschatz et al. [SILB08]. For these reasons, FIFO is not considered a good page replacement algorithm.

LEAST RECENTLY USED PAGE REPLACEMENT The **least recently used (LRU) algorithm** replaces the page that has not been used for the longest time, on the assumption that the page probably will not be needed again. This algorithm performs fairly well, but requires a considerable amount of overhead. To implement the LRU algorithm, the page tables must record the time every time the page is referenced. Then, when page replacement is required, every page must be checked to find the page with the oldest recorded time. If the number of pages is large, this can take a considerable amount of time.

LEAST FREQUENTLY USED PAGE REPLACEMENT Another possibility is to select the page that has been used the least frequently. Intuitively, this algorithm has appeal, since it would seem that a page not used much is more replaceable than one that has received a lot of use. The flaw with this algorithm is that a page that has just been brought into memory has not been used much, compared to a page that has been in memory for a while. Still, the new page was brought into memory because it was needed, and it is likely that it will be needed again.

NOT USED RECENTLY PAGE REPLACEMENT The **not used recently (NUR) algorithm** is a simplification of the least recently used algorithm. In this method, the computer system hardware provides two additional bits for each entry in the page tables. One bit is set whenever the page is referenced (*used*). The other bit is set whenever the data on the page is modified, that is, written to. This second bit is called a **dirty bit.** Periodically, the system resets all the reference bits.

The memory manager will attempt to find a page with both bits set to 0. Presumably, this is a page that has not been used for a while. Furthermore, it is a page that has not been modified, so it is necessary only to write the new page over it. The page being replaced does not have to be saved back to the backing store, since it has not been modified. The second choice will be a page whose dirty bit is set, but whose reference bit is unset.

This situation can occur if the page has not been accessed for a while, but was modified when it was accessed, prior to the resetting of the reference bits. This page must be written back to the backing store before a new frame can be read into its spot. Third choice will be a page that has been referenced, but not modified. And finally, least desirable will be a page that has been recently referenced and modified. This is a commonly used algorithm.

One difficulty with this algorithm is that gradually all the *used* bits fill up, making selection difficult or impossible. There are a number of variations on this algorithm that solve this problem by selectively resetting used bits at regular intervals or each time a page replacement occurs. The most common approach pictures the process pages as numerals on a clock. When a page replacement must be found, the clock hand moves until it finds an unset used bit and the corresponding page is replaced. Pages with set used bits that the hand passes over are reset. The hand remains at the found replacement page awaiting the next replacement requirement. This variation on NUR is called the **clock page replacement algorithm**.

SECOND CHANCE PAGE REPLACEMENT ALGORITHMS One second chance algorithm uses an interesting variation on FIFO, using a referenced bit similar to that of NUR. When the oldest page is selected for replacement, its referenced bit is checked. If the referenced bit is set, the bit is reset, and the time is upgraded, as though the page had just entered memory. This gives the page a second pass through the list of pages. If the referenced bit is not set, then the page is replaced, since it is safe to assume that it has not been referenced in some time.

Another second chance algorithm keeps a small pool of free pages that are not assigned. When a page is replaced, it is not removed from memory but, instead, is moved into the free pool. The oldest page in the free pool is removed to make room. If the page is accessed while in the free pool, it is moved out of the free pool and back into the active pages by replacing another page.

Both second chance algorithms reduce the number of disk swaps by keeping what would otherwise be swapped-out pages in memory. However, the first of these algorithms has the potential of keeping a page beyond its usefulness, and the second decreases the number of possible pages in memory by using some of those pages for the free pool.

Thrashing

A condition that can arise when a system is heavily loaded is called **thrashing.** Thrashing is every system administrator's nightmare. Thrashing occurs when every frame of memory is in use, and programs are allocated just enough pages to meet their minimum requirement. A page fault occurs in a program, and the page is replaced by another page that will itself be needed for replacement almost immediately. Thrashing is most serious when global page replacement is used. In this case, the stolen page may come from another program. When the second program tries to execute, it is immediately faced with its own page fault. Unfortunately, the time required to swap a page from the disk is long compared to CPU execution time, and as the page fault is passed around from program to program, no program is able to execute, and the system as a whole slows to a crawl or crashes. The programs simply continue to steal pages from each other. With local page replacement, the number of thrashing programs is more limited, but thrashing can still have a serious effect on system performance.

Page Table Implementation

As we mentioned previously, the data in the page table must be stored in memory. You should realize that data in the page table must be accessed during the fetch-execute cycle, possibly several times, if the fetch-execute cycle is executing an instruction with a complex addressing mode. Thus, it is important that the page table be accessed as quickly as possible, since the use of paging can negatively affect the performance of the system in a major way otherwise. To improve access, many systems provide a small amount of a special type of memory called **associative memory.** Associative memory differs from regular memory in that the addresses in associative memory are not consecutive. Instead, the addresses in associative memory are assigned to each location as labels. When associative memory is accessed, every address is checked at the same time, but only the location whose address label matches the address to be accessed is activated. Then the data at that location can be read or written. (Cache memory lines are accessed similarly.)

A mailbox analogy might be useful in helping you to understand associative memory. Instead of having mailboxes that are numbered consecutively, picture mailboxes that have those little brass inserts that you slide a paper label into. On each label is written the address of that particular box. By looking at all the boxes, you can find the one that contains your mail. For a human, this technique would be slower than going directly to a mailbox in a known location. The computer, however, is able to look at every address label simultaneously.

