OPERATING SYSTEMS
(PGDIT06)
(PG DIPLOMA - IT)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

OPERATING SYSTEMS

MCA- 105

Lesson Writer

D.NAGA RAJU
ASST. PROFESSOR
DEPT.OF COMPUTER SCIENCE & ENGG.
ACHARAYA NAGARJUNA UNIVERSITY

Editor

P. AMMI REDDY, ***M.Sc., M.C.A., M.Tech.***
Associate Professor
Vasireddy Venkatadri Institute of Technology
Nambur (P.O.), GUNTUR – Dt.

Director

Prof.V.CHANDRASEKHARA RAO, M.Com., Ph.D.
CENTRE FOR DISTANCE EDUCATION
ACHARAYA NAGARJUNA UNIVERSITY
NAGARJUNA NAGAR – 522 510

Ph: 0863-2293299,2293356,08645-211023,Cell:98482 85518
08645-21102 4 (Study Material)
Website: www.anucde.com, e-mail:anucde@yahoo.com

CONTENTS

OPERATING SYSTEMS

UNIT – I

1 INTRODUCTION

An **operating system** is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how varied they are in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between.

Operating systems for handheld computers are designed to provide an environment in which a user can easily interface with the computer to execute programs. Thus, some operating systems are designed to be *convenient*, others to be *efficient*, and others some combination of the two. To truly understand what operating systems are, we must first understand how they have developed. In this chapter, after offering a general description of what operating systems do, we trace the development of operating systems from the first hands-on systems through multiprogrammed and time-shared systems to PCs and handheld computers. We also discuss operating system variations, such as parallel, real-time, and embedded systems. As we move through the various stages, we see how the components of operating systems evolved as natural solutions to problems in early computer systems.

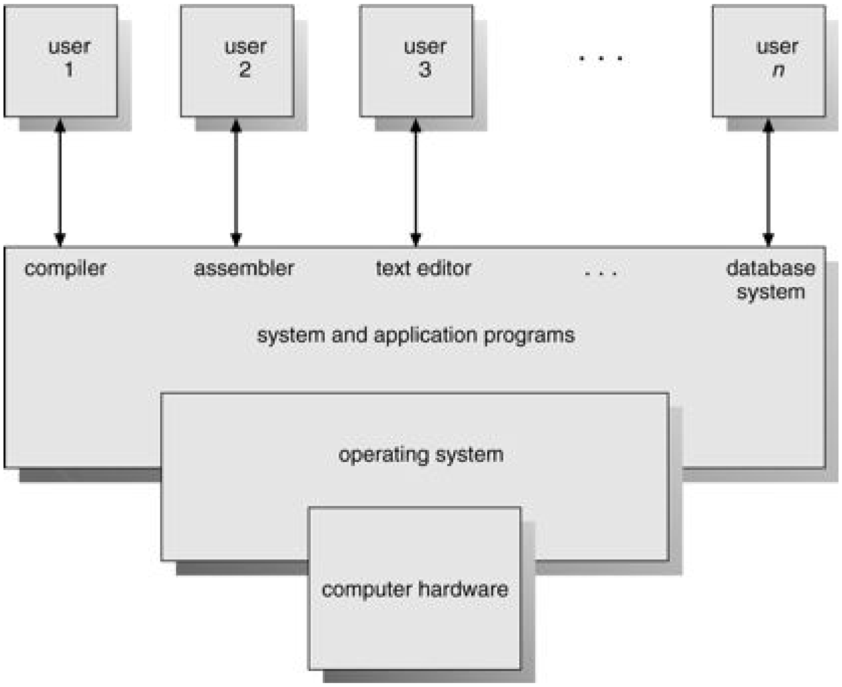
1.1 What Operating Systems Do

We begin our discussions by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: the *hardware*, the *operating system*, the *application programs*, and the *Users* (Figure 1.1).

The **hardware**—the **central processing unit (CPU)**, the **memory**, and the **input/output (I/O) devices**—provides the basic computing resources for the system. The **application programs**—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls and coordinates the use of the hardware among the various application programs for the various users. We can also look at a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system.

An operating system is similar to a *government*. Like a government, it performs no useful function by itself. It simply provides an *environment* within which other programs can do useful work. To understand more fully the operating systems role, we next explore operating systems from two viewpoints: that of the user and that of the system.

Figure 1.1 Abstract view of the components of a computer system.



1.1.1 User View

The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and none paid to **resource utilization**—how various hardware and software resources are shared. Performance is, of course, important to the user; but the demands placed on the system by a single user are too low to make resource utilization an issue.

In some cases, a user sits at a terminal connected to a **mainframe** or **minicomputer**. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization—to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share. In still other cases, users sit at **workstations** connected to networks of other workstations and servers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers—file, compute and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

Recently, many varieties of handheld computers have come into fashion. These devices are mostly standalone units used singly by individual users. Some are connected to networks, either directly by wire or (more often) through wireless modems. Because of power and interface limitations, they perform relatively few remote operations. Their operating systems are designed mostly for individual usability, but performance per amount of battery life is important as well. Some computers have little or no user view. For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but mostly they and their operating systems are designed to run without user intervention.

1.1.2 System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources—hardware and software—that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on.

The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer. A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

1.1.3 Defining Operating Systems

We have looked at the operating system's role from the views of the user and of the system. How, though, can we define what an operating system is? In general, we have no completely adequate definition of an operating system. Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Toward this goal, computer hardware is constructed. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

In addition, we have no universally accepted definition of what is part of the operating system. A simple viewpoint is that it includes everything a vendor ships when you order "the operating system." The features included, however, vary greatly across systems. Some systems take up less than 1 megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are entirely based on graphical windowing systems. (A kilobyte, or KB, is 1,024 bytes; a megabyte, or MB, is

1, 0242bytes; and a gigabyte, or GB, is 1, 0243 bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes.)

A more common definition is that the operating system is the one program running at all times on the computer (usually called the **kernel**), with all else being systems programs and application programs. This last definition is the one that we generally follow. The matter of what constitutes an operating system has become increasingly important. In 1998, the United States Department of Justice filed suit against Microsoft, in essence claiming that Microsoft included too much functionality in its operating systems and thus prevented application vendors from competing. For example, a web browser was an integral part of the operating system. As a result, Microsoft was found guilty of using its operating system monopoly to limit competition.

1.1.4 System Goals

It is easier to define an operating system by what it *does* than by what it *is*, but even this can be tricky. The primary goal of some operating systems is *convenience for the user*. Operating systems exist because computing with them is supposedly easier than computing without them. As we have seen, this view is particularly clear when you look at operating systems for small PCs. The primary goal of other operating systems is *efficient* operation of the computer system. This is the case for large, shared, multiuser systems. These systems are expensive, so it is desirable to make them as efficient as possible. These two goals—convenience and efficiency—are sometimes contradictory. In the past, efficiency was often more important than convenience .Thus, much of operating-system theory concentrates on optimal use of computing resources.

Operating systems have also evolved over time in ways that have affected system goals. For example, UNIX started with a keyboard and printer as its interface, limiting its convenience for users. Over time, hardware changed, and UNIX was ported to new hardware with more user-friendly interfaces. Many **graphic user interfaces (GUIs)** were added, allowing UNIX to be more convenient to use while still concentrating on efficiency.

Designing any operating system is a complex task. Designers face many tradeoffs, and many people are involved not only in bringing the operating system to fruition but also in constantly revising and updating it. How well any given operating system meets its design goals is open to debate and involves subjective judgments on the part of different users.

To examine more closely what operating systems are and what they do, we next consider how they have developed over the past 50 years. By tracing that evolution, we can identify the common elements of operating systems and see how and why these systems have developed as they have. Operating systems and computer architecture have influenced each other a great deal.

To facilitate the use of the hardware, researchers developed operating systems. Users of the operating systems then proposed changes in hardware design to simplify them. In this short historical review, notice how identification of operating system problems led to the introduction of new hardware features.

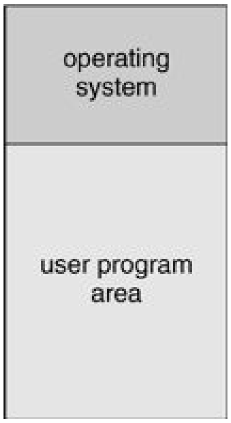
1.2 Mainframe Systems

Mainframe computer systems were the first computers used to tackle many commercial and scientific applications. In this section, we trace the growth of mainframe systems from simple **batch systems**, in which the computer runs one—and only one—application, to **time-shared systems**, which allow user interaction with the computer system.

1.2.1 Batch Systems

Early computers were physically enormous machines run from consoles. The common input devices were card readers and tape drives. The common output devices were line printers, tape drives, and card punches.

Figure 1.2 Memory layout for a simple batch system.



The user did not interact directly with the computer system. Rather, the user prepared a job—which The user did not interact directly with the computer system. Rather, the user prepared a job—which consisted of the program, the data, and some control information about the nature of the job (control cards)—and submitted it to the computer operator.

The job was usually in the form of punch cards. At some later time (after minutes, hours, or days), the output appeared. The output consisted of the result of the program, as well as a dump of the final memory and register contents for debugging. The operating system in these early computers was fairly simple. Its major task was to transfer control automatically from one job to the next. The operating system was always resident in memory (Figure 1.2).

