OPERATING SYSTEM CONCEPTS

Abraham Silberschatz
Peter Baer Galvin
Greg Gagne

Ninth Edition

is increasing. Mobile operating systems often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—Apple's iOS and Google's Android—features a core kernel along with middleware that supports databases, multimedia, and graphics (to name a only few).

1.2 Computer-System Organization

Before we can explore the details of how computer systems operate, we need general knowledge of the structure of a computer system. In this section, we look at several parts of this structure. The section is mostly concerned with computer-system organization, so you can skim or skip it if you already understand the concepts.

1.2.1 Computer-System Operation

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored within the computer hardware in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware**. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish

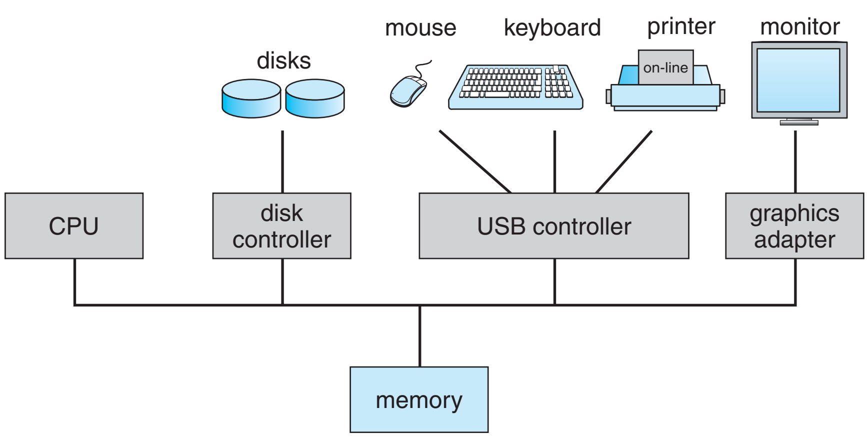


Figure 1.2 A modern computer system.

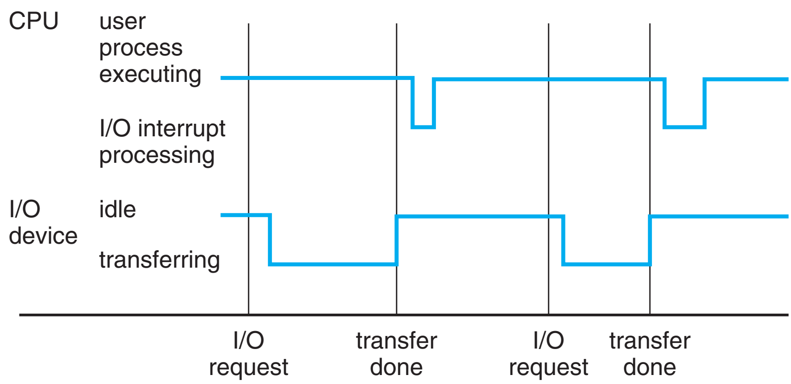


Figure 1.3 Interrupt timeline for a single process doing output.

this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running. On UNIX, the first system process is "init," and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**).

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A timeline of this operation is shown in Figure 1.3.

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn, would call the interrupt-specific handler. However, interrupts must be handled quickly. Since only a predefined number of interrupts is possible, a table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector**, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for

STORAGE DEFINITIONS AND NOTATION

The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A **kilobyte**, or **KB**, is 1,024 bytes; a **megabyte**, or **MB**, is $1{,}024^2$ bytes; a **gigabyte**, or **GB**, is $1{,}024^3$ bytes; a **terabyte**, or **TB**, is $1{,}024^4$ bytes; and a **petabyte**, or **PB**, is $1{,}024^5$ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

the interrupting device. Operating systems as different as Windows and UNIX dispatch interrupts in this manner.

The interrupt architecture must also save the address of the interrupted instruction. Many old designs simply stored the interrupt address in a fixed location or in a location indexed by the device number. More recent architectures store the return address on the system stack. If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

1.2.2 Storage Structure

The CPU can load instructions only from memory, so any programs to run must be stored there. General-purpose computers run most of their programs from rewritable memory, called main memory (also called **random-access memory**, or **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory** (**DRAM**).

Computers use other forms of memory as well. We have already mentioned read-only memory, ROM) and electrically erasable programmable read-only memory, EEPROM). Because ROM cannot be changed, only static programs, such as the bootstrap program described earlier, are stored there. The immutability of ROM is of use in game cartridges. EEPROM can be changed but cannot be changed frequently and so contains mostly static programs. For example, smartphones have EEPROM to store their factory-installed programs.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of `load` or `store` instructions to specific memory addresses. The `load` instruction moves a byte or word from main memory to an internal register within the CPU, whereas the `store` instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution.

A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses. It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory is a **volatile** storage device that loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data. Most programs (system and application) are stored on a disk until they are loaded into memory. Many programs then use the disk as both the source and the destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system, as we discuss in Chapter 10.

In a larger sense, however, the storage structure that we have described—consisting of registers, main memory, and magnetic disks—is only one of many possible storage systems. Others include cache memory, CD-ROM, magnetic tapes, and so on. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, cost, size, and volatility.

The wide variety of storage systems can be organized in a hierarchy (Figure 1.4) according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases. This trade-off is reasonable; if a given storage system were both faster and less expensive than another—other properties being the same—then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper

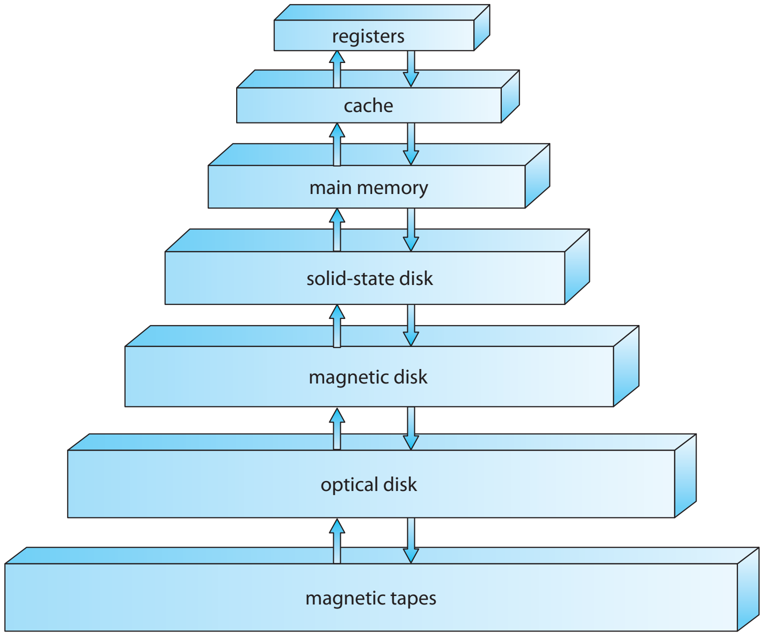


Figure 1.4 Storage-device hierarchy.

tape and core memories, are relegated to museums now that magnetic tape and **semiconductor memory** have become faster and cheaper. The top four levels of memory in Figure 1.4 may be constructed using semiconductor memory.

In addition to differing in speed and cost, the various storage systems are either volatile or nonvolatile. As mentioned earlier, **volatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping. In the hierarchy shown in Figure 1.4, the storage systems above the solid-state disk are volatile, whereas those including the solid-state disk and below are nonvolatile.

Solid-state disks have several variants but in general are faster than magnetic disks and are nonvolatile. One type of solid-state disk stores data in a large DRAM array during normal operation but also contains a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, this solid-state disk's controller copies the data from RAM to the magnetic disk. When external power is restored, the controller copies the data back into RAM. Another form of solid-state disk is flash memory, which is popular in cameras and **personal digital assistants** (**PDAs**), in robots, and increasingly for storage on general-purpose computers. Flash memory is slower than DRAM but needs no power to retain its contents. Another form of nonvolatile storage is **NVRAM**, which is DRAM with battery backup power. This memory can be as fast as DRAM and (as long as the battery lasts) is nonvolatile.

The design of a complete memory system must balance all the factors just discussed: it must use only as much expensive memory as necessary while providing as much inexpensive, nonvolatile memory as possible. Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

1.2.3 I/O Structure

Storage is only one of many types of I/O devices within a computer. A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices. Next, we provide an overview of I/O.

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, more than one device may be attached. For instance, seven or more devices can be attached to the **small computer-systems interface (SCSI)** controller. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. Typically, operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.

To start an I/O operation, the device driver loads the appropriate registers within the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard"). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation. The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information.

This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, **direct memory access (DMA)** is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective. Figure 1.5 shows the interplay of all components of a computer system.

1.3 Computer-System Architecture

In Section 1.2, we introduced the general structure of a typical computer system. A computer system can be organized in a number of different ways, which we

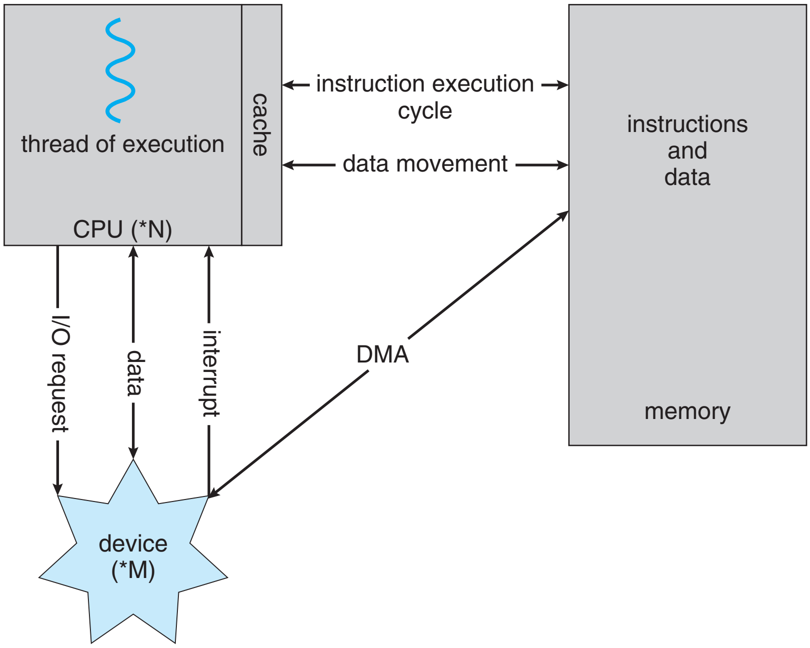


Figure 1.5 How a modern computer system works.

can categorize roughly according to the number of general-purpose processors used.

1.3.1 Single-Processor Systems

Until recently, most computer systems used a single processor. On a single-processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes. Almost all single-processor systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers; or, on mainframes, they may come in the form of more general-purpose processors, such as I/O processors that move data rapidly among the components of the system.

All of these special-purpose processors run a limited instruction set and do not run user processes. Sometimes, they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status. For example, a disk-controller microprocessor receives a sequence of requests from the main CPU and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU, then the system is a single-processor system.

1.3.2 Multiprocessor Systems

Within the past several years, **multiprocessor systems** (also known as **parallel systems** or **multicore systems**) have begun to dominate the landscape of computing. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices. Multiprocessor systems first appeared prominently appeared in servers and have since migrated to desktop and laptop systems. Recently, multiple processors have appeared on mobile devices such as smartphones and tablet computers.

Multiprocessor systems have three main advantages:

1. **Increased throughput**. By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with N processors is not N, however; rather, it is less than N. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors. Similarly, N programmers working closely together do not produce N times the amount of work a single programmer would produce.

2. **Economy of scale**. Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.

3. **Increased reliability**. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

Increased reliability of a computer system is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected. The HP NonStop (formerly Tandem) system uses both hardware and software duplication to ensure continued operation despite faults. The system consists of multiple pairs of CPUs, working in lockstep. Both processors in the pair execute each instruction and compare the results. If the results differ, then one CPU of the pair is at fault, and both are halted. The process that was being executed is then moved to another pair of CPUs, and the instruction that failed is restarted. This solution is expensive, since it involves special hardware and considerable hardware duplication.

The multiple-processor systems in use today are of two types. Some systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A ***boss*** processor controls the system; the other processors either look to the boss for instruction or have predefined tasks. This scheme defines a boss–worker relationship. The boss processor schedules and allocates work to the worker processors.

The most common systems use **symmetric multiprocessing** (**SMP**), in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no boss–worker relationship exists between processors. Figure 1.6 illustrates a typical SMP architecture. Notice that each processor has its own set of registers, as well as a private—or local—cache. However, all processors share physical memory. An example of an SMP system is AIX, a commercial version of UNIX designed by IBM. An AIX system can be configured to employ dozens of processors. The benefit of this model is that many processes can run simultaneously—N processes can run if there are N CPUs—without causing performance to deteriorate significantly. However, we must carefully control I/O to ensure that the data reach the appropriate processor. Also, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow processes and resources—such as memory—to be shared dynamically among the various processors and can lower the variance among the processors. Such a system must be written carefully, as we shall see in Chapter 5. Virtually all modern operating systems—including Windows, Mac OS X, and Linux—now provide support for SMP.

The difference between symmetric and asymmetric multiprocessing may result from either hardware or software. Special hardware can differentiate the multiple processors, or the software can be written to allow only one boss and multiple workers. For instance, Sun Microsystems' operating system SunOS Version 4 provided asymmetric multiprocessing, whereas Version 5 (Solaris) is symmetric on the same hardware.

Multiprocessing adds CPUs to increase computing power. If the CPU has an integrated memory controller, then adding CPUs can also increase the amount

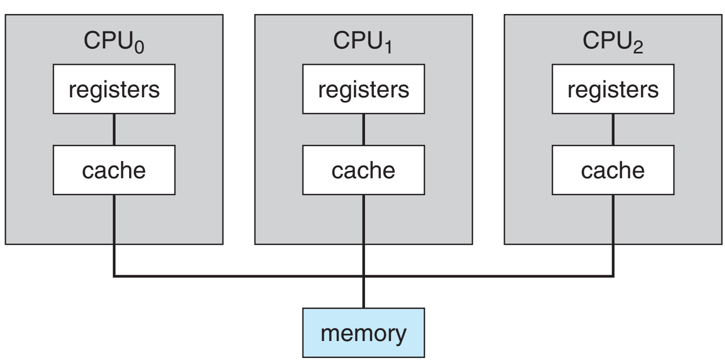


Figure 1.6 Symmetric multiprocessing architecture.

of memory addressable in the system. Either way, multiprocessing can cause a system to change its memory access model from uniform memory access (**UMA**) to non-uniform memory access (**NUMA**). UMA is defined as the situation in which access to any RAM from any CPU takes the same amount of time. With NUMA, some parts of memory may take longer to access than other parts, creating a performance penalty. Operating systems can minimize the NUMA penalty through resource management, as discussed in Section 9.5.4.

A recent trend in CPU design is to include multiple computing **cores** on a single chip. Such multiprocessor systems are termed **multicore**. They can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips.

It is important to note that while multicore systems are multiprocessor systems, not all multiprocessor systems are multicore, as we shall see in Section 1.3.3. In our coverage of multiprocessor systems throughout this text, unless we state otherwise, we generally use the more contemporary term ***multicore***, which excludes some multiprocessor systems.

In Figure 1.7, we show a dual-core design with two cores on the same chip. In this design, each core has its own register set as well as its own local cache. Other designs might use a shared cache or a combination of local and shared caches. Aside from architectural considerations, such as cache, memory, and bus contention, these multicore CPUs appear to the operating system as N standard processors. This characteristic puts pressure on operating system designers—and application programmers—to make use of those processing cores.

Finally, **blade servers** are a relatively recent development in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

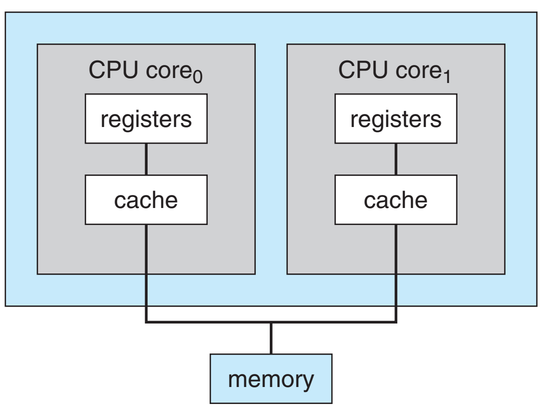


Figure 1.7 A dual-core design with two cores placed on the same chip.

1.3.3 Clustered Systems

Another type of multiprocessor system is a **clustered system**, which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems described in Section 1.3.2 in that they are composed of two or more individual systems—or nodes—joined together. Such systems are considered **loosely coupled**. Each node may be a single processor system or a multicore system. We should note that the definition of ***clustered*** is not concrete; many commercial packages wrestle to define a clustered system and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via a local-area network LAN (as described in Chapter 17) or a faster interconnect, such as InfiniBand.

Clustering is usually used to provide **high-availability** service—that is, service will continue even if one or more systems in the cluster fail. Generally, we obtain high availability by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

Clustering can be structured asymmetrically or symmetrically. In **asymmetric clustering**, one machine is in **hot-standby mode** while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server. In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware. However it does require that more than one application be available to run.

Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide **high-performance computing** environments. Such systems can supply significantly greater computational power than single-processor or even SMP systems because they can run an application concurrently on all computers in the cluster. The application must have been written specifically to take advantage of the cluster, however. This involves a technique known as **parallelization**, which divides a program into separate components that run in parallel on individual computers in the cluster. Typically, these applications are designed so that once each computing node in the cluster has solved its portion of the problem, the results from all the nodes are combined into a final solution.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN) (as described in Chapter 17). Parallel clusters allow multiple hosts to access the same data on shared storage. Because most operating systems lack support for simultaneous data access by multiple hosts, parallel clusters usually require the use of special versions of software and special releases of applications. For example, Oracle Real Application Cluster is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access, the system must also supply access control and locking to

BEOWULF CLUSTERS

Beowulf clusters are designed to solve high-performance computing tasks. A Beowulf cluster consists of commodity hardware—such as personal computers—connected via a simple local-area network. No single specific software package is required to construct a cluster. Rather, the nodes use a set of open-source software libraries to communicate with one another. Thus, there are a variety of approaches to constructing a Beowulf cluster. Typically, though, Beowulf computing nodes run the Linux operating system. Since Beowulf clusters require no special hardware and operate using open-source software that is available free, they offer a low-cost strategy for building a high-performance computing cluster. In fact, some Beowulf clusters built from discarded personal computers are using hundreds of nodes to solve computationally expensive scientific computing problems.

ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager** (**DLM**), is included in some cluster technology.

Cluster technology is changing rapidly. Some cluster products support dozens of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks** (**SANs**), as described in Section 10.3.3, which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability. Figure 1.8 depicts the general structure of a clustered system.

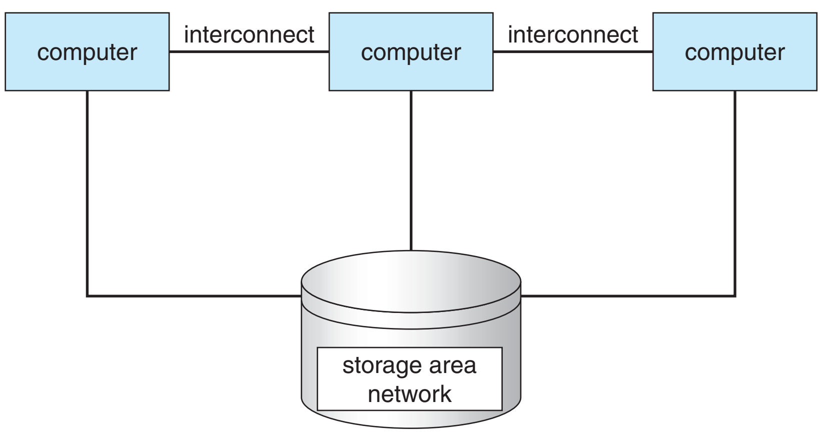


Figure 1.8 General structure of a clustered system.

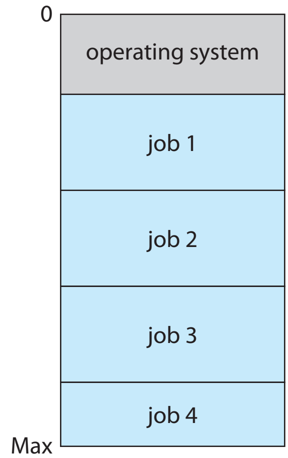


Figure 1.9 Memory layout for a multiprogramming system.

1.4 Operating-System Structure

Now that we have discussed basic computer-system organization and architecture, we are ready to talk about operating systems. An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. There are, however, many commonalities, which we consider in this section.

One of the most important aspects of operating systems is the ability to multiprogram. A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. **Multiprogramming** increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.9). Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory.

The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job. When ***that*** job needs to wait, the CPU switches to ***another*** job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If he has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system. **Time sharing** (or **multitasking**) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an **interactive** computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is called a **process**. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device. Since interactive I/O typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.

Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision involves **job scheduling**, which we discuss in Chapter 6. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management, which we cover in Chapters 8 and 9. In addition, if several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU scheduling**, which is also discussed in Chapter 6. Finally, running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. We discuss these considerations throughout the text.

In a time-sharing system, the operating system must ensure reasonable response time. This goal is sometimes accomplished through **swapping**, whereby processes are swapped in and out of main memory to the disk. A more common method for ensuring reasonable response time is **virtual memory**, a technique that allows the execution of a process that is not completely in memory (Chapter 9). The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

A time-sharing system must also provide a file system (Chapters 11 and 12). The file system resides on a collection of disks; hence, disk management must be provided (Chapter 10). In addition, a time-sharing system provides a mechanism for protecting resources from inappropriate use (Chapter 14). To ensure orderly execution, the system must provide mechanisms for job synchronization and communication (Chapter 5), and it may ensure that jobs do not get stuck in a deadlock, forever waiting for one another (Chapter 7).

1.5 Operating-System Operations

As mentioned earlier, modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided to deal with the interrupt.

Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program running. With sharing, many processes could be adversely affected by a bug in one program. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself.

Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

1.5.1 Dual-Mode and Multimode Operation

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user-defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution.

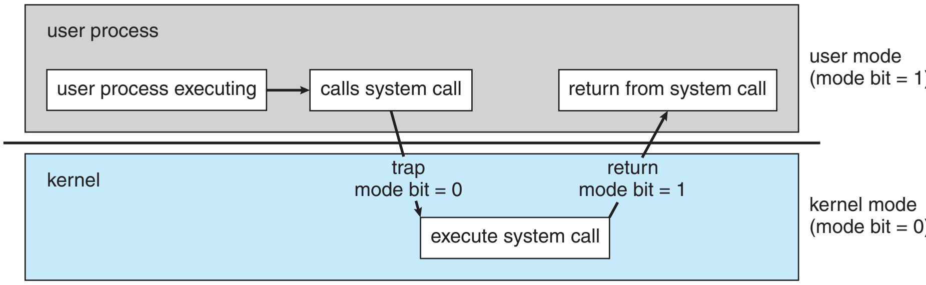


Figure 1.10 Transition from user to kernel mode.

At the very least, we need two separate ***modes*** of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.10. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. As we shall see throughout the text, there are many additional privileged instructions.

The concept of modes can be extended beyond two modes (in which case the CPU uses more than one bit to set and test the mode). CPUs that support virtualization (Section 16.1) frequently have a separate mode to indicate when the **virtual machine manager (VMM)**—and the virtualization management software—is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel. It needs that level of privilege so it can create and manage virtual machines, changing the CPU state to do so. Sometimes, too, different modes are used by various kernel components. We should note that, as an alternative to modes, the CPU designer may use other methods to differentiate operational privileges. The Intel 64 family of CPUs supports four ***privilege levels***, for example, and supports virtualization but does not have a separate mode for virtualization.

We can now see the life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic `trap` instruction, although some systems (such as MIPS) have a specific `syscall` instruction to invoke a system call.

When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call. We describe system calls more fully in Section 2.3.

The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system. For instance, MS-DOS was written for the Intel 8088 architecture, which has no mode bit and therefore no dual mode. A user program running awry can wipe out the operating system by writing over it with data; and multiple programs are able to write to a device at the same time, with potentially disastrous results. Modern versions of the Intel CPU do provide dual-mode operation. Accordingly, most contemporary operating systems—such as Microsoft Windows 7, as well as Unix and Linux—take advantage of this dual-mode feature and provide greater protection for the operating system.

Once hardware protection is in place, it detects errors that violate modes. These errors are normally handled by the operating system. If a user program fails in some way—such as by making an attempt either to execute an illegal instruction or to access memory that is not in the user's address space—then the hardware traps to the operating system. The trap transfers control through the interrupt vector to the operating system, just as an interrupt does. When a program error occurs, the operating system must terminate the program abnormally. This situation is handled by the same code as a user-requested abnormal termination. An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it and perhaps correct it and restart the program.

1.5.2 Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

We can use the timer to prevent a user program from running too long. A simple technique is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for example, would have its counter initialized to 420. Every second, the timer interrupts, and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program. When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

1.6 Process Management

A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A time-shared user program such as a compiler is a process. A word-processing program being run by an individual user on a PC is a process. A system task, such as sending output to a printer, can also be a process (or at least part of one). For now, you can consider a process to be a job or a time-shared program, but later you will learn that the concept is more general. As we shall see in Chapter 3, it is possible to provide system calls that allow processes to create subprocesses to execute concurrently.

A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are either given to the process when it is created or allocated to it while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along. For example, consider a process whose function is to display the status of a file on the screen of a terminal. The process will be given the name of the file as an input and will execute the appropriate instructions and system calls to obtain and display the desired information on the terminal. When the process terminates, the operating system will reclaim any reusable resources.

We emphasize that a program by itself is not a process. A program is a ***passive*** entity, like the contents of a file stored on disk, whereas a process is an ***active*** entity. A single-threaded process has one **program counter** specifying the next instruction to execute. (Threads are covered in Chapter 4.) The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, one instruction at most is executed on behalf of the process. Thus, although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

A process is the unit of work in a system. A system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently—by multiplexing on a single CPU, for example.

The operating system is responsible for the following activities in connection with process management:

- Scheduling processes and threads on the CPUs
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

We discuss process-management techniques in Chapters 3 through 5.

1.7 Memory Management

As we discussed in Section 1.2.2, the main memory is central to the operation of a modern computer system. Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a von Neumann architecture). As noted earlier, the main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU-generated I/O calls. In the same way, instructions must be in memory for the CPU to execute them.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management. Many different memory-management schemes are used. These schemes reflect various approaches, and the effectiveness of any given algorithm depends on the situation. In selecting a memory-management scheme for a specific system, we must take into account many factors—especially the ***hardware*** design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and who is using them
- Deciding which processes (or parts of processes) and data to move into and out of memory
- Allocating and deallocating memory space as needed

Memory-management techniques are discussed in Chapters 8 and 9.

1.8 Storage Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. The operating system maps files onto physical media and accesses these files via the storage devices.

1.8.1 File-System Management

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic disk, optical disk, and magnetic tape are the most common. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields). Clearly, the concept of a file is an extremely general one.

The operating system implements the abstract concept of a file by managing mass-storage media, such as tapes and disks, and the devices that control them. In addition, files are normally organized into directories to make them easier to use. Finally, when multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for example, read, write, append).

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files

- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

File-management techniques are discussed in Chapters 11 and 12.

1.8.2 Mass-Storage Management

As we have already seen, because main memory is too small to accommodate all data and programs, and because the data that it holds are lost when power is lost, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the principal on-line storage medium for both programs and data. Most programs—including compilers, assemblers, word processors, editors, and formatters—are stored on a disk until loaded into memory. They then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system. The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

Because secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the disk subsystem and the algorithms that manipulate that subsystem.

There are, however, many uses for storage that is slower and lower in cost (and sometimes of higher capacity) than secondary storage. Backups of disk data, storage of seldom-used data, and long-term archival storage are some examples. Magnetic tape drives and their tapes and CD and DVD drives and platters are typical **tertiary storage** devices. The media (tapes and optical platters) vary between **WORM** (write-once, read-many-times) and **RW** (read–write) formats.

Tertiary storage is not crucial to system performance, but it still must be managed. Some operating systems take on this task, while others leave tertiary-storage management to application programs. Some of the functions that operating systems can provide include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

Techniques for secondary and tertiary storage management are discussed in Chapter 10.

1.8.3 Caching

Caching is an important principle of computer systems. Here's how it works. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache. If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

In addition, internal programmable registers, such as index registers, provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.

Other caches are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy. We are not concerned with these hardware-only caches in this text, since they are outside the control of the operating system.

Because caches have limited size, **cache management** is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance. Figure 1.11 compares storage performance in large workstations and small servers. Various replacement algorithms for software-controlled caches are discussed in Chapter 9.

Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use and data must be in main memory before being moved to secondary storage for safekeeping. The file-system data, which resides permanently on secondary storage, may appear on several levels in the storage hierarchy. At the highest level, the operating system may maintain a cache of file-system data in main memory. In addition, solid-state disks may be used for high-speed storage that is accessed through the file-system interface. The bulk of secondary storage is on magnetic disks. The magnetic-disk storage, in turn, is often backed up onto magnetic tapes or removable disks to protect against data loss in case of a hard-disk failure. Some systems automatically archive old file data from secondary storage to tertiary storage, such as tape jukeboxes, to lower the storage cost (see Chapter 10).

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 1.11 Performance of various levels of storage.

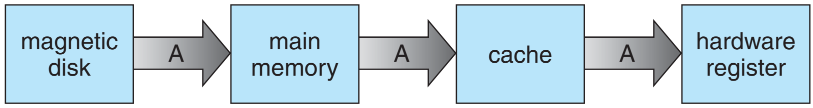


Figure 1.12 Migration of integer A from disk to register.

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on magnetic disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register (see Figure 1.12). Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy. However, in a multitasking environment, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.

The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache (Figure 1.6). In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute in parallel, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**, and it is usually a hardware issue (handled below the operating-system level).

In a distributed environment, the situation becomes even more complex. In this environment, several copies (or replicas) of the same file can be kept on different computers. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible. There are various ways to achieve this guarantee, as we discuss in Chapter 17.

1.8.4 I/O Systems

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the **I/O subsystem**. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

We discussed in Section 1.2.3 how interrupt handlers and device drivers are used in the construction of efficient I/O subsystems. In Chapter 13, we discuss how the I/O subsystem interfaces to the other system components, manages devices, transfers data, and detects I/O completion.

1.9 Protection and Security

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system. For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage, as we discuss in Chapter 14.

A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information (her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system). Prevention of some of these attacks is considered an operating-system function on some systems, while other systems leave it to policy or additional software. Due to the alarming rise in security incidents, operating-system security features represent a fast-growing area of research and implementation. We discuss security in Chapter 15.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifiers** (**user IDs**). In Windows parlance, this is a **security ID** (**SID**). These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be readable by a user, it is translated back to the user name via the user name list.

In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may be allowed only to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group. Group functionality can be implemented as a system-wide list of group names and **group identifiers**. A user can be in one or more groups, depending on operating-system design decisions. The user's group IDs are also included in every associated process and thread.

In the course of normal system use, the user ID and group ID for a user are sufficient. However, a user sometimes needs to **escalate privileges** to gain extra permissions for an activity. The user may need access to a device that is restricted, for example. Operating systems provide various methods to allow privilege escalation. On UNIX, for instance, the *setuid* attribute on a program causes that program to run with the user ID of the owner of the file, rather than the current user's ID. The process runs with this **effective UID** until it turns off the extra privileges or terminates.

1.10 Kernel Data Structures

We turn next to a topic central to operating-system implementation: the way data are structured in the system. In this section, we briefly describe several fundamental data structures used extensively in operating systems. Readers who require further details on these structures, as well as others, should consult the bibliography at the end of the chapter.

1.10.1 Lists, Stacks, and Queues

An array is a simple data structure in which each element can be accessed directly. For example, main memory is constructed as an array. If the data item being stored is larger than one byte, then multiple bytes can be allocated to the item, and the item is addressed as item number $\times$ item size. But what about storing an item whose size may vary? And what about removing an item if the relative positions of the remaining items must be preserved? In such situations, arrays give way to other data structures.

After arrays, lists are perhaps the most fundamental data structures in computer science. Whereas each item in an array can be accessed directly, the items in a list must be accessed in a particular order. That is, a **list** represents a collection of data values as a sequence. The most common method for

CHAPTER 3

Processes

Early computers allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern time-sharing system.

The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: operating-system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive. In this chapter, you will read about what processes are and how they work.

CHAPTER OBJECTIVES

- To introduce the notion of a process — a program in execution, which forms the basis of all computation.
- To describe the various features of processes, including scheduling, creation, and termination.
- To explore interprocess communication using shared memory and message passing.
- To describe communication in client–server systems.

3.1 Process Concept

A question that arises in discussing operating systems involves what to call all the CPU activities. A batch system executes **jobs**, whereas a time-shared system has **user programs**, or **tasks**. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package. And even if a user can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them **processes**.

The terms ***job*** and ***process*** are used almost interchangeably in this text. Although we personally prefer the term ***process***, much of operating-system theory and terminology was developed during a time when the major activity of operating systems was job processing. It would be misleading to avoid the use of commonly accepted terms that include the word ***job*** (such as ***job scheduling***) simply because ***process*** has superseded ***job***.