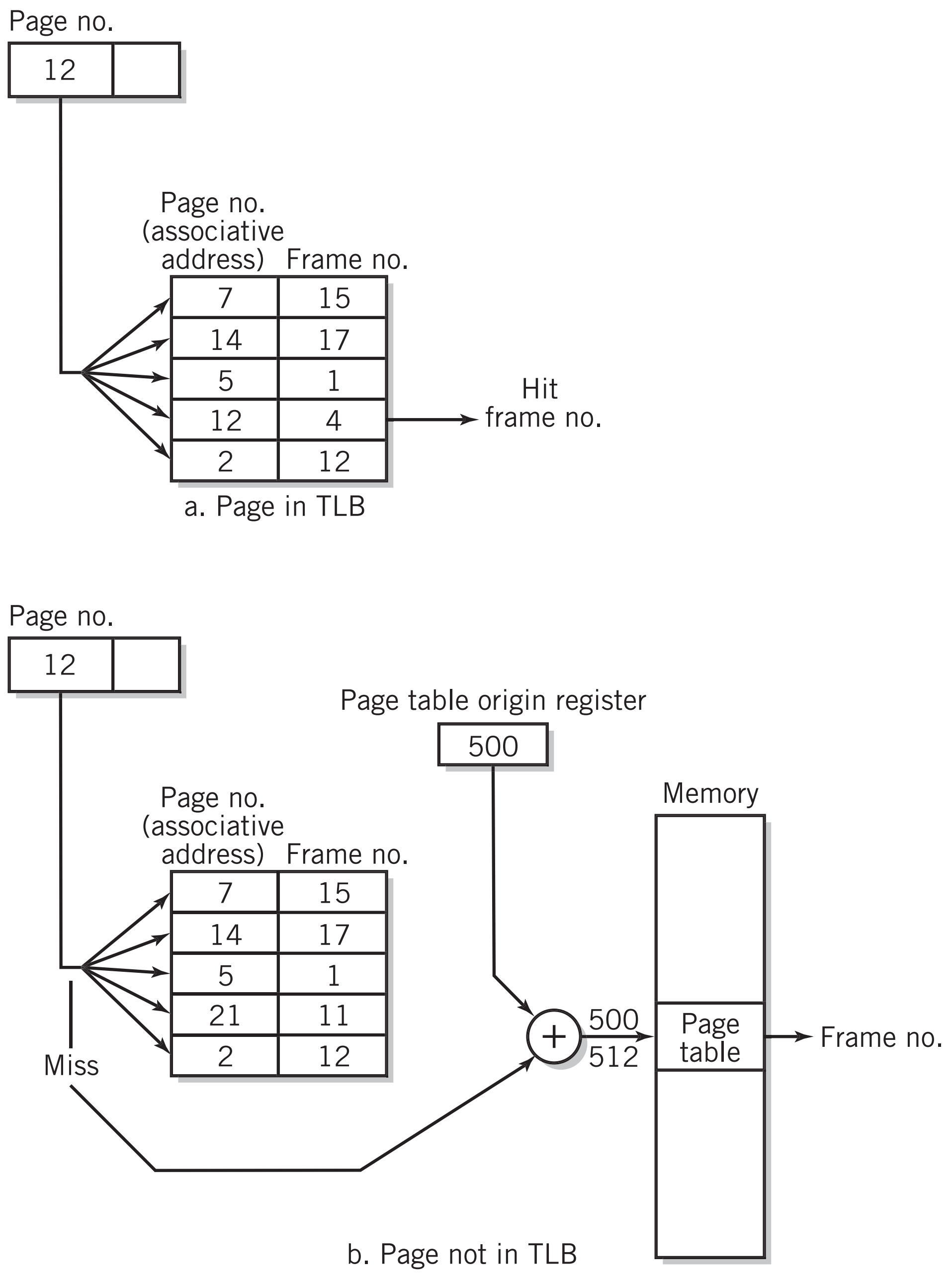
Suppose, then, that the most frequently used pages are stored in this associative memory. They may be stored in any order, since the address labels of all locations are checked simultaneously. The page number is used as the address label that is being accessed. Then, the only frame number that will be read is the one that corresponds to that page. A page table that is constructed this way is known as a **translation lookaside buffer** (**TLB**), table.

The number of locations available in a TLB table is small because associative memory is expensive. There must be a second, larger, page table that contains all the page entries for the program. When the desired page is found in the TLB table, known as a **hit,** the frame can be used without further delay. When the desired page is not found in the TLB table, called a **miss,** the memory management unit defaults to conventional memory, where the larger page table is stored. Access to the table in memory does, in fact, require an extra memory access, which will significantly slow down the fetch-execute cycle, but that can't be helped.

To locate the correct entry in the larger page table, most computers provide a special register in the memory management unit that stores the address of the origin of the page table in memory. Then, the nth page can be located quickly, since its address in memory is the address of the origin plus the offset. The process of page table lookup is shown in Figure 18.22. Figure 18.22a shows how the page is accessed when the page is found in associative memory; Figure 18.22b shows the procedure when the TLB does not contain the desired page.

FIGURE 18.22

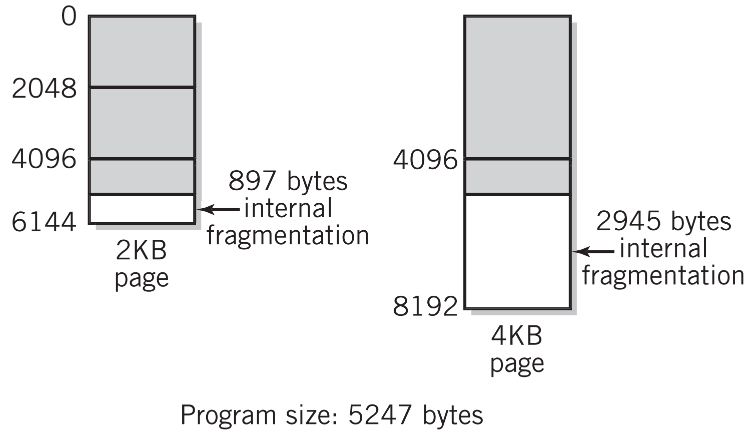
Frame Lookup Procedures: (a) Page in TLB, (b) Page Not in TLB



Beyond the requirement that the frame or page size conform exactly to a fixed number of bits, the size of a frame or page is determined by the computer system designer as a fundamental characteristic of the system. It is not changeable. There are several trade-offs in the determination of page size. The page table for a program must contain an entry for every page in the program. The number of pages is inversely proportional to the page size, so as the page size is decreased, the number of page table entries required increases.

FIGURE 18.23

Internal Fragmentation


0
2048
4096
6144
897 bytes internal fragmentation
2KB page
4096
8192
2945 bytes internal fragmentation
4KB page
Program size: 5247 bytes

On the other hand, we have assumed that the size of the program corresponds exactly to the amount of memory occupied by the pages required for the program. This is not usually true. More commonly, the last page is partly empty. The wasted space is internal fragmentation. An example is shown in Figure 18.23.

Also, if the pages are small, memory will consist of more pages, which allows more programs to be resident. Conversely, smaller pages will require more swapping, since each program will have less code and data available to it at any given time. Experimentally, designers have determined that 2 KB or 4 KB pages seem to optimize overall performance.

Page tables on large machines can, themselves, require considerable memory space. One solution is to store the page tables in virtual memory. Page tables, or portions of page tables, in current use will occupy frames, as usual. Other portions or tables will reside only in virtual memory until needed.

Segmentation

Segmentation is essentially similar to paging conceptually, but differs in many details. A segment is usually defined as a logically self-contained part of a program, as determined by a programmer or by a compiler translation program. Thus, in most systems, segments can be variable in size. (A few systems define segments instead as large pages, of fixed size, but of 1 MB, 2 MB, or 4 MB, or even more. This definition does not interest us here, since the previous discussion of paging applies in this case. When a fixed size segment is further divided into pages, the program address is divided into three parts, a segment, a page, and an offset, and the mapping process takes place in two steps, but the procedure is otherwise identical to our previous discussion.) Program segments can represent parts of a program such as main routines and subroutines or functions, or they can represent program code and data, even separate data tables. The crucial difference between segments and pages is that due to their variability in size, the boundaries between segments do not fall on natural borders, as pages do.

Therefore, in the **segment table,** it is necessary to provide the entire physical address for the start of the segment instead of just a page number. It is also necessary to record the size or upper limit location of the segment, so that the system can check to make sure that the requested location does not fall outside the limit of the segment. Otherwise, it would be possible to read or write data to a location belonging to another segment, which would compromise the integrity of the system. This is not a problem with paging, since it is impossible for the offset to exceed the size of a page.

The program segment numbers are stored with each segment and are treated similarly to page numbers. For each segment number, there is an entry in the segment table containing the starting location of the segment in physical memory plus the limit of the segment. The physical address is calculated by adding the program segment offset from the start of the segment to the memory starting location and checking this value against the limit. As with the page table, part of the segment table can be stored in associative memory for faster access. When segmentation and paging are both provided, there may be two TLB tables, one for each. When both are provided, the translation process performs its mapping in two steps. First, the segment table is used to determine the location of the pages that make up the segment. Then, the page table locates the desired frame. Since the programmer establishes the segments, segmentation is less invisible to the programmer than paging, even though during operation it is still invisible. This provides a few advantages to the programmer, stemming from the fact that each segment can be treated independently. This means that a particular segment could be shared among different programs, for example. Nonetheless, segmentation is harder to operate and maintain than paging and has rapidly fallen out of favor as a virtual storage technique.