To speed up processing, operators batched together jobs with similar needs and ran them through the computer as a group. Thus, the programmers would leave their programs with the operator. The operator would sort programs into batches with similar requirements and, as the computer became available, would run each batch. The output from each job would be sent back to the appropriate programmer.

In this execution environment, the CPU is often idle, because it works so much faster than the mechanical I/O devices. Even a slow CPU works in the microsecond range, executing thousands of instructions per second. A fast card reader, in contrast, might read 1200 cards per minute (or 20 cards per second). Thus, the difference in speed between the CPU and its I/O devices may be three orders of magnitude or more.

Over time, of course, improvements in technology and the introduction of disks resulted in faster I/O devices. However, CPU speeds increased even more, so the problem was not only unresolved but exacerbated. The introduction of disk technology allowed the operating system to keep all jobs on a disk, rather than in a serial card reader. With direct access to several jobs, the operating system could perform job scheduling to use resources and perform tasks more efficiently. We discuss a few important aspects of job and CPU scheduling here;

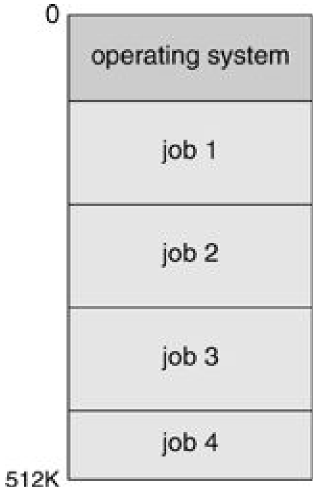
1.2.2 Multiprogrammed Systems

The most important aspect of job scheduling is the ability to multi program. A single user cannot, in general, keep either the CPU or the I/O devices busy at all times.

Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute. The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.3). This set of jobs is a subset of the jobs kept in the job pool—which contains all jobs that enter the system—since the number of jobs that can be kept simultaneously in memory is usually much smaller than the number of jobs that can be kept in the job pool. The operating system picks and begins to execute one of the jobs in the 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 is switched 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.)

Figure 1.3 Memory layout for a multiprogramming system.



Multiprogramming represents the first instance in which the operating system was required to make decisions for the users. Multiprogrammed operating systems are therefore fairly sophisticated. As mentioned, all the jobs that enter the system are kept in the job pool. This pool consists of all processes residing on disk awaiting allocation of main 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. When the operating system selects a job from the job pool, it loads that job into memory for execution.

1.2.3 Time-Sharing Systems

Multiprogrammed batched systems provided an environment in which the various system resources (for example, CPU, memory, peripheral devices) were utilized effectively, but it did 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** (or **hands-on**) **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 keyboard or a mouse, and waits for immediate results. 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, ach user is given the impression that the entire computer system is dedicated to her use, even though it is being shared among many users.

A time-shared operating system uses CPU scheduling and multi programming 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 commonly referred to as 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 operating systems are even more complex than multiprogrammed operating systems. In both, several jobs must be kept simultaneously in memory, so the system must have memory management and protection. To obtain a reasonable response time, the system may have to swap jobs in and out of main memory to the disk that now serves as a backing store for main memory.

A common method for achieving this goal is **virtual memory**, a technique that allows the execution of a job that is not completely in memory. 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.

The idea of time sharing was demonstrated as early as 1960; but since time-shared systems are difficult and expensive to build, they did not become common until the early 1970s. Although some batch processing is still done, most systems today, including personal computers, use time sharing.

1.3 Desktop Systems

Personal computers (PCs) appeared in the 1970s. During their first decade, the CPUs in PCs lacked the features needed to protect an operating system from user programs. PC operating systems therefore were neither multiuser nor multitasking. Contemporary CPUs in PCs typically now provide such features. However, unlike multiprogrammed

and time-shared systems whose primary goals are maximizing CPU and peripheral utilization, PC operating systems must also maximize user convenience and responsiveness. These systems include PCs running Microsoft Windows and the Apple Macintosh.

The MS-DOS operating system from Microsoft has been superseded by multiple flavors of Microsoft Windows. The Apple Macintosh operating system has been ported to more advanced hardware and now includes features such as virtual memory and multitasking. With the release of Mac OS X, the core of the operating system is now based on Mach and FreeBSD UNIX for scalability, performance, and features, but it retains the same rich GUI.

Linux, an open source UNIX operating system available for PCs, has all of these features as well and has become quite popular. Operating systems for these computers have benefited in several ways from the development of operating systems for mainframes. Of course, some of the design decisions made for mainframe operating systems are not appropriate for smaller systems. Because hardware costs for microcomputers are relatively low, one individual generally has sole use of a microcomputer.

Thus, whereas efficient CPU utilization is a primary concern for mainframes, it is no longer so important for systems supporting a single user. Other design decisions still apply, however. For example, file protection was, at first, not necessary on personal machines. However, these computers are now often connected to other computers over local-area networks or other Internet connections. When other computers and other users can access the files on a PC, file protection again becomes a necessary feature of the operating system.

Indeed, the lack of such protection has made it easy for malicious programs to destroy data on systems such as MS-DOS. These programs may be self-replicating and may spread rapidly via **worm** or **virus** mechanisms and disrupt entire companies or even worldwide networks. Advanced time-sharing features such as protected memory and file permissions are not enough, on their own, to safeguard a system from attack.

1.4 Multiprocessor Systems

Most systems to date have been single-processor systems; that is, they have had only one main CPU. However, **multiprocessor systems** (also known as **parallel systems** or **tightly coupled systems**) are growing in importance. Such systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices.

Multiprocessor systems have three main advantages.

1. Increased throughput. By increasing the number of processors, we hope to get more work done in less time. The speed-up ratio with *N* processors is not *N*; 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, a group of *N* programmers working closely together does 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. Blade servers are a recent development in which multiple processor boards, I/O boards, and networking boards can be 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.

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 must pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether. This ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Systems designed for graceful degradation are also called **fault tolerant**.

Note that the third advantage, continued operation in the presence of failures, requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected. The Tandem system uses both hardware and software duplication to ensure continued operation despite faults. The system consists of two identical processors, each with its own local memory. The processors are connected by a bus. One processor is the primary and the other is the backup. Two copies are kept of each process: one on the primary processor and the other on the backup. At fixed checkpoints in the execution of the system, the state information of each job—including a copy of the memory image—is copied from the primary machine to the backup. If a failure is detected, the backup copy is activated and is restarted from the most recent checkpoint. This solution is expensive, since it involves considerable hardware duplication.

Two types of multiple-processor systems are in use today. The most common use **symmetric multiprocessing (SMP)**, in which each processor runs a copy of the operating system and these copies communicate with one another as needed. Some systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master slave relationship. The master processor schedules and allocates work to the slave processors.

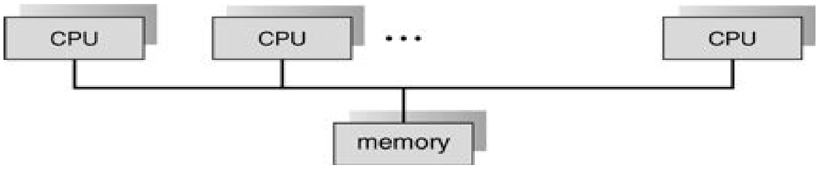
SMP means that all processors are peers; no master-slave relationship exists between processors. Each processor, as mentioned, concurrently runs a copy of the operating system. Figure 1.4 illustrates a typical SMP architecture. An example of the SMP system is Solaris, a commercial version of UNIX designed by Sun Microsystems. A Solaris system can be configured to employ dozens of processors, all running copies of UNIX.

The benefit of this model is that many processes can run simultaneously—*N* processes can run if there are *N* CPUs—without causing a significant deterioration of performance. 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. Virtually all modern operating systems—including Windows 2000, Windows XP, 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 master and multiple slaves. For instance, Sun's operating system SunOS Version 4 provides asymmetric multiprocessing, whereas Version 5 (Solaris) is symmetric on the same hardware.

As microprocessors become less expensive and more powerful, additional operating system functions are off-loaded to slave processors (or **back-ends**). For example, it is fairly easy to add a microprocessor with its own memory to manage a disk system. The microprocessor could receive a sequence of requests from the main CPU and implement 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 fact, this use of microprocessors has become so common that it is no longer considered multiprocessing.

Figure 1.4 Symmetric multiprocessing architecture



1.5 EVOLUTION OF OPERATING SYSTEMS

Operating systems have been evolving over the years. We can roughly divide them into five distinct generations that are characterised by hardware component technology, software development, and mode of delivery of computer services.

0th Generation

The term 0th generation is used to refer to the period of development of computing, which predated the commercial production and sale of computer equipment. We consider that the period might be way back when Charles Babbage invented the Analytical Engine. Afterwards the computers by John Atanasoff in 1940; the Mark I, built by Howard Aiken and a group of IBM engineers at Harvard in 1944; the ENIAC, designed and constructed at the University of Pencylvania by Wallace Eckert and John Mauchly and the EDVAC, developed in 1944-46 by John Von Neumann, Arthur Burks, and Herman Goldstine (which was the first to fully implement the idea of the stored program and serial execution of instructions) were designed. The development of EDVAC set the stage for the evolution of commercial computing and operating system software. The hardware component technology of this period was *electronic vacuum tubes*.

The actual operation of these early computers took place **without the benefit of an operating system**. Early programs were written in machine language and each contained code for initiating operation of the computer itself.