3.1.1 The Process

Informally, as mentioned earlier, a process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure 3.1.

We emphasize that a program by itself is not a process. A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (often called an **executable file**). In contrast, a process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files

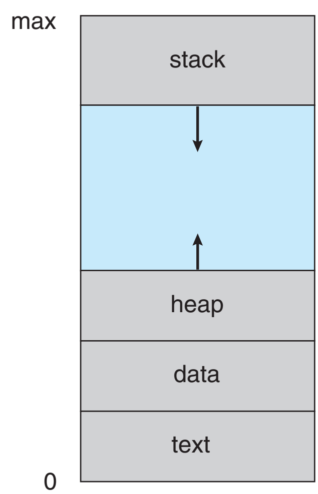


Figure 3.1 Process in memory.

are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in `prog.exe` or `a.out`).

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs. We discuss such matters in Section 3.4.

Note that a process itself can be an execution environment for other code. The Java programming environment provides a good example. In most circumstances, an executable Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code and takes actions (via native machine instructions) on behalf of that code. For example, to run the compiled Java program `Program.class`, we would enter

```
java Program
```

The command `java` runs the JVM as an ordinary process, which in turns executes the Java program `Program` in the virtual machine. The concept is the same as simulation, except that the code, instead of being written for a different instruction set, is written in the Java language.

3.1.2 Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting*, however. The state diagram corresponding to these states is presented in Figure 3.2.

3.1.3 Process Control Block

Each process is represented in the operating system by a **process control block** (**PCB**)—also called a **task control block**. A PCB is shown in Figure 3.3. It contains many pieces of information associated with a specific process, including these:

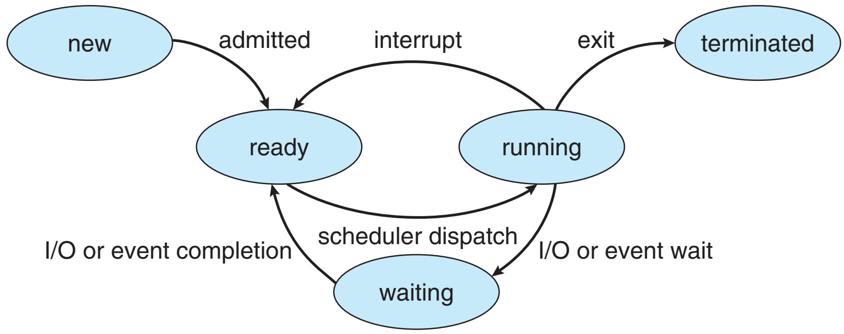


Figure 3.2 Diagram of process state.

- **Process state**. The state may be new, ready, running, waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 3.4).
- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 6 describes process scheduling.)
- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 8).

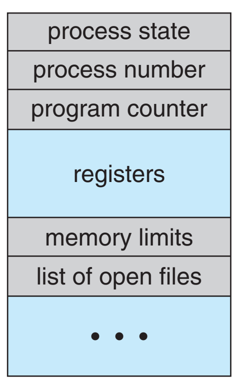


Figure 3.3 Process control block (PCB).

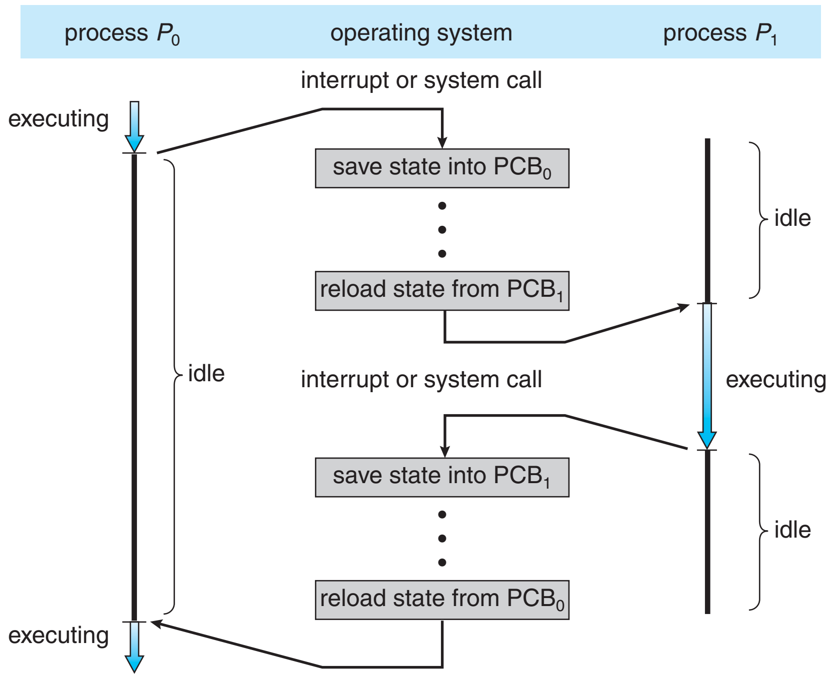


Figure 3.4 Diagram showing CPU switch from process to process.

- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

3.1.4 Threads

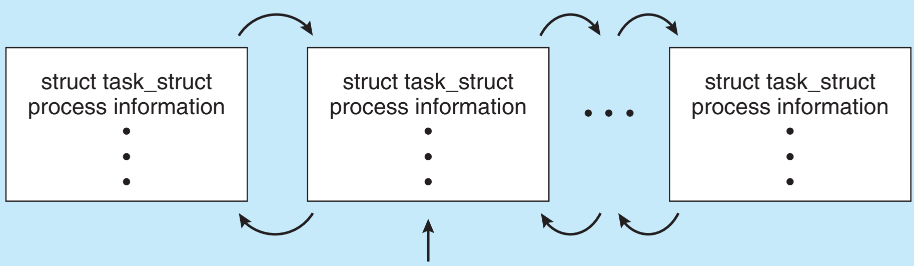
The process model discussed so far has implied that a process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads. Chapter 4 explores threads in detail.

PROCESS REPRESENTATION IN LINUX

The process control block in the Linux operating system is represented by the C structure `task_struct`, which is found in the `<linux/sched.h>` include file in the kernel source-code directory. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and a list of its children and siblings. (A process's **parent** is the process that created it; its **children** are any processes that it creates. Its **siblings** are children with the same parent process.) Some of these fields include:

```
long state; /* state of the process */
struct sched_entity se; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

For example, the state of a process is represented by the field `long state` in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`. The kernel maintains a pointer—`current`—to the process currently executing on the system, as shown below:



As an illustration of how the kernel might manipulate one of the fields in the `task_struct` for a specified process, let's assume the system would like to change the state of the process currently running to the value `new_state`. If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

3.2 Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program

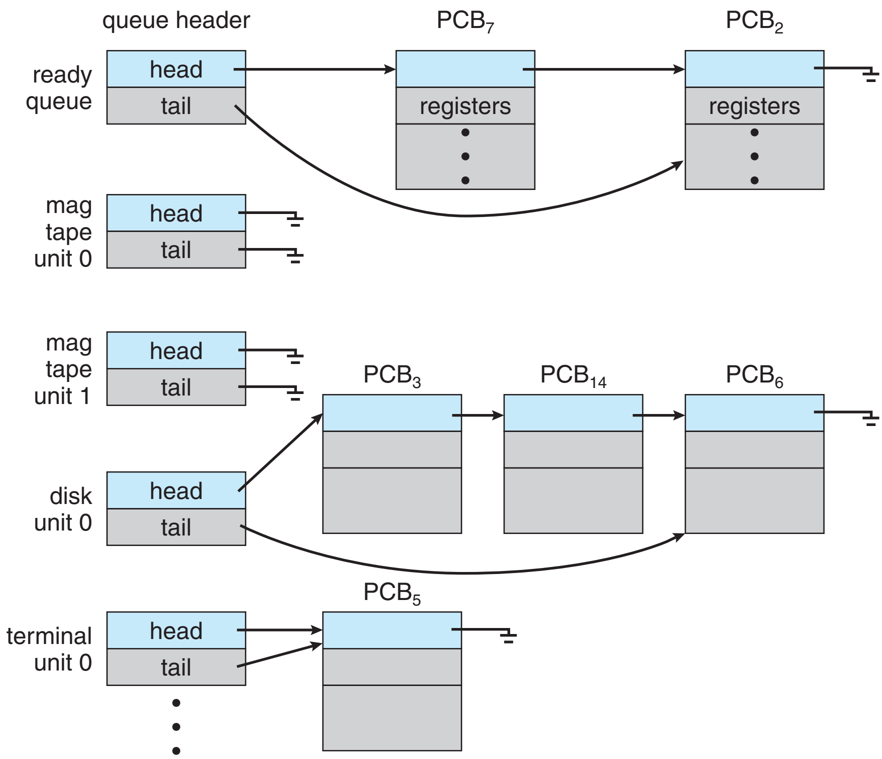


Figure 3.5 The ready queue and various I/O device queues.

while it is running. To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

3.2.1 Scheduling Queues

As processes enter the system, they are put into a **job queue**, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue (Figure 3.5).

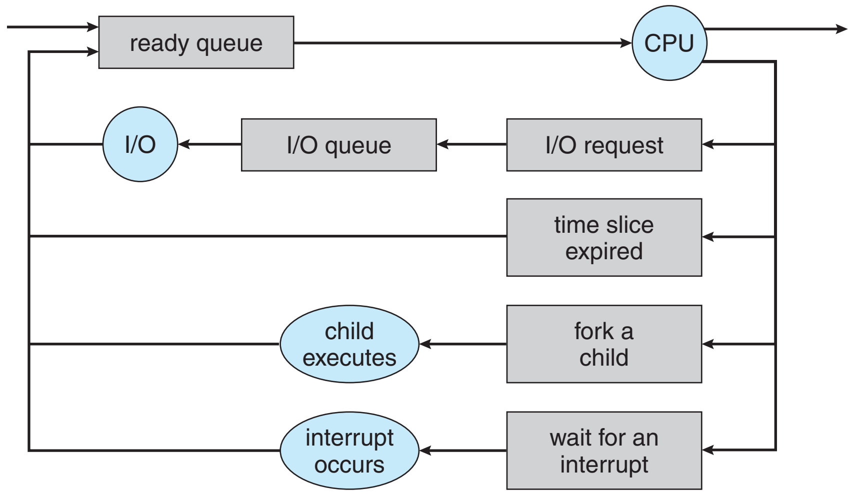


Figure 3.6 Queueing-diagram representation of process scheduling.

A common representation of process scheduling is a **queueing diagram**, such as that in Figure 3.6. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

3.2.2 Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution. The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100 + 10) = 9 percent of the CPU is being used (wasted) simply for scheduling the work.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either I/O bound or CPU bound. An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations. A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that the long-term scheduler select a good ***process mix*** of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler. The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users. If performance declines to unacceptable levels on a multiuser system, some users will simply quit.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is diagrammed in Figure 3.7. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. Swapping is discussed in Chapter 8.

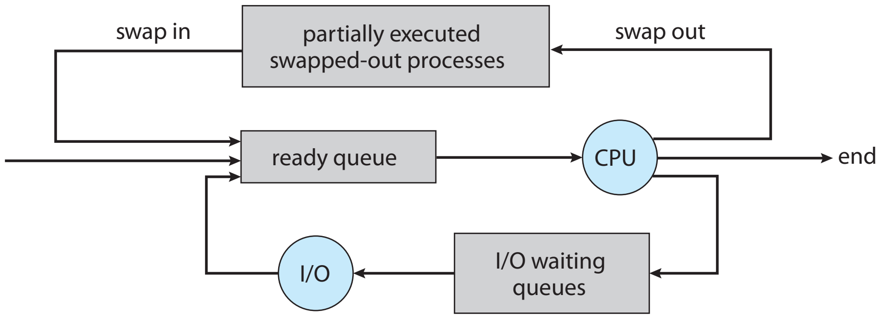


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

3.2.3 Context Switch

As mentioned in Section 1.2.1, interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information. Generically, we perform a **state save** of the current state of the CPU, be it in kernel or user mode, and then a **state restore** to resume operations.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch. As we will see in Chapter 8, advanced memory-management techniques may require that extra data be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.

MULTITASKING IN MOBILE SYSTEMS

Because of the constraints imposed on mobile devices, early versions of iOS did not provide user-application multitasking; only one application runs in the foreground and all other user applications are suspended. Operating-system tasks were multitasked because they were written by Apple and well behaved. However, beginning with iOS 4, Apple now provides a limited form of multitasking for user applications, thus allowing a single foreground application to run concurrently with multiple background applications. (On a mobile device, the **foreground** application is the application currently open and appearing on the display. The **background** application remains in memory, but does not occupy the display screen.) The iOS 4 programming API provides support for multitasking, thus allowing a process to run in the background without being suspended. However, it is limited and only available for a limited number of application types, including applications

- running a single, finite-length task (such as completing a download of content from a network);
- receiving notifications of an event occurring (such as a new email message);
- with long-running background tasks (such as an audio player.)

Apple probably limits multitasking due to battery life and memory use concerns. The CPU certainly has the features to support multitasking, but Apple chooses to not take advantage of some of them in order to better manage resource use.

Android does not place such constraints on the types of applications that can run in the background. If an application requires processing while in the background, the application must use a **service**, a separate application component that runs on behalf of the background process. Consider a streaming audio application: if the application moves to the background, the service continues to send audio files to the audio device driver on behalf of the background application. In fact, the service will continue to run even if the background application is suspended. Services do not have a user interface and have a small memory footprint, thus providing an efficient technique for multitasking in a mobile environment.

3.3 Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

3.3.1 Process Creation

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

Figure 3.8 illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid. (We use the term ***process*** rather loosely, as Linux prefers the term ***task*** instead.) The `init` process (which always has a pid of 1) serves as the root parent process for all user processes. Once the system has booted, the `init` process can also create various user processes, such as a web or print server, an `ssh` server, and the like. In Figure 3.8, we see two children of `init`—`kthreadd` and `sshd`. The `kthreadd` process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, `khelper` and `pdflush`). The `sshd` process is responsible for managing clients that connect to the system by using `ssh` (which is short for ***secure shell***). The `login` process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the `bash` shell, which has been assigned pid 8416. Using the `bash` command-line interface, this user has created the process `ps` as well as the `emacs` editor.

On UNIX and Linux systems, we can obtain a listing of processes by using the `ps` command. For example, the command

```
ps -el
```

will list complete information for all processes currently active in the system. It is easy to construct a process tree similar to the one shown in Figure 3.8 by recursively tracing parent processes all the way to the `init` process.

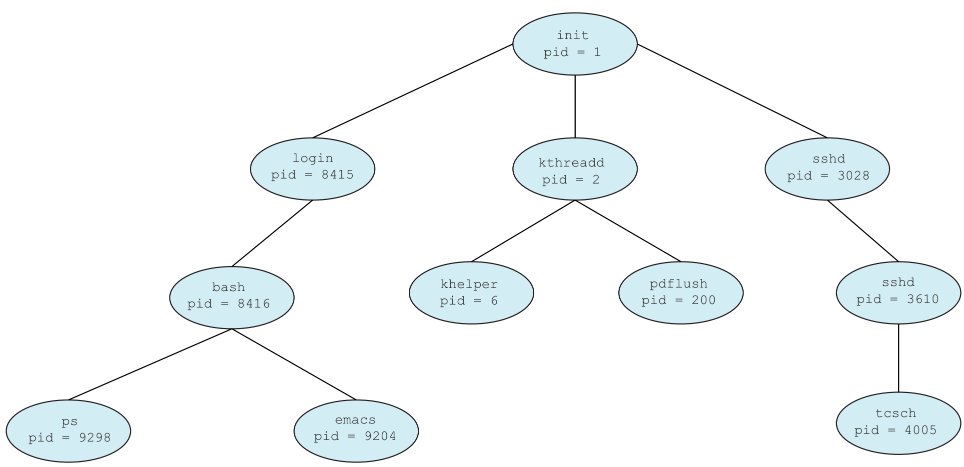


Figure 3.8 A tree of processes on a typical Linux system.

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process. For example, consider a process whose function is to display the contents of a file —say, `image.jpg`—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file ***image.jpg***. Using that file name, it will open the file and write the contents out. It may also get the name of the output device. Alternatively, some operating systems pass resources to child processes. On such a system, the new process may get two open files, `image.jpg` and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the `fork()`, with one difference: the return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

After a `fork()` system call, one of the two processes typically uses the `exec()` system call to replace the process's memory space with a new program. The `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a `wait()` system call to move itself off the ready queue until the termination of the child. Because the

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

Figure 3.9 Creating a separate process using the UNIX `fork()` system call.

call to `exec()` overlays the process's address space with a new program, the call to `exec()` does not return control unless an error occurs.

The C program shown in Figure 3.9 illustrates the UNIX system calls previously described. We now have two different processes running copies of the same program. The only difference is that the value of `pid` (the process identifier) for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files. The child process then overlays its address space with the UNIX command `/bin/ls` (used to get a directory listing) using the `execlp()` system call (`execlp()` is a version of the `exec()` system call). The parent waits for the child process to complete with the `wait()` system call. When the child process completes (by either implicitly or explicitly invoking `exit()`), the parent process resumes from the call to `wait()`, where it completes using the `exit()` system call. This is also illustrated in Figure 3.10.

Of course, there is nothing to prevent the child from ***not*** invoking `exec()` and instead continuing to execute as a copy of the parent process. In this scenario, the parent and child are concurrent processes running the same code

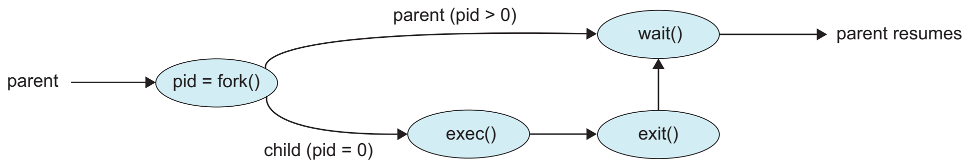


Figure 3.10 Process creation using the `fork()` system call.

instructions. Because the child is a copy of the parent, each process has its own copy of any data.

As an alternative example, we next consider process creation in Windows. Processes are created in the Windows API using the `CreateProcess()` function, which is similar to `fork()` in that a parent creates a new child process. However, whereas `fork()` has the child process inheriting the address space of its parent, `CreateProcess()` requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas `fork()` is passed no parameters, `CreateProcess()` expects no fewer than ten parameters.

The C program shown in Figure 3.11 illustrates the `CreateProcess()` function, which creates a child process that loads the application `mspaint.exe`. We opt for many of the default values of the ten parameters passed to `CreateProcess()`. Readers interested in pursuing the details of process creation and management in the Windows API are encouraged to consult the bibliographical notes at the end of this chapter.

The two parameters passed to the `CreateProcess()` function are instances of the `STARTUPINFO` and `PROCESS_INFORMATION` structures. `STARTUPINFO` specifies many properties of the new process, such as window size and appearance and handles to standard input and output files. The `PROCESS_INFORMATION` structure contains a handle and the identifiers to the newly created process and its thread. We invoke the `ZeroMemory()` function to allocate memory for each of these structures before proceeding with `CreateProcess()`.

The first two parameters passed to `CreateProcess()` are the application name and command-line parameters. If the application name is NULL (as it is in this case), the command-line parameter specifies the application to load. In this instance, we are loading the Microsoft Windows `mspaint.exe` application. Beyond these two initial parameters, we use the default parameters for inheriting process and thread handles as well as specifying that there will be no creation flags. We also use the parent's existing environment block and starting directory. Last, we provide two pointers to the `STARTUPINFO` and `PROCESS_INFORMATION` structures created at the beginning of the program. In Figure 3.9, the parent process waits for the child to complete by invoking the `wait()` system call. The equivalent of this in Windows is `WaitForSingleObject()`, which is passed a handle of the child process—`pi.hProcess`—and waits for this process to complete. Once the child process exits, control returns from the `WaitForSingleObject()` function in the parent process.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figure 3.11 Creating a separate process using the Windows API.

3.3.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its parent process (via the `wait()` system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, `TerminateProcess()` in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the `exit()` system call, providing an exit status as a parameter:

```
/* exit with status 1 */
exit(1);
```

In fact, under normal termination, `exit()` may be called either directly (as shown above) or indirectly (by a `return` statement in `main()`).

A parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid;
int status;

pid = wait(&status);
```

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls `wait()`, because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie** process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.

Now consider what would happen if a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**. Linux and UNIX address this scenario by assigning the `init` process as the new parent to orphan processes. (Recall from Figure 3.8 that the `init` process is the root of the process hierarchy in UNIX and Linux systems.) The `init` process periodically invokes `wait()`, thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

3.4 Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads, as we discussed in Chapter 2.
- **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Cooperating processes require an **interprocess communication** (**IPC**) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: **shared memory** and **message passing**. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.12.

Both of the models just mentioned are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory. (Although there are systems that provide distributed shared memory, we do not consider them in this text.) Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls

MULTIPROCESS ARCHITECTURE—CHROME BROWSER

Many websites contain active content such as JavaScript, Flash, and HTML5 to provide a rich and dynamic web-browsing experience. Unfortunately, these web applications may also contain software bugs, which can result in sluggish response times and can even cause the web browser to crash. This isn't a big problem in a web browser that displays content from only one website. But most contemporary web browsers provide tabbed browsing, which allows a single instance of a web browser application to open several websites at the same time, with each site in a separate tab. To switch between the different sites , a user need only click on the appropriate tab. This arrangement is illustrated below:

A problem with this approach is that if a web application in any tab crashes, the entire process—including all other tabs displaying additional websites —crashes as well.

Google's Chrome web browser was designed to address this issue by using a multiprocess architecture. Chrome identifies three different types of processes: browser, renderers, and plug-ins.

- The **browser** process is responsible for managing the user interface as well as disk and network I/O. A new browser process is created when Chrome is started. Only one browser process is created.
- **Renderer** processes contain logic for rendering web pages. Thus, they contain the logic for handling HTML, Javascript, images, and so forth. As a general rule, a new renderer process is created for each website opened in a new tab, and so several renderer processes may be active at the same time.
- A **plug-in** process is created for each type of plug-in (such as Flash or QuickTime) in use. Plug-in processes contain the code for the plug-in as well as additional code that enables the plug-in to communicate with associated renderer processes and the browser process.

The advantage of the multiprocess approach is that websites run in isolation from one another. If one website crashes, only its renderer process is affected; all other processes remain unharmed. Furthermore, renderer processes run in a **sandbox**, which means that access to disk and network I/O is restricted, minimizing the effects of any security exploits.

and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared-

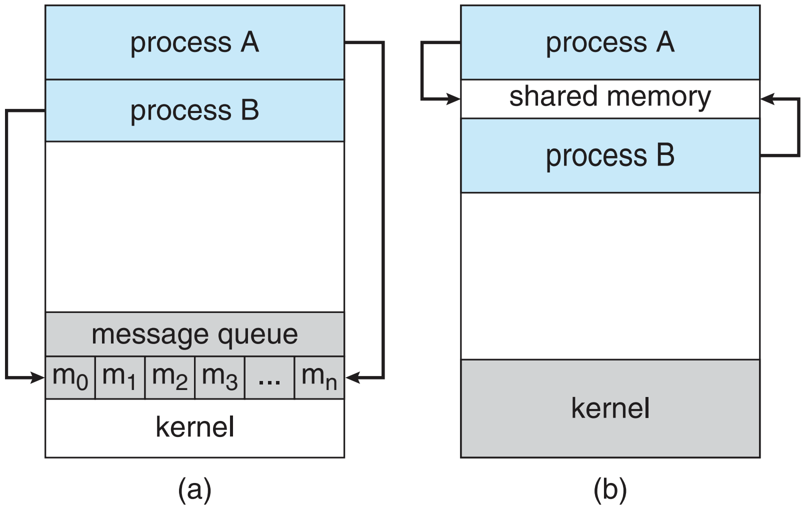


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Recent research on systems with several processing cores indicates that message passing provides better performance than shared memory on such systems. Shared memory suffers from cache coherency issues, which arise because shared data migrate among the several caches. As the number of processing cores on systems increases, it is possible that we will see message passing as the preferred mechanism for IPC.

In the remainder of this section, we explore shared-memory and message-passing systems in more detail.

3.4.1 Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader. The producer–consumer problem

```
item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.13 The producer process using shared memory.

also provides a useful metaphor for the client–server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Let's look more closely at how the bounded buffer illustrates interprocess communication using shared memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
   . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared `buffer` is implemented as a circular array with two logical pointers: `in` and `out`. The variable `in` points to the next free position in the buffer; `out` points to the first full position in the buffer. The buffer is empty when `in` == `out`; the buffer is full when ((`in` + 1) % BUFFER_SIZE) == `out`.

The code for the producer process is shown in Figure 3.13, and the code for the consumer process is shown in Figure 3.14. The producer process has a

```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Figure 3.14 The consumer process using shared memory.

local variable `next_produced` in which the new item to be produced is stored. The consumer process has a local variable `next_consumed` in which the item to be consumed is stored.

This scheme allows at most BUFFER_SIZE − 1 items in the buffer at the same time. We leave it as an exercise for you to provide a solution in which BUFFER_SIZE items can be in the buffer at the same time. In Section 3.5.1, we illustrate the POSIX API for shared memory.

One issue this illustration does not address concerns the situation in which both the producer process and the consumer process attempt to access the shared buffer concurrently. In Chapter 5, we discuss how synchronization among cooperating processes can be implemented effectively in a shared-memory environment.

3.4.2 Message-Passing Systems

In Section 3.4.1, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet `chat` program could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations:

```
send(message)          receive(message)
```

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating-system design.

If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other: a ***communication link*** must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, which are covered in Chapter 17) but rather with its logical implementation. Here are several methods for logically implementing a link and the `send()`/`receive()` operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

We look at issues related to each of these features next.

3.4.2.1 Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the `send()` and `receive()` primitives are defined as:

- `send(P, message)`—Send a `message` to process `P`.
- `receive(Q, message)`—Receive a `message` from process `Q`.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits ***symmetry*** in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs ***asymmetry*** in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the `send()` and `receive()` primitives are defined as follows:

- `send(P, message)`—Send a `message` to process `P`.
- `receive(id, message)`—Receive a `message` from any process. The variable `id` is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such ***hard-coding*** techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

With ***indirect communication***, the messages are sent to and received from ***mailboxes***, or ***ports***. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The `send()` and `receive()` primitives are defined as follows:

- `send(A, message)`—Send a `message` to mailbox `A`.
- `receive(A, message)`—Receive a `message` from mailbox `A`.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Now suppose that processes P_1, P_2, and P_3 all share mailbox A. Process P_1 sends a message to A, while both P_2 and P_3 execute a `receive()` from A. Which process will receive the message sent by P_1? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a `receive()` operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P_2 or P_3, but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message (for example, ***round robin,*** where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

3.4.2.2 Synchronization

Communication between processes takes place through calls to `send()` and `receive()` primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**—also known as **synchronous** and **asynchronous**. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.
- **Nonblocking receive**. The receiver retrieves either a valid message or a null.

Different combinations of `send()` and `receive()` are possible. When both `send()` and `receive()` are blocking, we have a **rendezvous** between the sender and the receiver. The solution to the producer–consumer problem becomes trivial when we use blocking `send()` and `receive()` statements. The producer merely invokes the blocking `send()` call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes `receive()`, it blocks until a message is available. This is illustrated in Figures 3.15 and 3.16.

3.4.2.3 Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

Figure 3.15 The producer process using message passing.

- **Zero capacity**. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**. The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

3.5 Examples of IPC Systems

In this section, we explore three different IPC systems. We first cover the POSIX API for shared memory and then discuss message passing in the Mach operating system. We conclude with Windows, which interestingly uses shared memory as a mechanism for providing certain types of message passing.

3.5.1 An Example: POSIX Shared Memory

Several IPC mechanisms are available for POSIX systems, including shared memory and message passing. Here, we explore the POSIX API for shared memory.

POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file. A process must first create

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

Figure 3.16 The consumer process using message passing.

CHAPTER 6

CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms. We also consider the problem of selecting an algorithm for a particular system.

In Chapter 4, we introduced threads to the process model. On operating systems that support them, it is kernel-level threads—not processes—that are in fact being scheduled by the operating system. However, the terms "process scheduling" and "thread scheduling" are often used interchangeably. In this chapter, we use ***process scheduling*** when discussing general scheduling concepts and ***thread scheduling*** to refer to thread-specific ideas.

CHAPTER OBJECTIVES

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems.
- To describe various CPU-scheduling algorithms.
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system.
- To examine the scheduling algorithms of several operating systems.

6.1 Basic Concepts

In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When

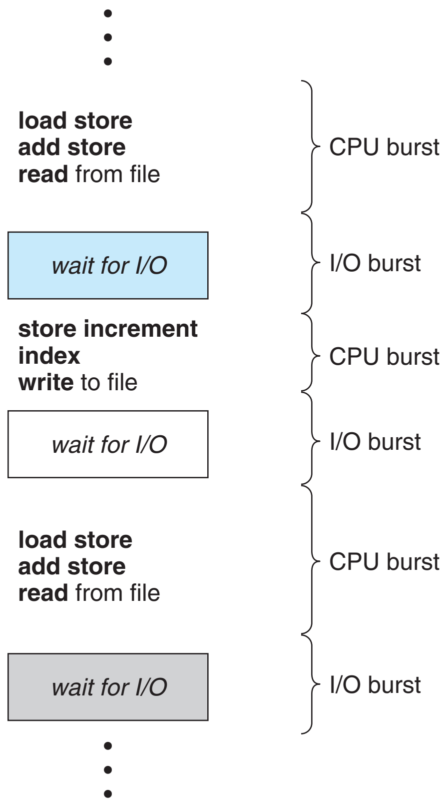


Figure 6.1 Alternating sequence of CPU and I/O bursts.

one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

6.1.1 CPU–I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 6.1).

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure 6.2. The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts.

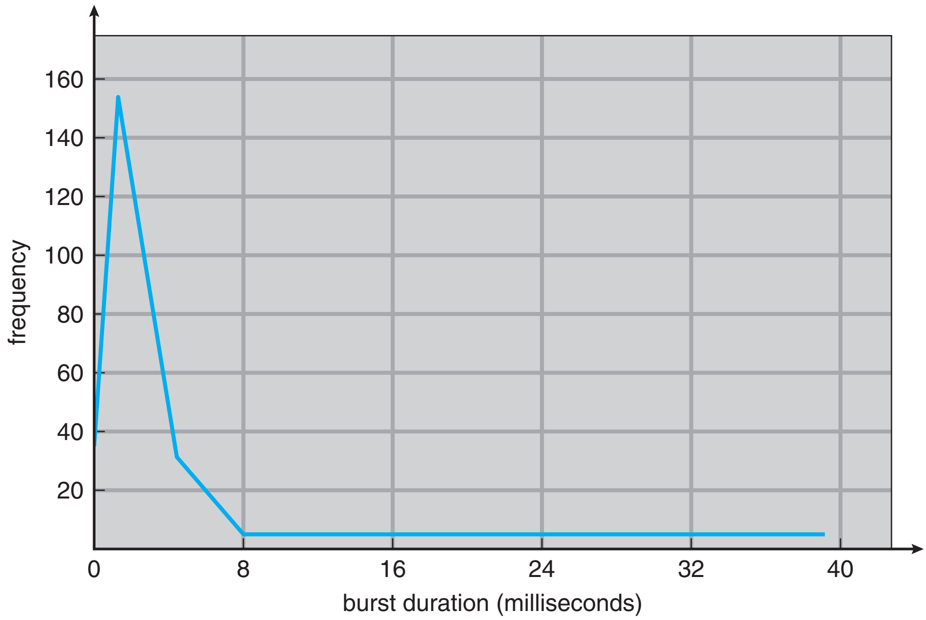


Figure 6.2 Histogram of CPU-burst durations.

An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

6.1.2 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler**, or CPU scheduler. The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

6.1.3 Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of `wait()` for the termination of a child process)

2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**. Otherwise, it is **preemptive**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method was used by Microsoft Windows 3.x. Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling. The Mac OS X operating system for the Macintosh also uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling. Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes. Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. This issue was explored in detail in Chapter 5.

Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure? Chaos ensues. Certain operating systems, including most versions of UNIX, deal with this problem by waiting either for a system call to complete or for an I/O block to take place before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel-execution model is a poor one for supporting real-time computing where tasks must complete execution within a given time frame. In Section 6.6, we explore scheduling demands of real-time systems.

Because interrupts can, by definition, occur at any time, and because they cannot always be ignored by the kernel, the sections of code affected by interrupts must be guarded from simultaneous use. The operating system needs to accept interrupts at almost all times. Otherwise, input might be lost or output overwritten. So that these sections of code are not accessed concurrently by several processes, they disable interrupts at entry and reenable interrupts at exit. It is important to note that sections of code that disable interrupts do not occur very often and typically contain few instructions.

6.1.4 Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

6.2 Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called **throughput**. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
- **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being

output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Investigators have suggested that, for interactive systems (such as desktop systems), it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance.

As we discuss various CPU-scheduling algorithms in the following section, we illustrate their operation. An accurate illustration should involve many processes, each a sequence of several hundred CPU bursts and I/O bursts. For simplicity, though, we consider only one CPU burst (in milliseconds) per process in our examples. Our measure of comparison is the average waiting time. More elaborate evaluation mechanisms are discussed in Section 6.8.