Process Separation

The use of virtual storage offers one additional benefit that should be mentioned. Under normal program execution without virtual storage, every memory access has the potential to address a portion of memory that belongs to a different process. This would violate system security and data integrity; for example, a program in a partitioned memory could access data belonging to another process simply by overflowing an array. Prior to virtual storage memory management, this was a difficult problem. It was necessary to implement memory access limits for each process in hardware, because there is no way for operating system software to check every attempted memory access while a program is executing. With virtual storage, every memory access request points to a logical address, not a physical one. Since the logical address is within the space of the process itself, the translation process assures that it is not possible to point to a physical address belonging to another process, unless the page tables have been set up intentionally to share frames between the processes. Thus, virtual storage provides simple, effective separation protection between processes.

18.8 SECONDARY STORAGE SCHEDULING

On a busy system, it is common to have a number of disk requests pending at any given time. The operating system software will attempt to process these requests in a way that enhances the performance of the system. As you might expect by now, there are several different disk scheduling algorithms in use.

First-Come, First-Served Scheduling

First-come, first-served (FCFS) scheduling is the simplest algorithm. As requests arrive, they are placed in a queue and are satisfied in order. Although this may seem like a fair algorithm, its inefficiency may result in poorer service to every request in the queue. The problem is that seek time on a disk is long and somewhat proportional to the distance that the head has to move. With FCFS, one can expect the head to move all over the disk to satisfy requests. It would be preferable to use an algorithm that minimizes seek distances.

This would suggest processing requests that are on nearby tracks first. The other algorithms in use attempt to do so.

Shortest Distance First Scheduling

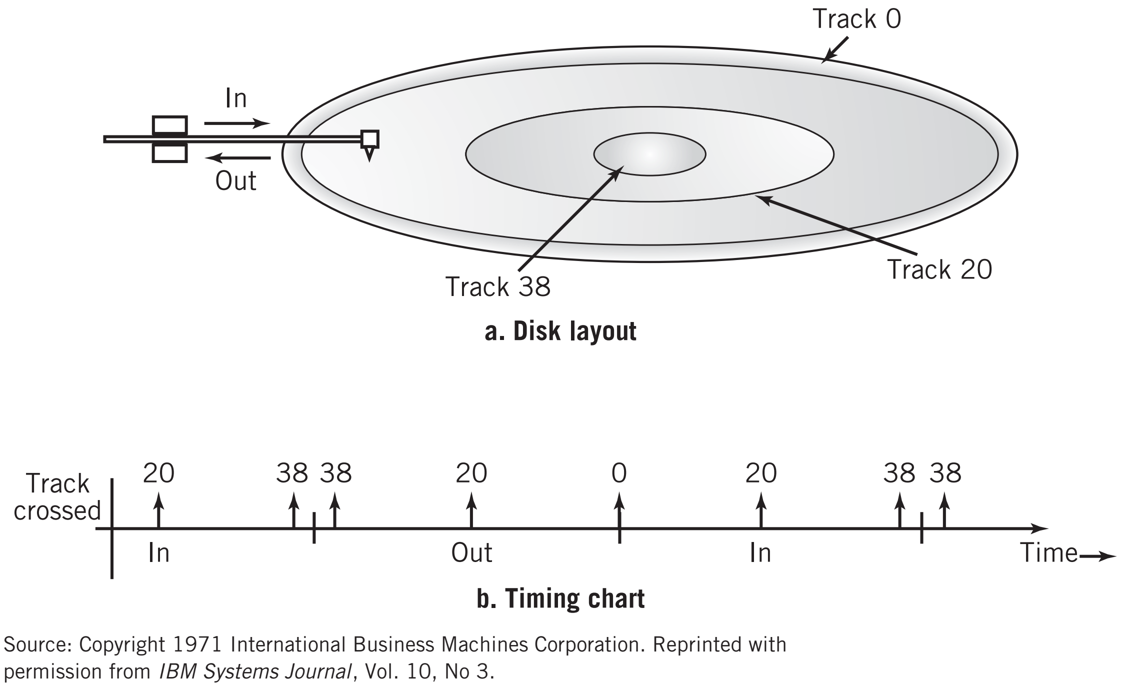
The **shortest distance first** (**SDF**) **scheduling** algorithm looks at all the requests in the queue and processes the one nearest to the current location of the head. This algorithm suffers from the possibility of **indefinite postponement.** If the head is near the middle track on the disk, a request near the edge of the disk may never get serviced if requests continue to join the queue.

Scan Scheduling

The **scan scheduling** algorithm attempts to satisfy the limitation of SDF scheduling. The head scans back and forth across the disk surface, processing requests as it goes. Although this method is fairer than SDF, it suffers from a different limitation, namely, that blocks near the middle tracks are processed twice as often as blocks near the edge. To see this more clearly, consider the diagram in Figure 18.24. Consider the head moving smoothly back and forth across the disk at a constant speed. The diagram shows the time at which the head crosses various tracks. Note that the middle track is crossed in both directions, at about equal intervals. Tracks near either the inside or outside track, however, are crossed twice in quick succession. Then there is a long interval in which they are not touched. A

FIGURE 18.24

Scan Scheduling Algorithm



Source: Copyright 1971 International Business Machines Corporation. Reprinted with permission from *IBM Systems Journal*, Vol. 10, No 3.

track at the very edge, inside or outside, is touched only once for every two times that a track in the middle is touched.

N-STEP C-SCAN Scheduling

Two changes improve the ***n*-step c-scan scheduling algorithm.** One is to cycle in only one direction, then return to the other end before accessing blocks again. This assures that each block is treated equally, even though a bit of time is wasted returning the head to its original position. The other change is to maintain two separate queues. Once the head has started to traverse the disk, it will read only blocks that were already waiting when the traverse started. This prevents block requests that are just ahead of the head from jumping into the queue. Instead, such a block would be placed in the alternate queue to wait for the next pass. This approach is fairer to requests that have already been waiting. Practically, there is no reason to move the head beyond the last block sought, and reversal will take place at that time. Some writers refer to this as c-look scheduling.

Figure 18.25 compares the head movement for different scheduling algorithms. These drawings, based on an example and drawings by Silberschatz et al. [SILB08], assume a disk queue containing blocks in tracks 98, 183, 37, 122, 14, 124, 65, and 67. The head starts at track 53.

18.9 NETWORK OPERATING SYSTEM SERVICES

To take advantage of networking, the operating system must include services that support networking and provide the features offered by networking capability. These services include implementation of network software protocols, augmentation of the file system to support the transfer and use of files from other locations, remote login capability, and additional utilities and tools. Modern operating systems include networking facilities as part of the base system.