The mode of operation was called “open-shop” and this meant that users signed up for computer time and when a user’s time arrived, the entire (in those days quite large) computer system was turned over to the user. The individual user (programmer) was responsible for all machine set up and operation, and subsequent clean-up and preparation for the next user. This system was clearly inefficient and dependent on the varying competencies of the individual programmer as operators.

First Generation (1951-1956)

The first generation marked the beginning of commercial computing, including the introduction of Eckert and Mauchly’s UNIVAC I in early 1951, and a bit later, the IBM 701 which was also known as the Defence Calculator. The first generation was characterised again by the vacuum tube as the active component technology.

Operation continued without the benefit of an operating system for a time. The mode was called “closed shop” and was characterised by the appearance of hired operators who would select the job to be run, initial program load the system, run the user’s program, and then select another job, and so forth. Programs began to be written in higher level, procedure-oriented languages, and thus the operator’s routine expanded. The operator now selected a job, ran the translation program to assemble or compile the source program, and combined the translated object program along with any existing library programs that the program might need for input to the linking program, loaded and ran the composite linked program, and then handled the next job in a similar fashion.

Application programs were run one at a time, and were translated with absolute computer addresses that bound them to be loaded and run from these reassigned storage addresses set by the translator, obtaining their data from specific physical I/O device. There was no provision for moving a program to different location in storage for any reason. Similarly, a program bound to specific devices could not be run at all if any of these devices were busy or broken.

The inefficiencies inherent in the above methods of operation led to the development of the ***mono -programmed operating system***, which eliminated some of the human intervention in running job and provided programmers with a number of desirable functions. The OS consisted of a permanently resident kernel in main storage, and a job scheduler and a number of utility programs kept in secondary storage. User application programs were preceded by control or specification cards (in those day, computer program were submitted on data cards) which informed the OS of what system resources (software resources such as compilers and loaders; and hardware resources such as tape drives and printer) were needed to run a particular application. The systems were designed to be operated as batch processing system.

These systems continued to operate under the control of a human operator who initiated operation by mounting a magnetic tape that contained the operating system executable code onto a "boot device", and then pushing the IPL (Initial Program Load) or "boot" button to initiate the bootstrap loading of the operating system. Once the system was loaded, the operator entered the date and time, and then initiated the operation of the job scheduler program which read and interpreted the control statements, secured the needed resources, executed the first user program, recorded timing and accounting information, and then went back to begin processing of another user program, and so on, as long as there were programs waiting in the input queue to be executed.

The first generation saw the evolution from hands-on operation to closed shop operation to the development of mono-programmed operating systems. At the same time, the development of programming languages was moving away from the basic machine languages; first to assembly language, and later to procedure oriented languages, the most significant being the development of FORTRAN by John W. Backus in 1956. Several problems remained, however, the most obvious was the inefficient use of system resources, which was most evident when the CPU waited while the relatively slower, mechanical I/O devices were reading or writing program data. In addition, system protection was a problem because the operating system kernel was not protected from being overwritten by an erroneous application program.

Second Generation (1956-1964)

The second generation of computer hardware was most notably characterised by ***transistors*** replacing vacuum tubes as the hardware component technology. In addition, some very important changes in hardware and software architectures occurred during this period. For the most part, computer systems remained card and tape-oriented systems. Significant use of random access devices, that is, disks, did not appear until towards the end of the second generation. Program processing was, for the most part, provided by large centralised computers operated under ***mono-programmed batch processing operating systems***.

The most significant innovations addressed the problem of excessive central processor delay due to waiting for input/output operations. Recall that programs were executed by processing the machine instructions in a strictly sequential order. As a result, the CPU, with its high speed electronic component, was often forced to wait for completion of I/O operations which involved mechanical devices (card readers and tape drives) that were order of magnitude slower. This problem led to the introduction of the data channel, an integral and special-purpose computer with its own instruction set, registers, and control unit designed to process input/output operations separately and asynchronously from the operation of the computer's main CPU near the end of the first generation, and its widespread adoption in the second generation.

The data channel allowed some I/O to be buffered. That is, a program's input data could be read "ahead" from data cards or tape into a special block of memory called a buffer. Then, when the user's program came to an input statement, the data could be transferred from the buffer locations at the faster main memory access speed rather than the slower I/O device speed. Similarly, a program's output could be written another buffer and later moved from the buffer to the printer, tape, or card punch. What made this all work was the data channel's ability to work asynchronously and concurrently with the main processor. Thus, the slower mechanical I/O could be happening concurrently with main program processing. This process was called I/O overlap.

The data channel was controlled by a channel program set up by the operating system I/O control routines and initiated by a special instruction executed by the CPU. Then, the channel independently processed data to or from the buffer. This provided communication from the CPU to the data channel to initiate an I/O operation. It remained for the channel to communicate to the CPU such events as data errors and the completion of a transmission. At first, this communication was handled by polling – the CPU stopped its work periodically and polled the channel to determine if there is any message.

Polling was obviously inefficient (imagine stopping your work periodically to go to the post office to see if an expected letter has arrived) and led to another significant innovation of the second generation – the interrupt. The data channel was able to interrupt the CPU with a message – usually "I/O complete." Infact, the interrupt idea was later extended from I/O to allow signalling of number of exceptional conditions such as arithmetic overflow, division by zero and time-run-out. Of course, interval clocks were added in conjunction with the latter, and thus operating system came to have a way of regaining control from an exceptionally long or indefinitely looping program.

These hardware developments led to enhancements of the operating system. I/O and data channel communication and control became functions of the operating system, both to relieve the application programmer from the difficult details of I/O programming and to protect the integrity of the system to provide improved service to users by segmenting jobs and running shorter jobs first (during "prime time") and relegating longer jobs to lower priority or night time runs. System libraries became more widely available and more comprehensive as new utilities and application software components were available to programmers.

In order to further mitigate the I/O wait problem, system were set up to spool the input batch from slower I/O devices such as the card reader to the much higher speed tape drive and similarly, the output from the higher speed tape to the slower printer. In this scenario, the user submitted a job at a window, a batch of jobs was accumulated and spooled from cards to tape "off line," the tape was moved to the main computer, the jobs were run, and their output was collected on another tape that later was taken to a satellite computer for off line tape-to-printer output. User then picked up their output at the submission windows.

Toward the end of this period, as random access devices became available, tape-oriented operating system began to be replaced by disk-oriented systems. With the more sophisticated disk hardware and the operating system supporting a greater portion of the programmer’s work, the computer system that users saw was more and more removed from the actual hardware-users saw a virtual machine.

The second generation was a period of ***intense operating system development***. Also it was the period for sequential batch processing. But the sequential processing of one job at a time remained a significant limitation. Thus, there continued to be low CPU utilisation for I/O bound jobs and low I/O device utilisation for CPU bound jobs. This was a major concern, since computers were still very large (room-size) and expensive machines. Researchers began to experiment with multiprogramming and multiprocessing in their computing services called the time-sharing system. A noteworthy example is the Compatible Time Sharing System (CTSS), developed at MIT during the early 1960s.

Third Generation (1964-1979)

The third generation officially began in April 1964 with IBM’s announcement of its System/360 family of computers. Hardware technology began to use integrated circuits (ICs) which yielded significant advantages in both speed and economy.

Operating system development continued with the introduction and widespread adoption of multiprogramming. This marked first by the appearance of more sophisticated I/O buffering in the form of spooling operating systems, such as the HASP (Houston Automatic Spooling) system that accompanied the IBM OS/360 system. These systems worked by introducing two new systems programs, a system reader to move input jobs from cards to disk, and a system writer to move job output from disk to printer, tape, or cards. Operation of spooling system was, as before, transparent to the computer user who perceived input as coming directly from the cards and output going directly to the printer.

The idea of taking fuller advantage of the computer’s data channel I/O capabilities continued to develop. That is, designers recognised that I/O needed only to be initiated by a CPU instruction – the actual I/O data transmission could take place under control of separate and asynchronously operating channel program. Thus, by switching control of the CPU between the currently executing user program, the system reader program, and the system writer program, it was possible to keep the slower mechanical I/O device running and minimizes the amount of time the CPU spent waiting for I/O completion. The net result was an increase in system throughput and resource utilisation, to the benefit of both user and providers of computer services.

This concurrent operation of three programs (more properly, apparent concurrent operation, since systems had only one CPU, and could, therefore execute just one instruction at a time) required that additional features and complexity be added to the operating system. First, the fact that the input queue was now on disk, a direct access device, freed the system scheduler from the first-come-first-served policy so that it could select the “best” next job to enter the system (looking for either the shortest job or the highest priority job in the queue). Second, since the CPU was to be shared by the user program, the system reader, and the system writer, some processor allocation rule or policy was needed. Since the goal of spooling was to increase resource utilisation by enabling the slower I/O devices to run asynchronously with user program processing, and since I/O processing required the CPU only for short periods to initiate data channel instructions, the CPU was dispatched to the reader, the writer, and the program in that order. Moreover, if the writer or the user program was executing when something became available to read, the reader program would preempt the currently executing program to regain control of the CPU for its initiation instruction, and the writer program would preempt the user program for the same purpose. This rule, called the static priority rule with preemption, was implemented in the operating system as a system dispatcher program.

The spooling operating system in fact had multiprogramming since more than one program was resident in main storage at the same time. Later this basic idea of multiprogramming was extended to include more than one active user program in memory at time. To accommodate this extension, both the scheduler and the dispatcher were enhanced. The scheduler became able to manage the diverse resource needs of the several concurrently active used programs, and the dispatcher included policies for allocating processor resources among the competing user programs. In addition, memory management became more sophisticated in order to assure that the program code for each job or at least that part of the code being executed, was resident in main storage.