6.3 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms. In this section, we describe several of them.

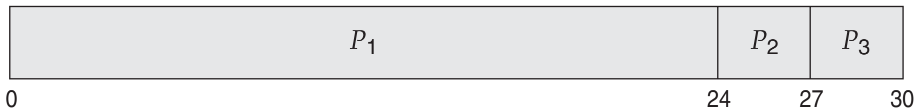
6.3.1 First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS)** scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

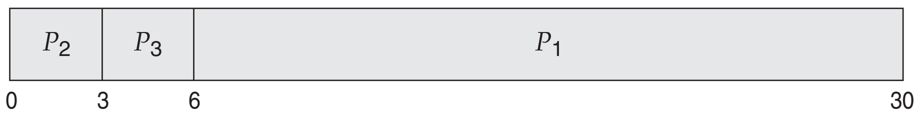
On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process P_1, 24 milliseconds for process P_2, and 27 milliseconds for process P_3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P_2, P_3, P_1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Note also that the FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

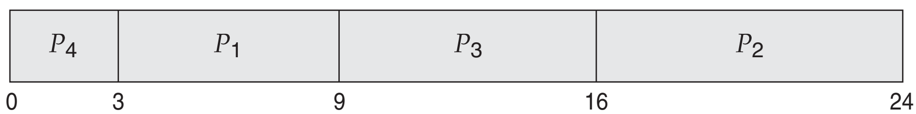
6.3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the **shortest-job-first (SJF)** scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the ***shortest-next-CPU-burst*** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



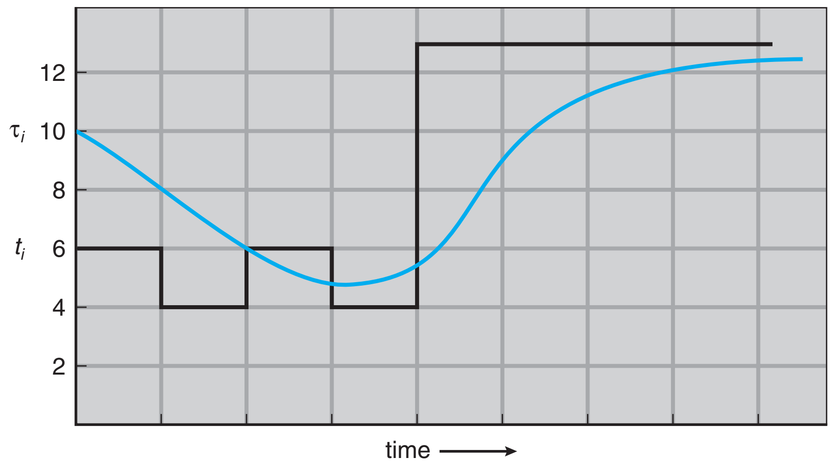
The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

The real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use the process time limit that a user specifies when he submits the job. In this situation, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response but too low a value will cause a time-limit-exceeded error and require resubmission. SJF scheduling is used frequently in long-term scheduling.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. With short-term scheduling, there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts. We can define the exponential



CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

Figure 6.3 Prediction of the length of the next CPU burst.

average with the following formula. Let t_n be the length of the nth CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for α, $0 \leq \alpha \leq 1$, define

$$\tau_{n+1} = \alpha\ t_n + (1 - \alpha)\tau_n.$$

The value of t_n contains our most recent information, while τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient). If $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted. The initial τ_0 can be defined as a constant or as an overall system average. Figure 6.3 shows an exponential average with $\alpha = 1/2$ and $\tau_0 = 10$.

To understand the behavior of the exponential average, we can expand the formula for τ_{n+1} by substituting for τ_n to find

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1}\tau_0.$$

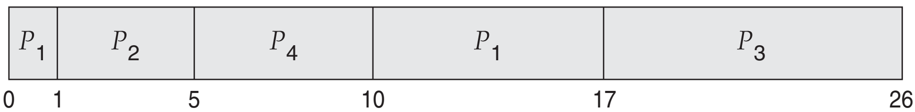
Typically, α is less than 1. As a result, $(1 - \alpha)$ is also less than 1, and each successive term has less weight than its predecessor.

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6.5$ milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

6.3.3 Priority Scheduling

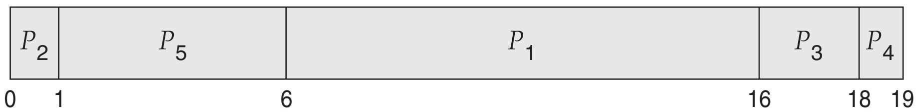
The SJF algorithm is a special case of the general **priority-scheduling** algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of ***high*** priority and ***low*** priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order $P_1, P_2, \cdots, P_5$, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes. (Rumor has it that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

6.3.4 Round-Robin Scheduling

The **round-robin** (**RR**) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue.

The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

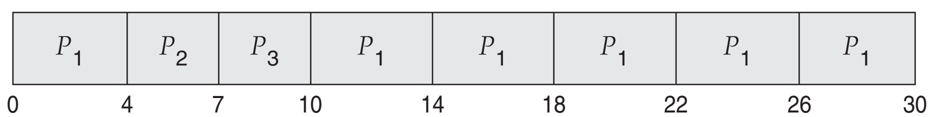
To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2. Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. P_1 waits for 6 milliseconds (10 - 4), P_2 waits for 4 milliseconds, and P_3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy

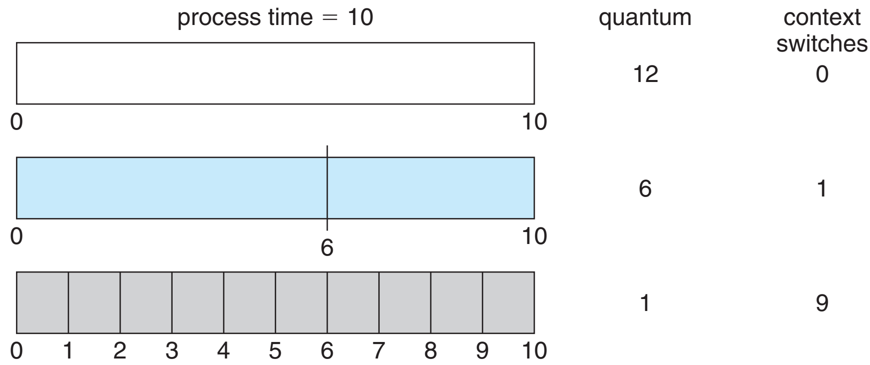


Figure 6.4 How a smaller time quantum increases context switches.

is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 6.4).

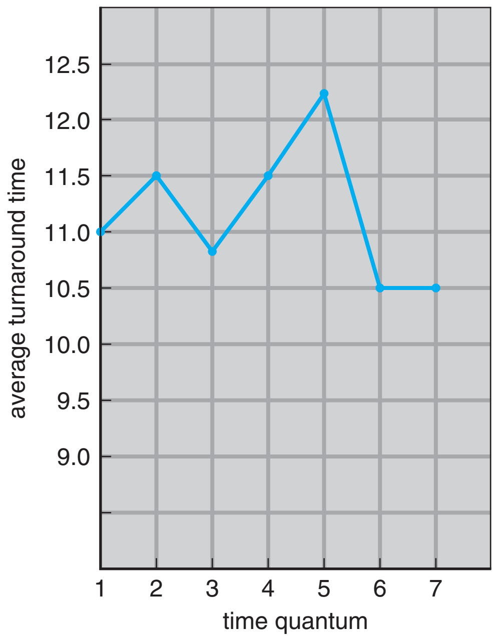
Thus, we want the time quantum to be large with respect to the context-switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

Turnaround time also depends on the size of the time quantum. As we can see from Figure 6.5, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context-switch time, it should not be too large. As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

6.3.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 6.5 How turnaround time varies with the time quantum.

common division is made between **foreground** (interactive) processes and **background** (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues (Figure 6.6). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

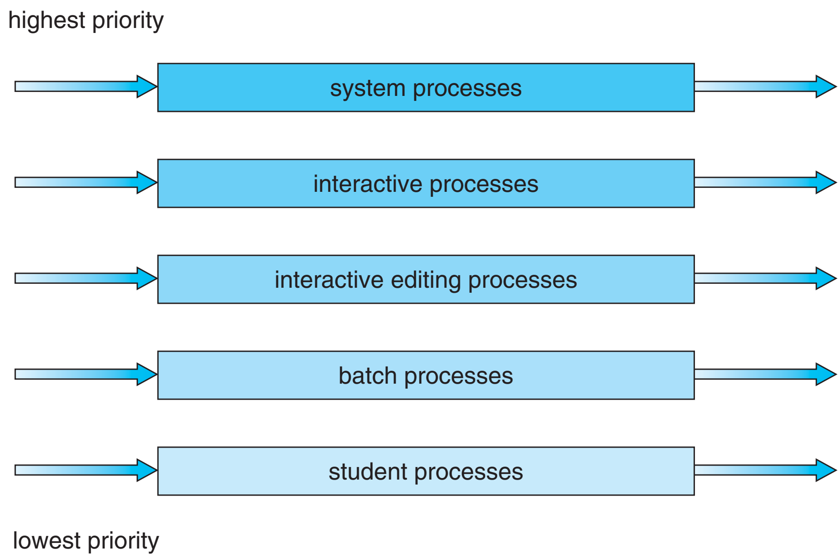


Figure 6.6 Multilevel queue scheduling.

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

6.3.6 Multilevel Feedback Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 6.7). The scheduler first executes all

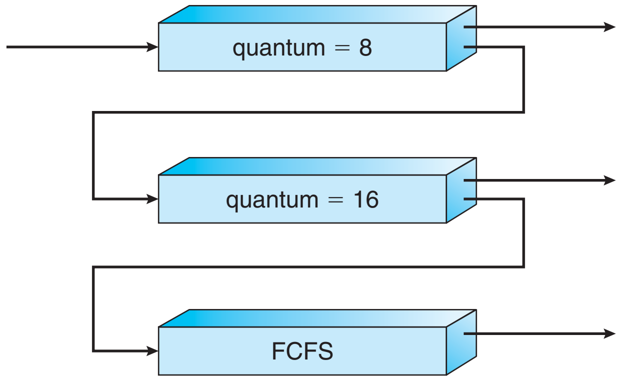


Figure 6.7 Multilevel feedback queues.

processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm,

CHAPTER 7

Deadlocks

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**. We discussed this issue briefly in Chapter 5 in connection with semaphores.

Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part: “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

In this chapter, we describe methods that an operating system can use to prevent or deal with deadlocks. Although some applications can identify programs that may deadlock, operating systems typically do not provide deadlock-prevention facilities, and it remains the responsibility of programmers to ensure that they design deadlock-free programs. Deadlock problems can only become more common, given current trends, including larger numbers of processes, multithreaded programs, many more resources within a system, and an emphasis on long-lived file and database servers rather than batch systems.

CHAPTER OBJECTIVES

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.

7.1 System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type *CPU* has two instances. Similarly, the resource type *printer* may have five instances.

If a process requests an instance of a resource type, the allocation of ***any*** instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly. For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer.

Chapter 5 discussed various synchronization tools, such as mutex locks and semaphores. These tools are also considered system resources, and they are a common source of deadlock. However, a lock is typically associated with protecting a specific data structure—that is, one lock may be used to protect access to a queue, another to protect access to a linked list, and so forth. For that reason, each lock is typically assigned its own resource class, and definition is not a problem.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request**. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. **Release**. The process releases the resource.

The request and release of resources may be system calls, as explained in Chapter 2. Examples are the `request()` and `release()` device, `open()` and `close()` file, and `allocate()` and `free()` memory system calls. Similarly, as we saw in Chapter 5, the request and release of semaphores can be accomplished through the `wait()` and `signal()` operations on semaphores or through `acquire()` and `release()` of a mutex lock. For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource. A system table records whether each resource is free or allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, semaphores, mutex locks, and files). However, other types of events may result in deadlocks (for example, the IPC facilities discussed in Chapter 3).

To illustrate a deadlocked state, consider a system with three CD RW drives. Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlocked state. Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process P_i is holding the DVD and process P_j is holding the printer. If P_i requests the printer and P_j requests the DVD drive, a deadlock occurs.

Developers of multithreaded applications must remain aware of the possibility of deadlocks. The locking tools presented in Chapter 5 are designed to avoid race conditions. However, in using these tools, developers must pay careful attention to how locks are acquired and released. Otherwise, deadlock can occur, as illustrated in the dining-philosophers problem in Section 5.7.3.