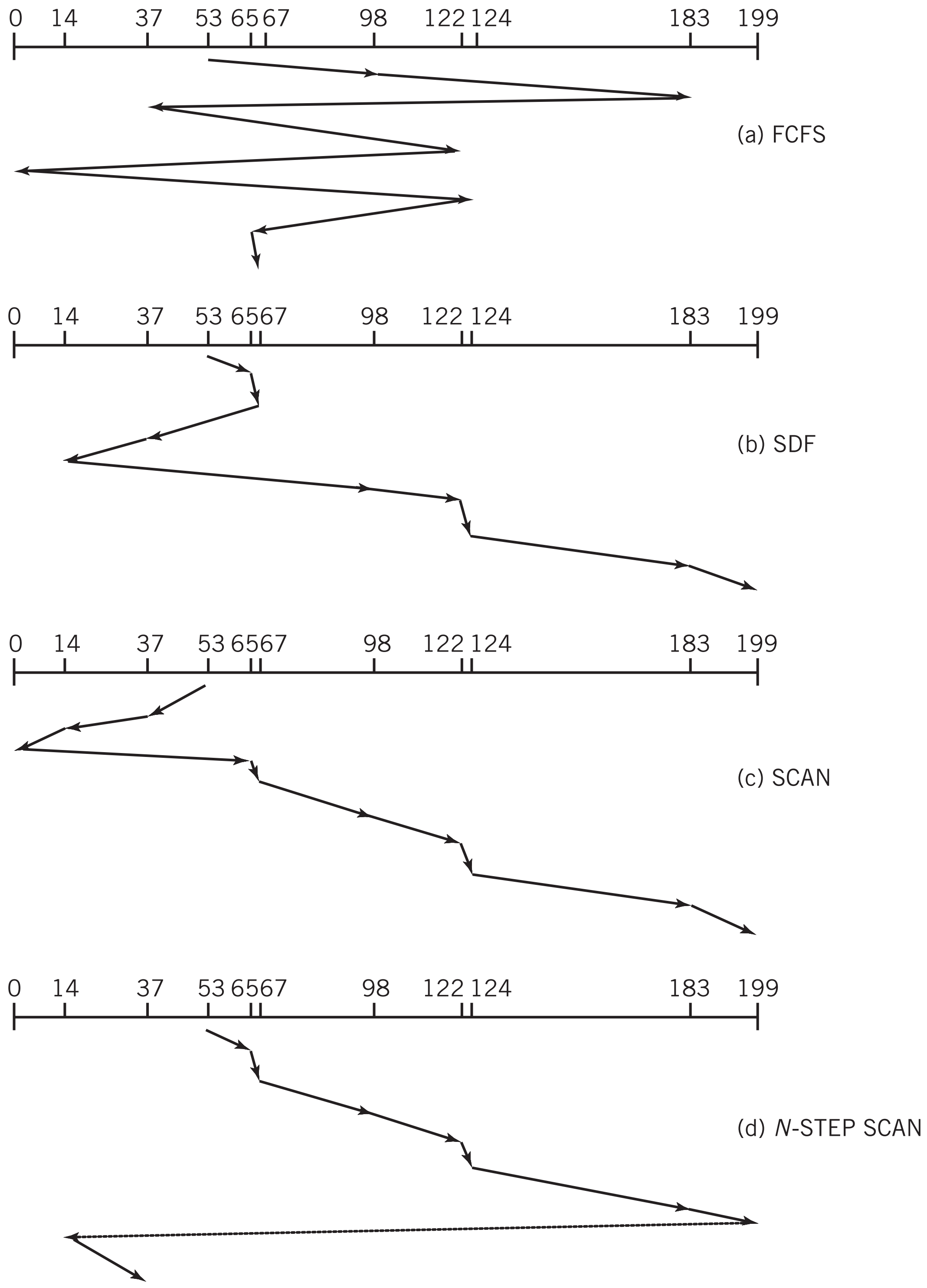
OS Protocol Support and Other Services

The operating system implements the protocols that are required for network communication and provides a variety of additional services to the user and to application programs. Most operating systems recognize and support a number of different protocols. This contributes to open system connectivity, since the network can then pass packets with less concern for the protocols available on the network stations. In addition to standard communication protocol support, the operating system commonly provides some or all of the following services:

- File services transfer programs and data files from one computer on the network to another. Network file services require that identification of the network node occur ahead of the file manager in the operating system hierarchy. This allows file requests to be directed to the appropriate file manager. Local requests are passed on to the local file manager; other requests go to the network for service by the file manager on the machine where the file resides. This concept is shown in Figure 18.26.

FIGURE 18.25

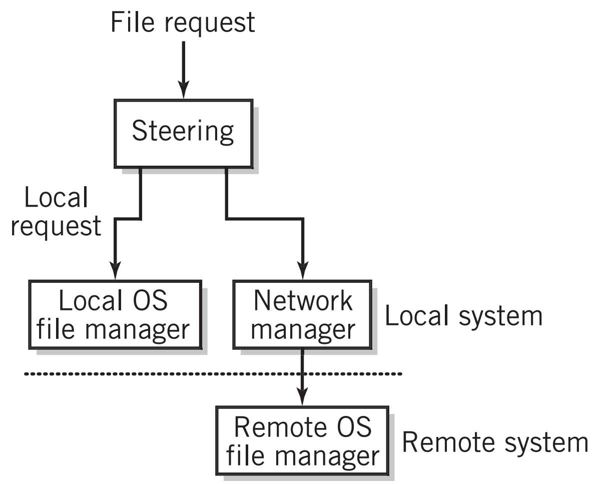
Comparison of Different Disk Algorithms



Source: A. Silberschatz, J. Peterson, P. Galvin, *Operating Systems Concepts*, 5/e. © 1998 by John Wiley & Sons, Inc. Reprinted by permission of John Wiley & Sons, Inc.

FIGURE 18.26

The Access for a Networked Operating System



- Some file services require a logical name for the machine to be included on network file requests. For example, Windows assigns pseudodrive letters to file systems accessible through the network. To the user, a file might reside on drive "M:". Although this system is simple, it has one potential shortcoming: different computers on the network might access the same drive by different letters if care isn't taken to prevent this situation. This can make it difficult for users to find their network-based files when they move between computers. Other systems allow the network administrator to assign names to each machine. Many Bentley University machines, for example, are named after Greek gods. To access a file on the "zeus" computer, the user types "zeus:" ahead of the file path name.
- Some operating systems provide transparent access to files on the network. On these systems, network files are mounted to the file system in such a way that network files simply appear in the directory structure like any other file. The operating system uses whatever method is appropriate, local or network, to access the requested files. The user need not know the actual location of the file.
- Print services work similarly to file services. Print requests are redirected by the operating system to the network station that manages the requested printer. This allows users to share expensive printer resources.
- Other peripherals and facilities can also be managed as network services. System-intensive operations, such as database services, can be processed on large computers with the capability and then passed over the network to other computers. This technique places the processing burden on the system that is most qualified to handle it and has the additional benefit of making the data available wherever it is needed.
- Web services accept requests from the network connections and return answers in the form of HTML files, image files, and more. Frequently, Web pages require data processing on the server to prepare dynamically created pages. Operating

system scripts and servers are often used for this purpose. The common gateway interface (CGI) protocol provides a standard connection between the Web server and the scripts and operating system services.

- Messaging services allow users and application programs to pass messages from one to another. The most familiar application of messaging services is e-mail and chat facilities. The network operating system not only passes these messages, it also formats them for display on different systems.
- Application program interface services allow a program to access network services. Some network operating systems also provide access to services on remote machines that might not be available locally. These services are called **remote procedure calls (RPCs)**. RPCs can be used to implement distributed computing.
- Security and network management services provide security across the network and allow users to manage and control the network from computers on the network. These services also include protection against data loss that can occur when multiple computers access data simultaneously.
- Remote processing services allow a user or application to log in to another system on the network and use its facilities for processing. Thus, the processing workload can be distributed among computers on the network, and users have access to remote computers from their own system. The most familiar services of this type are probably *telnet*, and *SSH*.

When considered together, the network services provided by a powerful network operating system transform a user's computer into a **distributed system.** Tanenbaum [TAN95] defines a distributed system as follows:

> A distributed system is a collection of independent computers that appear to the users of the system as a single computer.