The advent of large-scale multiprogramming was made possible by several important hardware innovations such as:

- The widespread availability of large capacity, high-speed disk units to accommodate the spooled input streams and the memory overflow together with the maintenance of several concurrently active program in execution.
- Relocation hardware which facilitated the moving of blocks of code within memory without any undue overhead penalty.
- The availability of storage protection hardware to ensure that user jobs are protected from one another and that the operating system itself is protected from user programs.
- Some of these hardware innovations involved extensions to the interrupt system in order to handle a variety of external conditions such as program malfunctions, storage protection violations, and machine checks in addition to I/O interrupts. In addition, the interrupt system became the technique for the user program to request services from the operating system kernel.
- The advent of privileged instructions allowed the operating system to maintain coordination and control over the multiple activities now going on with in the system.

Successful implementation of multiprogramming opened the way for the development of a new way of delivering computing services-time-sharing. In this environment, several terminals, sometimes up to 200 of them, were attached (hard wired or via telephone lines) to a central computer. Users at their terminals, “logged in” to the central system, and worked interactively with the system. The system’s apparent concurrency was enabled by the multiprogramming operating system. Users shared not only the system hardware but also its software resources and file system disk space.

Fourth Generation (1979 – Present)

The fourth generation is characterised by the appearance of the personal computer and the workstation. Miniaturisation of electronic circuits and components continued and large scale integration (LSI), the component technology of the third generation, was replaced by very large scale integration (VLSI), which characterizes the fourth generation. VLSI with its capacity for containing thousands of transistors on a small chip, made possible the development of desktop computers with capabilities exceeding those that filled entire rooms and floors of building just twenty years earlier.

The operating systems that control these desktop machines have brought us back in a full circle, to the open shop type of environment where each user occupies an entire computer for the duration of a job's execution. This works better now, not only because the progress made over the years has made the virtual computer resulting from the operating system/hardware combination so much easier to use, or, in the words of the popular press "user-friendly."

However, improvements in hardware miniaturisation and technology have evolved so fast that we now have inexpensive workstation – class computers capable of supporting multiprogramming and time-sharing. Hence the operating systems that supports today's personal computers and workstations look much like those which were available for the minicomputers of the third generation. Examples are Microsoft's DOS for IBM-compatible personal computers and UNIX for workstation.

However, many of these desktop computers are now connected as networked or distributed systems. Computers in a networked system each have their operating systems augmented with communication capabilities that enable users to remotely log into any system on the network and transfer information among machines that are connected to the network. The machines that make up distributed system operate as a virtual single processor system from the user's point of view; a central operating system controls and makes transparent the location in the system of the particular processor or processors and file systems that are handling any given program.

1.6 FUNCTIONS OF OS

The main functions of an operating system are as follows:

- Process Management
- Memory Management
- Secondary Storage Management
- I/O Management
- File Management
- Protection
- Networking Management
- Command Interpretation.

1.6.1 Process Management

The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general.

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

- The creation and deletion of both user and system processes
- The suspension and resumption of processes.
- The provision of mechanisms for process synchronization
- The provision of mechanisms for deadlock handling.

1.6.2 Memory Management

Memory is the most expensive part in the computer system. Memory is a large array of words or bytes, each with its own address. Interaction is achieved through a sequence of reads or writes of specific memory address. The CPU fetches from and stores in memory.

There are various algorithms that depend on the particular situation to manage the memory. Selection of a memory management scheme for a specific system depends upon many factors, but especially upon 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.

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and deallocate memory space as needed.

1.6.3 Secondary Storage Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory during execution. Since the main memory is too small to permanently accommodate all data and program, the computer system must provide secondary storage to backup main memory. Most modem computer systems use disks as the primary on-line storage of information, of both programs and data. Most programs, like compilers, assemblers, sort routines, editors, formatters, and so on, are stored on the disk until loaded into memory, and 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.

There are few alternatives. Magnetic tape systems are generally too slow. In addition, they are limited to sequential access. Thus tapes are more suited for storing infrequently used files, where speed is not a primary concern.

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

- Free space management
- Storage allocation

- Disk scheduling.

1.6.4 I/O Management

One of the purposes of an operating system is to hide the peculiarities or 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 system. The operating system is responsible for the following activities in connection to I/O management:

- A buffer caching system
- To activate a general device driver code
- To run the driver software for specific hardware devices as and when required.

1.6.5 File Management

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms: magnetic tape, disk, and drum are the most common forms. Each of these devices has it own characteristics and physical organisation.

For convenient use of the computer system, 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. Files are mapped, by the operating system, onto physical devices.

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 or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general a files is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. It is a very general concept.

The operating system implements the abstract concept of the file by managing mass storage device, such as types and disks. Also files are normally organised into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed.

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

- The creation and deletion of files.
- The creation and deletion of directory.
- The support of primitives for manipulating files and directories.
- The mapping of files onto disk storage.
- Backup of files on stable (non volatile) storage.
- Protection and security of the files.

1.6.6 Protection

The various processes in an operating system must be protected from each other's activities. For that purpose, various mechanisms which can be used to ensure that the files, memory segment, CPU and other resources can be operated on only by those processes that have gained proper authorisation from the operating system.

For example, memory addressing hardware ensures that a process can only execute within its own address space. The timer ensures that no process can gain control of the CPU without relinquishing it. Finally, no process is allowed to do its own I/O, to protect the integrity of the various peripheral devices. Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement.

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 subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorised or incompetent user. More on protection and security can be studied in Unit-4 of Block-3.

1.6.7 Networking

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with each other through various communication lines, such as high speed buses or telephone lines. Distributed systems vary in size and function. They may involve microprocessors, workstations, minicomputers, and large general purpose computer systems.

The processors in the system are connected through a communication network, which can be configured in the number of different ways. The network may be fully or partially connected. The communication network design must consider routing and connection strategies and the problems of connection and security.

A distributed system provides the user with access to the various resources the system maintains. Access to a shared resource allows computation speed-up, data availability, and reliability.

1.6.8 Command Interpretation

One of the most important components of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system.

Many commands are given to the operating system by control statements. When a new job is started in a batch system or when a user logs-in to a time-shared system, a program which reads and interprets control statements is automatically executed. This program is variously called (1) the control card interpreter, (2) the command line interpreter, (3) the shell (in Unix), and so on. Its function is quite simple: get the next command statement, and execute it.

The command statements themselves deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking.

Exercises

1.1 What are the two main purposes of an operating system?

1.2 List the four steps necessary to run a program on a completely dedicated machine.

1.3 What is the main advantage of multiprogramming?

1.4 What are the main differences between operating systems for

mainframe computers and PCs?

1.5 In a multiprogramming and time-sharing environment, several users share the system simultaneously. This situation can result in various security problems.

a. What are two such problems?

b. Can we ensure the same degree of security in a time-shared machine as we have in a dedicated machine? Explain your answer.

1.6 Define the essential properties of the following types of operating systems:

a. Batch

b. Interactive

c. Time sharing

d. Real time

e. Network

f. Parallel

g. Distributed

h. Clustered

i. Handheld

1.7 We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to "waste" resources? Why is such a system not really wasteful?

1.8 Under what circumstances would a user be better off using a time-sharing system, rather than a PC or single-user workstation?

1.9 Describe the differences between symmetric and asymmetric multiprocessing. What are three advantages and one disadvantage of multiprocessor systems?

1.10 What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?

1.11 Consider the various definitions of operating system. Consider whether the operating system should include applications such as web browsers and mail programs. Argue both pro and con positions, and support your answers.

1.12 What are the tradeoffs inherent in handheld computers?

1.13 Consider a computing cluster consisting of two nodes running a database. Describe two ways in which the cluster software can manage access to the data on the disk. Discuss the benefits and detriments of each.

2. COMPUTER SYSTEM STRUCTURE

2.1 Introduction

A modern, general-purpose computer system consists of a CPU and a number of device controllers that are connected through a common bus that provides access to shared memory. Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays). The CPU and the device controllers can execute concurrently, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.

2.2 Objectives

After going through this unit, you should be able to:

- understand the I/O structure of a system;
- discuss the storage hierarchy;
- describe various elements of storage system;
- describe hardware protection;

2.3 I/0 Structure

A general-purpose computer system consists of a CPU 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, there may be more than one attached device. For instance, the small computer-systems interface (SCSI) controller can have seven or more devices attached to it. 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. The size of the local buffer within a device controller varies from one controller

to another, depending on the particular device being controlled. For example, the size of the buffer of a disk controller is the same as or a multiple of the size of the smallest addressable portion of a disk, called a sector, which is usually 512 bytes.