7.2 Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

DEADLOCK WITH MUTEX LOCKS

Let's see how deadlock can occur in a multithreaded Pthread program using mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created in the following code example:

```
/* Create and initialize the mutex locks */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

Next, two threads—`thread_one` and `thread_two`—are created, and both these threads have access to both mutex locks. `thread_one` and `thread_two`

DEADLOCK WITH MUTEX LOCKS (Continued)

run in the functions do_work_one() and do_work_two(), respectively, as shown below:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

In this example, thread_one attempts to acquire the mutex locks in the order (1) first_mutex, (2) second_mutex, while thread_two attempts to acquire the mutex locks in the order (1) second_mutex, (2) first_mutex. Deadlock is possible if thread_one acquires first_mutex while thread_two acquires second_mutex.

Note that, even though deadlock is possible, it will not occur if thread_one can acquire and release the mutex locks for first_mutex and second_mutex before thread_two attempts to acquire the locks. And, of course, the order in which the threads run depends on how they are scheduled by the CPU scheduler. This example illustrates a problem with handling deadlocks: it is difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances.

7.2.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion**. At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait**. A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent. We shall see in Section 7.4, however, that it is useful to consider each condition separately.

7.2.2 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge $P_i \rightarrow R_j$ is called a **request edge**; a directed edge $R_j \rightarrow P_i$ is called an **assignment edge**.

Pictorially, we represent each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.

When process P_i requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is ***instantaneously*** transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure 7.1 depicts the following situation.

- The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$

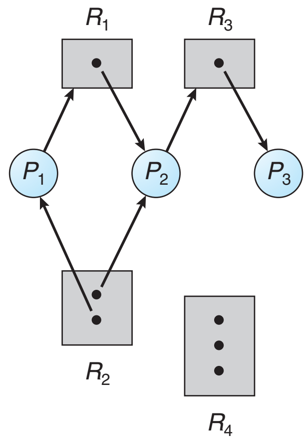


Figure 7.1 Resource-allocation graph.

- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

- Resource instances:
  - One instance of resource type R_1
  - Two instances of resource type R_2
  - One instance of resource type R_3
  - Three instances of resource type R_4
- Process states:
  - Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
  - Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
  - Process P_3 is holding an instance of R_3.

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in Figure 7.1. Suppose that process P_3 requests an instance of resource

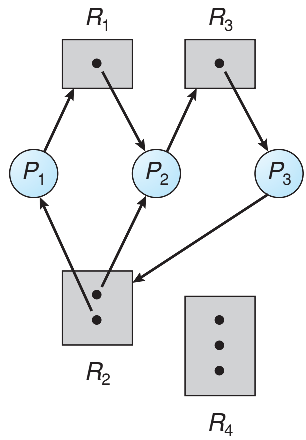


Figure 7.2 Resource-allocation graph with a deadlock.

type R_2. Since no resource instance is currently available, we add a request edge $P_3 \rightarrow R_2$ to the graph (Figure 7.2). At this point, two minimal cycles exist in the system:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

Processes P_1, P_2, and P_3 are deadlocked. Process P_2 is waiting for the resource R_3, which is held by process P_3. Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2. In addition, process P_1 is waiting for process P_2 to release resource R_1.

Now consider the resource-allocation graph in Figure 7.3. In this example, we also have a cycle:

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

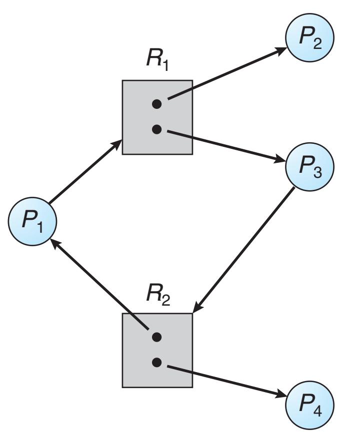


Figure 7.3 Resource-allocation graph with a cycle but no deadlock.

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is ***not*** in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

7.3 Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including Linux and Windows. It is then up to the application developer to write programs that handle deadlocks.

Next, we elaborate briefly on each of the three methods for handling deadlocks. Then, in Sections 7.4 through 7.7, we present detailed algorithms. Before proceeding, we should mention that some researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems. The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme. **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions (Section 7.2.1) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. We discuss these methods in Section 7.4.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. We discuss these schemes in Section 7.5.

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred). We discuss these issues in Section 7.6 and Section 7.7.

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened. In this case, the undetected deadlock will cause the system's performance to deteriorate, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems, as mentioned earlier. Expense is one important consideration. Ignoring the possibility of deadlocks is cheaper than the other approaches. Since in many systems, deadlocks occur infrequently (say, once per year), the extra expense of the other methods may not seem worthwhile. In addition, methods used to recover from other conditions may be put to use to recover from deadlock. In some circumstances, a system is in a frozen state but not in a deadlocked state. We see this situation, for example, with a real-time process running at the highest priority (or any process running on a nonpreemptive scheduler) and never returning control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

7.4 Deadlock Prevention

As we noted in Section 7.2.1, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

7.4.1 Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be nonsharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable. For example, a mutex lock cannot be simultaneously shared by several processes.

7.4.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that we can use requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

programs would need to be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Versions with minor changes retain the same version number, whereas versions with major changes increment the number. Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries**.

Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses. We elaborate on this concept when we discuss paging in Section 8.5.4.

8.2 Swapping

A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution (Figure 8.5). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

8.2.1 Standard Swapping

Standard swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk. It must be large

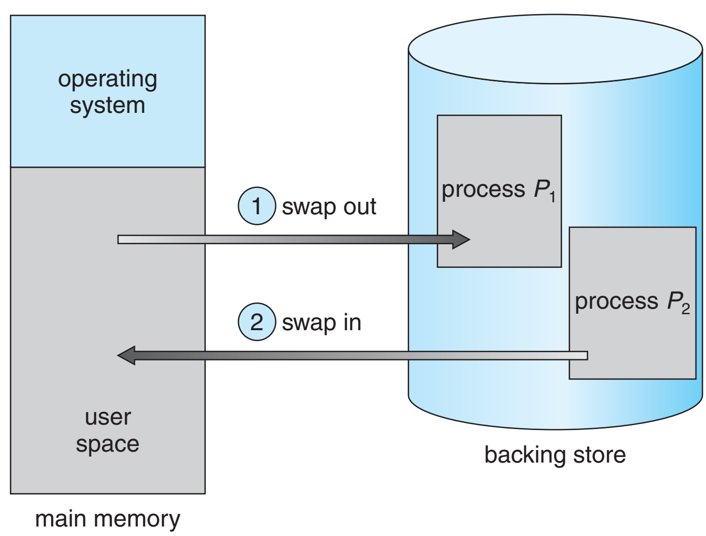


Figure 8.5 Swapping of two processes using a disk as a backing store.

enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

The context-switch time in such a swapping system is fairly high. To get an idea of the context-switch time, let's assume that the user process is 100 MB in size and the backing store is a standard hard disk with a transfer rate of 50 MB per second. The actual transfer of the 100-MB process to or from main memory takes

$$100 \text{ MB}/50 \text{ MB per second} = 2 \text{ seconds}$$

The swap time is 200 milliseconds. Since we must swap both out and in, the total swap time is about 4,000 milliseconds. (Here, we are ignoring other disk performance aspects, which we cover in Chapter 10.)

Notice that the major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped. If we have a computer system with 4 GB of main memory and a resident operating system taking 1 GB, the maximum size of the user process is 3 GB. However, many user processes may be much smaller than this—say, 100 MB. A 100-MB process could be swapped out in 2 seconds, compared with the 60 seconds required for swapping 3 GB. Clearly, it would be useful to know exactly how much memory a user process ***is*** using, not simply how much it ***might*** be using. Then we would need to swap only what is actually used, reducing swap time. For this method to be effective, the user must keep the system informed of any changes in memory requirements. Thus, a process with dynamic memory requirements will need to issue system calls (`request_memory()` and `release_memory()`) to inform the operating system of its changing memory needs.

Swapping is constrained by other factors as well. If we want to swap a process, we must be sure that it is completely idle. Of particular concern is any pending I/O. A process may be waiting for an I/O operation when we want to swap that process to free up memory. However, if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped. Assume that the I/O operation is queued because the device is busy. If we were to swap out process P_1 and swap in process P_2, the I/O operation might then attempt to use memory that now belongs to process P_2. There are two main solutions to this problem: never swap a process with pending I/O, or execute I/O operations only into operating-system buffers. Transfers between operating-system buffers and process memory then occur only when the process is swapped in. Note that this **double buffering** itself adds overhead. We now need to copy the data again, from kernel memory to user memory, before the user process can access it.

Standard swapping is not used in modern operating systems. It requires too much swapping time and provides too little execution time to be a reasonable memory-management solution. Modified versions of swapping, however, are found on many systems, including UNIX, Linux, and Windows. In one common variation, swapping is normally disabled but will start if the amount of free memory (unused memory available for the operating system or processes to use) falls below a threshold amount. Swapping is halted when the amount of free memory increases. Another variation involves swapping portions of processes—rather than entire processes—to decrease swap time. Typically, these modified forms of swapping work in conjunction with virtual memory, which we cover in Chapter 9.

8.2.2 Swapping on Mobile Systems

Although most operating systems for PCs and servers support some modified version of swapping, mobile systems typically do not support swapping in any form. Mobile devices generally use flash memory rather than more spacious hard disks as their persistent storage. The resulting space constraint is one reason why mobile operating-system designers avoid swapping. Other reasons include the limited number of writes that flash memory can tolerate before it becomes unreliable and the poor throughput between main memory and flash memory in these devices.

Instead of using swapping, when free memory falls below a certain threshold, Apple's iOS ***asks*** applications to voluntarily relinquish allocated memory. Read-only data (such as code) are removed from the system and later reloaded from flash memory if necessary. Data that have been modified (such as the stack) are never removed. However, any applications that fail to free up sufficient memory may be terminated by the operating system.

Android does not support swapping and adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its **application state** to flash memory so that it can be quickly restarted.

Because of these restrictions, developers for mobile systems must carefully allocate and release memory to ensure that their applications do not use too much memory or suffer from memory leaks. Note that both iOS and Android support paging, so they do have memory-management abilities. We discuss paging later in this chapter.

8.3 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Thus, in this text, we discuss only the situation in which the operating system resides in low memory. The development of the other situation is similar.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

8.3.1 Memory Protection

Before discussing memory allocation further, we must discuss the issue of memory protection. We can prevent a process from accessing memory it does not own by combining two ideas previously discussed. If we have a system with a relocation register (Section 8.1.3), together with a limit register (Section 8.1.1), we accomplish our goal. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory (Figure 8.6).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called **transient** operating-system code; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

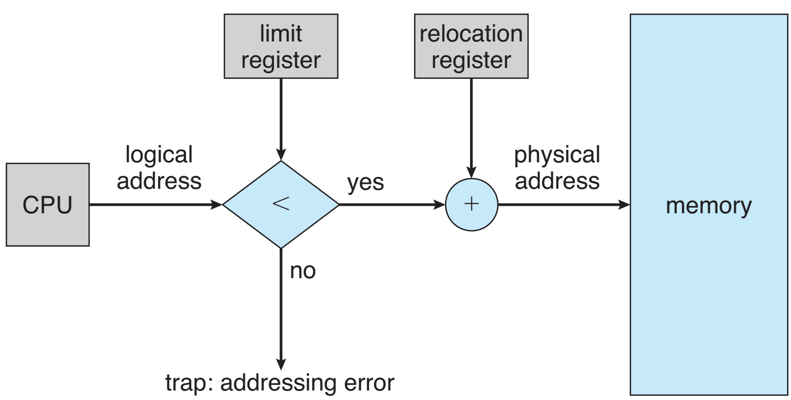


Figure 8.6 Hardware support for relocation and limit registers.

8.3.2 Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this **multiple-partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT)but is no longer in use. The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management (Section 8.4).

In the **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**. Eventually, as you will see, memory contains a set of holes of various sizes.

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, then, we have a list of available block sizes and an input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, as mentioned, the memory blocks available comprise a ***set*** of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general **dynamic storage-allocation problem**, which concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

- **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

8.3.3 Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece—the one on the top or the one on the bottom?) No matter which algorithm is used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another 0.5 N blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. Two complementary techniques achieve this solution: segmentation (Section 8.4) and paging (Section 8.5). These techniques can also be combined.

Fragmentation is a general problem in computing that can occur wherever we must manage blocks of data. We discuss the topic further in the storage management chapters (Chapters 10 through and 12).

8.4 Segmentation

As we've already seen, the user's view of memory is not the same as the actual physical memory. This is equally true of the programmer's view of memory. Indeed, dealing with memory in terms of its physical properties is inconvenient to both the operating system and the programmer. What if the hardware could provide a memory mechanism that mapped the programmer's view to the actual physical memory? The system would have more freedom to manage memory, while the programmer would have a more natural programming environment. Segmentation provides such a mechanism.

8.4.1 Basic Method

Do programmers think of memory as a linear array of bytes, some containing instructions and others containing data? Most programmers would say "no." Rather, they prefer to view memory as a collection of variable-sized segments, with no necessary ordering among the segments (Figure 8.7).

When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. The programmer talks about "the stack," "the math library," and "the main program" without caring what addresses in memory these elements occupy. She is not concerned with whether the stack is stored before or after the `Sqrt()` function. Segments vary in length, and the length of each is intrinsically defined by its purpose in the program. Elements within a segment are identified by their offset from the beginning of the segment: the first statement of the program, the seventh stack frame entry in the stack, the fifth instruction of the `Sqrt()`, and so on.

Segmentation is a memory-management scheme that supports this programmer view of memory. A logical address space is a collection of segments.

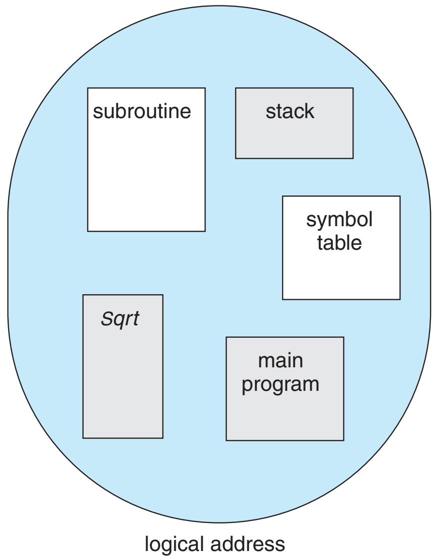


Figure 8.7 Programmer's view of a program.

Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset.

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:*

<segment-number, offset>.

Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

A C compiler might create separate segments for the following:

1. The code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

8.4.2 Segmentation Hardware

Although the programmer can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical

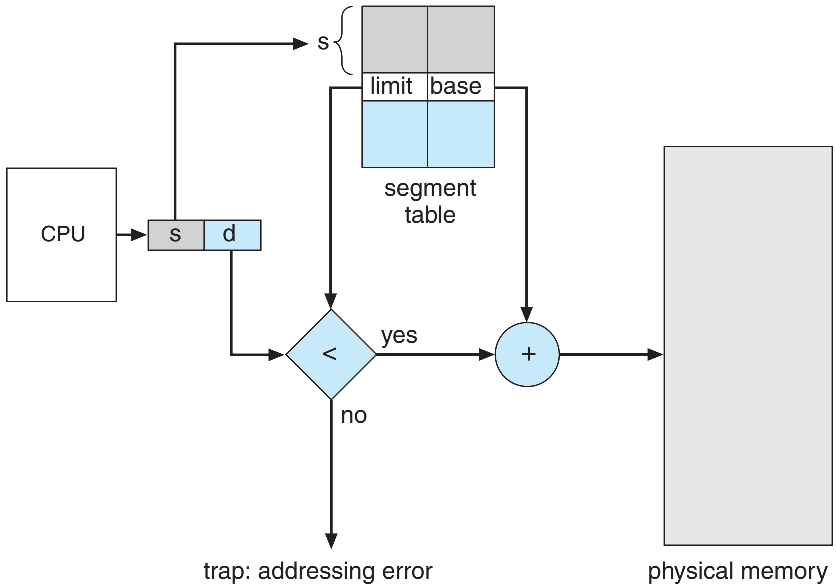


Figure 8.8 Segmentation hardware.

addresses. This mapping is effected by a **segment table**. Each entry in the segment table has a **segment base** and a **segment limit**. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure 8.8. A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d*. The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base–limit register pairs.

As an example, consider the situation shown in Figure 8.9. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

8.5 Paging

Segmentation permits the physical address space of a process to be noncontiguous. **Paging** is another memory-management scheme that offers this advantage. However, paging avoids external fragmentation and the need for

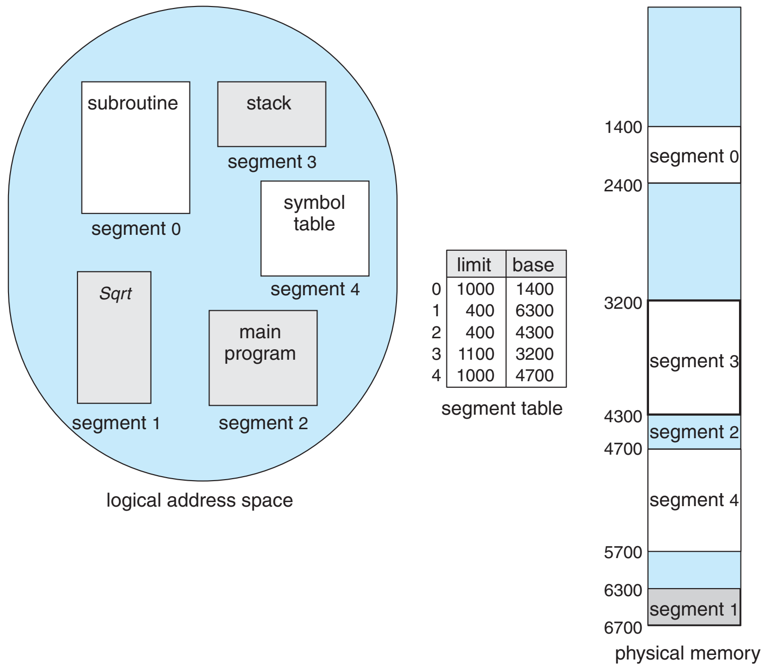


Figure 8.9 Example of segmentation.

compaction, whereas segmentation does not. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store. Most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems, from those for mainframes through those for smartphones. Paging is implemented through cooperation between the operating system and the computer hardware.

8.5.1 Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames. This rather simple idea has great functionality and wide ramifications. For example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 2^{64} bytes of physical memory.

The hardware support for paging is illustrated in Figure 8.10. Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page**

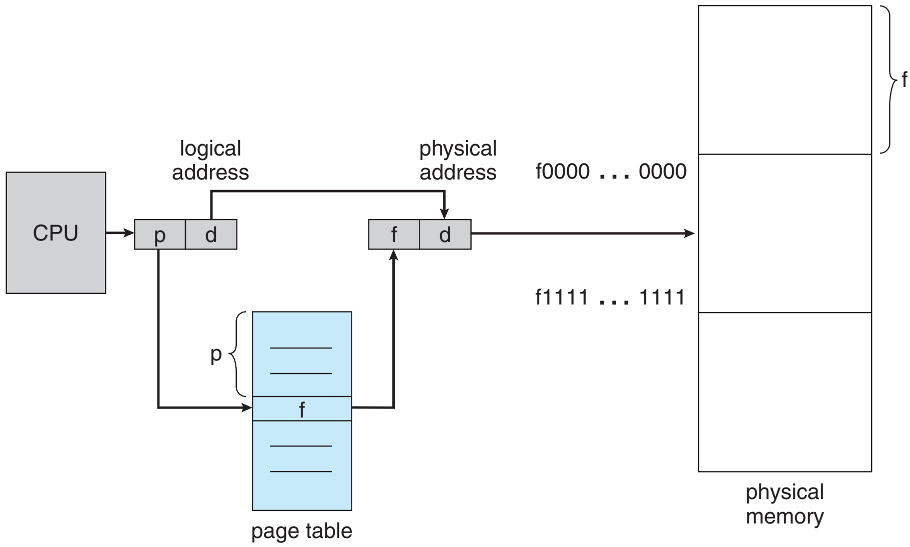


Figure 8.10 Paging hardware.

offset (d). The page number is used as an index into a **page table**. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 8.11.

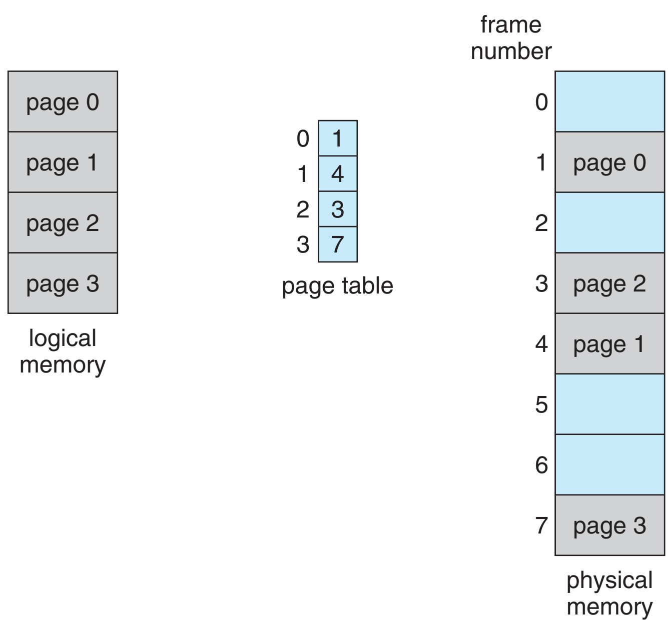
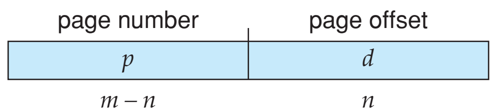


Figure 8.11 Paging model of logical and physical memory.

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order $m - n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



where p is an index into the page table and d is the displacement within the page.

As a concrete (although minuscule) example, consider the memory in Figure 8.12. Here, in the logical address, $n= 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0

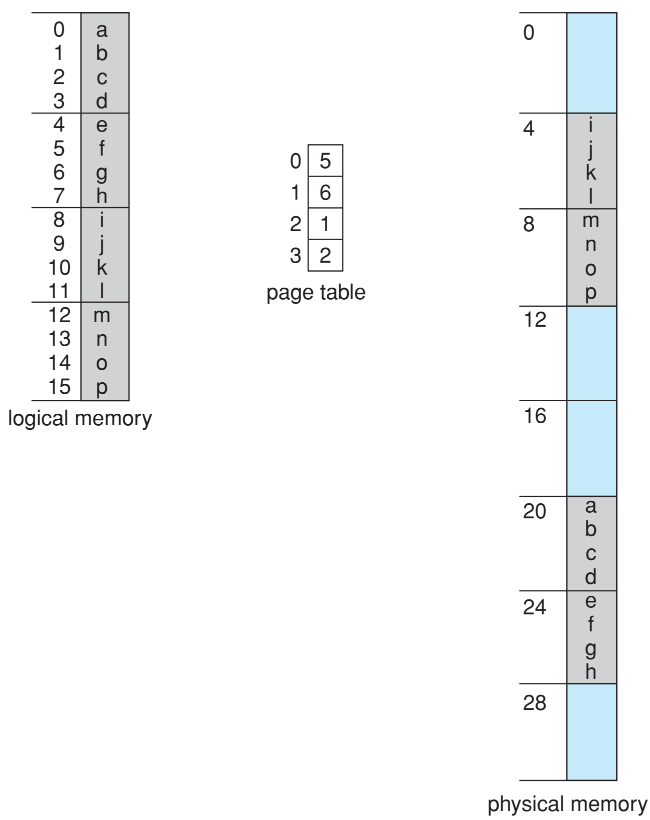


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.

> ***OBTAINING THE PAGE SIZE ON LINUX SYSTEMS***
>
> On a Linux system, the page size varies according to architecture, and there are several ways of obtaining the page size. One approach is to use the `getpagesize()` system call. Another strategy is to enter the following command on the command line:
>
> ```
> getconf PAGESIZE
> ```
>
> Each of these techniques returns the page size as a number of bytes.

is in frame 5. Thus, logical address 0 maps to physical address 20 [= $(5 \times 4) + 0$]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= $(5 \times 4) + 3$]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= $(6 \times 4) + 0$]. Logical address 13 maps to physical address 9.

You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes. In the worst case, a process would need n pages plus 1 byte. It would be allocated $n + 1$ frames, resulting in internal fragmentation of almost an entire frame.

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount data being transferred is larger (Chapter 10). Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages typically are between 4 KB and 8 KB in size, and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages. Researchers are now developing support for variable on-the-fly page size.

Frequently, on a 32-bit CPU, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 2^{32} physical page frames. If frame size is 4 KB (2^{12}), then a system with 4-byte entries can address 2^{44} bytes (or 16 TB) of physical memory. We should note here that the size of physical memory in a paged memory system is different from the maximum logical size of a process. As we further explore paging, we introduce other information that must be kept in the page-table entries. That information reduces the number

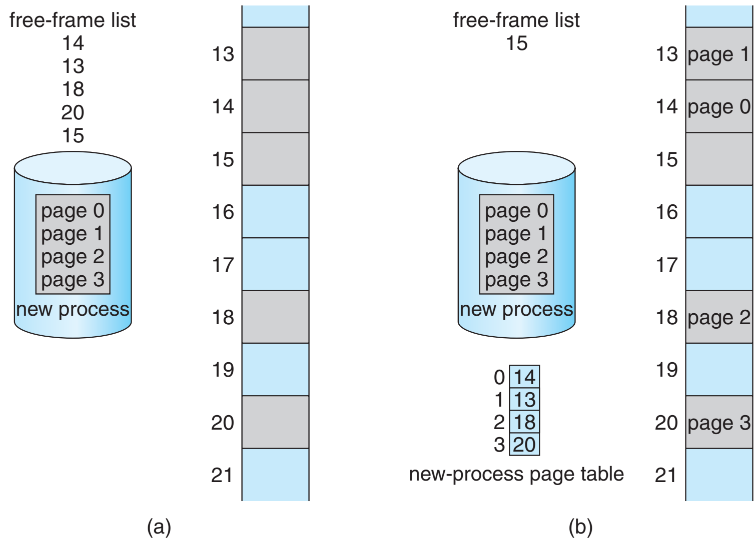


Figure 8.13 Free frames (a) before allocation and (b) after allocation.

of bits available to address page frames. Thus, a system with 32-bit page-table entries may address less physical memory than the possible maximum. A 32-bit CPU uses 32-bit addresses, meaning that a given process space can only be 2^{32} bytes (4 TB). Therefore, paging lets us use physical memory that is larger than what can be addressed by the CPU's address pointer length.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on (Figure 8.13).

An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the programmer's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the programmer and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a **frame table**. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

8.5.2 Hardware Support

Each operating system has its own methods for storing page tables. Some allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table. Other operating systems provide one or at most a few page tables, which decreases the overhead involved when processes are context-switched.

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated **registers**. These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map. The DEC PDP-11 is an example of such an architecture. The address consists of 16 bits, and the page size is 8 KB. The page table thus consists of eight entries that are kept in fast registers.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register** (**PTBR**) points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

The problem with this approach is the time required to access a user memory location. If we want to access location *i*, we must first index into the page table, using the value in the PTBR offset by the page number for *i*. This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, ***two*** memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances. We might as well resort to swapping!

The standard solution to this problem is to use a special, small, fast-lookup hardware cache called a **translation look-aside buffer** (**TLB**). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size. Some CPUs implement separate instruction and data address TLBs. That can double the number of TLB entries available, because those lookups occur in different pipeline steps. We can see in this development an example of the evolution of CPU technology: systems have evolved from having no TLBs to having multiple levels of TLBs, just as they have multiple levels of caches.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made. Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system. When the frame number is obtained, we can use it to access memory (Figure 8.14). In addition, we add the page number and frame number to the TLB, so

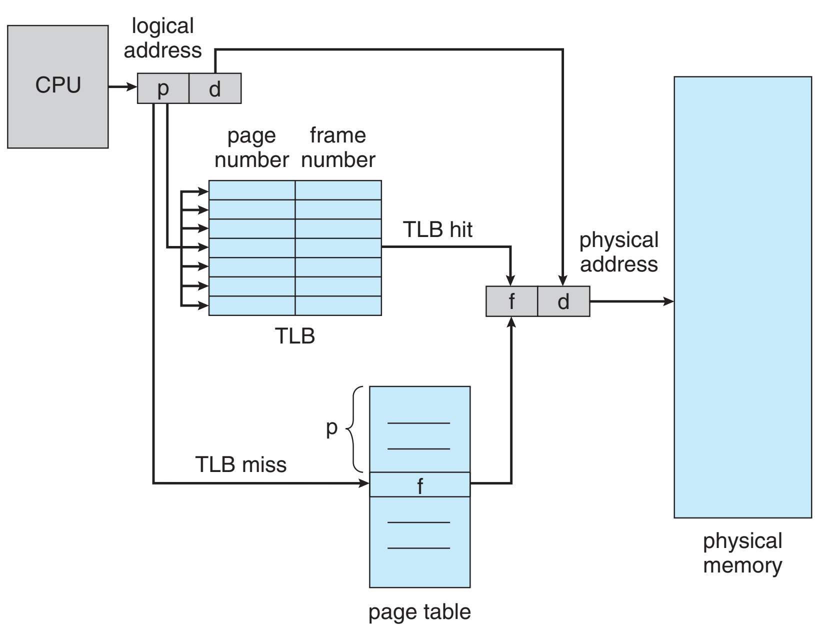


Figure 8.14 Paging hardware with TLB.

that they will be found quickly on the next reference. If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random. Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves. Furthermore, some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.) To find the **effective memory-access time**, we weight the case by its probability:

$$
\begin{aligned}
\text{effective access time} &= 0.80 \times 100 + 0.20 \times 200 \\
&= 120 \text{ nanoseconds}
\end{aligned}
$$

In this example, we suffer a 20-percent slowdown in average memory-access time (from 100 to 120 nanoseconds).

For a 99-percent hit ratio, which is much more realistic, we have

$$
\begin{aligned}
\text{effective access time} &= 0.99 \times 100 + 0.01 \times 200 \\
&= 101 \text{ nanoseconds}
\end{aligned}
$$

This increased hit rate produces only a 1 percent slowdown in access time.

As we noted earlier, CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above. For instance, the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB. A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work.

A complete performance analysis of paging overhead in such a system would require miss-rate information about each TLB tier. We can see from the general information above, however, that hardware features can have a significant effect on memory performance and that operating-system improvements (such as paging) can result in and, in turn, be affected by hardware changes (such as TLBs). We will further explore the impact of the hit ratio on the TLB in Chapter 9.

TLBs are a hardware feature and therefore would seem to be of little concern to operating systems and their designers. But the designer needs to understand the function and features of TLBs, which vary by hardware platform. For optimal operation, an operating-system design for a given platform must implement paging according to the platform's TLB design. Likewise, a change in the TLB design (for example, between generations of Intel CPUs) may necessitate a change in the paging implementation of the operating systems that use it.

8.5.3 Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read–write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.

One additional bit is generally attached to each entry in the page table: a **valid–invalid** bit. When this bit is set to *valid,* the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to*invalid,* the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in Figure 8.15. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Notice that this scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This

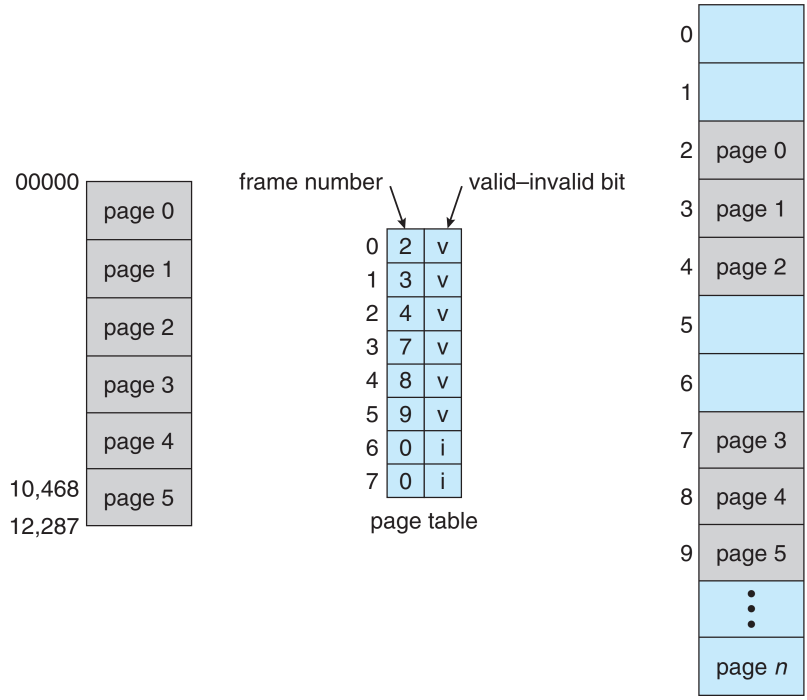


Figure 8.15 Valid (v) or invalid (i) bit in a page table.

problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.

Rarely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused but would take up valuable memory space. Some systems provide hardware, in the form of a **page-table length register** (**PTLR**), to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

8.5.4 Shared Pages

An advantage of paging is the possibility of ***sharing*** common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is **reentrant code** (or **pure code**), however, it can be shared, as shown in Figure 8.16. Here, we see three processes sharing a three-page editor—each page 50 KB in size (the large page size is used to simplify the figure). Each process has its own data page.

Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time.

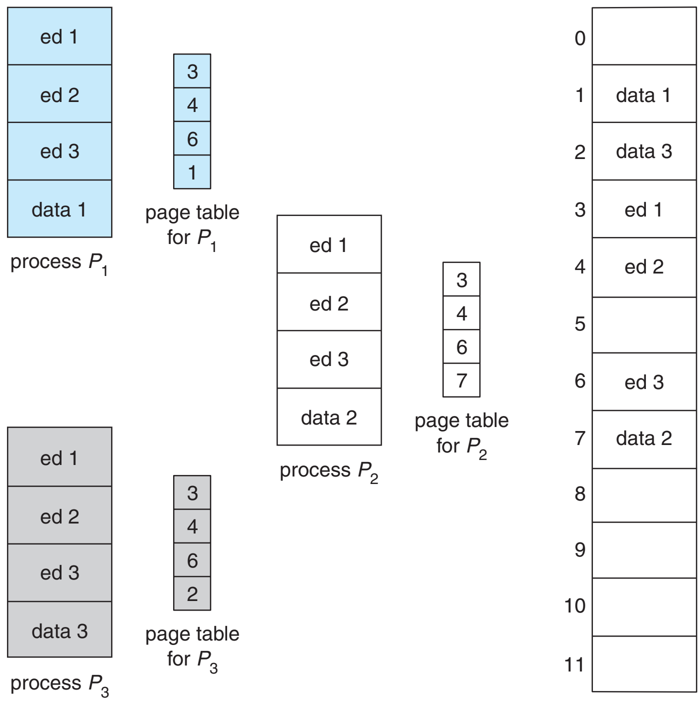


Figure 8.16 Sharing of code in a paging environment.

Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different.

Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings.

Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.

The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads, described in Chapter 4. Furthermore, recall that in Chapter 3 we described shared memory as a method of interprocess communication. Some operating systems implement shared memory using shared pages.

Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages. We cover several other benefits in Chapter 9.

8.6 Structure of the Page Table

In this section, we explore some of the most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

8.6.1 Hierarchical Paging

Most modern computer systems support a large logical address space (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries ($2^{32}/2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.

One way is to use a two-level paging algorithm, in which the page table itself is also paged (Figure 8.17). For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided

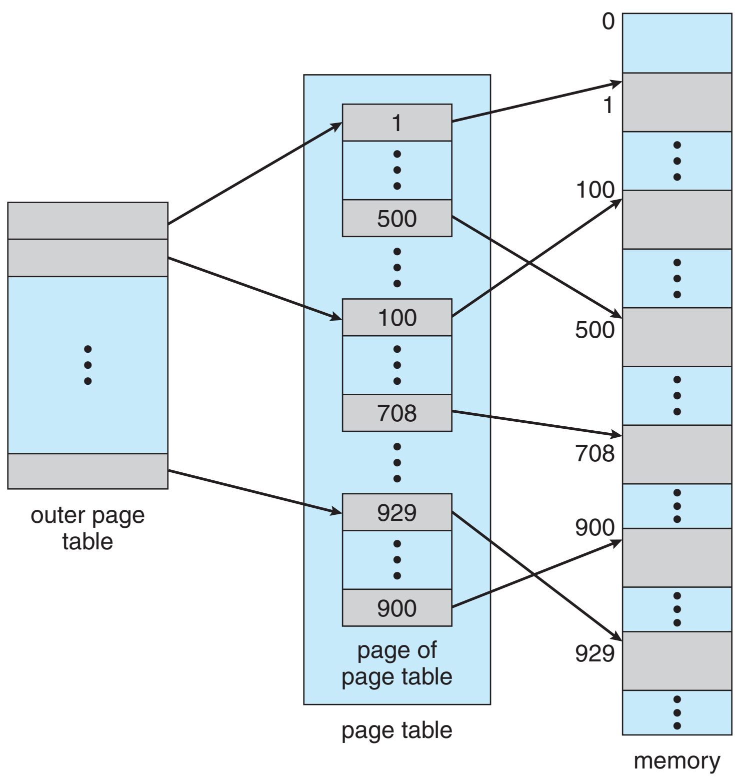


Figure 8.17 A two-level page-table scheme.

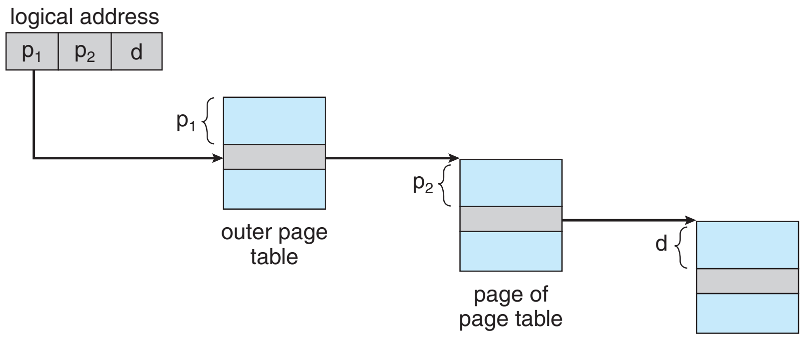


Figure 8.18 Address translation for a two-level 32-bit paging architecture.

into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table and p_2 is the displacement within the page of the inner page table. The address-translation method for this architecture is shown in Figure 8.18. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**.

Consider the memory management of one of the classic systems, the **VAX** minicomputer from **Digital Equipment Corporation** (**DEC**). The VAX was the most popular minicomputer of its time and was sold from 1977 through 2000. The VAX architecture supported a variation of two-level paging. The VAX is a 32-bit machine with a page size of 512 bytes. The logical address space of a process is divided into four equal sections, each of which consists of 2^{30} bytes. Each section represents a different part of the logical address space of a process. The first 2 high-order bits of the logical address designate the appropriate section. The next 21 bits represent the logical page number of that section, and the final 9 bits represent an offset in the desired page. By partitioning the page table in this manner, the operating system can leave partitions unused until a process needs them. Entire sections of virtual address space are frequently unused, and multilevel page tables have no entries for these spaces, greatly decreasing the amount of memory needed to store virtual memory data structures.

An address on the VAX architecture is as follows:

section	page	offset
s	p	d
2	21	9

where s designates the section number, p is an index into the page table, and d is the displacement within the page. Even when this scheme is used, the size of a one-level page table for a VAX process using one section is 2^{21} bits $*$ 4 bytes per entry = 8 MB. To further reduce main-memory use, the VAX pages the user-process page tables.

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let's suppose that the page size in such a system is 4 KB (2^{12}). In this case, the page table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses look like this:

outer page	inner page	offset
p_1	p_2	d
42	10	12

The outer page table consists of 2^{42} entries, or 2^{44} bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. (This approach is also used on some 32-bit processors for added flexibility and efficiency.)

We can divide the outer page table in various ways. For example, we can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes). In this case, a 64-bit address space is still daunting:

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

The outer page table is still 2^{34} bytes (16 GB) in size.

The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth. The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses—to translate each logical address. You can see from this example why, for 64-bit architectures, hierarchical page tables are generally considered inappropriate.

8.6.2 Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 8.19.

A variation of this scheme that is useful for 64-bit address spaces has been proposed. This variation uses **clustered page tables**, which are similar to

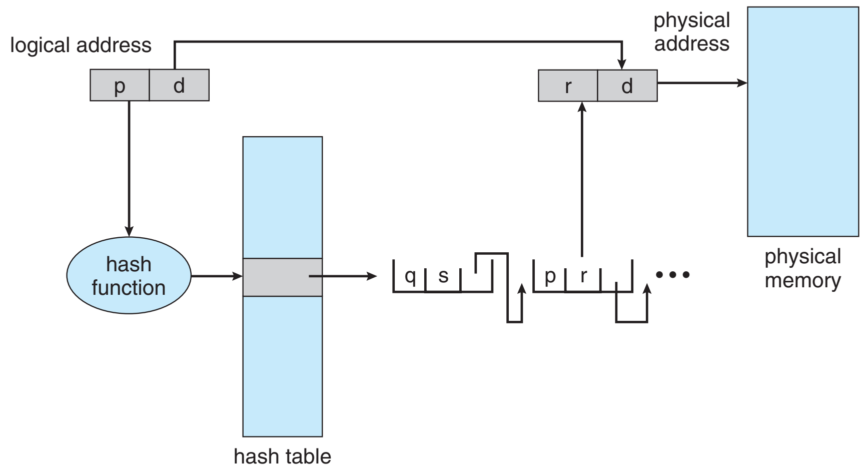


Figure 8.19 Hashed page table.

hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

8.6.3 Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

To solve this problem, we can use an **inverted page table**. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure 8.20 shows the operation of an inverted page table. Compare it with Figure 8.10, which depicts a standard page table in operation. Inverted page tables often require that an address-space identifier (Section 8.5.2) be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame. Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.

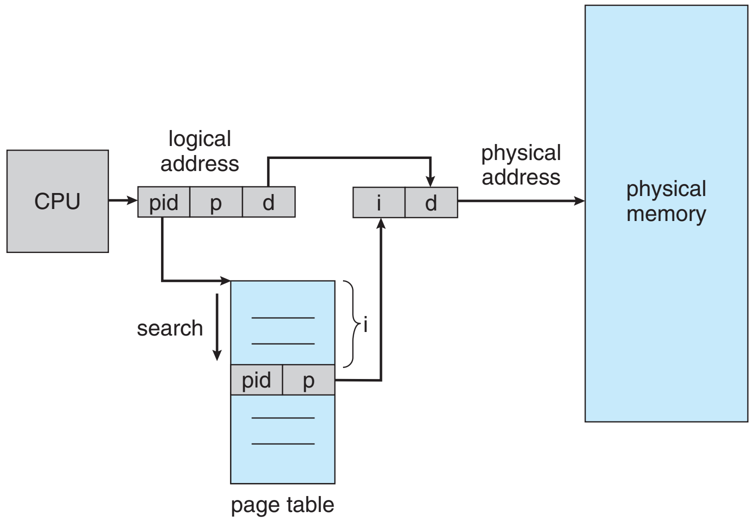


Figure 8.20 Inverted page table.

To illustrate this method, we describe a simplified version of the inverted page table used in the ***IBM RT***. IBM was the first major company to use inverted page tables, starting with the IBM System 38 and continuing through the RS/6000 and the current IBM Power CPUs. For the IBM RT, each virtual address in the system consists of a triple:

<process-id, page-number, offset>.

Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, page-number>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say, at entry *i*—then the physical address <*i*, offset> is generated. If no match is found, then an illegal address access has been attempted.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long. To alleviate this problem, we use a hash table, as described in Section 8.6.2, to limit the search to one—or at most a few—page-table entries. Of course, each access to the hash table adds a memory reference to the procedure, so one virtual memory reference requires at least two real memory reads—one for the hash-table entry and one for the page table. (Recall that the TLB is searched first, before the hash table is consulted, offering some performance improvement.)

Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as multiple virtual addresses (one for each process sharing the memory) that are mapped to one physical address. This standard method cannot be used with inverted page tables; because there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses. A simple technique for addressing this issue is to allow the page table to contain only one mapping of a virtual address to the shared physical address. This means that references to virtual addresses that are not mapped result in page faults.

8.6.4 Oracle SPARC Solaris

Consider as a final example a modern 64-bit CPU and operating system that are tightly integrated to provide low-overhead virtual memory. **Solaris** running on the **SPARC** CPU is a fully 64-bit operating system and as such has to solve the problem of virtual memory without using up all of its physical memory by keeping multiple levels of page tables. Its approach is a bit complex but solves the problem efficiently using hashed page tables. There are two hash tables—one for the kernel and one for all user processes. Each maps memory addresses from virtual to physical memory. Each hash-table entry represents a contiguous area of mapped virtual memory, which is more efficient than having a separate hash-table entry for each page. Each entry has a base address and a span indicating the number of pages the entry represents.

Virtual-to-physical translation would take too long if each address required searching through a hash table, so the CPU implements a TLB that holds translation table entries (TTEs) for fast hardware lookups. A cache of these TTEs reside in a translation storage buffer (TSB), which includes an entry per recently accessed page. When a virtual address reference occurs, the hardware searches the TLB for a translation. If none is found, the hardware walks through the in-memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup. This **TLB walk** functionality is found on many modern CPUs. If a match is found in the TSB, the CPU copies the TSB entry into the TLB, and the memory translation completes. If no match is found in the TSB, the kernel is interrupted to search the hash table. The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit. Finally, the interrupt handler returns control to the MMU, which completes the address translation and retrieves the requested byte or word from main memory.

8.7 Example: Intel 32 and 64-bit Architectures

The architecture of Intel chips has dominated the personal computer landscape for several years. The 16-bit Intel 8086 appeared in the late 1970s and was soon followed by another 16-bit chip—the Intel 8088—which was notable for being the chip used in the original IBM PC. Both the 8086 chip and the 8088 chip were based on a segmented architecture. Intel later produced a series of 32-bit chips—the IA-32—which included the family of 32-bit Pentium processors. The IA-32 architecture supported both paging and segmentation. More recently, Intel has produced a series of 64-bit chips based on the x86-64 architecture. Currently, all the most popular PC operating systems run on Intel chips, including Windows, Mac OS X, and Linux (although Linux, of course, runs on several other architectures as well). Notably, however, Intel's dominance has not spread to mobile systems, where the ARM architecture currently enjoys considerable success (see Section 8.8).

9.2 Demand Paging

Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially ***need*** the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for ***all*** options, regardless of whether or not an option is ultimately selected by the user. An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are thus never loaded into physical memory.

A demand-paging system is similar to a paging system with swapping (Figure 9.4) where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, though, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed. In the context of a demand-paging system, use of the term "swapper" is technically incorrect. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process. We thus use "pager," rather than "swapper," in connection with demand paging.

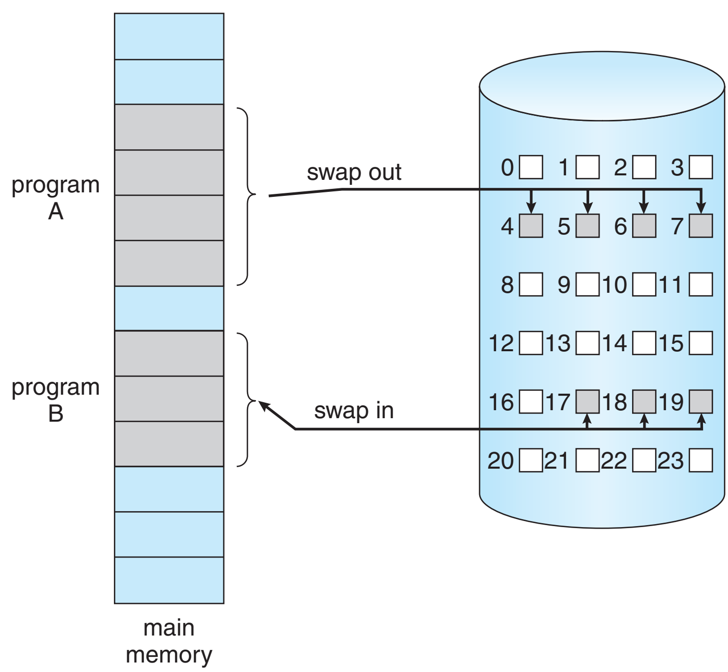


Figure 9.4 Transfer of a paged memory to contiguous disk space.

9.2.1 Basic Concepts

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid–invalid bit scheme described in Section 8.5.3 can be used for this purpose. This time, however, when this bit is set to "valid," the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. This situation is depicted in Figure 9.5.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all pages that are actually needed and only those pages, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are **memory resident**, execution proceeds normally.

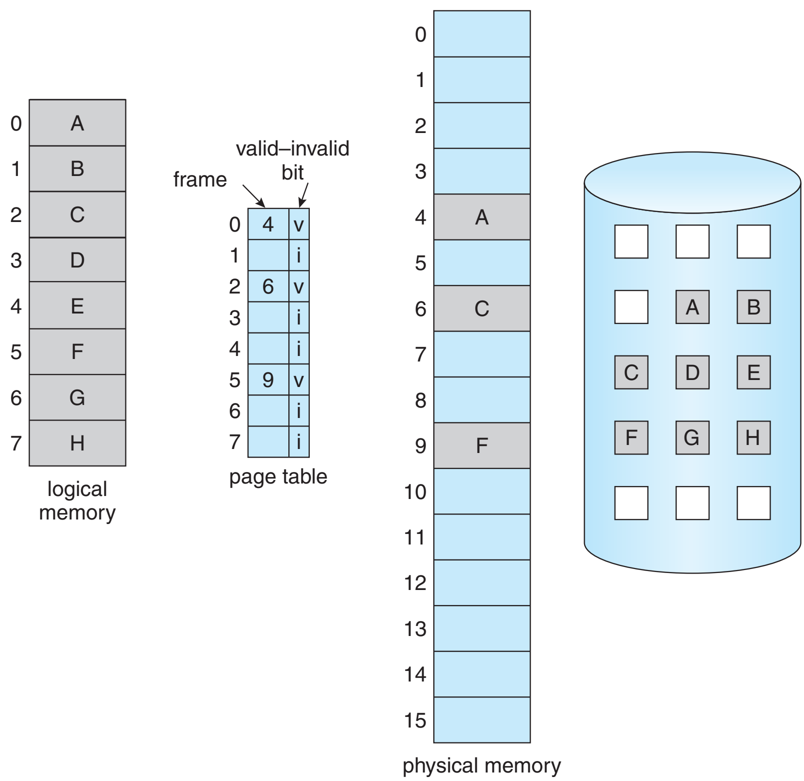


Figure 9.5 Page table when some pages are not in main memory.

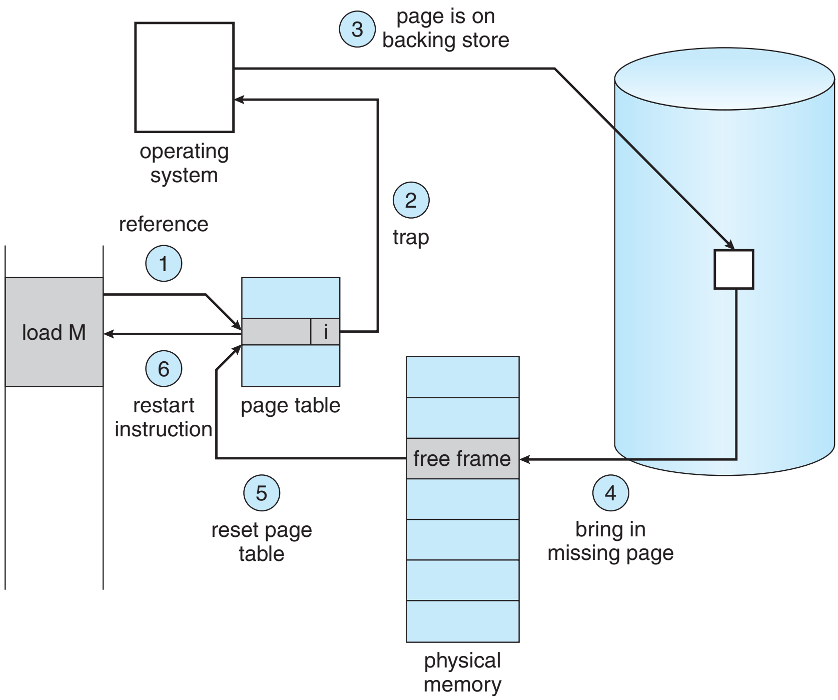


Figure 9.6 Steps in handling a page fault.

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page fault**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure 9.6):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with ***no*** pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging**: never bring a page into memory until it is required.

Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behavior is exceedingly unlikely. Programs tend to have **locality of reference**, described in Section 9.6.1, which results in reasonable performance from demand paging.

The hardware to support demand paging is the same as the hardware for paging and swapping:

- **Page table**. This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.
- **Secondary memory**. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space**. Swap-space allocation is discussed in Chapter 10.

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in ***exactly*** the same place and state, except that the desired page is now in memory and is accessible. In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

As a worst-case example, consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

If we fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs.

The major difficulty arises when one instruction may modify several different locations. For example, consider the IBM System 360/370 MVC (move character) instruction, which can move up to 256 bytes from one location to another (possibly overlapping) location. If either block (source or destination) straddles a page boundary, a page fault might occur after the move is partially done. In addition, if the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction.

This problem can be solved in two different ways. In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place; we know that no page fault can occur, since all the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.

This is by no means the only architectural problem resulting from adding paging to an existing architecture to allow demand paging, but it illustrates some of the difficulties involved. Paging is added between the CPU and the memory in a computer system. It should be entirely transparent to the user process. Thus, people often assume that paging can be added to any system. Although this assumption is true for a non-demand-paging environment, where a page fault represents a fatal error, it is not true where a page fault means only that an additional page must be brought into memory and the process restarted.

9.2.2 Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the **effective access time** for a demand-paged memory. For most computer systems, the memory-access time, denoted *ma*, ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let p be the probability of a page fault ($0 \le p \le 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults. The **effective access time** is then

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}.$$

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.

3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
    a. Wait in a queue for this device until the read request is serviced.
    b. Wait for the device seek and/or latency time.
    c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Not all of these steps are necessary in every case. For example, we are assuming that, in step 6, the CPU is allocated to another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization but requires additional time to resume the page-fault service routine when the I/O transfer is complete.

In any case, we are faced with three major components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page-switch time, however, will probably be close to 8 milliseconds. (A typical hard disk has an average latency of 3 milliseconds, a seek of 5 milliseconds, and a transfer time of 0.05 milliseconds. Thus, the total paging time is about 8 milliseconds, including hardware and software time.) Remember also that we are looking at only the device-service time. If a queue of processes is waiting for the device, we have to add device-queueing time as we wait for the paging device to be free to service our request, increasing even more the time to swap.

With an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, the effective access time in nanoseconds is

$$\begin{aligned}\text{effective access time} &= (1 - p) \times (200) + p \text{ (8 milliseconds)}\\ &= (1 - p) \times 200 + p \times 8{,}000{,}000\\ &= 200 + 7{,}999{,}800 \times p.\end{aligned}$$

We see, then, that the effective access time is directly proportional to the **page-fault rate**. If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. The computer will be slowed down by a factor of 40 because of demand paging! If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:

$$\begin{aligned}&220 > 200 + 7{,}999{,}800 \times p,\\ &20 > 7{,}999{,}800 \times p,\\ &p < 0.0000025.\end{aligned}$$

That is, to keep the slowdown due to paging at a reasonable level, we can allow fewer than one memory access out of 399,990 to page-fault. In sum, it is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

An additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is a faster file system because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used (Chapter 10). The system can therefore gain better paging throughput by copying an entire file image into the swap space at process startup and then performing demand paging from the swap space. Another option is to demand pages from the file system initially but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are read from the file system but that all subsequent paging is done from swap space.

Some systems attempt to limit the amount of swap space used through demand paging of binary files. Demand pages for such files are brought directly from the file system. However, when page replacement is called for, these frames can simply be overwritten (because they are never modified), and the pages can be read in from the file system again if needed. Using this approach, the file system itself serves as the backing store. However, swap space must still be used for pages not associated with a file (known as **anonymous memory**); these pages include the stack and heap for a process. This method appears to be a good compromise and is used in several systems, including Solaris and BSD UNIX.

Mobile operating systems typically do not support swapping. Instead, these systems demand-page from the file system and reclaim read-only pages (such as code) from applications if memory becomes constrained. Such data can be demand-paged from the file system if it is later needed. Under iOS, anonymous memory pages are never reclaimed from an application unless the application is terminated or explicitly releases the memory.

CHAPTER 11

File-System Interface

For most users, the file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system. File systems live on devices, which we described in the preceding chapter and will continue to discuss in the following one. In this chapter, we consider the various aspects of files and the major directory structures. We also discuss the semantics of sharing files among multiple processes, users, and computers. Finally, we discuss ways to handle file protection, necessary when we have multiple users and we want to control who may access files and how files may be accessed.

CHAPTER OBJECTIVES

- To explain the function of file systems.
- To describe the interfaces to file systems.
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- To explore file-system protection.

11.1 File Concept

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of stored information. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent between system reboots.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

The information in a file is defined by its creator. Many different types of information may be stored in a file—source or executable programs, numeric or text data, photos, music, video, and so on. A file has a certain defined structure, which depends on its type. A **text file** is a sequence of characters organized into lines (and possibly pages). A **source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements. An **executable file** is a series of code sections that the loader can bring into memory and execute.

11.1.1 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as `example.c`. Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file `example.c`, and another user might edit that file by specifying its name. The file's owner might write the file to a USB disk, send it as an e-mail attachment, or copy it across a network, and it could still be called `example.c` on the destination system.

A file's attributes vary from one operating system to another but typically consist of these:

- **Name**. The symbolic file name is the only information kept in human-readable form.
- **Identifier**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**. This information is needed for systems that support different types of files.
- **Location**. This information is a pointer to a device and to the location of the file on that device.
- **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification**. This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

Figure 11.1 A file info window on Mac OS X.

Some newer file systems also support **extended file attributes**, including character encoding of the file and security features such as a file checksum. Figure 11.1 illustrates a **file info window** on Mac OS X, which displays a file's attributes.

The information about all files is kept in the directory structure, which also resides on secondary storage. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this information for each file. In a system with many files, the size of the directory itself may be megabytes. Because directories, like files, must be nonvolatile, they must be stored on the device and brought into memory piecemeal, as needed.

11.1.2 File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these six basic file operations. It should then be easy to see how other similar operations, such as renaming a file, can be implemented.

- **Creating a file**. Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file in Chapter 12. Second, an entry for the new file must be made in the directory.
- **Writing a file**. To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a **write pointer** to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- **Reading a file**. To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a **read pointer** to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **current-file-position pointer**. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within a file**. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file **seek**.
- **Deleting a file**. To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file**. The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.

These six basic operations comprise the minimal set of required file operations. Other common operations include appending new information to the end of an existing file and renaming an existing file. These primitive operations can then be combined to perform other file operations. For instance, we can create a copy of a file—or copy the file to another I/O device, such as a printer or a display—by creating a new file and then reading from the old and writing to the new. We also want to have operations that allow a user to get and set the various attributes of a file. For example, we may want to have operations that allow a user to determine the status of a file, such as the file's length, and to set file attributes, such as the file's owner.

Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an `open()` system call be made before a file is first used. The operating system keeps a table, called the **open-file table**, containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required. When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table. `create()` and `delete()` are system calls that work with closed rather than open files.

Some systems implicitly open a file when the first reference to it is made. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that the programmer open a file explicitly with the `open()` system call before that file can be used. The `open()` operation takes a file name and searches the directory, copying the directory entry into the open-file table. The `open()` call can also accept access-mode information—create, read-only, read–write, append-only, and so on. This mode is checked against the file's permissions. If the request mode is allowed, the file is opened for the process. The `open()` system call typically returns a pointer to the entry in the open-file table. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching and simplifying the system-call interface.

The implementation of the `open()` and `close()` operations is more complicated in an environment where several processes may open the file simultaneously. This may occur in a system where several different applications open the same file at the same time. Typically, the operating system uses two levels of internal tables: a per-process table and a system-wide table. The per-process table tracks all files that a process has open. Stored in this table is information regarding the process's use of the file. For instance, the current file pointer for each file is found here. Access rights to the file and accounting information can also be included.

Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an `open()` call, a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system-wide table. Typically, the open-file table also has an **open count** associated with each file to indicate how many processes have the file open. Each `close()` decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

In summary, several pieces of information are associated with an open file.

- **File pointer**. On systems that do not include a file offset as part of the `read()` and `write()` system calls, the system must track the last read–write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
- **File-open count**. As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Multiple processes may have opened a file, and the system must wait for the last file to close before removing the open-file table entry. The file-open count tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Disk location of the file**. Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
- **Access rights**. Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

Some operating systems provide facilities for locking an open file (or sections of a file). File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes—for example, a system log file that can be accessed and modified by a number of processes in the system.

File locks provide functionality similar to reader–writer locks, covered in Section 5.7.2. A **shared lock** is akin to a reader lock in that several processes can acquire the lock concurrently. An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock. It is important to note that not all operating systems provide both types of locks: some systems only provide exclusive file locking.

FILE LOCKING IN JAVA

In the Java API, acquiring a lock requires first obtaining the `FileChannel` for the file to be locked. The `lock()` method of the `FileChannel` is used to acquire the lock. The API of the `lock()` method is