Network operating systems are characterized by the distribution of control that they provide. **Client-server systems** centralize control in the server computer. Client computers have their network access limited to services provided by the server(s). Novell NetWare is an example of a client-server system. The operating system software on the server can communicate with every computer on the network, but client software communicates only with the server. In contrast, **peer-to-peer network software** permits communication between any two computers on the network, within security constraints, of course.

18.10 OTHER OPERATING SYSTEM ISSUES

There are many challenges in the design of an operating system. In this section we make a few comments about one of the more interesting operating system issues, deadlock.

Deadlock

It is not unusual for more than one process to need the same computer resource. If the resource is capable of handling multiple concurrent requests, then there is no problem. However, some resources can operate with only one process at a time. A printer is one example. If one process is printing, it is not acceptable to allow other processes access to the printer at that time.

When one process has a resource that another process needs to proceed, and the other process has a resource that the first process needs, then both are waiting for an event that can never occur, namely, the release by the other process of the needed resource. This situation can be extended to any number of processes, arranged in a circle.

This situation is called **deadlock**, and it is not unfamiliar to you in other forms. The most familiar example is the automobile gridlock situation depicted in Figure 18.27. Each vehicle is waiting for the one to its right to move, but of course no one can move.

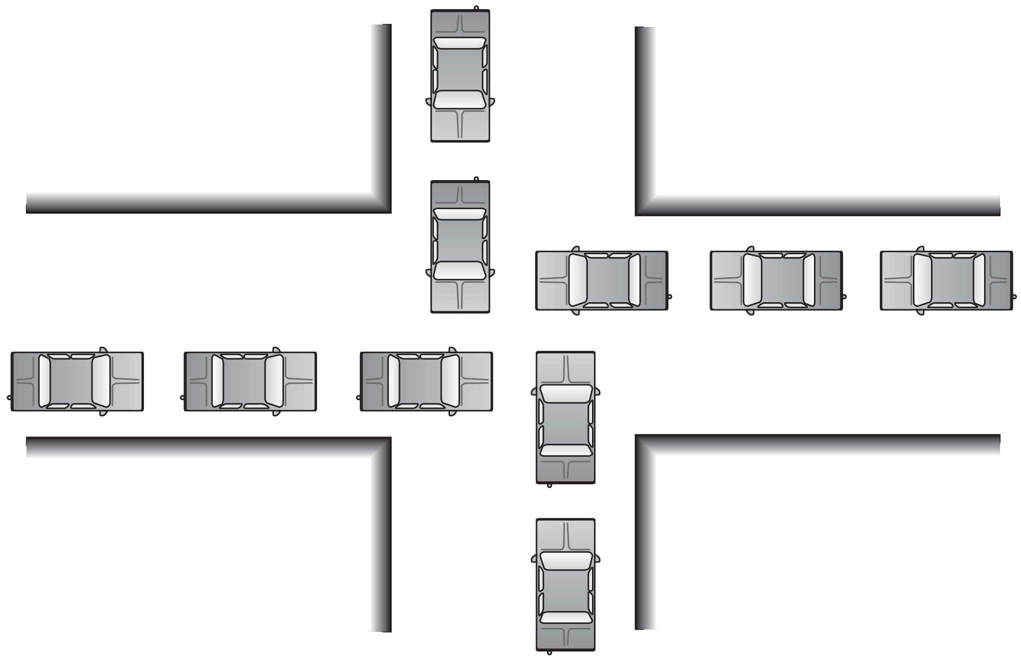
In a computer system, deadlock is a serious problem. Much theoretical study has been done on deadlock. This has resulted in three basic ways in which deadlock is managed. These are **deadlock prevention**, **deadlock avoidance**, and **deadlock detection and recovery**.

Deadlock prevention is the safest method; however, it also has the most severe effect on system performance. Deadlock prevention works by eliminating in general any condition that could create a deadlock. It is equivalent to closing one of the streets.

Deadlock avoidance provides a somewhat weaker form of protection. It works by continually monitoring the resource requirements, looking for situations in which a deadlock potential exists and then not allowing that situation to occur. If the fourth car is not allowed into the street because there are three other cars already in the intersection, that is deadlock avoidance. In a computer system, the equivalent would be a refusal by the operating system to allocate a resource because doing so would have a potential to cause deadlock.

FIGURE 18.27

A Familiar Deadlock Situation



Deadlock detection and recovery is the simplest method to implement, but the most costly when things go wrong. This methodology allows deadlocks to occur. The operating system monitors the resources. If everything stops, it assumes that a deadlock has occurred. It may take some time to notice the condition, time that is lost to productive system work. Recovery techniques include terminating processes and preempting resources. Terminated processes must be rerun. Much work could be lost and require re-creation. Deadlock recovery is generally considered the least satisfactory solution. To drivers too!

Other Issues

There are other issues that must be considered in the design of an operating system. Operating systems require a method for communication between processes. In some systems, interprocess communication may be as simple as sharing variables in a special pool or sending semaphore messages that indicate completion of a task. In others, there may be a complex message passing arrangement, with mailboxes set up for each process. Interprocess communication has increased in importance over the past few years, due to the desire to move data and program execution more easily from one application to another.

One form of communication that is sometimes very important is the ability to synchronize processes with each other. Two or more processes may be cooperating on the solution of a complex problem, and one may be dependent on the solution provided by another. Furthermore, both may be required to access the same data, the order and timing in which access takes place can be critical, and these conditions can affect the overall results. This requires a solution to the problem of **process synchronization**.

As a simple example, consider an address card file shared by you and your roommate or partner. A friend calls to tell you that she has moved and to give you her new phone number. You place a new card with this information in the card file box. Meanwhile, your roommate has taken the old card from the box and has used it to write a letter. He returns to the box, sees the new card, figures that it must be obsolete, and so throws it away and replaces it with the original. The new data is now lost. Similar situations can occur with data being shared by more than one process.

As another simple example, consider two processes, with the goal to produce the result *c*, where process 1 solves the program statement

$$\mathtt{a = a + b}$$

with initial values $a = 2$ and $b = 3$.

The second process solves the statement

$$\mathtt{c = a + 5}$$

where the value of *a* is to be taken from the first process.

Clearly, it is important that the first process complete before the value of *a* is used by process 2. If process 2 looks at the value of *a* too early, the result, *c*, will be $2 + 5 = 7$. The correct value is $5 + 5 = 10$. The solutions to the problems of interprocess communication and process synchronization are beyond the scope of this textbook. They are both difficult and interesting. Various books, such as Stallings [STAL08], Silberschatz et al. [SILB08], and Tanenbaum [TAN07], discuss these issues at length.

18.11 VIRTUAL MACHINES

From Chapter 11, you're aware that it is possible to combine the processing power of a number of computers to form a cluster that acts as a single, more powerful computer. The inverse is also true. It is possible to use a powerful computer to simulate a number of smaller computers. The process for doing so is called **virtualization**. The individual simulations that result are called **virtual machines**. Each virtual machine has its own access to the hardware resources of the host machine and an operating system that operates as a **guest** of the host machine. On a desktop or laptop computer, the user interface for each virtual machine typically appears in a separate GUI window on a display. A user can switch from one to another simply by clicking in a different window.

The use of virtualization has increased rapidly in volume and importance in recent years. There are a number of factors that account for this:

- Although computer hardware is relatively inexpensive to purchase, the overhead costs—software, networking, power consumption, space requirements, and support costs of various kinds—make the overall cost of ownership of each additional machine a significant burden.
- Modern computers generally have processing capability far in excess of usage or need.
- The development of virtualization technology has reached the state that even small computers can be virtualized easily, effectively, and securely, with complete isolation between virtual machines operating on the same host. Recent virtualization software and hardware also supports a wider range of different operating systems.