I/O Interrupts

To start an I/O operation, the CPU 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. For example, if it finds a read request, the controller will start the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the CPU that it has finished its operation. It accomplishes this communication by triggering an interrupt. This situation will occur, in general, as the result of a user process requesting I/O. Once the I/O is started, two courses of action are possible. In the simplest case, the I/O is started; then, at I/O completion, control is returned to the user process. This case is known as synchronous I/O. The other possibility, called asynchronous I/O, returns control to the user program without waiting for the 1/0 to complete. The 1/0 then can continue while other system operations occur. Waiting for I/O completion may be accomplished in one of two ways. Some computers have a special wait instruction that idles the CPU until the next interrupt. Machines that do not have such an instruction may have a wait loop: Loop: j mp Loop This tight loop simply continues until an interrupt occurs, transferring control to another part of the operating system. Such a loop might also need to poll any 1/0 devices that do not support the interrupt structure; instead, these devices simply set a flag in one of their registers and expect the operating system to notice that flag. If the CPU always waits for 1/0 completion, at most one 1/0 request is outstanding at a time. Thus, whenever an 1/0 interrupt occurs, the operating system knows exactly which device is interrupting. On the other hand, this approach excludes concurrent I/O operations to several devices, and also excludes the possibility of overlapping useful computation with I/O. A better alternative is to start the I/O and then to continue processing other operating-system or user program code. A system call is then needed to allow the user program to wait for I/O completion, if desired. If no user programs are ready to run, and the operating system has no other work to do, we still require the wait instruction or idle loop, as before. We also need to be able to keep track of many I/O requests at the same time. For this purpose, the operating system uses a table containing an entry for each I/O device: the device-status table. Each table entry indicates the device's type, address, and state (not functioning, idle, or busy). If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that device. Since it is possible for other processes to issue requests to the same device, the operating system will also maintain a wait queue-a list of waiting requests-for each I/O device.

An I/O device interrupts when it needs service. When an interrupt occurs, the operating system first determines which I/O device caused the interrupt. It then indexes into the I/O device table to determine the status of that device, and modifies the table entry to reflect the occurrence of the interrupt. For most devices, an interrupt signals completion of an I/O request. If there are additional requests waiting in the queue for this device, the operating system starts processing the next request.

Finally, control is returned from the I/O interrupt. If a process was waiting for this request to complete (as recorded in the device-status table), we can now return control to it. Otherwise, we return to whatever we were doing before the I/O interrupt: to the execution of the user program (the program started an I/O operation and that operation has now finished, but the program has not yet waited for the operation to complete) or to the wait loop (the program started two or more I/O operations and is waiting for a particular one to finish, but this interrupt was from one of the other operations). In a time-sharing system, the operating system could switch to another ready-to-run process.

The schemes used by specific input devices may vary from this one. Many interactive systems allow users to type ahead-to enter data before the data are requested-on the keyboard. In this case, interrupts may occur, signaling the arrival of characters from the terminal, while the device-status block indicates that no program has requested input from this device. If typeahead is to be allowed, then a buffer must be provided to store the typeahead characters until some program wants them. In general, we may need a buffer for each input device.

The main advantage of asynchronous 1/0 is increased system efficiency. While 1/0 is taking place, the system CPU can be used for processing or starting I/Os to other devices. Because 1/0 can be slow compared to processor speed, the system makes efficient use of its facilities. In Section 2.2.2, we describe another mechanism for improving system performance.

DMA Structure

In a simple terminal-input driver, when a line is to be read from the terminal, the first character typed is sent to the computer. When that character is received, the asynchronous-communication (or serial-port) device to which the terminal line is connected interrupts the CPU. When the interrupt request from the terminal arrives, the CPU is about to execute some instruction. (If the CPU is in the middle of executing an instruction, the interrupt is normally held pending the completion of instruction execution.) The address of this interrupted instruction is saved, and control is transferred to the interrupt service routine for the appropriate device.

The interrupt service routine saves the contents of any CPU registers that it will need to use. It checks for any error conditions that might have resulted from the most recent input operation. It then takes the character from the device, and stores that character in a buffer. The interrupt routine must also adjust pointer and counter variables, to be sure that the next input character will be stored at the next location in the buffer. The interrupt routine next sets a flag in memory indicating to the other parts of the operating system that new input has been received. The other parts are responsible for processing the data in the buffer, and for transferring the characters to the program that is requesting input (see Section 2.5). Then, the interrupt service routine restores the contents of any saved registers, and transfers control back to the interrupted instruction. If characters are being typed to a 9600-baud terminal, the terminal can accept and transfer one character approximately every 1 millisecond, or 1000 microseconds. A well-written interrupt service routine to input characters into a buffer may require 2 microseconds per character, leaving 998 microseconds out of every 1000 for CPU computation (and for servicing of other interrupts). Given this disparity, asynchronous I/O is usually assigned a low interrupt priority, allowing other, more important interrupts to be processed first, or even to preempt the current interrupt for another. A high-speed device, however- such as a tape, disk, or communications network- may be able to transmit information at close to memory speeds; if the CPU needs two microseconds to respond to each interrupt and interrupts arrive every four microseconds, for example, that does not leave much time for process execution. To solve this problem, direct memory access (DMA) is used for high-speed I/O devices. 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, rather than the one interrupt per byte (or word) generated for low-speed devices.

The basic operation of the CPU is the same. A user program, or the operating system itself, may request data transfer. The operating system finds a buffer (an empty buffer for input, or a full buffer for output) from a pool of buffers for the transfer. (A buffer is typically 128 to 4,096 bytes, depending on the device type.) Next, a portion of the operating system called a device driver sets the DMA controller registers to use appropriate source and destination addresses, and transfer length. The DMA controller is then instructed to start the 1/0 operation. While the DMA controller is performing the data transfer, the CPU is free to perform other tasks. Since the memory generally can transfer only one word at a time, the DMA controller "steals" memory cycles from the CPU. This cycle stealing can s low down the CPU execution while a DMA transfer is in progress. The DMA controller interrupts the CPU when the transfer has been completed.

2.4 Storage Structure

Computer programs must be in main memory (also called random access memory or RAM) to be executed. Main memory is the only large storage area (millions to billions of bytes) that the processor can access directly. It is implemented in a semiconductor technology called dynamic random-access memory (DRAM), which forms an array of memory words. Each word has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a 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.

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, and so on) 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.

2.4 Storage System

Ideally, we want the programs and data to reside in main memory permanently. This arrangement 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 of both programs and data. Most programs (web browsers, compilers, word processors, spreadsheets, and so on) are stored on a disk until they are loaded into memory. Many programs then use the disk as both a source and a destination of the information for their processing. Hence, the proper management of disk storage is of central importance to a computer system.

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. There are also cache memory, CD-ROM, magnetic tapes, and so on. Each storage system provides the basic functions of storing a datum, and of 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.

Main Memory

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

In the case of I/O, each I/O controller includes registers to hold commands and the data being transferred. Usually, special I/O instructions allow data transfers between these registers and system memory.

To allow more convenient access to I/O devices, many computer architectures provide memory-mapped I/O. In this case, ranges of memory addresses are set aside, and are mapped to the device registers. Reads and writes to these memory addresses cause the data to be transferred to and from the device registers. This method is appropriate for devices that have fast response times, such as video controllers. In the IBM PC, each location on the screen is mapped to a memory location. Displaying text on the screen is almost as easy as writing the text into the appropriate memory-mapped locations.

Memory-mapped I/O is also convenient for other devices, such as the serial and parallel ports used to connect modems and printers to a computer. The CPU transfers data through these kinds of devices by reading and writing a few device registers, called an 1/0 port. To send out a long string of bytes through a memory-mapped serial port, the CPU writes one data byte to the data register, then sets a bit in the control register to signal that the byte is available. The device takes the data byte, and then clears the bit in the control register to signal that it is ready for the next byte. Then, the CPU can transfer the next byte. If the CPU uses polling to watch the control bit, constantly looping to see whether the device is ready, this method of operation is called programmed 110 (PIO). If the CPU does not poll the control bit, but instead receives an interrupt when the device is ready for the next byte, the data transfer is said to be interrupt driven.

Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Memory access may take many cycles of the CPU clock to complete, in which case the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a cache.

Secondary Memory

Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple. Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 5.25 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters. A read-write head "flies" just above each surface of every platter. The heads are attached to a disk arm, which moves all the heads as a unit. The surface of a platter is logically divided into circular tracks, which are subdivided into sectors. The set of tracks that are at one arm position forms a cylinder. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.

When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 200 times per second. Disk speed has two parts. The transfer rate is the rate at which data flow between the drive and the computer. The positioning time, sometimes called the random-access time, consists of the time to move the disk arm to the desired cylinder, called the seek time, and the time for the desired sector to rotate to the disk head, called the rotational latency. Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latencies of several milliseconds. Because the disk head flies on an extremely thin (measured in microns) cushion of air, there is a danger of the head making contact with the disk surface. Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a head crash. A head crash normally cannot be repaired; the entire disk must be replaced. A disk can be removable, allowing different disks to be mounted as needed.

Removable magnetic disks generally consist of one platter, held in a plastic case to prevent damage while not in the disk drive. Floppy disks are inexpensive removable magnetic disks that have a soft plastic case containing a flexible platter. The head of a floppy-disk drive generally sits directly on the disk surface, so the drive is designed to rotate more slowly than a hard-disk drive to reduce the wear on the disk surface. The storage capacity of a floppy disk is typically only 1 MB or so. Removable disks are available that work much like normal hard disks and have capacities measured in gigabytes.