```
FileLock lock(long begin, long end, boolean shared)
```

where `begin` and `end` are the beginning and ending positions of the region being locked. Setting `shared` to `true` is for shared locks; setting `shared` to `false` acquires the lock exclusively. The lock is released by invoking the `release()` of the `FileLock` returned by the `lock()` operation.

The program in Figure 11.2 illustrates file locking in Java. This program acquires two locks on the file `file.txt`. The first half of the file is acquired as an exclusive lock; the lock for the second half is a shared lock.

FILE LOCKING IN JAVA (Continued)

```
import java.io.*;
import java.nio.channels.*;

public class LockingExample {
  public static final boolean EXCLUSIVE = false;
  public static final boolean SHARED = true;

  public static void main(String args[]) throws IOException {
    FileLock sharedLock = null;
    FileLock exclusiveLock = null;

    try {
      RandomAccessFile raf = new RandomAccessFile("file.txt","rw");

      // get the channel for the file
      FileChannel ch = raf.getChannel();

      // this locks the first half of the file - exclusive
      exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);

      /** Now modify the data . . . */

      // release the lock
      exclusiveLock.release();

      // this locks the second half of the file - shared
      sharedLock = ch.lock(raf.length()/2+1,raf.length(),SHARED);

      /** Now read the data . . . */

      // release the lock
      sharedLock.release();
    } catch (java.io.IOException ioe) {
      System.err.println(ioe);
    }
    finally {
      if (exclusiveLock != null)
            exclusiveLock.release();
      if (sharedLock != null)
            sharedLock.release();
    }
  }
}
```

Figure 11.2 File-locking example in Java.

Furthermore, operating systems may provide either **mandatory** or **advisory** file-locking mechanisms. If a lock is mandatory, then once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file. For example, assume a process acquires an exclusive lock on the file `system.log`. If we attempt to open `system.log` from another process—for example, a text editor—the operating system will prevent access until the exclusive lock is released. This occurs even if the text editor is not written explicitly to acquire the lock. Alternatively, if the lock is advisory, then the operating system will not prevent the text editor from acquiring access to `system.log`. Rather, the text editor must be written so that it manually acquires the lock before accessing the file. In other words, if the locking scheme is mandatory, the operating system ensures locking integrity. For advisory locking, it is up to software developers to ensure that locks are appropriately acquired and released. As a general rule, Windows operating systems adopt mandatory locking, and UNIX systems employ advisory locks.

The use of file locks requires the same precautions as ordinary process synchronization. For example, programmers developing on systems with mandatory locking must be careful to hold exclusive file locks only while they are accessing the file. Otherwise, they will prevent other processes from accessing the file as well. Furthermore, some measures must be taken to ensure that two or more processes do not become involved in a deadlock while trying to acquire file locks.

11.1.3 File Types

When we design a file system—indeed, an entire operating system—we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to output the binary-object form of a program. This attempt normally produces garbage; however, the attempt can succeed if the operating system has been told that the file is a binary-object program.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an extension, usually separated by a period (Figure 11.3). In this way, the user and the operating system can tell from the name alone what the type of a file is. Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters. Examples include `resume.docx`, `server.c`, and `ReaderThread.cpp`.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a `.com`, `.exe`, or `.sh` extension can be executed, for instance. The `.com` and `.exe` files are two forms of binary executable files, whereas the `.sh` file is a **shell script** containing, in ASCII format, commands to the operating system. Application programs also use extensions to indicate file types in which they are interested. For example, Java compilers expect source files to have a `.java` extension, and the Microsoft Word word processor expects its files to end with a `.doc` or `.docx` extension. These extensions are not always required, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered “hints” to the applications that operate on them.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure 11.3 Common file types.

Consider, too, the Mac OS X operating system. In this system, each file has a type, such as `.app` (for application). Each file also has a creator attribute containing the name of the program that created it. This attribute is set by the operating system during the `create()` call, so its use is enforced and supported by the system. For instance, a file produced by a word processor has the word processor's name as its creator. When the user opens that file, by double-clicking the mouse on the icon representing the file, the word processor is invoked automatically and the file is loaded, ready to be edited.

The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file—executable program, shell script, PDF file, and so on. Not all files have magic numbers, so system features cannot be based solely on this information. UNIX does not record the name of the creating program, either. UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the operating system; they are meant mostly to aid users in determining what type of contents the file contains. Extensions can be used or ignored by a given application, but that is up to the application's programmer.

11.1.4 File Structure

File types also can be used to indicate the internal structure of the file. As mentioned in Section 11.1.3, source and object files have structures that match the expectations of the programs that read them. Further, certain files must conform to a required structure that is understood by the operating system. For example, the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures.

This point brings us to one of the disadvantages of having the operating system support multiple file structures: the resulting size of the operating system is cumbersome. If the operating system defines five different file structures, it needs to contain the code to support these file structures. In addition, it may be necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by the operating system, severe problems may result.

For example, assume that a system supports two types of files: text files (composed of ASCII characters separated by a carriage return and line feed) and executable binary files. Now, if we (as users) want to define an encrypted file to protect the contents from being read by unauthorized people, we may find neither file type to be appropriate. The encrypted file is not ASCII text lines but rather is (apparently) random bits. Although it may appear to be a binary file, it is not executable. As a result, we may have to circumvent or misuse the operating system's file-type mechanism or abandon our encryption scheme.

Some operating systems impose (and support) a minimal number of file structures. This approach has been adopted in UNIX, Windows, and others. UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility but little support. Each application program must include its own code to interpret an input file as to the appropriate structure. However, all operating systems must support at least one structure—that of an executable file—so that the system is able to load and run programs.

11.1.5 Internal File Structure

Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

For example, the UNIX operating system defines all files to be simply streams of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record size is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks—say, 512 bytes per block—as necessary.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks. All the basic I/O

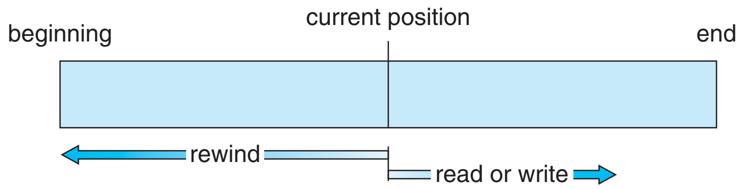


Figure 11.4 Sequential-access file.

functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.

Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The waste incurred to keep everything in units of blocks (instead of bytes) is internal fragmentation. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

11.2 Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. while others support many access methods, and choosing the right one for a particular application is a major design problem.

11.2.1 Sequential Access

The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

Reads and writes make up the bulk of the operations on a file. A read operation—`read_next()`—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation—`write_next()`—appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and on some systems, a program may be able to skip forward or backward n records for some integer n—perhaps only for $n = 1$. Sequential access, which is depicted in Figure 11.4, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

11.2.2 Direct Access

Another method is **direct access** (or **relative access**). Here, a file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's names or search a small in-memory index to determine a block to read and search.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have `read(n)`, where n is the block number, rather than `read_next()`, and `write(n)` rather than `write_next()`. An alternative approach is to retain `read_next()` and `write_next()`, as with sequential access, and to add an operation `position_file(n)` where n is the block number. Then, to effect a `read(n)`, we would `position_file(n)` and then `read_next()`.

The block number provided by the user to the operating system is normally a **relative block number**. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the absolute disk address may be 14703 for the first block and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed (called the **allocation problem**, as we discuss in Chapter 12) and helps to prevent the user from accessing portions of the file system that may not be part of her file. Some systems start their relative block numbers at 0; others start at 1.

How, then, does the system satisfy a request for record N in a file? Assuming we have a logical record length L, the request for record N is turned into an I/O request for L bytes starting at location $L * (N)$ within the file (assuming the first record is $N = 0$). Since logical records are of a fixed size, it is also easy to read, write, or delete a record.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created. Such a file can be accessed only in a manner consistent with its declaration. We can easily simulate sequential access on a direct-access file by simply keeping a variable *cp* that defines our current position, as shown in Figure 11.5. Simulating a direct-access file on a sequential-access file, however, is extremely inefficient and clumsy.

11.2.3 Other Access Methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The **index**, like an index in the back of a book, contains pointers to the various blocks. To

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

Figure 11.5 Simulation of sequential access on a direct-access file.

find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists of a 10-digit UPC and a 6-digit price, for a 16-byte record. If our disk has 1,024 bytes per block, we can store 64 records per block. A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory. To find the price of a particular item, we can make a binary search of the index. From this search, we learn exactly which block contains the desired record and access that block. This structure allows us to search a large file doing little I/O.

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.

For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. Figure 11.6 shows a similar situation as implemented by VMS index and relative files.

11.3 Directory and Disk Structure

Next, we consider how to store files. Certainly, no general-purpose computer stores just one file. There are typically thousands, millions, even billions of files within a computer. Files are stored on random-access storage devices, including hard disks, optical disks, and solid-state (memory-based) disks.

A storage device can be used in its entirety for a file system. It can also be subdivided for finer-grained control. For example, a disk can be **partitioned** into quarters, and each quarter can hold a separate file system. Storage devices can also be collected together into RAID sets that provide protection from the failure of a single disk (as described in Section 10.7). Sometimes, disks are subdivided and also collected into RAID sets.

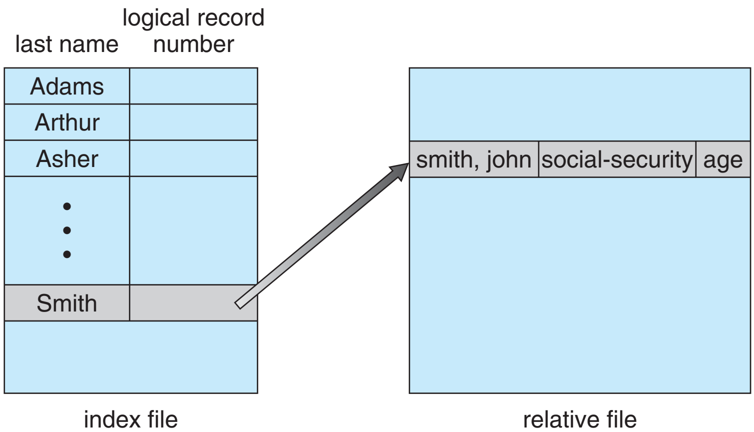


Figure 11.6 Example of index and relative files.

Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses, such as swap space or unformatted (raw) disk space. A file system can be created on each of these parts of the disk. Any entity containing a file system is generally known as a **volume**. The volume may be a subset of a device, a whole device, or multiple devices linked together into a RAID set. Each volume can be thought of as a virtual disk. Volumes can also store multiple operating systems, allowing a system to boot and run more than one operating system.

Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents**. The device directory (more commonly known simply as the **directory**) records information—such as name, location, size, and type—for all files on that volume. Figure 11.7 shows a typical file-system organization.

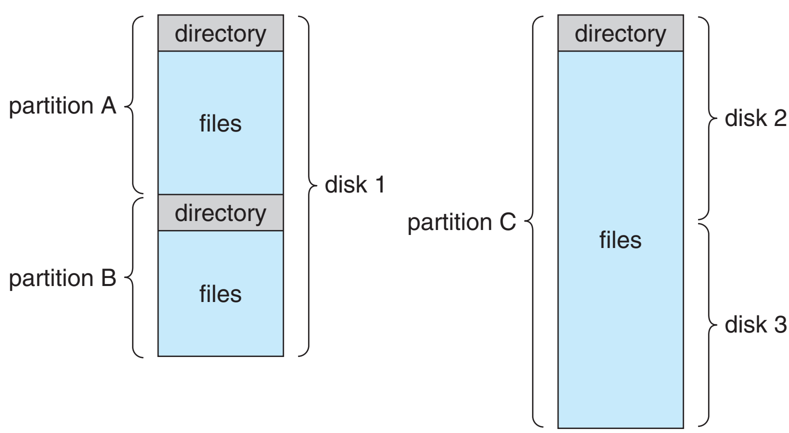


Figure 11.7 A typical file-system organization.

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

Figure 11.8 Solaris file systems.

11.3.1 Storage Structure

As we have just seen, a general-purpose computer system has multiple storage devices, and those devices can be sliced up into volumes that hold file systems. Computer systems may have zero or more file systems, and the file systems may be of varying types. For example, a typical Solaris system may have dozens of file systems of a dozen different types, as shown in the file system list in Figure 11.8.

In this book, we consider only general-purpose file systems. It is worth noting, though, that there are many special-purpose file systems. Consider the types of file systems in the Solaris example mentioned above:

- **tmpfs**—a "temporary" file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **objfs**—a "virtual" file system (essentially an interface to the kernel that looks like a file system) that gives debuggers access to kernel symbols
- **ctfs**—a virtual file system that maintains "contract" information to manage which processes start when the system boots and must continue to run during operation
- **lofs**—a "loop back" file system that allows one file system to be accessed in place of another one
- **procfs**—a virtual file system that presents information on all processes as a file system
- **ufs, zfs**—general-purpose file systems

The file systems of computers, then, can be extensive. Even within a file system, it is useful to segregate files into groups and manage and act on those groups. This organization involves the use of directories. In the remainder of this section, we explore the topic of directory structure.

11.3.2 Directory Overview

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In this section, we examine several schemes for defining the logical structure of the directory system.

When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.
- **Create a file**. New files need to be created and added to the directory.
- **Delete a file**. When a file is no longer needed, we want to be able to remove it from the directory.
- **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

In the following sections, we describe the most common schemes for defining the logical structure of a directory.

11.3.3 Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 11.9).

A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call

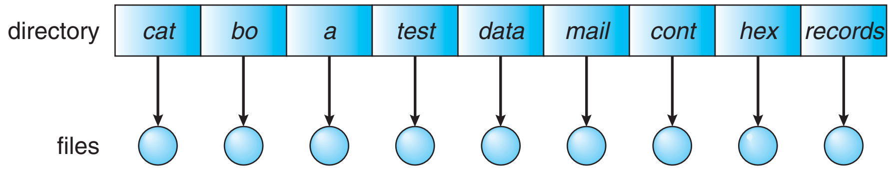


Figure 11.9 Single-level directory.

their data file `test.txt`, then the unique-name rule is violated. For example, in one programming class, 23 students called the program for their second assignment `prog2.c`; another 11 called it `assign2.c`. Fortunately, most file systems support file names of up to 255 characters, so it is relatively easy to select unique file names.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

11.3.4 Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has his own **user file directory** (**UFD**). The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's **master file directory** (**MFD**) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user (Figure 11.10).

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

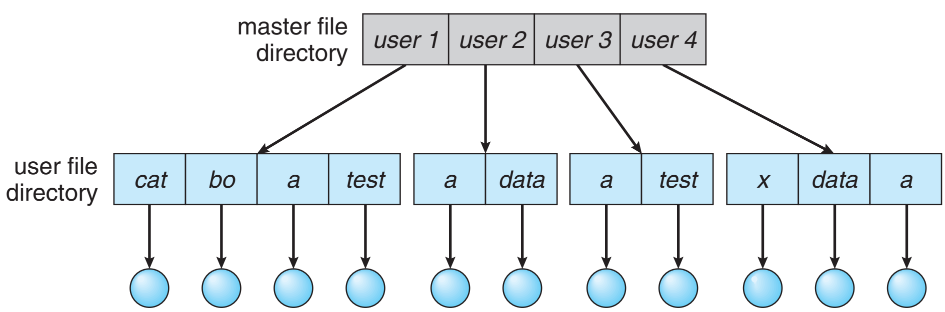


Figure 11.10 Two-level directory structure.

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new UFD and adds an entry for it to the MFD. The execution of this program might be restricted to system administrators. The allocation of disk space for user directories can be handled with the techniques discussed in Chapter 12 for files themselves.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file). Thus, a user name and a file name define a **path name**. Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

For example, if user A wishes to access her own test file named `test.txt`, she can simply refer to `test.txt`. To access the file named `test.txt` of user B (with directory-entry name `userb`), however, she might have to refer to `/userb/test.txt`. Every system has its own syntax for naming files in directories other than the user's own.

Additional syntax is needed to specify the volume of a file. For instance, in Windows a volume is specified by a letter followed by a colon. Thus, a file specification might be `C:\userb\test`. Some systems go even further and separate the volume, directory name, and file name parts of the specification. In VMS, for instance, the file `login.com` might be specified as: `u:[sst.jdeck]login.com;1`, where `u` is the name of the volume, `sst` is the name of the directory, `jdeck` is the name of the subdirectory, and `1` is the version number. Other systems—such as UNIX and Linux—simply treat the volume name as part of the directory name. The first name given is that of the volume, and the rest is the directory and file. For instance, `/u/pbg/test` might specify volume `u`, directory `pbg`, and file `test`.

A special instance of this situation occurs with the system files. Programs provided as part of the system—loaders, assemblers, compilers, utility routines, libraries, and so on—are generally defined as files. When the appropriate commands are given to the operating system, these files are read by the loader and executed. Many command interpreters simply treat such a command as the name of a file to load and execute. In the directory system as we defined it above, this file name would be searched for in the current UFD. One solution would be to copy the system files into each UFD. However, copying all the system files would waste an enormous amount of space. (If the system files require 5 MB, then supporting 12 users would require $5 \times 12 = 60$ MB just for copies of the system files.)

The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files (for example, user 0). Whenever a file name is given to be loaded, the operating system first searches the local UFD. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the **search path**. The search path can be extended to contain an unlimited list of directories to search when a command name is given. This method is the one most used in UNIX and Windows. Systems can also be designed so that each user has his own search path.

11.3.5 Tree-Structured Directories

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (Figure 11.11). This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.

In normal use, each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must

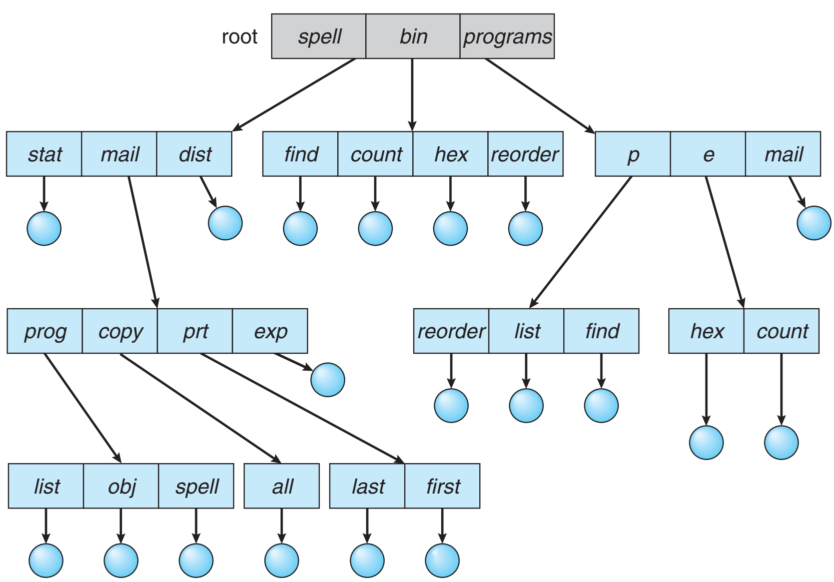


Figure 11.11 Tree-structured directory structure.

either specify a path name or change the current directory to be the directory holding that file. To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory. Thus, the user can change her current directory whenever she wants. From one `change_directory()` system call to the next, all `open()` system calls search the current directory for the specified file. Note that the search path may or may not contain a special entry that stands for "the current directory."

The initial current directory of a user's login shell is designated when the user job starts or the user logs in. The operating system searches the accounting file (or some other predefined location) to find an entry for this user (for accounting purposes). In the accounting file is a pointer to (or the name of) the user's initial directory. This pointer is copied to a local variable for this user that specifies the user's initial current directory. From that shell, other processes can be spawned. The current directory of any subprocess is usually the current directory of the parent when it was spawned.

Path names can be of two types: absolute and relative. An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path. A **relative path name** defines a path from the current directory. For example, in the tree-structured file system of Figure 11.11, if the current directory is `root/spell/mail`, then the relative path name `prt/first` refers to the same file as does the absolute path name `root/spell/mail/prt/first`.

Allowing a user to define her own subdirectories permits her to impose a structure on her files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information (for example, the directory `programs` may contain source programs; the directory `bin` may store all the binaries).

An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory. If a directory is empty, its entry in the directory that contains it can simply be deleted. However, suppose the directory to be deleted is not empty but contains several files or subdirectories. One of two approaches can be taken. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also. This approach can result in a substantial amount of work. An alternative approach, such as that taken by the UNIX `rm` command, is to provide an option: when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but it is also more dangerous, because an entire directory structure can be removed with one command. If that command is issued in error, a large number of files and directories will need to be restored (assuming a backup exists).

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users. For example, user B can access a file of user A by specifying its path names. User B can specify either an absolute or a relative path name. Alternatively, user B can change her current directory to be user A's directory and access the file by its file names.

11.3.6 Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. In this situation, the common subdirectory should be ***shared.*** A shared directory or file exists in the file system in two (or more) places at once.

A tree structure prohibits the sharing of files or directories. An **acyclic graph** —that is, a graph with no cycles—allows directories to share subdirectories and files (Figure 11.12). The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

When people are working as a team, all the files they want to share can be put into one directory. The UFD of each team member will contain this directory of shared files as a subdirectory. Even in the case of a single user, the user's file organization may require that some file be placed in different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.

Shared files and subdirectories can be implemented in several ways. A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a link. A **link** is effectively a pointer to another file

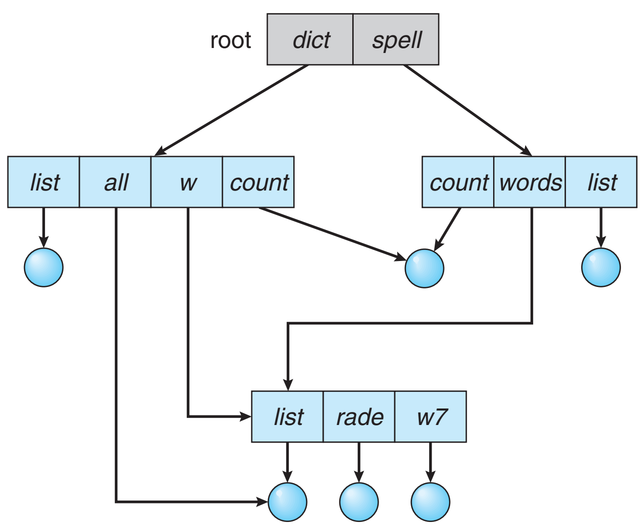


Figure 11.12 Acyclic-graph directory structure.

or subdirectory. For example, a link may be implemented as an absolute or a relative path name. When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information. We **resolve** the link by using that path name to locate the real file. Links are easily identified by their format in the directory entry (or by having a special type on systems that support types) and are effectively indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. Consider the difference between this approach and the creation of a link. The link is clearly different from the original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable. A major problem with duplicate directory entries is maintaining consistency when a file is modified.

An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex. Several problems must be considered carefully. A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system—to find a file, to accumulate statistics on all files, or to copy all files to backup storage—this problem becomes significant, since we do not want to traverse shared structures more than once.

Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link need not affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them as well, but unless a list of the associated links is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist and can fail to resolve the link name; the access is treated just as with any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is deleted and another file of the same name is created, before a symbolic link to the original file is used.) In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced. Microsoft Windows uses the same approach.

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list—we need to keep only a count of the number of references. Adding a new link or directory entry increments the reference count. Deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for nonsymbolic links (or **hard links**), keeping a reference count in the file information block (or inode; see Section A.7.2). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid problems such as the ones just discussed, some systems simply do not allow shared directories or links.

11.3.7 General Graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links, the tree structure is destroyed, resulting in a simple graph structure (Figure 11.13).

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file without finding it, we want to avoid searching that subdirectory again; the second search would be a waste of time.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution

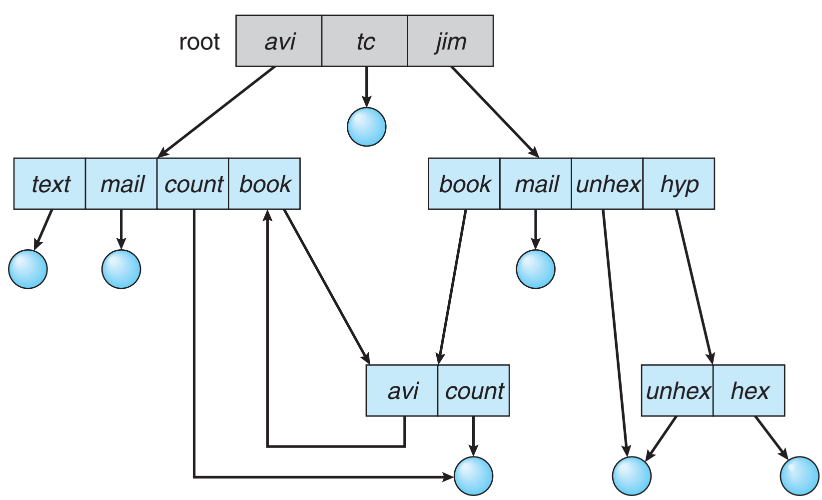


Figure 11.13 General graph directory.

is to limit arbitrarily the number of directories that will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (or a cycle) in the directory structure. In this case, we generally need to use a **garbage collection** scheme to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. (A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.) Garbage collection for a disk-based file system, however, is extremely time consuming and is thus seldom attempted.

Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles as new links are added to the structure. How do we know when a new link will complete a cycle? There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage. A simpler algorithm in the special case of directories and links is to bypass links during directory traversal. Cycles are avoided, and no extra overhead is incurred.

11.4 File-System Mounting

Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system. More specifically, the directory structure may be built out of multiple volumes, which must be mounted to make them available within the file-system name space.

The mount procedure is straightforward. The operating system is given the name of the device and the **mount point**—the location within the file structure where the file system is to be attached. Some operating systems require that a file system type be provided, while others inspect the structures of the device and determine the type of file system. Typically, a mount point is an empty directory. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as `/home`; then, to access the directory structure within that file system, we could precede the directory names with `/home`, as in `/home/jane`. Mounting that file system under `/users` would result in the path name `/users/jane`, which we could use to reach the same directory.

Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems, and even file systems of varying types, as appropriate.

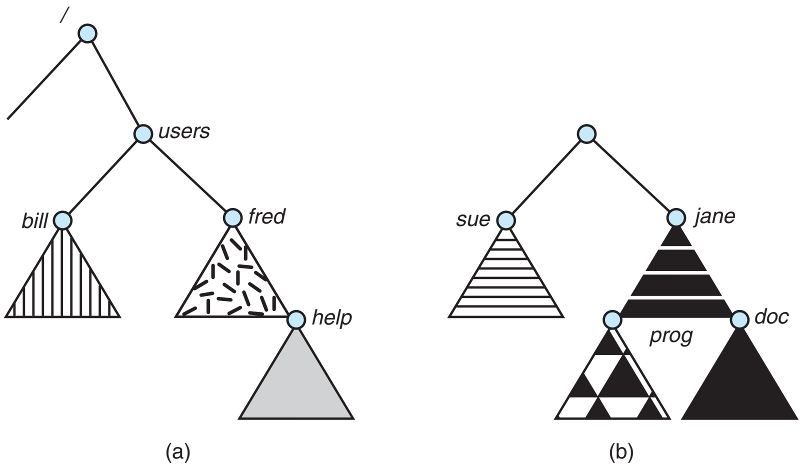


Figure 11.14 File system. (a) Existing system. (b) Unmounted volume.

To illustrate file mounting, consider the file system depicted in Figure 11.14, where the triangles represent subtrees of directories that are of interest. Figure 11.14(a) shows an existing file system, while Figure 11.14(b) shows an unmounted volume residing on `/device/dsk`. At this point, only the files on the existing file system can be accessed. Figure 11.15 shows the effects of mounting the volume residing on `/device/dsk` over `/users`. If the volume is unmounted, the file system is restored to the situation depicted in Figure 11.14.

Systems impose semantics to clarify functionality. For example, a system may disallow a mount over a directory that contains files; or it may make the mounted file system available at that directory and obscure the directory's existing files until the file system is unmounted, terminating the use of the file system and allowing access to the original files in that directory. As another example, a system may allow the same file system to be mounted repeatedly, at different mount points; or it may only allow one mount per file system.

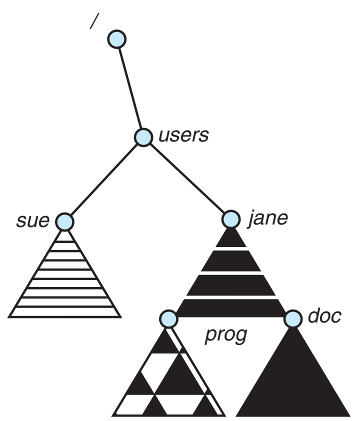


Figure 11.15 Mount point.

Consider the actions of the Mac OS X operating system. Whenever the system encounters a disk for the first time (either at boot time or while the system is running), the Mac OS X operating system searches for a file system on the device. If it finds one, it automatically mounts the file system under the `/Volumes` directory, adding a folder icon labeled with the name of the file system (as stored in the device directory). The user is then able to click on the icon and thus display the newly mounted file system.

The Microsoft Windows family of operating systems maintains an extended two-level directory structure, with devices and volumes assigned drive letters. Volumes have a general graph directory structure associated with the drive letter. The path to a specific file takes the form of `drive-letter:\path\to\file`. The more recent versions of Windows allow a file system to be mounted anywhere in the directory tree, just as UNIX does. Windows operating systems automatically discover all devices and mount all located file systems at boot time. In some systems, like UNIX, the mount commands are explicit. A system configuration file contains a list of devices and mount points for automatic mounting at boot time, but other mounts may be executed manually.

Issues concerning file system mounting are further discussed in Section 12.2.2 and in Section A.7.5.

11.5 File Sharing

In the previous sections, we explored the motivation for file sharing and some of the difficulties involved in allowing users to share files. Such file sharing is very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal. Therefore, user-oriented operating systems must accommodate the need to share files in spite of the inherent difficulties.

In this section, we examine more aspects of file sharing. We begin by discussing general issues that arise when multiple users share files. Once multiple users are allowed to share files, the challenge is to extend sharing to multiple file systems, including remote file systems; we discuss that challenge as well. Finally, we consider what to do about conflicting actions occurring on shared files. For instance, if multiple users are writing to a file, should all the writes be allowed to occur, or should the operating system protect the users' actions from one another?

11.5.1 Multiple Users

When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent. Given a directory structure that allows files to be shared by users, the system must mediate the file sharing. The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files. These are the issues of access control and protection, which are covered in Section 11.6.

To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system. Although many approaches have been taken to meet this requirement, most systems have evolved to use the concepts of file (or directory) **owner** (or **user**) and **group**. The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file. For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations. Exactly which operations can be executed by group members and other users is definable by the file's owner. More details on permission attributes are included in the next section.

The owner and group IDs of a given file (or directory) are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

Many systems have multiple local file systems, including volumes of a single disk or multiple volumes on multiple attached disks. In these cases, the ID checking and permission matching are straightforward, once the file systems are mounted.

11.5.2 Remote File Systems

With the advent of networks (Chapter 17), communication among remote computers became possible. Networking allows the sharing of resources spread across a campus or even around the world. One obvious resource to share is data in the form of files.

Through the evolution of network and file technology, remote file-sharing methods have changed. The first implemented method involves manually transferring files between machines via programs like `ftp`. The second major method uses a **distributed file system** (**DFS**) in which remote directories are visible from a local machine. In some ways, the third method, the **World Wide Web**, is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for `ftp`) are used to transfer files. Increasingly, cloud computing (Section 1.11.7) is being used for file sharing as well.

`ftp` is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system. The World Wide Web uses anonymous file exchange almost exclusively. DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files. This integration adds complexity, as we describe in this section.

11.5.2.1 The Client–Server Model

Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the **server**, and the machine seeking access to the files is the **client**. The client–server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client–server facility.

The server usually specifies the available files on a volume or directory level. Client identification is more difficult. A client can be specified by a network name or other identifier, such as an IP address, but these can be **spoofed**, or imitated. As a result of spoofing, an unauthorized client could be allowed access to the server. More secure solutions include secure authentication of the client via encrypted keys. Unfortunately, with security come many challenges, including ensuring compatibility of the client and server (they must use the same encryption algorithms) and security of key exchanges (intercepted keys could again allow unauthorized access). Because of the difficulty of solving these problems, unsecure authentication methods are most commonly used.

In the case of UNIX and its network file system (NFS), authentication takes place via the client networking information, by default. In this scheme, the user's IDs on the client and server must match. If they do not, the server will be unable to determine access rights to files. Consider the example of a user who has an ID of 1000 on the client and 2000 on the server. A request from the client to the server for a specific file will not be handled appropriately, as the server will determine if user 1000 has access to the file rather than basing the determination on the real user ID of 2000. Access is thus granted or denied based on incorrect authentication information. The server must trust the client to present the correct user ID. Note that the NFS protocols allow many-to-many relationships. That is, many servers can provide files to many clients. In fact, a given machine can be both a server to some NFS clients and a client of other NFS servers.

Once the remote file system is mounted, file operation requests are sent on behalf of the user across the network to the server via the DFS protocol. Typically, a file-open request is sent along with the ID of the requesting user. The server then applies the standard access checks to determine if the user has credentials to access the file in the mode requested. The request is either allowed or denied. If it is allowed, a file handle is returned to the client application, and the application then can perform read, write, and other operations on the file. The client closes the file when access is completed. The operating system may apply semantics similar to those for a local file-system mount or may use different semantics.

11.5.2.2 Distributed Information Systems

To make client–server systems easier to manage, **distributed information systems**, also known as **distributed naming services**, provide unified access to the information needed for remote computing. The **domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet. Before DNS became widespread, files containing the same information were sent via e-mail or `ftp` between all networked hosts. Obviously, this methodology was not scalable! DNS is further discussed in Section 17.4.1.

Other distributed information systems provide ***user name/password/user ID/group ID*** space for a distributed facility. UNIX systems have employed a wide variety of distributed information methods. Sun Microsystems (now part of Oracle Corporation) introduced **yellow pages** (since renamed **network information service**, or **NIS**), and most of the industry adopted its use. It centralizes storage of user names, host names, printer information, and the like.

Unfortunately, it uses unsecure authentication methods, including sending user passwords unencrypted (in clear text) and identifying hosts by IP address. Sun's NIS+ was a much more secure replacement for NIS but was much more complicated and was not widely adopted.

In the case of Microsoft's **common Internet file system** (**CIFS**), network information is used in conjunction with user authentication (user name and password) to create a network login that the server uses to decide whether to allow or deny access to a requested file system. For this authentication to be valid, the user names must match from machine to machine (as with NFS). Microsoft uses **active directory** as a distributed naming structure to provide a single name space for users. Once established, the distributed naming facility is used by all clients and servers to authenticate users.

The industry is moving toward use of the **lightweight directory-access protocol** (**LDAP**) as a secure distributed naming mechanism. In fact, active directory is based on LDAP. Oracle Solaris and most other major operating systems include LDAP and allow it to be employed for user authentication as well as system-wide retrieval of information, such as availability of printers. Conceivably, one distributed LDAP directory could be used by an organization to store all user and resource information for all the organization's computers. The result would be secure single sign-on for users, who would enter their authentication information once for access to all computers within the organization. It would also ease system-administration efforts by combining, in one location, information that is currently scattered in various files on each system or in different distributed information services.

11.5.2.3 Failure Modes

Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other disk-management information (collectively called **metadata**), disk-controller failure, cable failure, and host-adapter failure. User or system-administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention will be required to repair the damage.

Remote file systems have even more failure modes. Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems. In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration, or networking implementation issues. Although some networks have built-in resiliency, including multiple paths between hosts, many do not. Any single failure can thus interrupt the flow of DFS commands.

Consider a client in the midst of using a remote file system. It has files open from the remote host; among other activities, it may be performing directory lookups to open files, reading or writing data to files, and closing files. Now consider a partitioning of the network, a crash of the server, or even a scheduled shutdown of the server. Suddenly, the remote file system is no longer reachable. This scenario is rather common, so it would not be appropriate for the client system to act as it would if a local file system were lost. Rather, the system can either terminate all operations to the lost server or delay operations until the server is again reachable. These failure semantics are defined and implemented as part of the remote-file-system protocol. Termination of all operations can result in users' losing data—and patience. Thus, most DFS protocols either enforce or allow delaying of file-system operations to remote hosts, with the hope that the remote host will become available again.

To implement this kind of recovery from failure, some kind of **state information** may be maintained on both the client and the server. If both server and client maintain knowledge of their current activities and open files, then they can seamlessly recover from a failure. In the situation where the server crashes but must recognize that it has remotely mounted exported file systems and opened files, NFS takes a simple approach, implementing a **stateless** DFS. In essence, it assumes that a client request for a file read or write would not have occurred unless the file system had been remotely mounted and the file had been previously open. The NFS protocol carries all the information needed to locate the appropriate file and perform the requested operation. Similarly, it does not track which clients have the exported volumes mounted, again assuming that if a request comes in, it must be legitimate. While this stateless approach makes NFS resilient and rather easy to implement, it also makes it unsecure. For example, forged read or write requests could be allowed by an NFS server. These issues are addressed in the industry standard NFS Version 4, in which NFS is made stateful to improve its security, performance, and functionality.

11.5.3 Consistency Semantics

Consistency semantics represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. In particular, they specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.

Consistency semantics are directly related to the process synchronization algorithms of Chapter 5. However, the complex algorithms of that chapter tend not to be implemented in the case of file I/O because of the great latencies and slow transfer rates of disks and networks. For example, performing an atomic transaction to a remote disk could involve several network communications, several disk reads and writes, or both. Systems that attempt such a full set of functionalities tend to perform poorly. A successful implementation of complex sharing semantics can be found in the Andrew file system.

For the following discussion, we assume that a series of file accesses (that is, reads and writes) attempted by a user to the same file is always enclosed between the `open()` and `close()` operations. The series of accesses between the `open()` and `close()` operations makes up a **file session**. To illustrate the concept, we sketch several prominent examples of consistency semantics.

11.5.3.1 UNIX Semantics

The UNIX file system (Chapter 17) uses the following consistency semantics:

- Writes to an open file by a user are visible immediately to other users who have this file open.

- One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.

In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image causes delays in user processes.

11.5.3.2 Session Semantics

The Andrew file system (OpenAFS) uses the following consistency semantics:

- Writes to an open file by a user are not visible immediately to other users that have the same file open.
- Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay. Almost no constraints are enforced on scheduling accesses.

11.5.3.3 Immutable-Shared-Files Semantics

A unique approach is that of **immutable shared files**. Once a file is declared as shared by its creator, it cannot be modified. An immutable file has two key properties: its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed. The implementation of these semantics in a distributed system (Chapter 17) is simple, because the sharing is disciplined (read-only).

11.6 Protection

When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost. Reliability is covered in more detail in Chapter 10.

Protection can be provided in many ways. For a single-user laptop system, we might provide protection by locking the computer in a desk drawer or file cabinet. In a larger multiuser system, however, other mechanisms are needed.

11.6.1 Types of Access

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read**. Read from the file.
- **Write**. Write or rewrite the file.
- **Execute**. Load the file into memory and execute it.
- **Append**. Write new information at the end of the file.
- **Delete**. Delete the file and free its space for possible reuse.
- **List**. List the name and attributes of the file.

Other operations, such as renaming, copying, and editing the file, may also be controlled. For many systems, however, these higher-level functions may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests. In this case, a user with read access can also cause the file to be copied, printed, and so on.

Many protection mechanisms have been proposed. Each has advantages and disadvantages and must be appropriate for its intended application. A small computer system that is used by only a few members of a research group, for example, may not need the same types of protection as a large corporate computer that is used for research, finance, and personnel operations. We discuss some approaches to protection in the following sections and present a more complete treatment in Chapter 14.

11.6.2 Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an **access-control list** (**ACL**) specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list.

To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

- **Owner**. The user who created the file is the owner.
- **Group**. A set of users who are sharing the file and need similar access is a group, or work group.
- **Universe**. All other users in the system constitute the universe.

The most common recent approach is to combine access-control lists with the more general (and easier to implement) owner, group, and universe access-control scheme just described. For example, Solaris uses the three categories of access by default but allows access-control lists to be added to specific files and directories when more fine-grained access control is desired.

To illustrate, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named `book.tex.` The protection associated with this file is as follows:

- Sara should be able to invoke all operations on the file.
- Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
- All other users should be able to read, but not write, the file. (Sara is interested in letting as many people as possible read the text so that she can obtain feedback.)

To achieve such protection, we must create a new group—say, `text`—with members Jim, Dawn, and Jill. The name of the group, `text,` must then be associated with the file `book.tex,` and the access rights must be set in accordance with the policy we have outlined.

Now consider a visitor to whom Sara would like to grant temporary access to Chapter 1. The visitor cannot be added to the `text` group because that would give him access to all chapters. Because a file can be in only one group, Sara cannot add another group to Chapter 1. With the addition of access-control-list functionality, though, the visitor can be added to the access control list of Chapter 1.

PERMISSIONS IN A UNIX SYSTEM

In the UNIX system, directory protection and file protection are handled similarly. Associated with each subdirectory are three fields—owner, group, and universe—each consisting of the three bits `rwx`. Thus, a user can list the content of a subdirectory only if the `r` bit is set in the appropriate field. Similarly, a user can change his current directory to another current directory (say, `foo`) only if the `x` bit associated with the `foo` subdirectory is set in the appropriate field.

A sample directory listing from a UNIX environment is shown in below:

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5	pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	jwg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2012	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2012	program
drwx--x--x	4	tag	faculty	512	Jul 31 10:31	lib/
drwx------	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

The first field describes the protection of the file or directory. A `d` as the first character indicates a subdirectory. Also shown are the number of links to the file, the owner's name, the group's name, the size of the file in bytes, the date of last modification, and finally the file's name (with optional extension).

For this scheme to work properly, permissions and access lists must be controlled tightly. This control can be accomplished in several ways. For example, in the UNIX system, groups can be created and modified only by the manager of the facility (or by any superuser). Thus, control is achieved through human interaction. Access lists are discussed further in Section 14.5.2.

With the more limited protection classification, only three fields are needed to define protection. Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it. For example, the UNIX system defines three fields of 3 bits each—`rwx`, where `r` controls read access, `w` controls write access, and `x` controls execution. A separate field is kept for the file owner, for the file's group, and for all other users. In this scheme, 9 bits per file are needed to record protection information. Thus, for our example, the protection fields for the file `book.tex` are as follows: for the owner Sara, all bits are set; for the group `text`, the `r` and `w` bits are set; and for the universe, only the `r` bit is set.

One difficulty in combining approaches comes in the user interface. Users must be able to tell when the optional ACL permissions are set on a file. In the Solaris example, a "+" is appended to the regular permissions, as in:

```
19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1
```

A separate set of commands, `setfacl` and `getfacl`, is used to manage the ACLs.

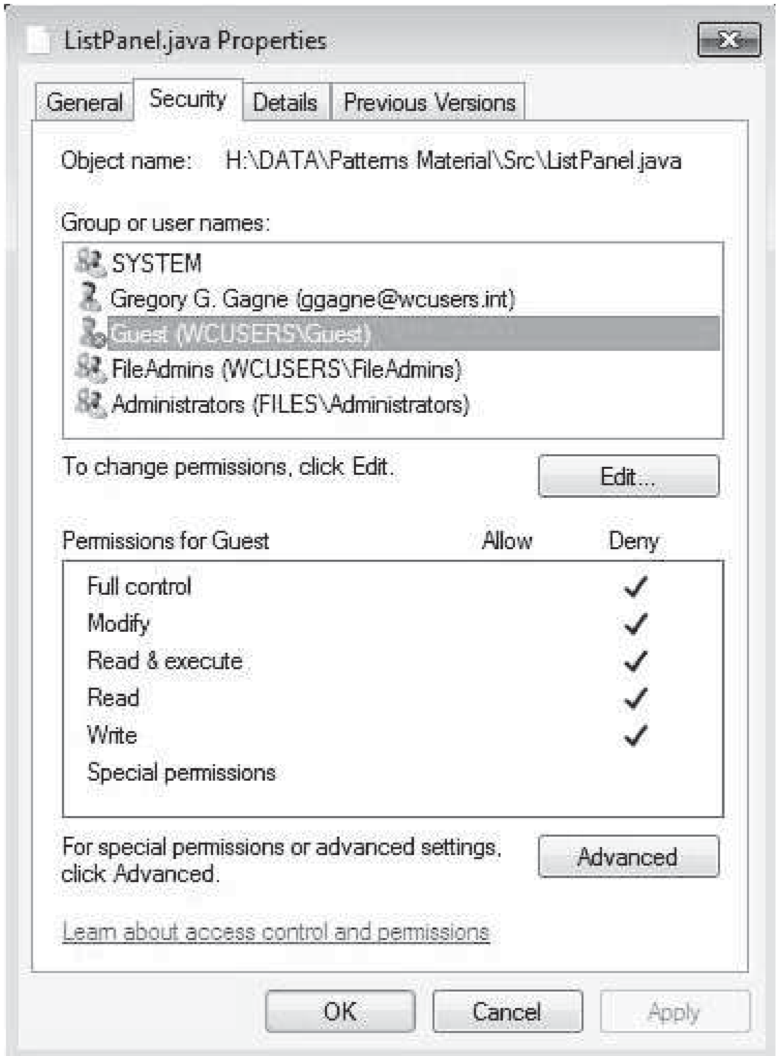


Figure 11.16 Windows 7 access-control list management.

Windows users typically manage access-control lists via the GUI. Figure 11.16 shows a file-permission window on Windows 7 NTFS file system. In this example, user "guest" is specifically denied access to the file `ListPanel.java`.

Another difficulty is assigning precedence when permission and ACLs conflict. For example, if Joe is in a file's group, which has read permission, but the file has an ACL granting Joe read and write permission, should a write by Joe be granted or denied? Solaris gives ACLs precedence (as they are more fine-grained and are not assigned by default). This follows the general rule that specificity should have priority.

11.6.3 Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis. Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to address this problem.

In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories; that is, we need to provide a mechanism for directory protection. The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory. In addition, we probably want to control whether a user can determine the existence of a file in a directory. Sometimes, knowledge of the existence and name of a file is significant in itself. Thus, listing the contents of a directory must be a protected operation. Similarly, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic and general graphs), a given user may have different access rights to a particular file, depending on the path name used.

11.7 Summary

A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line (of fixed or variable length), or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.

The major task for the operating system is to map the logical file concept onto physical storage devices such as magnetic disk or tape. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to order logical records into physical records. Again, this task may be supported by the operating system or left for the application program.

Each device in a file system keeps a volume table of contents or a device directory listing the location of the files on the device. In addition, it is useful to create directories to allow files to be organized. A single-level directory in a multiuser system causes naming problems, since each file must have a unique name. A two-level directory solves this problem by creating a separate directory for each user's files. The directory lists the files by name and includes the file's location on the disk, length, type, owner, time of creation, time of last use, and so on.

The natural generalization of a two-level directory is a tree-structured directory. A tree-structured directory allows a user to create subdirectories to organize files. Acyclic-graph directory structures enable users to share subdirectories and files but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories but sometimes requires garbage collection to recover unused disk space.

Disks are segmented into one or more volumes, each containing a file system or left "raw." File systems may be mounted into the system's naming

Protection

The processes in an operating system must be protected from one another's activities. To provide such protection, we can use various mechanisms to ensure that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources of a system.

Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specifying the controls to be imposed, together with a means of enforcement. We distinguish between protection and security, which is a measure of confidence that the integrity of a system and its data will be preserved. In this chapter, we focus on protection. Security assurance is a much broader topic, and we address it in Chapter 15.

CHAPTER OBJECTIVES

- To discuss the goals and principles of protection in a modern computer system.
- To explain how protection domains, combined with an access matrix, are used to specify the resources a process may access.
- To examine capability- and language-based protection systems.

14.1 Goals of Protection

As computer systems have become more sophisticated and pervasive in their applications, the need to protect their integrity has also grown. Protection was originally conceived as an adjunct to multiprogramming operating systems, so that untrustworthy users might safely share a common logical name space, such as a directory of files, or share a common physical name space, such as memory. Modern protection concepts have evolved to increase the reliability of any complex system that makes use of shared resources.

We need to provide protection for several reasons. The most obvious is the need to prevent the mischievous, intentional violation of an access restriction by a user. Of more general importance, however, is the need to ensure that each program component active in a system uses system resources only in ways consistent with stated policies. This requirement is an absolute one for a reliable system.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a malfunctioning subsystem. Also, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.

The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the management of a system. Still others are defined by the individual users to protect their own files and programs. A protection system must have the flexibility to enforce a variety of policies.

Policies for resource use may vary by application, and they may change over time. For these reasons, protection is no longer the concern solely of the designer of an operating system. The application programmer needs to use protection mechanisms as well, to guard resources created and supported by an application subsystem against misuse. In this chapter, we describe the protection mechanisms the operating system should provide, but application designers can use them as well in designing their own protection software.

Note that ***mechanisms*** are distinct from ***policies***. Mechanisms determine ***how*** something will be done; policies decide ***what*** will be done. The separation of policy and mechanism is important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, every change in policy would require a change in the underlying mechanism. Using general mechanisms enables us to avoid such a situation.

14.2 Principles of Protection

Frequently, a guiding principle can be used throughout a project, such as the design of an operating system. Following this principle simplifies design decisions and keeps the system consistent and easy to understand. A key, time-tested guiding principle for protection is the **principle of least privilege**. It dictates that programs, users, and even systems be given just enough privileges to perform their tasks.

Consider the analogy of a security guard with a passkey. If this key allows the guard into just the public areas that she guards, then misuse of the key will result in minimal damage. If, however, the passkey allows access to all areas, then damage from its being lost, stolen, misused, copied, or otherwise compromised will be much greater.

An operating system following the principle of least privilege implements its features, programs, system calls, and data structures so that failure or compromise of a component does the minimum damage and allows the minimum damage to be done. The overflow of a buffer in a system daemon might cause the daemon process to fail, for example, but should not allow the execution of code from the daemon process's stack that would enable a remote user to gain maximum privileges and access to the entire system (as happens too often today).

Such an operating system also provides system calls and services that allow applications to be written with fine-grained access controls. It provides mechanisms to enable privileges when they are needed and to disable them when they are not needed. Also beneficial is the creation of audit trails for all privileged function access. The audit trail allows the programmer, system administrator, or law-enforcement officer to trace all protection and security activities on the system.

Managing users with the principle of least privilege entails creating a separate account for each user, with just the privileges that the user needs. An operator who needs to mount tapes and back up files on the system has access to just those commands and files needed to accomplish the job. Some systems implement role-based access control (RBAC) to provide this functionality.

Computers implemented in a computing facility under the principle of least privilege can be limited to running specific services, accessing specific remote hosts via specific services, and doing so during specific times. Typically, these restrictions are implemented through enabling or disabling each service and through using access control lists, as described in Sections Section 11.6.2 and Section 14.6.

The principle of least privilege can help produce a more secure computing environment. Unfortunately, it frequently does not. For example, Windows 2000 has a complex protection scheme at its core and yet has many security holes. By comparison, Solaris is considered relatively secure, even though it is a variant of UNIX, which historically was designed with little protection in mind. One reason for the difference may be that Windows 2000 has more lines of code and more services than Solaris and thus has more to secure and protect. Another reason could be that the protection scheme in Windows 2000 is incomplete or protects the wrong aspects of the operating system, leaving other areas vulnerable.

14.3 Domain of Protection

A computer system is a collection of processes and objects. By ***objects,*** we mean both **hardware objects** (such as the CPU, memory segments, printers, disks, and tape drives) and **software objects** (such as files, programs, and semaphores). Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations. Objects are essentially abstract data types.

The operations that are possible may depend on the object. For example, on a CPU, we can only execute. Memory segments can be read and written, whereas a CD-ROM or DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.

A process should be allowed to access only those resources for which it has authorization. Furthermore, at any time, a process should be able to access only those resources that it currently requires to complete its task. This second requirement, commonly referred to as the **need-to-know principle**, is useful in limiting the amount of damage a faulty process can cause in the system.

For example, when process p invokes procedure `A()`, the procedure should be allowed to access only its own variables and the formal parameters passed to it; it should not be able to access all the variables of process p. Similarly, consider the case in which process p invokes a compiler to compile a particular file. The compiler should not be able to access files arbitrarily but should have access only to a well-defined subset of files (such as the source file, listing file, and so on) related to the file to be compiled. Conversely, the compiler may have private files used for accounting or optimization purposes that process p should not be able to access. The need-to-know principle is similar to the principle of least privilege discussed in Section 14.2 in that the goals of protection are to minimize the risks of possible security violations.

14.3.1 Domain Structure

To facilitate the scheme just described, a process operates within a **protection domain**, which specifies the resources that the process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object. The ability to execute an operation on an object is an **access right**. A domain is a collection of access rights, each of which is an ordered pair `<object-name, rights-set>`. For example, if domain D has the access right `<file F, {read,write}>`, then a process executing in domain D can both read and write file F. It cannot, however, perform any other operation on that object.

Domains may share access rights. For example, in Figure 14.1, we have three domains: D_1, D_2, and D_3. The access right $<O_4, \{\text{print}\}>$ is shared by D_2 and D_3, implying that a process executing in either of these two domains can print object O_4. Note that a process must be executing in domain D_1 to read and write object O_1, while only processes in domain D_3 may execute object O_1.

The association between a process and a domain may be either **static**, if the set of resources available to the process is fixed throughout the process's lifetime, or **dynamic**. As might be expected, establishing dynamic protection domains is more complicated than establishing static protection domains.

If the association between processes and domains is fixed, and we want to adhere to the need-to-know principle, then a mechanism must be available to change the content of a domain. The reason stems from the fact that a process may execute in two different phases and may, for example, need read access in one phase and write access in another. If a domain is static, we must define the domain to include both read and write access. However, this arrangement provides more rights than are needed in each of the two phases, since we have read access in the phase where we need only write access, and vice versa.

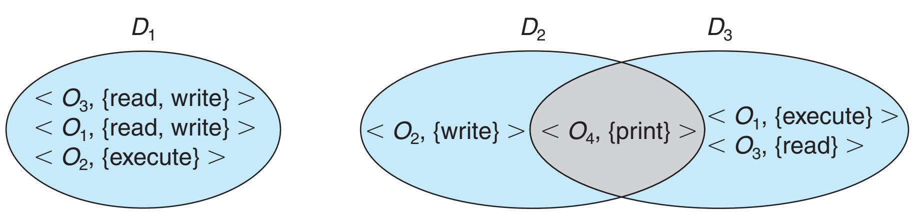


Figure 14.1 System with three protection domains.

Thus, the need-to-know principle is violated. We must allow the contents of a domain to be modified so that the domain always reflects the minimum necessary access rights.

If the association is dynamic, a mechanism is available to allow **domain switching**, enabling the process to switch from one domain to another. We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.

A domain can be realized in a variety of ways:

- Each ***user*** may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed—generally when one user logs out and another user logs in.
- Each ***process*** may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.
- Each ***procedure*** may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

We discuss domain switching in greater detail in Section 14.4.

Consider the standard dual-mode (monitor–user mode) model of operating-system execution. When a process executes in monitor mode, it can execute privileged instructions and thus gain complete control of the computer system. In contrast, when a process executes in user mode, it can invoke only nonprivileged instructions. Consequently, it can execute only within its predefined memory space. These two modes protect the operating system (executing in monitor domain) from the user processes (executing in user domain). In a multiprogrammed operating system, two protection domains are insufficient, since users also want to be protected from one another. Therefore, a more elaborate scheme is needed. We illustrate such a scheme by examining two influential operating systems—UNIX and MULTICS—to see how they implement these concepts.

14.3.2 An Example: UNIX

In the UNIX operating system, a domain is associated with the user. Switching the domain corresponds to changing the user identification temporarily. This change is accomplished through the file system as follows. An owner identification and a domain bit (known as the **setuid bit**) are associated with each file. When the setuid bit is `on`, and a user executes that file, the `userID` is set to that of the owner of the file. When the bit is `off`, however, the `userID` does not change. For example, when a user *A* (that is, a user with `userID` = *A*) starts executing a file owned by *B*, whose associated domain bit is `off`, the `userID` of the process is set to *A*. When the setuid bit is `on`, the `userID` is set to that of the owner of the file: *B*. When the process exits, this temporary `userID` change ends.

Other methods are used to change domains in operating systems in which `userID`s are used for domain definition, because almost all systems need to provide such a mechanism. This mechanism is used when an otherwise privileged facility needs to be made available to the general user population. For instance, it might be desirable to allow users to access a network without letting them write their own networking programs. In such a case, on a UNIX system, the setuid bit on a networking program would be set, causing the `userID` to change when the program was run. The `userID` would change to that of a user with network access privilege (such as `root`, the most powerful `userID`). One problem with this method is that if a user manages to create a file with `userID` `root` and with its setuid bit `on`, that user can become `root` and do anything and everything on the system. The setuid mechanism is discussed further in Appendix A.

An alternative to this method used in some other operating systems is to place privileged programs in a special directory. The operating system is designed to change the `userID` of any program run from this directory, either to the equivalent of `root` or to the `userID` of the owner of the directory. This eliminates one security problem, which occurs when intruders create programs to manipulate the setuid feature and hide the programs in the system for later use (using obscure file or directory names). This method is less flexible than that used in UNIX, however.

Even more restrictive, and thus more protective, are systems that simply do not allow a change of `userID`. In these instances, special techniques must be used to allow users access to privileged facilities. For instance, a **daemon process** may be started at boot time and run as a special `userID`. Users then run a separate program, which sends requests to this process whenever they need to use the facility. This method is used by the TOPS-20 operating system.

In any of these systems, great care must be taken in writing privileged programs. Any oversight can result in a total lack of protection on the system. Generally, these programs are the first to be attacked by people trying to break into a system. Unfortunately, the attackers are frequently successful. For example, security has been breached on many UNIX systems because of the setuid feature. We discuss security in Chapter 15.

14.3.3 An Example: MULTICS

In the MULTICS system, the protection domains are organized hierarchically into a ring structure. Each ring corresponds to a single domain (Figure 14.2). The rings are numbered from 0 to 7. Let D_i and D_j be any two domain rings. If $j < i$, then D_i is a subset of D_j. That is, a process executing in domain D_j has more privileges than does a process executing in domain D_i. A process executing in domain D_0 has the most privileges. If only two rings exist, this scheme is equivalent to the monitor–user mode of execution, where monitor mode corresponds to D_0 and user mode corresponds to D_1.

MULTICS has a segmented address space; each segment is a file, and each segment is associated with one of the rings. A segment description includes an entry that identifies the ring number. In addition, it includes three access bits

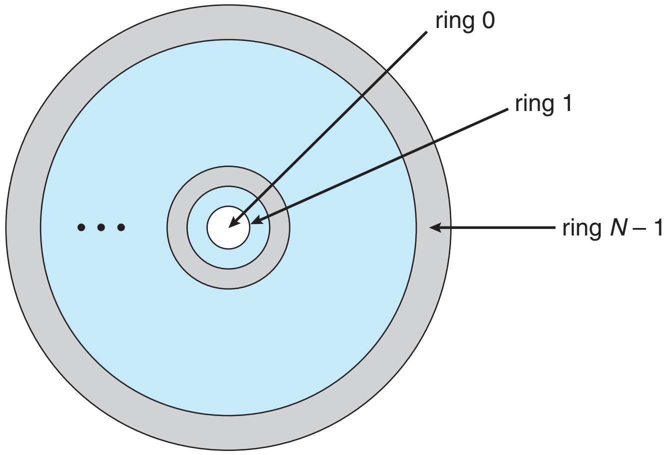


Figure 14.2 MULTICS ring structure.

to control reading, writing, and execution. The association between segments and rings is a policy decision with which we are not concerned here.

A `current-ring-number counter` is associated with each process, identifying the ring in which the process is executing currently. When a process is executing in ring i, it cannot access a segment associated with ring j ($j < i$). It can access a segment associated with ring k ($k \geq i$). The type of access, however, is restricted according to the access bits associated with that segment.

Domain switching in MULTICS occurs when a process crosses from one ring to another by calling a procedure in a different ring. Obviously, this switch must be done in a controlled manner; otherwise, a process could start executing in ring 0, and no protection would be provided. To allow controlled domain switching, we modify the ring field of the segment descriptor to include the following:

- **Access bracket**. A pair of integers, $b1$ and $b2$, such that $b1 \leq b2$.
- **Limit**. An integer $b3$ such that $b3 > b2$.
- **List of gates**. Identifies the entry points (or **gates**) at which the segments may be called.

If a process executing in ring i calls a procedure (or segment) with access bracket ($b1$,$b2$), then the call is allowed if $b1 \leq i \leq b2$, and the current ring number of the process remains i. Otherwise, a trap to the operating system occurs, and the situation is handled as follows:

- If $i < b1$, then the call is allowed to occur, because we have a transfer to a ring (or domain) with fewer privileges. However, if parameters are passed that refer to segments in a lower ring (that is, segments not accessible to the called procedure), then these segments must be copied into an area that can be accessed by the called procedure.
- If $i > b2$, then the call is allowed to occur only if $b3$ is greater than or equal to i and the call has been directed to one of the designated entry points in

the list of gates. This scheme allows processes with limited access rights to call procedures in lower rings that have more access rights, but only in a carefully controlled manner.

The main disadvantage of the ring (or hierarchical) structure is that it does not allow us to enforce the need-to-know principle. In particular, if an object must be accessible in domain D_j but not accessible in domain D_i, then we must have $j < i$. But this requirement means that every segment accessible in D_i is also accessible in D_j.

The MULTICS protection system is generally more complex and less efficient than are those used in current operating systems. If protection interferes with the ease of use of the system or significantly decreases system performance, then its use must be weighed carefully against the purpose of the system. For instance, we would want to have a complex protection system on a computer used by a university to process students' grades and also used by students for classwork. A similar protection system would not be suited to a computer being used for number crunching, in which performance is of utmost importance. We would prefer to separate the mechanism from the protection policy, allowing the same system to have complex or simple protection depending on the needs of its users. To separate mechanism from policy, we require a more general model of protection.

14.4 Access Matrix

Our general model of protection can be viewed abstractly as a matrix, called an **access matrix**. The rows of the access matrix represent domains, and the columns represent objects. Each entry in the matrix consists of a set of access rights. Because the column defines objects explicitly, we can omit the object name from the access right. The entry `access`(i,j) defines the set of operations that a process executing in domain D_i can invoke on object O_j.

To illustrate these concepts, we consider the access matrix shown in Figure 14.3. There are four domains and four objects—three files (F_1, F_2, F_3) and one laser printer. A process executing in domain D_1 can read files F_1 and F_3. A process executing in domain D_4 has the same privileges as one executing in

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 14.3 Access matrix.

domain D_1; but in addition, it can also write onto files F_1 and F_3. The laser printer can be accessed only by a process executing in domain D_2.

The access-matrix scheme provides us with the mechanism for specifying a variety of policies. The mechanism consists of implementing the access matrix and ensuring that the semantic properties we have outlined hold. More specifically, we must ensure that a process executing in domain D_i can access only those objects specified in row i, and then only as allowed by the access-matrix entries.

The access matrix can implement policy decisions concerning protection. The policy decisions involve which rights should be included in the $(i, j)^{th}$ entry. We must also decide the domain in which each process executes. This last policy is usually decided by the operating system.

The users normally decide the contents of the access-matrix entries. When a user creates a new object O_j, the column O_j is added to the access matrix with the appropriate initialization entries, as dictated by the creator. The user may decide to enter some rights in some entries in column j and other rights in other entries, as needed.

The access matrix provides an appropriate mechanism for defining and implementing strict control for both static and dynamic association between processes and domains. When we switch a process from one domain to another, we are executing an operation (`switch`) on an object (the domain). We can control domain switching by including domains among the objects of the access matrix. Similarly, when we change the content of the access matrix, we are performing an operation on an object: the access matrix. Again, we can control these changes by including the access matrix itself as an object. Actually, since each entry in the access matrix can be modified individually, we must consider each entry in the access matrix as an object to be protected. Now, we need to consider only the operations possible on these new objects (domains and the access matrix) and decide how we want processes to be able to execute these operations.

Processes should be able to switch from one domain to another. Switching from domain D_i to domain D_j is allowed if and only if the access right `switch` $\in$ `access`(i, j). Thus, in Figure 14.4, a process executing in domain D_2 can switch

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure 14.4 Access matrix of Figure 14.3 with domains as objects.

to domain D_3 or to domain D_4. A process in domain D_4 can switch to D_1, and one in domain D_1 can switch to D_2.

Allowing controlled change in the contents of the access-matrix entries requires three additional operations: `copy`, `owner`, and `control`. We examine these operations next.

The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right. The `copy` right allows the access right to be copied only within the column (that is, for the object) for which the right is defined. For example, in Figure 14.5(a), a process executing in domain D_2 can copy the read operation into any entry associated with file F_2. Hence, the access matrix of Figure 14.5(a) can be modified to the access matrix shown in Figure 14.5(b).

This scheme has two additional variants:

1. A right is copied from `access`(i, j) to `access`(k, j); it is then removed from `access`(i, j). This action is a of a right, rather than a copy.

2. Propagation of the `copy` right may be limited. That is, when the right R^* is copied from `access`(i, j) to `access`(k, j), only the right R (not R^*) is created. A process executing in domain D_k cannot further copy the right R.

A system may select only one of these three `copy` rights, or it may provide all three by identifying them as separate rights: `copy`, `transfer`, and `limited copy`.

We also need a mechanism to allow addition of new rights and removal of some rights. The `owner` right controls these operations. If `access`(i, j) includes the `owner` right, then a process executing in domain D_i can add and remove

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Figure 14.5 Access matrix with ***copy*** rights.

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Figure 14.6 Access matrix with `owner` rights.

any right in any entry in column j. For example, in Figure 14.6(a), domain D_1 is the owner of F_1 and thus can add and delete any valid right in column F_1. Similarly, domain D_2 is the owner of F_2 and F_3 and thus can add and remove any valid right within these two columns. Thus, the access matrix of Figure 14.6(a) can be modified to the access matrix shown in Figure 14.6(b).

The `copy` and `owner` rights allow a process to change the entries in a column. A mechanism is also needed to change the entries in a row. The `control` right is applicable only to domain objects. If `access`(i, j) includes the `control` right, then a process executing in domain D_i can remove any access right from row j. For example, suppose that, in Figure 14.4, we include the `control` right in `access`(D_2, D_4). Then, a process executing in domain D_2 could modify domain D_4, as shown in Figure 14.7.

The `copy` and `owner` rights provide us with a mechanism to limit the propagation of access rights. However, they do not give us the appropriate tools for preventing the propagation (or disclosure) of information. The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called the **confinement problem**. This problem is in general unsolvable (see the bibliographical notes at the end of the chapter).

These operations on the domains and the access matrix are not in themselves important, but they illustrate the ability of the access-matrix model to allow us to implement and control dynamic protection requirements. New objects and new domains can be created dynamically and included in the

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Figure 14.7 Modified access matrix of Figure 14.4.

access-matrix model. However, we have shown only that the basic mechanism exists. System designers and users must make the policy decisions concerning which domains are to have access to which objects in which ways.

14.5 Implementation of the Access Matrix

How can the access matrix be implemented effectively? In general, the matrix will be sparse; that is, most of the entries will be empty. Although data-structure techniques are available for representing sparse matrices, they are not particularly useful for this application, because of the way in which the protection facility is used. Here, we first describe several methods of implementing the access matrix and then compare the methods.

14.5.1 Global Table

The simplest implementation of the access matrix is a global table consisting of a set of ordered triples `<domain, object, rights-set>`. Whenever an operation M is executed on an object O_j within domain D_i, the global table is searched for a triple $<D_i, O_j, R_k>$, with $M \in R_k$. If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.

This implementation suffers from several drawbacks. The table is usually large and thus cannot be kept in main memory, so additional I/O is needed. Virtual memory techniques are often used for managing this table. In addition, it is difficult to take advantage of special groupings of objects or domains. For example, if everyone can read a particular object, this object must have a separate entry in every domain.

14.5.2 Access Lists for Objects

Each column in the access matrix can be implemented as an access list for one object, as described in Section 11.6.2. Obviously, the empty entries can be discarded. The resulting list for each object consists of ordered pairs `<domain, rights-set>`, which define all domains with a nonempty set of access rights for that object.

This approach can be extended easily to define a list plus a ***default*** set of access rights. When an operation M on an object O_j is attempted in domain

D_i, we search the access list for object O_j, looking for an entry $<D_i, R_k>$ with $M \in R_k$. If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs. For efficiency, we may check the default set first and then search the access list.

14.5.3 Capability Lists for Domains

Rather than associating the columns of the access matrix with the objects as access lists, we can associate each row with its domain. A **capability list** for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its physical name or address, called a **capability**. To execute operation M on object O_j, the process executes the operation M, specifying the capability (or pointer) for object O_j as a parameter. Simple **possession** of the capability means that access is allowed.

The capability list is associated with a domain, but it is never directly accessible to a process executing in that domain. Rather, the capability list is itself a protected object, maintained by the operating system and accessed by the user only indirectly. Capability-based protection relies on the fact that the capabilities are never allowed to migrate into any address space directly accessible by a user process (where they could be modified). If all capabilities are secure, the object they protect is also secure against unauthorized access.

Capabilities were originally proposed as a kind of secure pointer, to meet the need for resource protection that was foreseen as multiprogrammed computer systems came of age. The idea of an inherently protected pointer provides a foundation for protection that can be extended up to the application level.

To provide inherent protection, we must distinguish capabilities from other kinds of objects, and they must be interpreted by an abstract machine on which higher-level programs run. Capabilities are usually distinguished from other data in one of two ways:

- Each object has a **tag** to denote whether it is a capability or accessible data. The tags themselves must not be directly accessible by an application program. Hardware or firmware support may be used to enforce this restriction. Although only one bit is necessary to distinguish between capabilities and other objects, more bits are often used. This extension allows all objects to be tagged with their types by the hardware. Thus, the hardware can distinguish integers, floating-point numbers, pointers, Booleans, characters, instructions, capabilities, and uninitialized values by their tags.

- Alternatively, the address space associated with a program can be split into two parts. One part is accessible to the program and contains the program's normal data and instructions. The other part, containing the capability list, is accessible only by the operating system. A segmented memory space (Section 8.4) is useful to support this approach.

Several capability-based protection systems have been developed; we describe them briefly in Section 14.8. The Mach operating system also uses a version of capability-based protection; it is described in Appendix B.

14.5.4 A Lock–Key Mechanism

The **lock–key scheme** is a compromise between access lists and capability lists. Each object has a list of unique bit patterns, called **locks**. Similarly, each domain has a list of unique bit patterns, called **keys**. A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

As with capability lists, the list of keys for a domain must be managed by the operating system on behalf of the domain. Users are not allowed to examine or modify the list of keys (or locks) directly.

14.5.5 Comparison

As you might expect, choosing a technique for implementing an access matrix involves various trade-offs. Using a global table is simple; however, the table can be quite large and often cannot take advantage of special groupings of objects or domains. Access lists correspond directly to the needs of users. When a user creates an object, he can specify which domains can access the object, as well as what operations are allowed. However, because access-right information for a particular domain is not localized, determining the set of access rights for each domain is difficult. In addition, every access to the object must be checked, requiring a search of the access list. In a large system with long access lists, this search can be time consuming.

Capability lists do not correspond directly to the needs of users, but they are useful for localizing information for a given process. The process attempting access must present a capability for that access. Then, the protection system needs only to verify that the capability is valid. Revocation of capabilities, however, may be inefficient (Section 14.7).

The lock–key mechanism, as mentioned, is a compromise between access lists and capability lists. The mechanism can be both effective and flexible, depending on the length of the keys. The keys can be passed freely from domain to domain. In addition, access privileges can be effectively revoked by the simple technique of changing some of the locks associated with the object (Section 14.7).

Most systems use a combination of access lists and capabilities. When a process first tries to access an object, the access list is searched. If access is denied, an exception condition occurs. Otherwise, a capability is created and attached to the process. Additional references use the capability to demonstrate swiftly that access is allowed. After the last access, the capability is destroyed. This strategy is used in the MULTICS system and in the CAL system.

As an example of how such a strategy works, consider a file system in which each file has an associated access list. When a process opens a file, the directory structure is searched to find the file, access permission is checked, and buffers are allocated. All this information is recorded in a new entry in a file table associated with the process. The operation returns an index into this table for the newly opened file. All operations on the file are made by specification of the index into the file table. The entry in the file table then points to the file and its buffers. When the file is closed, the file-table entry is deleted. Since the file table is maintained by the operating system, the user cannot accidentally corrupt it. Thus, the user can access only those files that have been opened.

Since access is checked when the file is opened, protection is ensured. This strategy is used in the UNIX system.

The right to access must still be checked on each access, and the file-table entry has a capability only for the allowed operations. If a file is opened for reading, then a capability for read access is placed in the file-table entry. If an attempt is made to write onto the file, the system identifies this protection violation by comparing the requested operation with the capability in the file-table entry.

14.6 Access Control

In Section 11.6.2, we described how access controls can be used on files within a file system. Each file and directory is assigned an owner, a group, or possibly a list of users, and for each of those entities, access-control information is assigned. A similar function can be added to other aspects of a computer system. A good example of this is found in Solaris 10.

Solaris 10 advances the protection available in the operating system by explicitly adding the principle of least privilege via **role-based access control** (**RBAC**). This facility revolves around privileges. A privilege is the right to execute a system call or to use an option within that system call (such as opening a file with write access). Privileges can be assigned to processes, limiting them to exactly the access they need to perform their work. Privileges and programs can also be assigned to **roles**. Users are assigned roles or can take roles based on passwords to the roles. In this way, a user can take a role that enables a privilege, allowing the user to run a program to accomplish a specific task, as depicted in Figure 14.8. This implementation of privileges decreases the security risk associated with superusers and setuid programs.

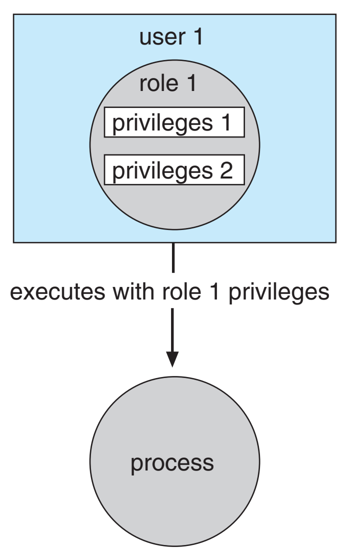


Figure 14.8 Role-based access control in Solaris 10.

Notice that this facility is similar to the access matrix described in Section 14.4. This relationship is further explored in the exercises at the end of the chapter.

14.7 Revocation of Access Rights

In a dynamic protection system, we may sometimes need to revoke access rights to objects shared by different users. Various questions about revocation may arise:

- **Immediate versus delayed**. Does revocation occur immediately, or is it delayed? If revocation is delayed, can we find out when it will take place?
- **Selective versus general**. When an access right to an object is revoked, does it affect all the users who have an access right to that object, or can we specify a select group of users whose access rights should be revoked?
- **Partial versus total**. Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?
- **Temporary versus permanent**. Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again?

With an access-list scheme, revocation is easy. The access list is searched for any access rights to be revoked, and they are deleted from the list. Revocation is immediate and can be general or selective, total or partial, and permanent or temporary.

Capabilities, however, present a much more difficult revocation problem, as mentioned earlier. Since the capabilities are distributed throughout the system, we must find them before we can revoke them. Schemes that implement revocation for capabilities include the following:

- **Reacquisition**. Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.
- **Back-pointers**. A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary. This scheme was adopted in the MULTICS system. It is quite general, but its implementation is costly.
- **Indirection**. The capabilities point indirectly, not directly, to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. Then, when an access is attempted, the capability is found to point to an illegal table entry. Table entries can be reused for other capabilities without difficulty, since both the capability and the table entry contain the unique name of the object. The object for a

CHAPTER 15

Security

Protection, as we discussed in Chapter 14, is strictly an ***internal*** problem: How do we provide controlled access to programs and data stored in a computer system? **Security**, on the other hand, requires not only an adequate protection system but also consideration of the ***external*** environment within which the system operates. A protection system is ineffective if user authentication is compromised or a program is run by an unauthorized user.

Computer resources must be guarded against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. These resources include information stored in the system (both data and code), as well as the CPU, memory, disks, tapes, and networking that are the computer. In this chapter, we start by examining ways in which resources may be accidentally or purposely misused. We then explore a key security enabler —cryptography. Finally, we look at mechanisms to guard against or detect attacks.

CHAPTER OBJECTIVES

- To discuss security threats and attacks.
- To explain the fundamentals of encryption, authentication, and hashing.
- To examine the uses of cryptography in computing.
- To describe various countermeasures to security attacks.

15.1 The Security Problem

In many applications, ensuring the security of the computer system is worth considerable effort. Large commercial systems containing payroll or other financial data are inviting targets to thieves. Systems that contain data pertaining to corporate operations may be of interest to unscrupulous competitors. Furthermore, loss of such data, whether by accident or fraud, can seriously impair the ability of the corporation to function.

In Chapter 14, we discussed mechanisms that the operating system can provide (with appropriate aid from the hardware) that allow users to protect their resources, including programs and data. These mechanisms work well only as long as the users conform to the intended use of and access to these resources. We say that a system is **secure** if its resources are used and accessed as intended under all circumstances. Unfortunately, total security cannot be achieved. Nonetheless, we must have mechanisms to make security breaches a rare occurrence, rather than the norm.

Security violations (or misuse) of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse than against malicious misuse. For the most part, protection mechanisms are the core of protection from accidents. The following list includes several forms of accidental and malicious security violations. We should note that in our discussion of security, we use the terms **intruder** and **cracker** for those attempting to breach security. In addition, a **threat** is the potential for a security violation, such as the discovery of a vulnerability, whereas an **attack** is the attempt to break security.

- **Breach of confidentiality**. This type of violation involves unauthorized reading of data (or theft of information). Typically, a breach of confidentiality is the goal of an intruder. Capturing secret data from a system or a data stream, such as credit-card information or identity information for identity theft, can result directly in money for the intruder.
- **Breach of integrity**. This violation involves unauthorized modification of data. Such attacks can, for example, result in passing of liability to an innocent party or modification of the source code of an important commercial application.
- **Breach of availability**. This violation involves unauthorized destruction of data. Some crackers would rather wreak havoc and gain status or bragging rights than gain financially. Website defacement is a common example of this type of security breach.
- **Theft of service**. This violation involves unauthorized use of resources. For example, an intruder (or intrusion program) may install a daemon on a system that acts as a file server.
- **Denial of service**. This violation involves preventing legitimate use of the system. **Denial-of-service (DOS)** attacks are sometimes accidental. The original Internet worm turned into a DOS attack when a bug failed to delay its rapid spread. We discuss DOS attacks further in Section 15.3.3.

Attackers use several standard methods in their attempts to breach security. The most common is **masquerading**, in which one participant in a communication pretends to be someone else (another host or another person). By masquerading, attackers breach **authentication**, the correctness of identification; they can then gain access that they would not normally be allowed or escalate their privileges—obtain privileges to which they would not normally be entitled. Another common attack is to replay a captured exchange of data. A **replay attack** consists of the malicious or fraudulent repeat of a valid data transmission. Sometimes the replay comprises the entire attack—for example, in a repeat of a request to transfer money. But frequently it is done along with **message modification**, again to escalate privileges. Consider

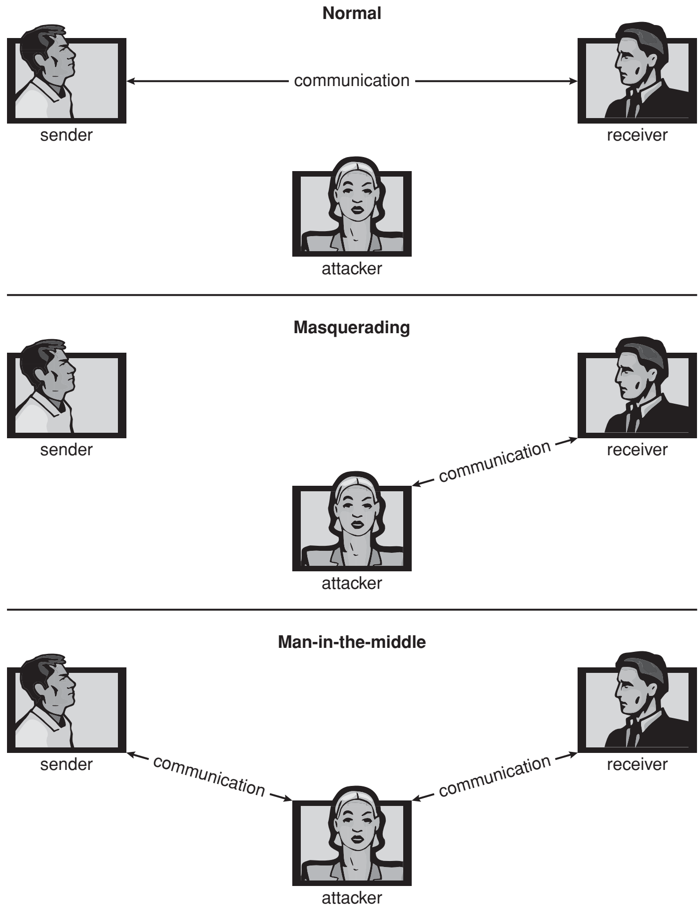


Figure 15.1 Standard security attacks.

the damage that could be done if a request for authentication had a legitimate user's information replaced with an unauthorized user's. Yet another kind of attack is the **man-in-the-middle attack**, in which an attacker sits in the data flow of a communication, masquerading as the sender to the receiver, and vice versa. In a network communication, a man-in-the-middle attack may be preceded by a **session hijacking**, in which an active communication session is intercepted. Several attack methods are depicted in Figure 15.1.

As we have already suggested, absolute protection of the system from malicious abuse is not possible, but the cost to the perpetrator can be made sufficiently high to deter most intruders. In some cases, such as a denial-of-service attack, it is preferable to prevent the attack but sufficient to detect the attack so that countermeasures can be taken.

To protect a system, we must take security measures at four levels:

1. **Physical**. The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders. Both the machine rooms and the terminals or workstations that have access to the machines must be secured.

2. **Human**. Authorization must be done carefully to assure that only appropriate users have access to the system. Even authorized users, however, may be "encouraged" to let others use their access (in exchange for a bribe, for example). They may also be tricked into allowing access via **social engineering**. One type of social-engineering attack is **phishing**. Here, a legitimate-looking e-mail or web page misleads a user into entering confidential information. Another technique is **dumpster diving**, a general term for attempting to gather information in order to gain unauthorized access to the computer (by looking through trash, finding phone books, or finding notes containing passwords, for example). These security problems are management and personnel issues, not problems pertaining to operating systems.

3. **Operating system**. The system must protect itself from accidental or purposeful security breaches. A runaway process could constitute an accidental denial-of-service attack. A query to a service could reveal passwords. A stack overflow could allow the launching of an unauthorized process. The list of possible breaches is almost endless.

4. **Network**. Much computer data in modern systems travels over private leased lines, shared lines like the Internet, wireless connections, or dial-up lines. Intercepting these data could be just as harmful as breaking into a computer, and interruption of communications could constitute a remote denial-of-service attack, diminishing users' use of and trust in the system.