The obvious application for virtual machines is the ability to consolidate servers by operating multiple servers on the same hardware platform, but there are a number of other useful purposes as well:

- A server can be set up to create a separate virtual machine for each client. This protects the underlying system and other clients from malware and other client-generated problems.
- A system analyst can evaluate software on a virtual machine without concern for its behavior. If the software crashes or damages the operating system, the analyst can simply kill the virtual machine without damage to the underlying system or any other virtual machine that is running on the host.
- A software developer or Web developer can test his software on different operating systems, with different configurations, all running on the same host. For example, a database specialist can test changes to the database without affecting the production system and then place them into production easily and efficiently.
- A user can operate in a **sandbox**. A sandbox is a user environment in which all activity is confined to the sandbox itself. A virtual machine is a sandbox. For example, a user can access dangerous resources on the Internet for testing a system against malware safely. Malware loaded into a virtual machine disappears

when the virtual machine is closed. The sandbox is also useful for Web research, where the safety of the Web sites accessed is not assured.

Virtualization creates an important illusion. The virtualization mechanism makes it appear that each virtual machine has the computer system entirely to itself. It allocates physical resources on a shared basis, processes on different machines can communicate with each other using built-in networking protocols, and there is a common set of interrupt routines, controlled by the virtualization software. In effect, each virtual machine offers an exact duplicate of the system hardware, providing the appearance of multiple machines, each the equivalent of a separate, fully configured system. The virtual machines can execute any operating system software that is compatible with the hardware. Each virtual machine supports its own operating system, isolated both from the actual hardware and from other virtual machines. The virtual machine mechanism is invisible to the software executing on a virtual machine.

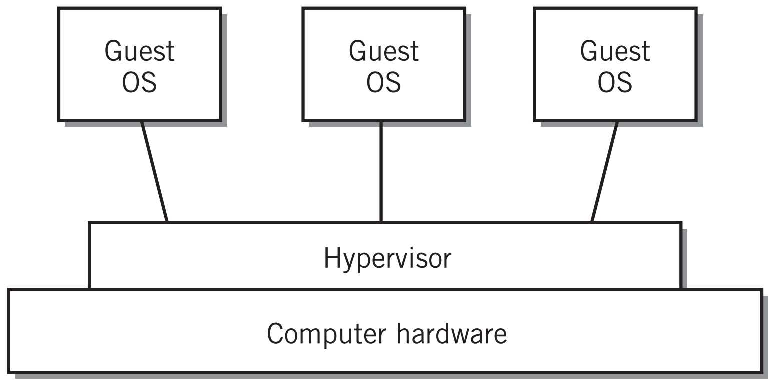
As an example, the IBM z/VM operating system simulates multiple copies of all the hardware resources of the IBM mainframe computer, registers, program counter, interrupts, I/O, and all. This allows the system to load and run one or more operating systems on top of z/VM, including even other copies of z/VM. The loaded operating systems each think that they are interacting with the hardware, but actually they are interacting with z/VM. Using virtualization, an IBM mainframe can support hundreds of virtual Linux machines simultaneously.

Figure 18.28 shows the basic design of virtualization. An additional layer called a **hypervisor** separates one or more operating systems from the hardware. The hypervisor may consist of software or a mixture of software and hardware, if the CPU provides hardware virtualization support. Most recent CPUs do so. There are two basic types of hypervisors.

- A *native*, or *type 1*, hypervisor is software that interfaces directly with the computer hardware. The hypervisor provides required software drivers, manages interrupts, and directs the results to the correct results of its work to the proper virtual machine. To the operating system or systems, the hypervisor looks like a

FIGURE 18.28

Virtual Machine Configuration



hardware interface. One way to implement a type 1 hypervisor is to use the facilities of a stripped-down operating system.

- A *hosted* or *type 2* hypervisor is software that runs as a program on a standard operating system. Some operating systems routinely provide hypervisor software as part of the standard package. Guest operating systems then run on top of the hypervisor.

SUMMARY AND REVIEW

An operating system is quite complex internally. This chapter has considered some of the more important components of the operating system in some detail. We began by looking at the critical components of a simple multitasking system, particularly scheduling and memory management.

Turning our attention to more general multitasking systems, we discussed the concepts of processes and threads. We showed you how the operating system creates and manages processes, including description of the standard process states. Threads are important in current systems, and we discussed threads as simplified processes, without the overhead.

Next, we introduced the two, and sometimes three, types of CPU scheduling. We described the difference between preemptive and nonpreemptive multitasking, described the different objectives that can be used to measure performance, and introduced several CPU dispatching algorithms, comparing the way in which these met different objectives.

The focus of memory management is to load programs in such a way as to enhance system performance. We briefly discussed the shortcomings of partitioning methods as a way to introduce virtual storage. The emphasis in this chapter is on the symbiosis between the hardware and the operating system to provide a memory management technique that addresses many of the shortcomings of other memory management techniques. The virtual storage methodology eliminates the requirement that the sum of programs to be loaded as a whole must fit all at once into available memory; instead, the active parts of each program are sufficient. It allows each program to exist in the same virtual memory space. It allows programs to be loaded anywhere in memory, and noncontiguously. And it eliminates the need for relocation procedures.

We explained the page fault procedure and discussed several page replacement algorithms. We considered the number of pages that are required to execute a program successfully and efficiently, and we considered the problem of thrashing.

Next, we discussed the algorithms used for secondary storage. Following that, we presented the operating system components that support networking. We next introduced briefly the issues of deadlock, process synchronization, and interprocess communication. These issues are representative of some of the more complex problems that must be faced by operating system designers and administrators. Finally, we introduced the concept of a virtual machine. We explained why virtualization is so important, explained how it's used, and showed how it works. The VM operating system provides virtual machines that can be treated as independent machines, each with its own operating system, applications, and users.

FOR FURTHER READING

Any of the references mentioned in the For Further Reading section of Chapter 15 also address the topics in this chapter. If you have become intrigued by operating systems and would like to know more, there are a large number of interesting problems and algorithms with intriguing names like "the dining philosophers problem." We have only barely touched upon the surface of operating system design and operation, especially in the areas of deadlock, process synchronization, and interprocess communication. We highly recommend the textbooks by Deitel [DEIT03], Tanenbaum [TAN07], Silberschatz, et al. [SILB08], and Stallings [STAL08] for thorough and detailed treatment of these and other topics. Information about network and distributed operating systems can be found in Tanenbaum [TAN/WOOD06, TAN/VANS06] and in most recent operating systems and networking texts. See the For Further Reading section in Chapter 13 for additional references.

Virtualization is currently a hot topic. Much information, including readable introductions, can be found at the vmware.com, xen.org, and sun.com websites. There are also numerous books and magazine articles on virtualization. Some of these are listed in the references at the back of this book.