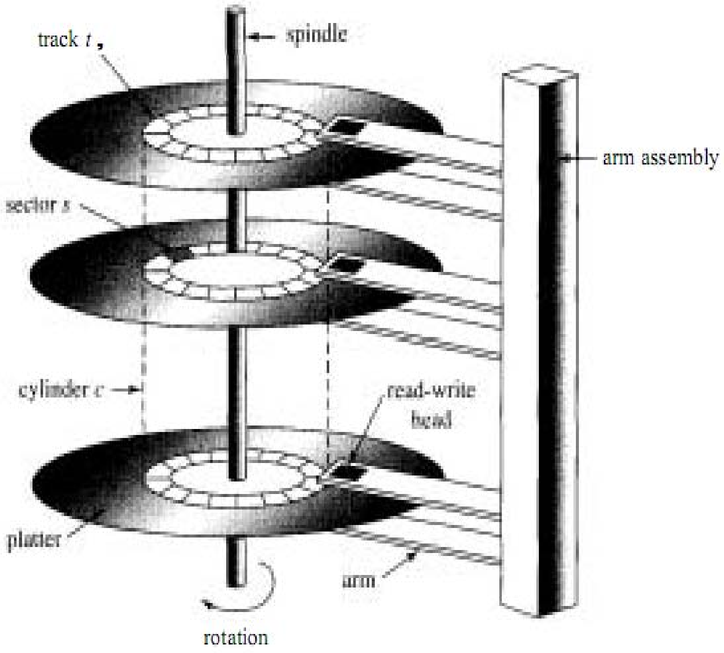


Figure 2.5 Moving-head disk mechanism.

A disk drive is attached to a computer by a set of wires called an 110 bus. Several kinds of buses are available, including enhanced integrated drive electronics (EIDE), advanced technology attachment (ATA), and SCSI buses.

The data transfers on a bus are carried out by special electronic processors called controllers. The host controller is the controller at the computer end of the bus. A disk controller is built into each disk drive. To perform a disk 1/0 operation, the computer places a command into the host controller, typically using memory-mapped I/O ports, as described in Section 2.3.1. The host controller then sends the command via messages to the disk controller, and the disk controller operates the disk-drive hardware to carry out the command.

Disk controllers usually have a built-in cache. Data transfer at the disk drive happens between the cache and the disk surface, and data transfer to the host, at fast electronic speeds, occurs between the cache and the host controller.

Magnetic Tapes

Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of data, its access time is slow in comparison to that of main memory. In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool, and is wound or rewound past a read-write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives. Tape capacities vary greatly, depending on the particular kind of tape drive. Some tapes hold 2 to 3 times more data than does a large disk drive. Tapes and their drivers are usually categorized by width, including 4,8, and 19 millimeters, 1 /4 and 1/2 inch.

2.5 Storage Hierarchy

The wide variety of storage systems in a computer system can be organized in a hierarchy 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 tradeoff 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 tape and core memories, are relegated to museums now that magnetic tape and semiconductor memory have become faster and cheaper. The top three levels of memory in Figure 2.6 may be constructed using semiconductor memory.

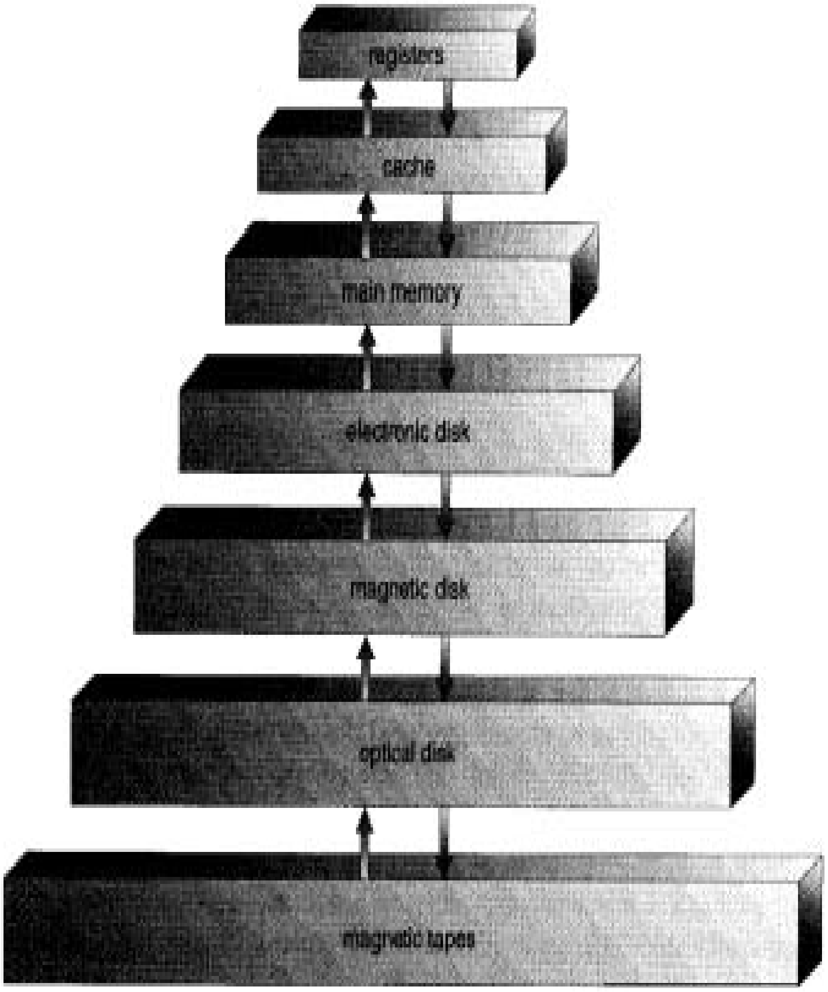


Figure 2.6 Storage-device hierarchy

In addition to having differing speed and cost, the various storage systems are either volatile or nonvolatile. 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 2.6, the storage systems above the electronic disk are volatile, whereas those below are nonvolatile. An electronic disk can be designed to be either volatile or nonvolatile. During normal operation, the electronic disk stores data in a large DRAM array, which is volatile. But many electronic-disk devices contain a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, the electronic-disk controller copies the data from RAM to the magnetic disk. When external power is restored, the controller copies the data back into the RAM.

The design of a complete memory system must balance all these factors:

It uses 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 access-time or transfer-rate disparity exists between two components.

2.6 . Hardware Protection

Early computer systems were single-user programmer-operated systems.

When the programmers operated the computer from the console, they had complete control over the system. As operating systems developed, however, this control was given to the operating system. Early operating systems were called resident monitors, and starting with the resident monitor, the operating system began to perform many of the functions, especially I/O, for which the programmer had previously been responsible.

In addition, to improve system utilization, the operating system began to share system resources among several programs simultaneously. With spooling, one program might have been executing while I/O occurred for other processes; the disk simultaneously held data for many processes. Multiprogramming put several programs in memory at the same time.

This sharing both improved utilization and increased problems. When the system was run without sharing, an error in a program could cause problems for only the one program that was running. With sharing, many processes could be adversely affected by a bug in one program.

For example, consider the simple batch operating system, which provides nothing more than automatic job sequencing. If a program gets stuck in a loop reading input cards, the program will read through all its data and, unless something stops it, will continue reading the cards of the next job, and the next, and so on. This loop could prevent the correct operation of many jobs.

Even more subtle errors can occur in a multiprogramming system, where one erroneous program might modify the program or data of another program, or even the resident monitor itself. MS-DOS and the Macintosh OS both allow this kind of error.

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.

Many programming errors are detected by the hardware. 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 will trap to the operating system. The trap transfers control through the interrupt vector to the operating system, just like an interrupt. Whenever a program error occurs, the operating system must abnormally terminate the program.

This situation is handled by the same code as is 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 can correct and restart the program.

2.6.1 Dual-Mode Operation

To ensure proper operation, we must protect the operating system and all other programs and their data from any malfunctioning program. Protection is needed for any shared resource. The approach taken by many operating systems provides hardware support that allows us to differentiate among various modes of execution.

At the very least, we need two separate modes of operation: user mode and monitor 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: monitor (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system, and one that is executed on behalf of the user. As we shall see, this architectural enhancement is useful for many other aspects of system operation.

At system boot time, the hardware starts in monitor mode. The operating system is then loaded, and starts user processes in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to monitor 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 monitor 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 monitor mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction, but rather treats the instruction as illegal and traps it to the operating system.

The concept of privileged instructions also provides us with the means for the user to interact with the operating system by asking the system to perform some designated tasks that only the operating system should do. Each such request is invoked by the user executing a privileged instruction. Such a request is known as a system call (also called a monitor call or an operating-system function call)-as described .

When a system call is executed, it is 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 monitor mode. The system-call service routine is a part of the operating system. The monitor 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 monitor verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

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 archtecture, 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 possibly disastrous results. More recent and advanced versions of the Intel CPU, such as the Pentium, do provide dual-mode operation. As a result, more recent operating systems, such as Microsoft Windows 2000 and IBM 05/2, take advantage of this feature and provide greater protection for the operating system.

2.5.2 I/O Protection

A user program may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to relinquish the CPU. We can use various mechanisms to ensure that such disruptions cannot take place in the system.

To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system. For I/O protection to be complete, we must be sure that a user program can never gain control of the computer in monitor mode. If it could, 1/0 protection could be compromised.

Consider a computer executing in user mode. It will switch to monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt vector. If a user program, as part of its execution, stores a new address in the interrupt vector, this new address could overwrite the previous address with an address in the user program. Then, when a corresponding trap or interrupt occurred, the hardware would switch to monitor mode, and would transfer control through the (modified) interrupt vector to the user program! The user program could gain control of the computer in monitor mode. In fact, user programs could gain control of the computer in monitor mode in many other ways. In addition, new bugs are discovered every day that can be exploited to bypass system protections. Those topics are discussed in Chapters 18 and 19. Thus, to do I/O, a user program executes a system call to request that the operating system perform 1/0 on its behalf (Figure 2.8).