Security at the first two levels must be maintained if operating-system security is to be ensured. A weakness at a high level of security (physical or human) allows circumvention of strict low-level (operating-system) security measures. Thus, the old adage that a chain is only as strong as its weakest link is especially true of system security. All of these aspects must be addressed for security to be maintained.

Furthermore, the system must provide protection (Chapter 14) to allow the implementation of security features. Without the ability to authorize users and processes, to control their access, and to log their activities, it would be impossible for an operating system to implement security measures or to run securely. Hardware protection features are needed to support an overall protection scheme. For example, a system without memory protection cannot be secure. New hardware features are allowing systems to be made more secure, as we shall discuss.

Unfortunately, little in security is straightforward. As intruders exploit security vulnerabilities, security countermeasures are created and deployed. This causes intruders to become more sophisticated in their attacks. For example, recent security incidents include the use of spyware to provide a conduit for spam through innocent systems (we discuss this practice in Section 15.2). This cat-and-mouse game is likely to continue, with more security tools needed to block the escalating intruder techniques and activities.

In the remainder of this chapter, we address security at the network and operating-system levels. Security at the physical and human levels, although important, is for the most part beyond the scope of this text. Security within the operating system and between operating systems is implemented in several not penetrated. Launching an attack that prevents legitimate use is frequently easier than breaking into a machine or facility.

Denial-of-service attacks are generally network based. They fall into two categories. Attacks in the first category use so many facility resources that, in essence, no useful work can be done. For example, a website click could download a Java applet that proceeds to use all available CPU time or to pop up windows infinitely. The second category involves disrupting the network of the facility. There have been several successful denial-of-service attacks of this kind against major websites. These attacks result from abuse of some of the fundamental functionality of TCP/IP. For instance, if the attacker sends the part of the protocol that says "I want to start a TCP connection," but never follows with the standard "The connection is now complete," the result can be partially started TCP sessions. If enough of these sessions are launched, they can eat up all the network resources of the system, disabling any further legitimate TCP connections. Such attacks, which can last hours or days, have caused partial or full failure of attempts to use the target facility. The attacks are usually stopped at the network level until the operating systems can be updated to reduce their vulnerability.

Generally, it is impossible to prevent denial-of-service attacks. The attacks use the same mechanisms as normal operation. Even more difficult to prevent and resolve are **distributed denial-of-service** (**DDOS**) attacks. These attacks are launched from multiple sites at once, toward a common target, typically by zombies. DDOS attacks have become more common and are sometimes associated with blackmail attempts. A site comes under attack, and the attackers offer to halt the attack in exchange for money.

Sometimes a site does not even know it is under attack. It can be difficult to determine whether a system slowdown is an attack or just a surge in system use. Consider that a successful advertising campaign that greatly increases traffic to a site could be considered a DDOS.

There are other interesting aspects of DOS attacks. For example, if an authentication algorithm locks an account for a period of time after several incorrect attempts to access the account, then an attacker could cause all authentication to be blocked by purposely making incorrect attempts to access all accounts. Similarly, a firewall that automatically blocks certain kinds of traffic could be induced to block that traffic when it should not. These examples suggest that programmers and systems managers need to fully understand the algorithms and technologies they are deploying. Finally, computer science classes are notorious sources of accidental system DOS attacks. Consider the first programming exercises in which students learn to create subprocesses or threads. A common bug involves spawning subprocesses infinitely. The system's free memory and CPU resources don't stand a chance.

15.4 Cryptography as a Security Tool

There are many defenses against computer attacks, running the gamut from methodology to technology. The broadest tool available to system designers and users is cryptography. In this section, we discuss cryptography and its use in computer security. Note that the cryptography discussed here has been simplified for educational purposes; readers are cautioned against using any of the schemes described here in the real world. Good cryptography libraries are widely available and would make a good basis for production applications.

In an isolated computer, the operating system can reliably determine the sender and recipient of all interprocess communication, since it controls all communication channels in the computer. In a network of computers, the situation is quite different. A networked computer receives bits "from the wire" with no immediate and reliable way of determining what machine or application sent those bits. Similarly, the computer sends bits onto the network with no way of knowing who might eventually receive them. Additionally, when either sending or receiving, the system has no way of knowing if an eavesdropper listened to the communication.

Commonly, network addresses are used to infer the potential senders and receivers of network messages. Network packets arrive with a source address, such as an IP address. And when a computer sends a message, it names the intended receiver by specifying a destination address. However, for applications where security matters, we are asking for trouble if we assume that the source or destination address of a packet reliably determines who sent or received that packet. A rogue computer can send a message with a falsified source address, and numerous computers other than the one specified by the destination address can (and typically do) receive a packet. For example, all of the routers on the way to the destination will receive the packet, too. How, then, is an operating system to decide whether to grant a request when it cannot trust the named source of the request? And how is it supposed to provide protection for a request or data when it cannot determine who will receive the response or message contents it sends over the network?

It is generally considered infeasible to build a network of any scale in which the source and destination addresses of packets can be ***trusted*** in this sense. Therefore, the only alternative is somehow to eliminate the need to trust the network. This is the job of cryptography. Abstractly, **cryptography** is used to constrain the potential senders and/or receivers of a message. Modern cryptography is based on secrets called **keys** that are selectively distributed to computers in a network and used to process messages. Cryptography enables a recipient of a message to verify that the message was created by some computer possessing a certain key. Similarly, a sender can encode its message so that only a computer with a certain key can decode the message. Unlike network addresses, however, keys are designed so that it is not computationally feasible to derive them from the messages they were used to generate or from any other public information. Thus, they provide a much more trustworthy means of constraining senders and receivers of messages. Note that cryptography is a field of study unto itself, with large and small complexities and subtleties. Here, we explore the most important aspects of the parts of cryptography that pertain to operating systems.

15.4.1 Encryption

Because it solves a wide variety of communication security problems, **encryption** is used frequently in many aspects of modern computing. It is used to send messages securely across across a network, as well as to protect database data, files, and even entire disks from having their contents read by unauthorized entities. An encryption algorithm enables the sender of a message to ensure that only a computer possessing a certain key can read the message, or ensure that the writer of data is the only reader of that data. Encryption of messages is an ancient practice, of course, and there have been many encryption algorithms, dating back to ancient times. In this section, we describe important modern encryption principles and algorithms.

An encryption algorithm consists of the following components:

- A set K of keys.
- A set M of messages.
- A set C of ciphertexts.
- An encrypting function $E : K \rightarrow (M \rightarrow C)$. That is, for each $k \in K$, E_k is a function for generating ciphertexts from messages. Both E and E_k for any k should be efficiently computable functions. Generally, E_k is a randomized mapping from messages to ciphertexts.
- A decrypting function $D : K \rightarrow (C \rightarrow M)$. That is, for each $k \in K$, D_k is a function for generating messages from ciphertexts. Both D and D_k for any k should be efficiently computable functions.

An encryption algorithm must provide this essential property: given a ciphertext $c \in C$, a computer can compute m such that $E_k(m) = c$ only if it possesses k. Thus, a computer holding k can decrypt ciphertexts to the plaintexts used to produce them, but a computer not holding k cannot decrypt ciphertexts. Since ciphertexts are generally exposed (for example, sent on a network), it is important that it be infeasible to derive k from the ciphertexts.

There are two main types of encryption algorithms: symmetric and asymmetric. We discuss both types in the following sections.

15.4.1.1 Symmetric Encryption

In a **symmetric encryption algorithm**, the same key is used to encrypt and to decrypt. Therefore, the secrecy of k must be protected. Figure 15.7 shows an example of two users communicating securely via symmetric encryption over an insecure channel. Note that the key exchange can take place directly between the two parties or via a trusted third party (that is, a certificate authority), as discussed in Section 15.4.1.4.

For the past several decades, the most commonly used symmetric encryption algorithm in the United States for civilian applications has been the **data-encryption standard (DES)** cipher adopted by the National Institute of Standards and Technology (NIST). DES works by taking a 64-bit value and a 56-bit key and performing a series of transformations that are based on substitution and permutation operations. Because DES works on a block of bits at a time, is known as a **block cipher**, and its transformations are typical of block ciphers. With block ciphers, if the same key is used for encrypting an extended amount of data, it becomes vulnerable to attack.

DES is now considered insecure for many applications because its keys can be exhaustively searched with moderate computing resources. (Note, though, that it is still frequently used.) Rather than giving up on DES, NIST created a modification called **triple DES**, in which the DES algorithm is repeated three times (two encryptions and one decryption) on the same plaintext using two

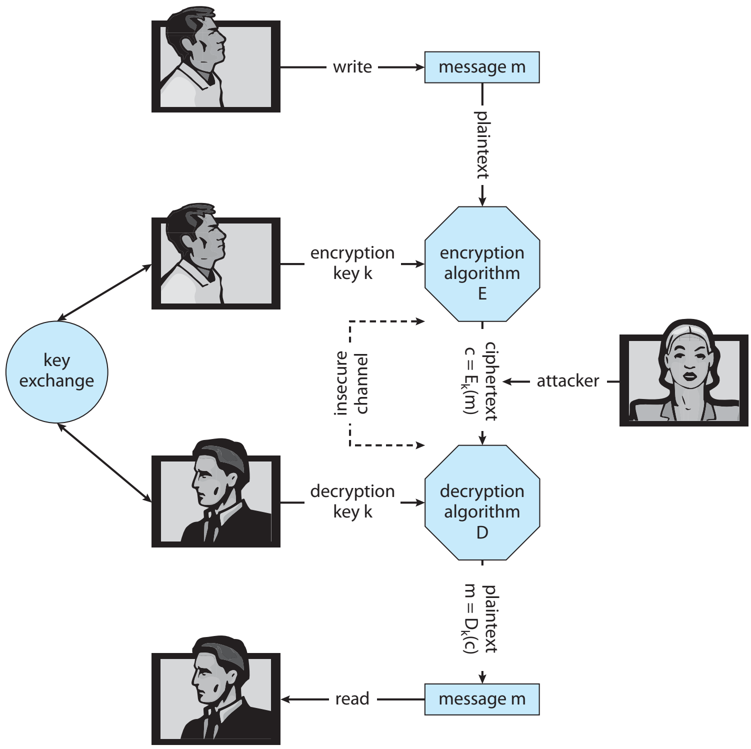


Figure 15.7 A secure communication over an insecure medium.

or three keys—for example, $c = E_{k3}(D_{k2}(E_{k1}(m)))$. When three keys are used, the effective key length is 168 bits. Triple DES is in widespread use today.

In 2001, NIST adopted a new block cipher, called the **advanced encryption standard** (**AES**), to replace DES. AES is another block cipher. It can use key lengths of 128, 192, or 256 bits and works on 128-bit blocks. Generally, the algorithm is compact and efficient.

Block ciphers are not in themselves secure encryption schemes. In particular, they do not directly handle messages longer than their required block sizes. However, there are many **modes of encryption** that are based on stream ciphers, which can be used to securely encrypt longer messages.

RC4 is perhaps the most common stream cipher. A **stream cipher** is designed to encrypt and decrypt a stream of bytes or bits rather than a block. This is useful when the length of a communication would make a block cipher too slow. The key is input into a pseudo–random-bit generator, which is an algorithm that attempts to produce random bits. The output of the generator when fed a key is a keystream. A **keystream** is an infinite set of bits that can be used to encrypt a plaintext stream by simply XORing it with the plaintext. (XOR, for "eXclusive OR" is an operation that compares two input bits and generates one output bit. If the bits are the same, the result is 0. If the bits are different, the result is 1.) RC4 is used in encrypting steams of data, such as in WEP, the wireless LAN protocol. Unfortunately, RC4 as used in WEP (IEEE standard 802.11) has been found to be breakable in a reasonable amount of computer time. In fact, RC4 itself has vulnerabilities.

15.4.1.2 Asymmetric Encryption

In an **asymmetric encryption algorithm**, there are different encryption and decryption keys. An entity preparing to receive encrypted communication creates two keys and makes one of them (called the public key) available to anyone who wants it. Any sender can use that key to encrypt a communication, but only the key creator can decrypt the communication. This scheme, known as **public-key encryption**, was a breakthrough in cryptography. No longer must a key be kept secret and delivered securely. Instead, anyone can encrypt a message to the receiving entity, and no matter who else is listening, only that entity can decrypt the message.

As an example of how public-key encryption works, we describe an algorithm known as **RSA**, after its inventors, Rivest, Shamir, and Adleman. RSA is the most widely used asymmetric encryption algorithm. (Asymmetric algorithms based on elliptic curves are gaining ground, however, because the key length of such an algorithm can be shorter for the same amount of cryptographic strength.)

In RSA, k_e is the **public key**, and k_d is the **private key**. N is the product of two large, randomly chosen prime numbers p and q (for example, p and q are 512 bits each). It must be computationally infeasible to derive $k_{d,N}$ from $k_{e,N}$, so that k_e need not be kept secret and can be widely disseminated. The encryption algorithm is $E_{ke,N}(m) = m^{k_e} \bmod N$, where k_e satisfies $k_e k_d \bmod (p-1)(q-1) = 1$. The decryption algorithm is then $D_{k_d,N}(c) = c^{k_d} \bmod N$.

An example using small values is shown in Figure 15.8. In this example, we make $p = 7$ and $q = 13$. We then calculate $N = 7*13 = 91$ and $(p-1)(q-1) = 72$. We next select k_e relatively prime to 72 and < 72, yielding 5. Finally, we calculate k_d such that $k_e k_d \bmod 72 = 1$, yielding 29. We now have our keys: the public key, $k_{e,N} = 5, 91$, and the private key, $k_{d,N} = 29, 91$. Encrypting the message 69 with the public key results in the message 62, which is then decoded by the receiver via the private key.

The use of asymmetric encryption begins with the publication of the public key of the destination. For bidirectional communication, the source also must publish its public key. "Publication" can be as simple as handing over an electronic copy of the key, or it can be more complex. The private key (or "secret key") must be zealously guarded, as anyone holding that key can decrypt any message created by the matching public key.

We should note that the seemingly small difference in key use between asymmetric and symmetric cryptography is quite large in practice. Asymmetric cryptography is much more computationally expensive to execute. It is much faster for a computer to encode and decode ciphertext by using the usual symmetric algorithms than by using asymmetric algorithms. Why, then, use an asymmetric algorithm? In truth, these algorithms are not used for general-purpose encryption of large amounts of data. However, they are used not only for encryption of small amounts of data but also for authentication, confidentiality, and key distribution, as we show in the following sections.

15.4.1.3 Authentication

We have seen that encryption offers a way of constraining the set of possible receivers of a message. Constraining the set of potential senders of a message is called **authentication**. Authentication is thus complementary to encryption.

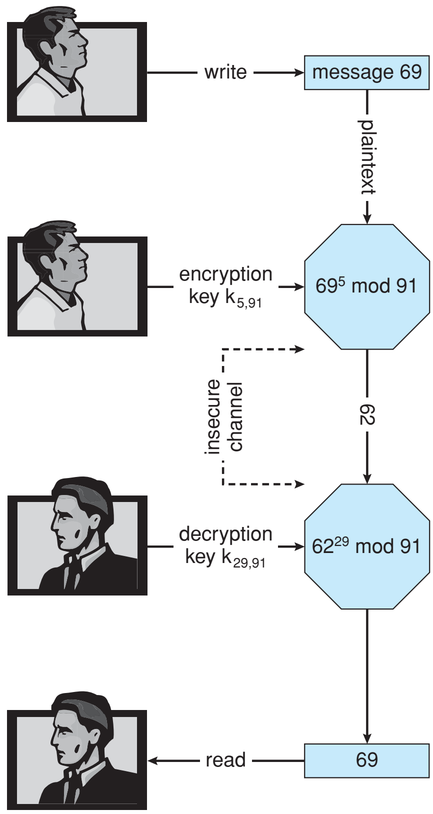


Figure 15.8 Encryption and decryption using RSA asymmetric cryptography.

Authentication is also useful for proving that a message has not been modified. In this section, we discuss authentication as a constraint on possible senders of a message. Note that this sort of authentication is similar to but distinct from user authentication, which we discuss in Section 15.5.

An authentication algorithm using symmetric keys consists of the following components:

- A set K of keys.
- A set M of messages.
- A set A of authenticators.
- A function $S : K \rightarrow (M \rightarrow A)$. That is, for each $k \in K$, S_k is a function for generating authenticators from messages. Both S and S_k for any k should be efficiently computable functions.
- A function $V : K \rightarrow (M \times A \rightarrow \{\texttt{true}, \texttt{false}\})$. That is, for each $k \in K$, V_k is a function for verifying authenticators on messages. Both V and V_k for any k should be efficiently computable functions.

The critical property that an authentication algorithm must possess is this: for a message m, a computer can generate an authenticator $a \in A$ such that $V_k(m, a) = \texttt{true}$ only if it possesses k. Thus, a computer holding k can generate authenticators on messages so that any computer possessing k can verify them. However, a computer not holding k cannot generate authenticators on messages that can be verified using V_k. Since authenticators are generally exposed (for example, sent on a network with the messages themselves), it must not be feasible to derive k from the authenticators. Practically, if $V_k(m, a)$ = `true`, then we know that m has not been modified, and that the sender of the message has k. If we share k with only one entity, then we know that the message originated from k.

Just as there are two types of encryption algorithms, there are two main varieties of authentication algorithms. The first step in understanding these algorithms is to explore hash functions. A **hash function** $H(m)$ creates a small, fixed-sized block of data, known as a **message digest** or **hash value**, from a message m. Hash functions work by taking a message, splitting it into blocks, and processing the blocks to produce an n-bit hash. H must be collision resistant—that is, it must be infeasible to find an $m' \neq m$ such that $H(m) = H(m')$. Now, if $H(m) = H(m')$, we know that $m = m'$—that is, we know that the message has not been modified. Common message-digest functions include **MD5**, now considered insecure, which produces a 128-bit hash, and **SHA-1**, which outputs a 160-bit hash. Message digests are useful for detecting changed messages but are not useful as authenticators. For example, $H(m)$ can be sent along with a message; but if H is known, then someone could modify m to m' and recompute $H(m')$, and the message modification would not be detected. Therefore, we must authenticate $H(m)$.

The first main type of authentication algorithm uses symmetric encryption. In a **message-authentication code** (**MAC**), a cryptographic checksum is generated from the message using a secret key. A MAC provides a way to securely authenticate short values. If we use it to authenticate $H(m)$ for an H that is collision resistant, then we obtain a way to securely authenticate long messages by hashing them first. Note that k is needed to compute both S_k and V_k, so anyone able to compute one can compute the other.

The second main type of authentication algorithm is a **digital-signature algorithm**, and the authenticators thus produced are called **digital signatures**. Digital signatures are very useful in that they enable ***anyone*** to verify the authenticity of the message. In a digital-signature algorithm, it is computationally infeasible to derive k_s from k_v. Thus, k_v is the public key, and k_s is the private key.

Consider as an example the RSA digital-signature algorithm. It is similar to the RSA encryption algorithm, but the key use is reversed. The digital signature of a message is derived by computing $S_{ks}(m) = H(m)^{k_s} \bmod N$. The key k_s again is a pair $\langle d, N\rangle$, where N is the product of two large, randomly chosen prime numbers p and q. The verification algorithm is then $V_{kv}(m, a) \stackrel{?}{=} a^{k_v} \bmod N = H(m))$, where k_v satisfies $k_v k_s \bmod (p-1)(q-1) = 1$.

Note that encryption and authentication may be used together or separately. Sometimes, for instance, we want authentication but not confidentiality. For example, a company could provide a software patch and could "sign" that patch to prove that it came from the company and that it hasn't been modified.

Authentication is a component of many aspects of security. For example, digital signatures are the core of **nonrepudiation**, which supplies proof that an entity performed an action. A typical example of nonrepudiation involves the filling out of electronic forms as an alternative to the signing of paper contracts. Nonrepudiation assures that a person filling out an electronic form cannot deny that he did so.

15.4.1.4 Key Distribution

Certainly, a good part of the battle between cryptographers (those inventing ciphers) and cryptanalysts (those trying to break them) involves keys. With symmetric algorithms, both parties need the key, and no one else should have it. The delivery of the symmetric key is a huge challenge. Sometimes it is performed **out-of-band**—say, via a paper document or a conversation. These methods do not scale well, however. Also consider the key-management challenge. Suppose a user wanted to communicate with N other users privately. That user would need N keys and, for more security, would need to change those keys frequently.

These are the very reasons for efforts to create asymmetric key algorithms. Not only can the keys be exchanged in public, but a given user needs only one private key, no matter how many other people she wants to communicate with. There is still the matter of managing a public key for each recipient of the communication, but since public keys need not be secured, simple storage can be used for that **key ring**.

Unfortunately, even the distribution of public keys requires some care. Consider the man-in-the-middle attack shown in Figure 15.9. Here, the person who wants to receive an encrypted message sends out his public key, but an attacker also sends her "bad" public key (which matches her private key). The person who wants to send the encrypted message knows no better and so uses the bad key to encrypt the message. The attacker then happily decrypts it.

The problem is one of authentication—what we need is proof of who (or what) owns a public key. One way to solve that problem involves the use of digital certificates. A **digital certificate** is a public key digitally signed by a trusted party. The trusted party receives proof of identification from some entity and certifies that the public key belongs to that entity. But how do we know we can trust the certifier? These **certificate authorities** have their public keys included within web browsers (and other consumers of certificates) before they are distributed. The certificate authorities can then vouch for other authorities (digitally signing the public keys of these other authorities), and so on, creating a web of trust. The certificates can be distributed in a standard X.509 digital certificate format that can be parsed by computer. This scheme is used for secure web communication, as we discuss in Section 15.4.3.

15.4.2 Implementation of Cryptography

Network protocols are typically organized in **layers**, like an onion or a parfait, with each layer acting as a client of the one below it. That is, when one protocol generates a message to send to its protocol peer on another machine, it hands its message to the protocol below it in the network-protocol stack for delivery to its peer on that machine. For example, in an IP network, TCP (a ***transport-layer*** protocol) acts as a client of IP (a ***network-layer*** protocol): TCP packets are passed down to IP for delivery to the IP peer at the other end of the connection. IP encapsulates the TCP packet in an IP packet, which it similarly passes down to the ***data-link layer*** to be transmitted across the network to its peer on the

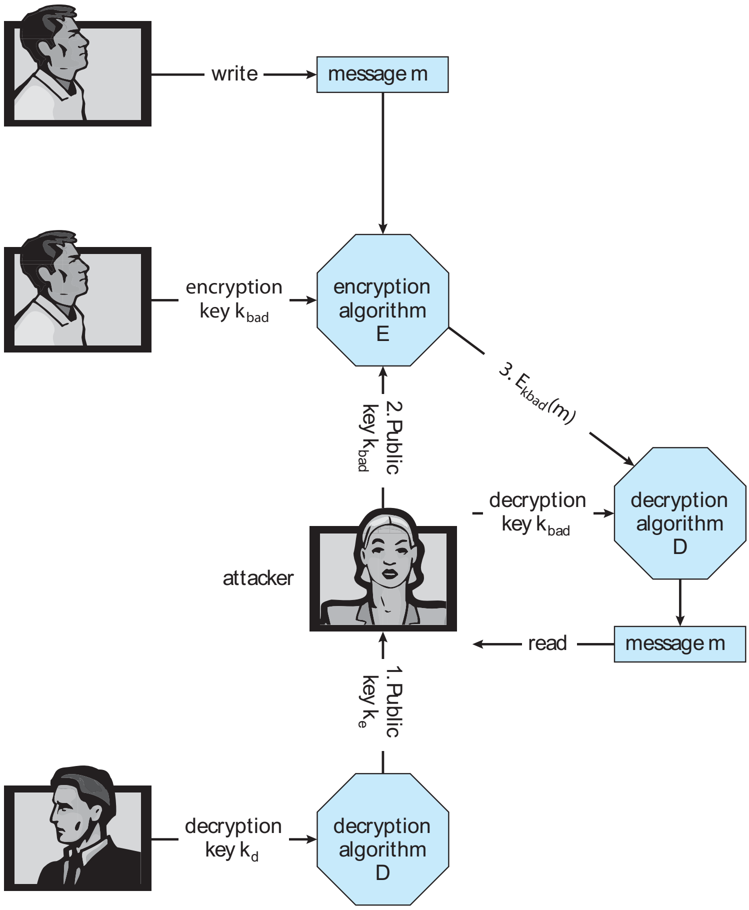


Figure 15.9 A man-in-the-middle attack on asymmetric cryptography.

destination computer. This IP peer then delivers the TCP packet up to the TCP peer on that machine.

Cryptography can be inserted at almost any layer in the OSI model. SSL (Section 15.4.3), for example, provides security at the transport layer. Network-layer security generally has been standardized on **IPSec**, which defines IP packet formats that allow the insertion of authenticators and the encryption of packet contents. IPSec uses symmetric encryption and uses the **Internet Key Exchange** (**IKE**) protocol for key exchange. IKE is based on pubic-key encryption. IPSec is becoming widely used as the basis for **virtual private networks** (**VPNs**), in which all traffic between two IPSec endpoints is encrypted to make a private network out of one that may otherwise be public. Numerous protocols also have been developed for use by applications, such as PGP for encrypting e-mail, but then the applications themselves must be coded to implement security.

Where is cryptographic protection best placed in a protocol stack? In general, there is no definitive answer. On the one hand, more protocols benefit from protections placed lower in the stack. For example, since IP packets encapsulate TCP packets, encryption of IP packets (using IPSec, for example) also hides the contents of the encapsulated TCP packets. Similarly, authenticators on IP packets detect the modification of contained TCP header information.

On the other hand, protection at lower layers in the protocol stack may give insufficient protection to higher-layer protocols. For example, an application server that accepts connections encrypted with IPSec might be able to authenticate the client computers from which requests are received. However, to authenticate a user at a client computer, the server may need to use an application-level protocol—the user may be required to type a password. Also consider the problem of e-mail. E-mail delivered via the industry-standard SMTP protocol is stored and forwarded, frequently multiple times, before it is delivered. Each of these transmissions could go over a secure or an insecure network. For e-mail to be secure, the e-mail message needs to be encrypted so that its security is independent of the transports that carry it.

15.4.3 An Example: SSL

SSL 3.0 is a cryptographic protocol that enables two computers to communicate securely—that is, so that each can limit the sender and receiver of messages to the other. It is perhaps the most commonly used cryptographic protocol on the Internet today, since it is the standard protocol by which web browsers communicate securely with web servers. For completeness, we should note that SSL was designed by Netscape and that it evolved into the industry- standard TLS protocol. In this discussion, we use SSL to mean both SSL and TLS.

SSL is a complex protocol with many options. Here, we present only a single variation of it. Even then, we describe it in a very simplified and abstract form, so as to maintain focus on its use of cryptographic primitives. What we are about to see is a complex dance in which asymmetric cryptography is used so that a client and a server can establish a secure **session key** that can be used for symmetric encryption of the session between the two—all of this while avoiding man-in-the-middle and replay attacks. For added cryptographic strength, the session keys are forgotten once a session is completed. Another communication between the two will require generation of new session keys.

The SSL protocol is initiated by a client c to communicate securely with a server. Prior to the protocol's use, the server s is assumed to have obtained a certificate, denoted cert_s, from certification authority CA. This certificate is a structure containing the following:

- Various attributes (attrs) of the server, such as its unique ***distinguished*** name and its ***common*** (DNS) name
- The identity of a asymmetric encryption algorithm $E()$ for the server
- The public key k_e of this server
- A validity interval ($\mathsf{interval}$) during which the certificate should be considered valid
- A digital signature a on the above information made by the CA—that is, $\mathsf{a} = \mathsf{S}_{kCA}(\langle\ \mathsf{attrs}, \mathrm{E}_{ke}, \mathsf{interval}\ \rangle)$

In addition, prior to the protocol's use, the client is presumed to have obtained the public verification algorithm V_{kCA} for CA. In the case of the Web, the user's browser is shipped from its vendor containing the verification algorithms and public keys of certain certification authorities. The user can add or delete these as she chooses.

When c connects to s, it sends a 28-byte random value n_c to the server, which responds with a random value n_s of its own, plus its certificate cert_s. The client verifies that $V_{kCA}(\langle \mathsf{attrs}, E_{ke}, \mathsf{interval}\rangle, \mathrm{a}) = \mathtt{true}$ and that the current time is in the validity interval interval. If both of these tests are satisfied, the server has proved its identity. Then the client generates a random 46-byte **premaster secret** pms and sends $\mathsf{cpms} = E_{ke}(\mathsf{pms})$ to the server. The server recovers $\mathsf{pms} = D_{kd}(\mathsf{cpms})$. Now both the client and the server are in possession of n_c, n_s, and pms, and each can compute a shared 48-byte **master secret** $\mathsf{ms} = H(n_c, n_s, \mathsf{pms})$. Only the server and client can compute ms, since only they know pms. Moreover, the dependence of ms on n_c and n_s ensures that ms is a ***fresh*** value —that is, a session key that has not been used in a previous communication. At this point, the client and the server both compute the following keys from the ms:

- A symmetric encryption key k_{cs}^{crypt} for encrypting messages from the client to the server
- A symmetric encryption key k_{sc}^{crypt} for encrypting messages from the server to the client
- A MAC generation key k_{cs}^{mac} for generating authenticators on messages from the client to the server
- A MAC generation key k_{sc}^{mac} for generating authenticators on messages from the server to the client

To send a message m to the server, the client sends

$$c = E_{k_{cs}^{\mathsf{crypt}}}(\langle m, S_{k_{cs}^{\mathsf{mac}}}(m)\rangle).$$

Upon receiving c, the server recovers

$$\langle m, a\rangle = D_{k_{cs}^{\mathsf{crypt}}}(c)$$

and accepts m if $V_{k_{cs}^{\mathsf{mac}}}(m, a) = \text{true}$. Similarly, to send a message m to the client, the server sends

$$c = E_{k_{sc}^{\mathsf{crypt}}}(\langle m, S_{k_{sc}^{\mathsf{mac}}}(m)\rangle)$$

and the client recovers

$$\langle m, a\rangle = D_{k_{sc}^{\mathsf{crypt}}}(c)$$

and accepts m if $V_{k_{sc}^{\mathsf{mac}}}(m, a) = \text{true}$.

This protocol enables the server to limit the recipients of its messages to the client that generated pms and to limit the senders of the messages it accepts to that same client. Similarly, the client can limit the recipients of the messages it sends and the senders of the messages it accepts to the party that knows k_d (that is, the party that can decrypt cpms). In many applications, such as web transactions, the client needs to verify the identity of the party that knows k_d. This is one purpose of the certificate cert_s. In particular, the attrs field contains information that the client can use to determine the identity—for example, the domain name—of the server with which it is communicating. For applications in which the server also needs information about the client, SSL supports an option by which a client can send a certificate to the server.

In addition to its use on the Internet, SSL is being used for a wide variety of tasks. For example, IPSec VPNs now have a competitor in SSL VPNs. IPSec is good for point-to-point encryption of traffic—say, between two company offices. SSL VPNs are more flexible but not as efficient, so they might be used between an individual employee working remotely and the corporate office.

15.5 User Authentication

Our earlier discussion of authentication involves messages and sessions. But what about users? If a system cannot authenticate a user, then authenticating that a message came from that user is pointless. Thus, a major security problem for operating systems is **user authentication**. The protection system depends on the ability to identify the programs and processes currently executing, which in turn depends on the ability to identify each user of the system. Users normally identify themselves. How do we determine whether a user's identity is authentic? Generally, user authentication is based on one or more of three things: the user's possession of something (a key or card), the user's knowledge of something (a user identifier and password), or an attribute of the user (fingerprint, retina pattern, or signature).

15.5.1 Passwords

The most common approach to authenticating a user identity is the use of **passwords**. When the user identifies herself by user ID or account name, she is asked for a password. If the user-supplied password matches the password stored in the system, the system assumes that the account is being accessed by the owner of that account.

Passwords are often used to protect objects in the computer system, in the absence of more complete protection schemes. They can be considered a special case of either keys or capabilities. For instance, a password may be associated with each resource (such as a file). Whenever a request is made to use the resource, the password must be given. If the password is correct, access is granted. Different passwords may be associated with different access rights. For example, different passwords may be used for reading files, appending files, and updating files.

In practice, most systems require only one password for a user to gain full rights. Although more passwords theoretically would be more secure, such systems tend not to be implemented due to the classic trade-off between security and convenience. If security makes something inconvenient, then the security is frequently bypassed or otherwise circumvented.

15.5.2 Password Vulnerabilities

Passwords are extremely common because they are easy to understand and use. Unfortunately, passwords can often be guessed, accidentally exposed, sniffed (read by an eavesdropper), or illegally transferred from an authorized user to an unauthorized one, as we show next.

There are two common ways to guess a password. One way is for the intruder (either human or program) to know the user or to have information about the user. All too frequently, people use obvious information (such as the names of their cats or spouses) as their passwords. The other way is to use brute force, trying enumeration—or all possible combinations of valid password characters (letters, numbers, and punctuation on some systems)—until the password is found. Short passwords are especially vulnerable to this method. For example, a four-character password provides only 10,000 variations. On average, guessing 5,000 times would produce a correct hit. A program that could try a password every millisecond would take only about 5 seconds to guess a four-character password. Enumeration is less successful where systems allow longer passwords that include both uppercase and lowercase letters, along with numbers and all punctuation characters. Of course, users must take advantage of the large password space and must not, for example, use only lowercase letters.

In addition to being guessed, passwords can be exposed as a result of visual or electronic monitoring. An intruder can look over the shoulder of a user (**shoulder surfing**) when the user is logging in and can learn the password easily by watching the keyboard. Alternatively, anyone with access to the network on which a computer resides can seamlessly add a network monitor, allowing him to **sniff**, or watch, all data being transferred on the network, including user IDs and passwords. Encrypting the data stream containing the password solves this problem. Even such a system could have passwords stolen, however. For example, if a file is used to contain the passwords, it could be copied for off-system analysis. Or consider a Trojan-horse program installed on the system that captures every keystroke before sending it on to the application.

Exposure is a particularly severe problem if the password is written down where it can be read or lost. Some systems force users to select hard-to-remember or long passwords, or to change their password frequently, which may cause a user to record the password or to reuse it. As a result, such systems provide much less security than systems that allow users to select easy passwords!

The final type of password compromise, illegal transfer, is the result of human nature. Most computer installations have a rule that forbids users to share accounts. This rule is sometimes implemented for accounting reasons but is often aimed at improving security. For instance, suppose one user ID is shared by several users, and a security breach occurs from that user ID. It is impossible to know who was using the ID at the time the break occurred or even whether the user was an authorized one. With one user per user ID, any user can be questioned directly about use of the account; in addition, the user might notice something different about the account and detect the break-in. Sometimes, users break account-sharing rules to help friends or to circumvent accounting, and this behavior can result in a system's being accessed by unauthorized users —possibly harmful ones.

Passwords can be either generated by the system or selected by a user. System-generated passwords may be difficult to remember, and thus users may write them down. As mentioned, however, user-selected passwords are often easy to guess (the user's name or favorite car, for example). Some systems will check a proposed password for ease of guessing or cracking before accepting it. Some systems also ***age*** passwords, forcing users to change their passwords at regular intervals (every three months, for instance). This method is not foolproof either, because users can easily toggle between two passwords. The solution, as implemented on some systems, is to record a password history for each user. For instance, the system could record the last N passwords and not allow their reuse.

Several variants on these simple password schemes can be used. For example, the password can be changed more frequently. At the extreme, the password is changed from session to session. A new password is selected (either by the system or by the user) at the end of each session, and that password must be used for the next session. In such a case, even if a password is used by an unauthorized person, that person can use it only once. When the legitimate user tries to use a now-invalid password at the next session, he discovers the security violation. Steps can then be taken to repair the breached security.

15.5.3 Securing Passwords

One problem with all these approaches is the difficulty of keeping the password secret within the computer. How can the system store a password securely yet allow its use for authentication when the user presents her password? The UNIX system uses secure hashing to avoid the necessity of keeping its password list secret. Because the list is hashed rather than encrypted, it is impossible for the system to decrypt the stored value and determine the original password.

Here's how this system works. Each user has a password. The system contains a function that is extremely difficult—the designers hope impossible—to invert but is simple to compute. That is, given a value x, it is easy to compute the hash function value $f(x)$. Given a function value $f(x)$, however, it is impossible to compute x. This function is used to encode all passwords. Only encoded passwords are stored. When a user presents a password, it is hashed and compared against the stored encoded password. Even if the stored encoded password is seen, it cannot be decoded, so the password cannot be determined. Thus, the password file does not need to be kept secret.

The flaw in this method is that the system no longer has control over the passwords. Although the passwords are hashed, anyone with a copy of the password file can run fast hash routines against it—hashing each word in a dictionary, for instance, and comparing the results against the passwords. If the user has selected a password that is also a word in the dictionary, the password is cracked. On sufficiently fast computers, or even on clusters of slow computers, such a comparison may take only a few hours. Furthermore, because UNIX systems use a well-known hashing algorithm, a cracker might keep a cache of passwords that have been cracked previously. For these reasons, systems include a "salt," or recorded random number, in the hashing algorithm. The salt value is added to the password to ensure that if two plaintext passwords are the same, they result in different hash values. In addition, the salt value makes hashing a dictionary ineffective, because each dictionary term would need to be combined with each salt value for comparison to the stored passwords. Newer versions of UNIX also store the hashed password entries in a file readable only by the superuser. The programs that compare the hash to the stored value are run `setuid` to root, so they can read this file, but other users cannot.

Another weakness in the UNIX password methods is that many UNIX systems treat only the first eight characters as significant. It is therefore extremely important for users to take advantage of the available password space. Complicating the issue further is the fact that some systems do not allow the use of dictionary words as passwords. A good technique is to generate your password by using the first letter of each word of an easily remembered phrase using both upper and lower characters with a number or punctuation mark thrown in for good measure. For example, the phrase "My mother's name is Katherine" might yield the password "Mmn.isK!". The password is hard to crack but easy for the user to remember. A more secure system would allow more characters in its passwords. Indeed, a system might also allow passwords to include the space character, so that a user could create a **passphrase**.

15.5.4 One-Time Passwords

To avoid the problems of password sniffing and shoulder surfing, a system can use a set of **paired passwords**. When a session begins, the system randomly selects and presents one part of a password pair; the user must supply the other part. In this system, the user is **challenged** and must **respond** with the correct answer to that challenge.

This approach can be generalized to the use of an algorithm as a password. Such algorithmic passwords are not susceptible to reuse. That is, a user can type in a password, and no entity intercepting that password will be able to reuse it. In this scheme, the system and the user share a symmetric password. The password pw is never transmitted over a medium that allows exposure. Rather, the password is used as input to the function, along with a **challenge** ch presented by the system. The user then computes the function $H(pw, ch)$. The result of this function is transmitted as the authenticator to the computer. Because the computer also knows pw and ch, it can perform the same computation. If the results match, the user is authenticated. The next time the user needs to be authenticated, another ch is generated, and the same steps ensue. This time, the authenticator is different. This **one-time password** system is one of only a few ways to prevent improper authentication due to password exposure.

One-time password systems are implemented in various ways. Commercial implementations use hardware calculators with a display or a display and numeric keypad. These calculators generally take the shape of a credit card, a key-chain dongle, or a USB device. Software running on computers or smartphones provides the user with $H(pw, ch)$; pw can be input by the user or generated by the calculator in synchronization with the computer. Sometimes, pw is just a **personal identification number (PIN)**. The output of any of these systems shows the one-time password. A one-time password generator that requires input by the user involves **two-factor authentication**. Two different types of components are needed in this case—for example, a one-time password generator that generates the correct response only if the PIN is valid. Two-factor authentication offers far better authentication protection than single-factor authentication because it requires "something you have" as well as "something you know."

Another variation on one-time passwords uses a **code book**, or **one-time pad**, which is a list of single-use passwords. Each password on the list is used once and then is crossed out or erased. The commonly used S/Key system uses either a software calculator or a code book based on these calculations as a source of one-time passwords. Of course, the user must protect his code book, and it is helpful if the code book does not identify the system to which the codes are authenticators.

15.5.5 Biometrics

Yet another variation on the use of passwords for authentication involves the use of biometric measures. Palm- or hand-readers are commonly used to secure physical access—for example, access to a data center. These readers match stored parameters against what is being read from hand-reader pads. The parameters can include a temperature map, as well as finger length, finger width, and line patterns. These devices are currently too large and expensive to be used for normal computer authentication.

Fingerprint readers have become accurate and cost-effective and should become more common in the future. These devices read finger ridge patterns and convert them into a sequence of numbers. Over time, they can store a set of sequences to adjust for the location of the finger on the reading pad and other factors. Software can then scan a finger on the pad and compare its features with these stored sequences to determine if they match. Of course, multiple users can have profiles stored, and the scanner can differentiate among them. A very accurate two-factor authentication scheme can result from requiring a password as well as a user name and fingerprint scan. If this information is encrypted in transit, the system can be very resistant to spoofing or replay attack.

Multifactor authentication is better still. Consider how strong authentication can be with a USB device that must be plugged into the system, a PIN, and a fingerprint scan. Except for having to place ones finger on a pad and plug the USB into the system, this authentication method is no less convenient than that using normal passwords. Recall, though, that strong authentication by itself is not sufficient to guarantee the ID of the user. An authenticated session can still be hijacked if it is not encrypted.

15.6 Implementing Security Defenses

Just as there are myriad threats to system and network security, there are many security solutions. The solutions range from improved user education, through technology, to writing bug-free software. Most security professionals subscribe to the theory of **defense in depth**, which states that more layers of defense are better than fewer layers. Of course, this theory applies to any kind of security. Consider the security of a house without a door lock, with a door lock, and with a lock and an alarm. In this section, we look at the major methods, tools, and techniques that can be used to improve resistance to threats.

15.6.1 Security Policy

The first step toward improving the security of any aspect of computing is to have a **security policy**. Policies vary widely but generally include a statement