KEY CONCEPTS AND TERMS

associative memory
backing store
Belady's anomaly
best-fit algorithm
blocked state
blocking
child process
client-server system
clock page replacement algorithm
cloning
concept of locality
cooperating processes
deadlock
deadlock avoidance
deadlock detection and recovery
deadlock prevention
demand paging
dirty bit
dispatcher
dispatching
distributed system
dynamic address translation (DAT)
dynamic priority scheduling
event
event-driven program
external fragmentation
first-come, first-served (FCFS) disk scheduling
first-fit algorithm
first-in, first-out (FIFO)
fixed partitioning
forking
fragmentation
frame (memory)
global page replacement
guest
high-level (long-term) scheduler
hit
hypervisor
indefinite postponement
independent processes
internal fragmentation
job
job steps
largest-fit algorithm
least recently used (LRU) page replacement algorithm
local page replacement
locked frame
miss
multilevel feedback queue algorithm
n-step c-scan scheduling algorithm
nonpreemptive systems
not used recently (NUR) page replacement algorithm
offset
page fault (trap)
page swapping
page replacement algorithm
page table

paging
parent process
peer-to-peer network software
preemptive systems
prepaging
priority scheduling
process
process control block (PCB)
process identifier (PID)
process state
process synchronization
ready state
remote procedure call (RPC)
response time
resumption
roll-out, roll-in
round robin
running state
sandbox
scan disk scheduling
segmentation
segment table
shortest job first (SJF)
shortest distance first (SDF) disk scheduling
spawning
starvation
suspended state
swap file
swap out
swap space
swapping
system process
task
thrashing
thread
time-out
translation lookaside buffer (TLB)
turnaround time
user-level thread
user process
variable partitioning
virtualization
virtual machine (VM)
virtual memory
virtual storage
wake-up
working set
worst-fit algorithm

READING REVIEW QUESTIONS

18.1 What is the fundamental purpose of any operating system? What is the role of the file manager? What other basic functions must the operating system be able to perform?

18.2 Where is the first stage of a bootstrap loader for a computer stored? What tasks does it perform?

18.3 What are the major items found in a *process control block*?

18.4 How does a process differ from a program?

18.5 What are user processes? What are system processes?

18.6 Explain the purpose of a *spawning* operation. What is the result when the spawning operation is complete?

18.7 Draw and label the process state diagram used for dispatching work to the CPU. Explain each state and the role of each connector.

18.8 What features characterize threads? How are threads used?

18.9 What is an *event-driven* program?

18.10 What are the potential difficulties that can occur when nonpreemptive dispatching is used in an interactive system?

18.11 Explain the *first-in-first-out* dispatch algorithm. Discuss the advantages and disadvantages of this algorithm. Is this a preemptive or nonpreemptive algorithm?

18.12 Explain how the *shortest job first* algorithm can result in starvation.

18.13 UNIX systems use a dynamic priority algorithm where the priority is based on the ratio of CPU time to the total time a process has been in the system. Explain how this reduces to *round robin* in the absence of any I/O.

18.14 What is the basic problem that memory management is supposed to solve? What is the shortcoming of memory partitioning as a solution?

18.15 What is a *page* in virtual storage? What is the relationship between a program and pages?

18.16 What is a *frame* in virtual storage? What is the relationship between a frame and physical memory?

18.17 What are the contents of a page table? Explain how a page table relates pages and frames.

18.18 Explain how page translation makes it possible to execute a program that is stored in memory noncontiguously.

18.19 A program's page table is shown in Figure 18Q.1. Assume that each page is 4 KB in size. (4 KB = 12 bits). The instruction currently being executed is to load data from location $5E24_{16}$. Where is the data located in physical memory?

18.20 Virtual storage makes it possible to execute a program that is larger than the available amount of memory. What obvious characteristic of program code makes this possible?

18.21 Describe the process that takes place when a *page fault* occurs? What happens if there are no frames available when a page fault occurs?

18.22 Explain the concept of a *working set*.

18.23 The *not used recently* page replacement algorithm stores two bits with each page to determine a page that is suitable for replacement. What does each bit represent? Which combination of bits makes a page the most desirable for replacement? Justify your answer. What combination would be second best?

18.24 Explain *thrashing*.

18.25 Describe at least three network services offered by most operating systems in addition to protocol services.

18.26 Explain *deadlock*. What are the three possible ways that an operating system can handle the issue of deadlock?

FIGURE 18Q.1

Page	Frame
0	2A
1	2B
2	5
3	17
4	18
5	2E
6	1F

18.27 State at least three advantages that result from the use of virtual machines.

18.28 Describe the tasks that are performed by a *hypervisor*.

EXERCISES

18.1 Describe, in step-by-step form, the procedure that the operating system would use to switch from one user to another in a multiuser time sharing system.

18.2 What values would you expect to find in the process state entry in a process control block? What purpose is served by the program counter and register save areas in a process control block? (Note that the program counter entry in a PCB is not the same as the program counter!)

18.3 Discuss the steps that take place when a process is moved (a) from ready state to running state, (b) from running state to blocked state, (c) from running state to ready state, and (d) from blocked state to ready state.

18.4 Why is there no path on the process diagram from blocked state to running state?

18.5 Describe what occurs when a user types a keystroke on a terminal connected to a multitasking system. Does the system respond differently for a preemptive or nonpreemptive system? Why or why not? If the response is different, *how* is it different?

18.6 The multilevel feedback queue scheduling method looks like FIFO at the upper levels and like round robin at the lowest level, yet it frequently behaves better than either in terms of the performance objectives mentioned in the text. Why is this so?

18.7 Discuss the shortest-job-first scheduling method in terms of the various objectives given in the text.

18.8 What is the risk that can result from the mixed nonpreemptive-preemptive scheduling system taken by Linux, as discussed in the text?

18.9 A VSOS (very simple operating system) uses a very simple approach to scheduling. Scheduling is done on a straight round-robin basis, where each job is given a time quantum sufficient to complete very short jobs. Upon completion by a job, another job is admitted to the system and immediately given one quantum. Thereafter, it enters the round-robin queue. Consider the scheduling objectives given in the text. Discuss the VSOS scheduling approach in terms of these objectives.

18.10 Earlier versions of Windows used an essentially nonpreemptive dispatching technique that Microsoft called "cooperative multitasking." In cooperative multitasking, each program was expected to voluntarily give up the CPU periodically to give other processes a chance to execute. Discuss. What potential difficulties can this method cause?

18.11 In the memory management schemes used in earlier operating systems, it was necessary to modify the addresses of most programs when they were loaded into memory because they were usually not loaded into memory starting at location 0. The OS program loader was assigned this task, which was called *program relocation*. Why is program relocation unnecessary when virtual storage is used for memory management?

18.12 Discuss the impact of virtual storage on the design of an operating system. Consider the tasks that must be performed, the various methods of performing those tasks, and the resulting effect on system performance.

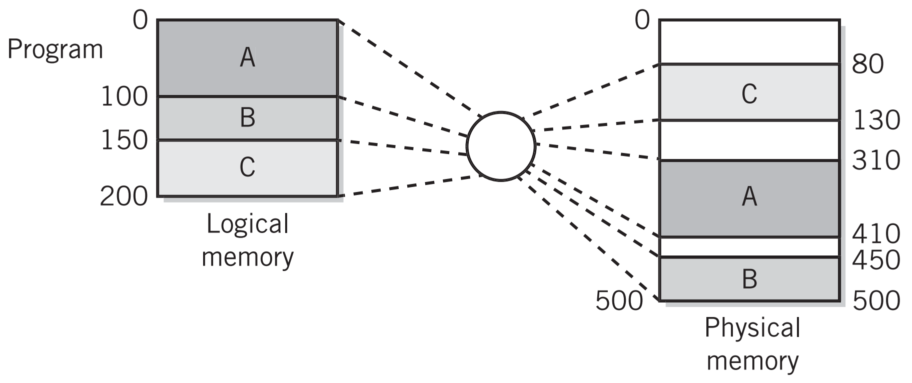
18.13 There are a number of different factors, both hardware and OS software, that affect the operating speed of a virtual storage system. Explain carefully each of the factors and its resulting impact on system performance.