The operating system, executing in monitor mode, checks that the request is valid, and (if the request is valid) does the I/O requested. The operating system then returns to the user.

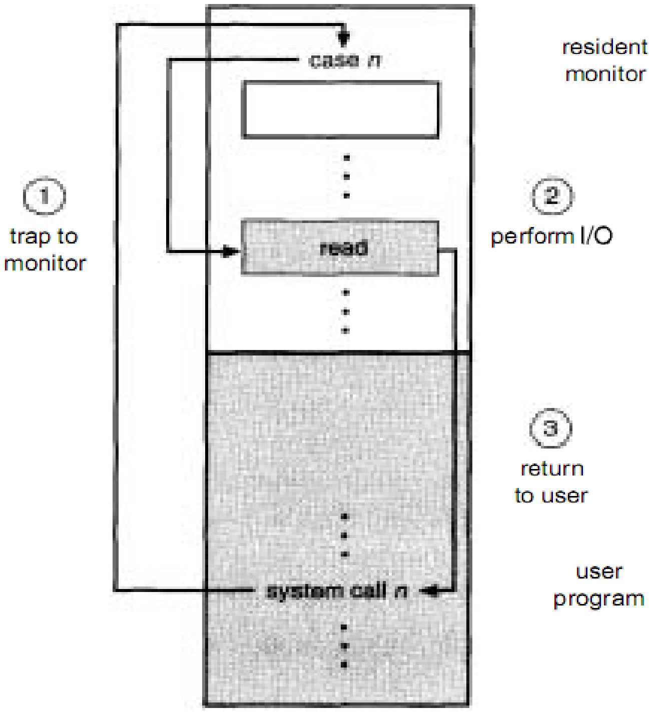


Figure 2.8 Use of a system call to perform I/O.

2.6.3 Memory Protection

To ensure correct operation, we must protect the interrupt vector from modification by a user program. In addition, we must also protect the interrupt-service routines in the operating system from modification. Even if the user did not gain unauthorized control of the computer, modifying the interrupt service routines would probably disrupt the proper operation of the computer system and of its spooling and buffering.

We see then that we must provide memory protection at least for the interrupt vector and the interrupt-service routines of the operating system. In general, we want to protect the operating system from access by user programs, and, in addition, to protect user programs from one another. This protection must be provided by the hardware. It can be implemented in several ways, as we describe in Chapter 9. Here, we outline one such possible implementation.

To separate each program's memory space, we need the ability to determine the range of legal addresses that the program may access, and to protect the memory outside that space. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 2.9. The base register holds the smallest legal physical memory address; the limit register contains the size of the range. For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 inclusive.

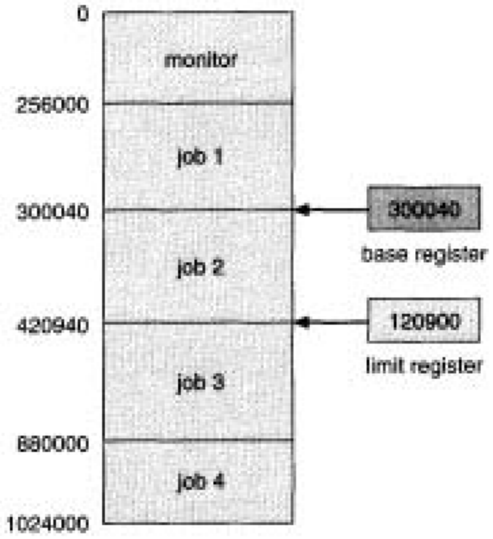


Figure 2.9 A base and a limit register define a logical address space.

Exercises

2.1 Prefetching is a method of overlapping the I/O of a job with that job's own computation. The idea is simple. After a read operation completes and the job is about to start operating on the data, the input device is instructed to begin the next read immediately. The CPU and input device are then both busy. With luck, by the time that the job is ready for the next data item, the input device will have finished reading that data item. The CPU can then begin processing the newly read data, while the input device starts to read the following data. A similar idea can be used for output. In this case, the job creates data that are put into a buffer until an output device can accept them.

Compare the perfecting scheme with spooling, where the CPU overlaps the input of one job with the computation and output of other jobs.

2.2 How does the distinction between monitor mode and user mode function as a rudimentary form of protection (security) system?

2.3 What are the differences between a trap and an interrupt? What is the use of each function?

2.4 For what types of operations is DMA useful? Explain your answer.

2.5 Which of the following instructions should be privileged?

a. Set value of timer.

b. Read the clock.

c. Clear memory.

d. Turn off interrupts.

e. Switch from user to monitor mode.

2.6 Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computers? Give arguments both that it is and that it is not possible.

2.7 Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.

2.8 Protecting the operating system is crucial to ensuring that the computer system operates correctly. Provision of this protection is the reason for dual-mode operation, memory protection, and the timer. To allow maximum flexibility, however, you should also place minimal constraints on the user.

The following is a list of instructions that are normally protected. What is the minimal set of instructions that must be protected?

a. Change to user mode.

b. Change to monitor mode.

c. Read from monitor memory.

d. Write into monitor memory.

e. Fetch an instruction from monitor memory.

f. Turn on timer interrupt.

g. Turn off timer interrupt.

2.9 Give two reasons why caches are useful. What problems do they solve?

What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why

not make it that large and eliminate the device?

2.10 Writing an operating system that can operate without interference from malicious or undebugged user programs requires hardware assistance.

Name three hardware aids for writing an operating system, and describe how they could be used together to protect the operating system.

2.11 Some CPUs provide for more than two modes of operation. What are two possible uses of these multiple modes?

2.12 What are the main differences between a WAN and a LAN?

2.13 What network configuration would best suit the following environments?

a. A dormitory floor

b. A university campus

c. A state

d. A nation

3. PROCESSES

3. PROCESSES

Early computer systems 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, current-day 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, we look more closely at processes. We begin by discussing the process concept and go on to describe various features of processes, including scheduling, creation and termination, and communication. We end the chapter by describing 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 such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. Even if the user can execute only one program at a time, 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 method 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.

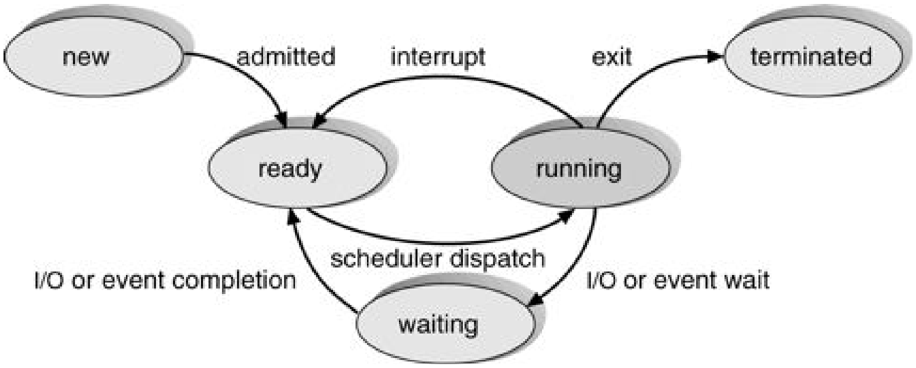
We emphasize that a program by itself is not a process; a program is a *passive* entity, such as a file containing a list of instructions stored on disk, whereas a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources. Although 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 editor program. Each of these is a separate process; and although the text sections are equivalent, the data sections vary. It is also common to have a process that spawns many processes as it runs.

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. Each process may be in one of the following states:

- **New:** The process is being created.
- **Running:** Instructions are being executed.

Figure 3.1 Diagram of process state.



- **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 3.1.

3.1.3 Process Control Block

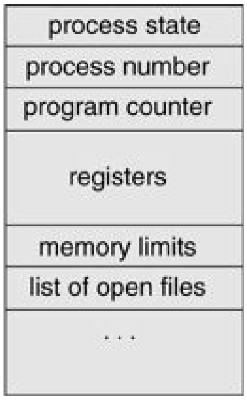
Each process is represented in the operating system by a **process control block (PCB)**—also called a *task control block*. A PCB is shown in Figure 3.2. It contains many pieces of information associated with a specific process, including these:

- *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 3.3).

Figure 3.2 Process control block ((PCB).