18.14 Show in a drawing similar to Figure 18.18 how two different programs with the same logical address space can be transformed by virtual storage into independent parts of physical memory.

18.15 Show in a drawing similar to Figure 18.18 how two different programs with the same logical address space can be transformed by virtual storage partially into the same part of physical memory and partially into independent parts of physical memory. Assume that the two programs use the same program code, located from logical addresses 0 to 100, and that they each have their own data region, located from logical addresses 101 to 165.

18.16 Create a page table that meets the translation requirements of Figure 18.E1. Assume a page size of 10.

FIGURE 18.E1



18.17 Explain why the installation of additional physical memory in a virtual memory system often results in substantial improvement in overall system performance.

18.18 Develop an example that explains thrashing clearly.

18.19 What kind of fragmentation would you find in virtual storage? Is this a serious problem? Justify your answer. Discuss the relationship between fragmentation and page size.

18.20 Explain why page sharing can reduce the number of page faults that occur in a virtual storage system.

18.21 The manual for a popular operating system points out that the number of concurrent users on the system can be increased if the users are sharing programs, such as editors, mail readers, or compilers. What characteristics of virtual storage make this possible?

18.22 Explain deadlocking.

18.23 The CPU scheduling algorithm (in UNIX) is a simple priority algorithm. The priority for a process is computed as the ratio of the CPU time actually used by the process to the real time that has passed. The lower the figure, the higher the priority. Priorities are recalculated every tenth of a second.

- **a.** What kind of jobs are favored by this type of algorithm?
- **b.** If there is no I/O being performed, this algorithm reduces to a round-robin algorithm. Explain.
- **c.** Discuss this algorithm in terms of the scheduling objectives given in the text.

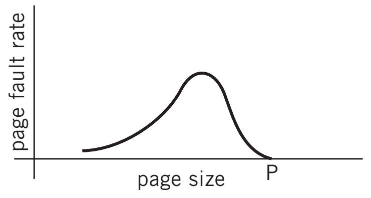
18.24 Explain the working set concept. What is the relationship between the working set concept and the principle of locality?

18.25 Why is the working set concept much more effective if it is implemented dynamically, that is, recalculated while a process is executing?

18.26 What are the differences, trade-offs, advantages, and disadvantages between an OS that implements deadlock prevention versus deadlock avoidance versus deadlock detection and recovery?

18.27 Figure 18.E2 shows that, for a given process, the page fault rate in a virtual storage system increases as the page size is increased and then decreases to 0 as the page size approaches *P*, the size of the process. Explain the various parts of the curve.

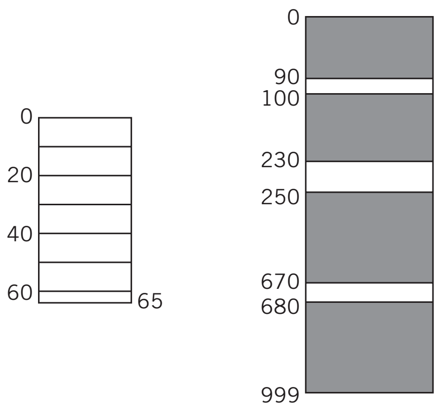
FIGURE 18.E2



18.28 Assume that you have a program to run on a Little Man-type computer that provides virtual storage paging. Each page holds ten locations (in other words, one digit). The system can support up to one hundred pages of memory. As Figure 18.E3 shows, your program is sixty-five instructions long. The available frames in physical memory are also shown in the diagram. All blocked-in areas are already occupied by other programs that are sharing the use of the Little Man.

- **a.** Create a starting page table for your program. Assume that your program will start executing at its location 0.
- **b.** Suppose a page fault occurs in your program. The OS has to decide whether to swap out one of your older pages, or one of somebody else's pages. Which strategy is less likely to cause thrashing? Why?

FIGURE 18.E3



18.29 What is a real-time system? Discuss the impact of a real-time system on the design of the operating systems, paying particular note to the various components and algorithms to be used.

18.30 Consider the operation of a jukebox. Each table has a jukebox terminal where customers can feed coins to play songs (50 cents apiece, three for a dollar). Prior to the iPod era, the queue to hear your songs in a busy restaurant could be quite long, sometimes longer than the average dining time, in fact.

Discuss the various disk scheduling algorithms as methods of selecting the order in which to play the requested songs. Be sure to consider the advantages and disadvantages of each method in terms of fairness, probability that each diner will get to hear their songs, ease of implementation, and any other important issues that you feel should be considered. You might note that multiple diners would sometimes request the same song.

18.31 Tanenbaum notes that the problem of scheduling an elevator in a tall building is similar to that of scheduling a disk arm. Requests come in continuously, calling the elevator to floors at random. One difference is that once inside, riders request that the elevator move to a different floor. Discuss the various disk scheduling algorithms as options for scheduling the elevator in terms of fairness, service, and ease of implementation.

18.32 Discuss possible tape scheduling algorithms for a tape controller. Assume that files are stored contiguously on tape. What effect would noncontiguous, linked files have on your algorithm?

18.33 You may have noticed a number of similarities between virtual storage paging and cache memory paging. One major difference, of course, is that main memory is much faster than disk access.

Consider the applicability and performance of the various paging algorithms in a memory caching system, and discuss the advantages and disadvantages of each.

18.34 The designer of a new operating system to be used especially for real-time applications has proposed the use of virtual storage memory management so that the system can handle programs too large to fit in the limited memory space sometimes provided on real-time systems. What are the implications of this decision in terms of the way that virtual storage works?

18.35 Discuss the various trade-offs and decisions involved in task dispatching and the options and methods used for implementing those trade-offs and decisions.

18.36 A system status report for a virtual storage operating system shows that between 2 p.m. and 4 p.m. CPU usage and I/O usage both climbed steadily. At 4 p.m., the I/O usage reached 100 percent, but continued to increase. After 4 p.m., the CPU usage, however, dropped off dramatically. What is the explanation for this behavior?

18.37 Discuss the network features and services provided in an operating system. Which services are mandatory? Why?

18.38 Explain the bootstrap procedure for a diskless workstation.

18.39 Consider the operation of an OS dispatcher on a computer with multiple cores operating under symmetric multiprocessing. Assuming that there are more processes being executed than there are cores, the dispatcher is responsible for maximizing the work load by keeping every core as busy as possible. In addition to the usual dispatch criteria and algorithms, there are two options for selecting in which core a process is to execute. The first option is to allow a process to execute in any core that is available each time it is selected to run; with this option, a process might execute in several different cores during its run. The second option is to require that the process run in the same core each time it is selected.

a. What are the advantages of the first option?

b. What are the advantages of the second option? (Hint: consider the interaction between a process and the cache memory that is assigned to each core.)

18.40 [courtesy of W. Wong] Assume that a program is to be executed on a computer with virtual storage. The machine supports 10,000 words of logical memory overall, broken into pages of 100 words each. This particular machine contains 400 physical memory locations. Suppose that the machine starts to execute a program. The page table is initially empty, and is filled as necessary. Suppose that the program references the following sequence of memory locations:

start `951, 952, 4730, 955, 2217, 3663, 2217, 4785, 957, 2401,`
`959, 2496, 3510, 962` ***end***

a. Indicate each of the points where page faults occur.

b. Show the page table at the end of the sequence for each of the following demand page replacement algorithms:

i. first in first out

ii. least recently used

iii. least frequently used (using FIFO as a tie breaker)