- *CPU-scheduling information:* This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

- *Memory-management information:* This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system

- *Accounting information:* This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

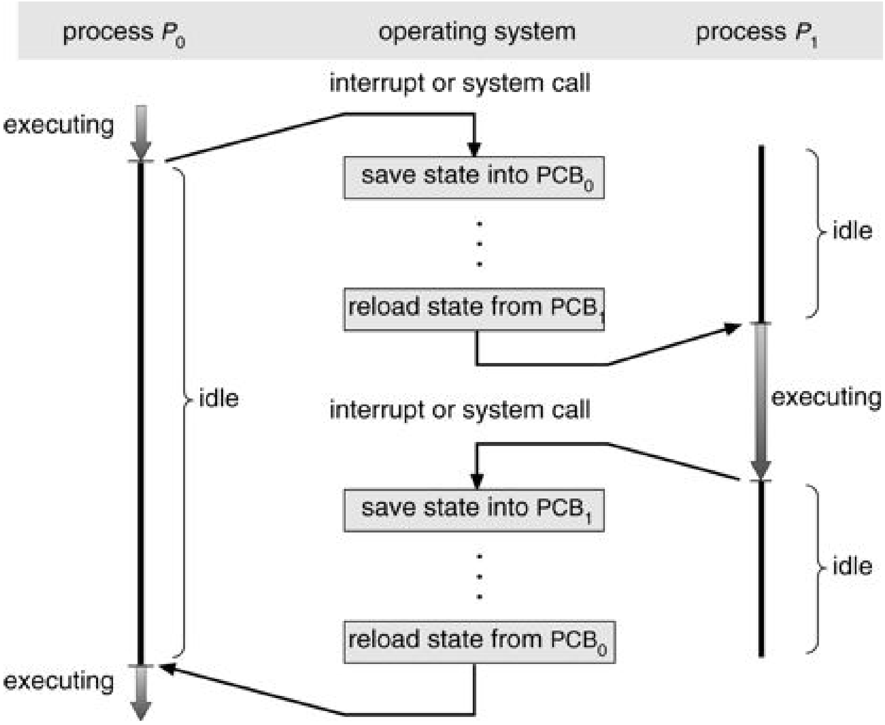
- *I/O status information:* This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

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 one time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Many 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.

Figure 3.3 Diagram showing CPU switch from process to process.



3.2 Process Scheduling

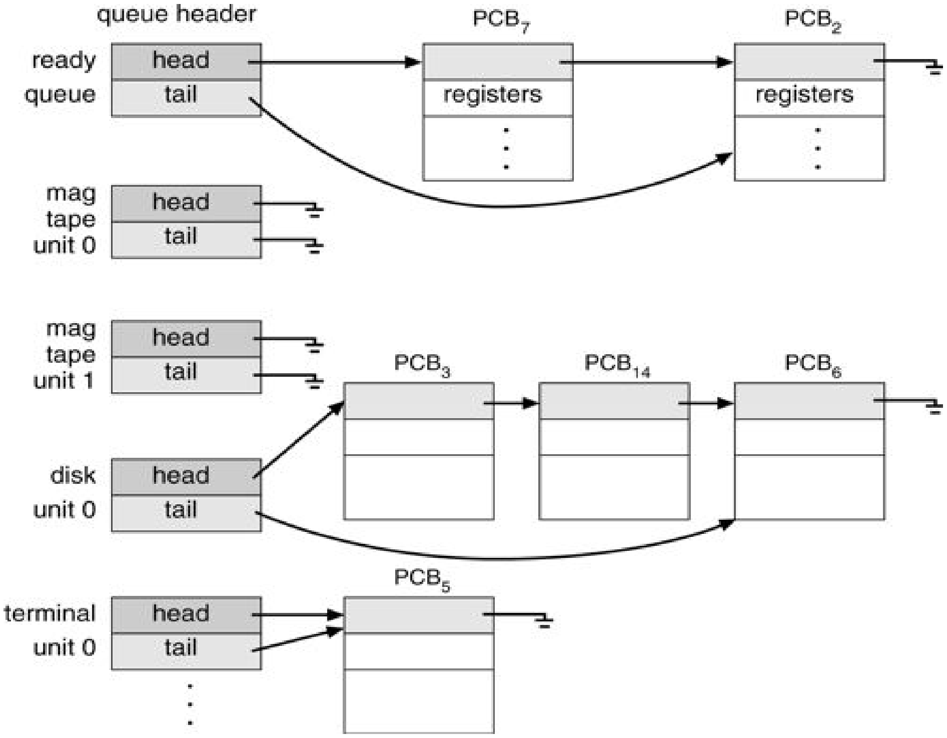
The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

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 uni-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.

Figure 3.4 The ready queue and various I/O device queues.



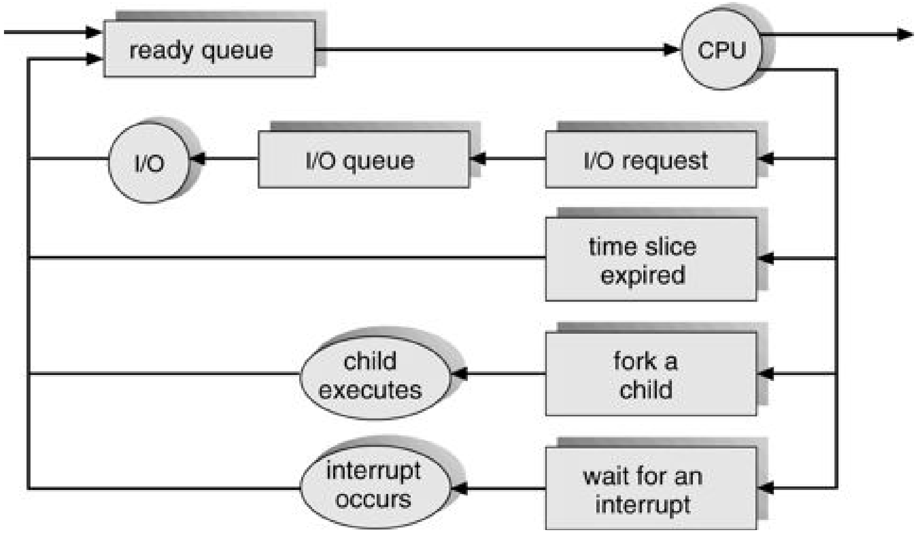
The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a shared device, such as a 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 3.4).

A common representation for a discussion of process scheduling is a **queueing diagram**, such as that in Figure 3.5. 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 is **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.

Figure 3.5 Queueing-diagram representation of process scheduling.



- The process could create a new sub process and wait for the sub process'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.

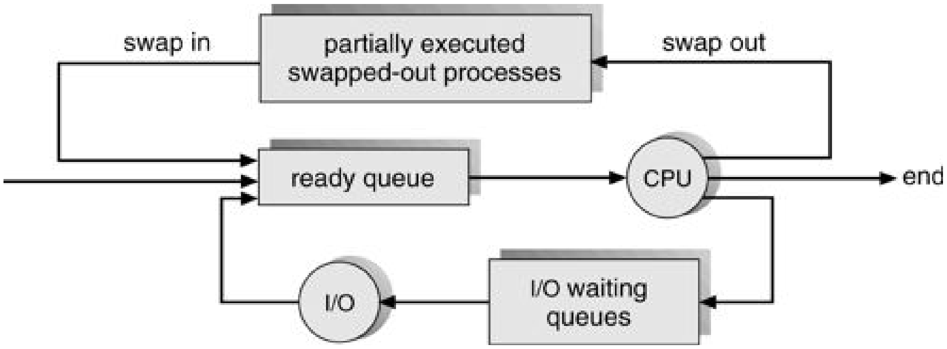
The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 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 the performance declines to unacceptable levels on a multiuser system, some users will simply quit.

Figure 3.6 Addition of medium-term scheduling to the queueing diagram.



Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is diagrammed in Figure 3.6. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes 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.

3.2.3 Context Switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state of the new process. This task is known as a **context switch**. The **context** of a process is represented in the PCB of the process; it includes the value of the CPU registers, the process state (see Figure 3.1), and memory management information. 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. Its 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). Typical speeds are less than 10 milliseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun ultra SPARC) 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 more work must be done during a context switch.

Advanced memory-management techniques may require extra data to 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.

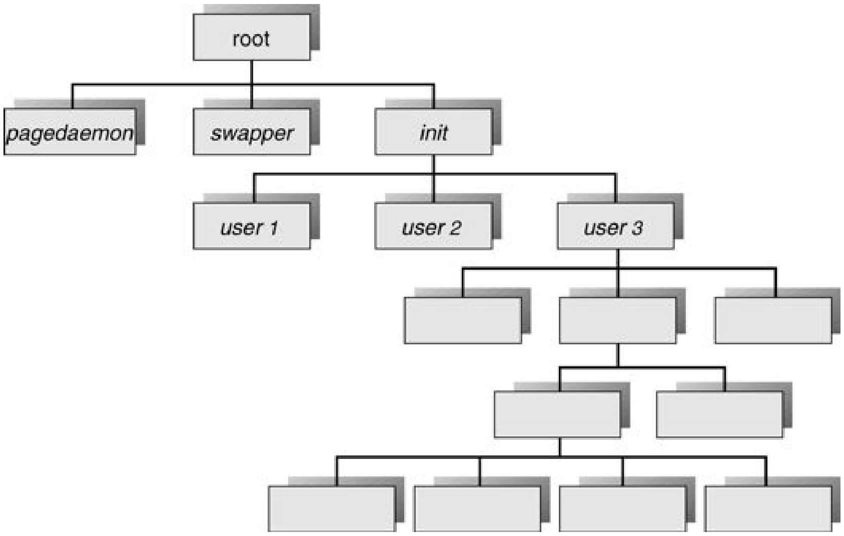
3.3 Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these operating systems must provide a mechanism for process creation and termination.

3.3.1 Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. 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 (Figure 3.7). In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a sub process, that sub 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 sub processes.

Figure 3.7 A tree of processes on a typical UNIX system.



In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process. For example, consider a process whose function is to display the contents of a file—say, *F1*—on the screen of a terminal. When it is created, it will get, as an input from its parent process, the name of the file *F1*, and it will use that file name, open the file, and write the contents out. It may also get the name of the output device. Some operating systems pass resources to child processes. On such a system, the new process may get two open files, *F1* and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities exist in terms of execution:

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 possibilities in terms of the address space of 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.

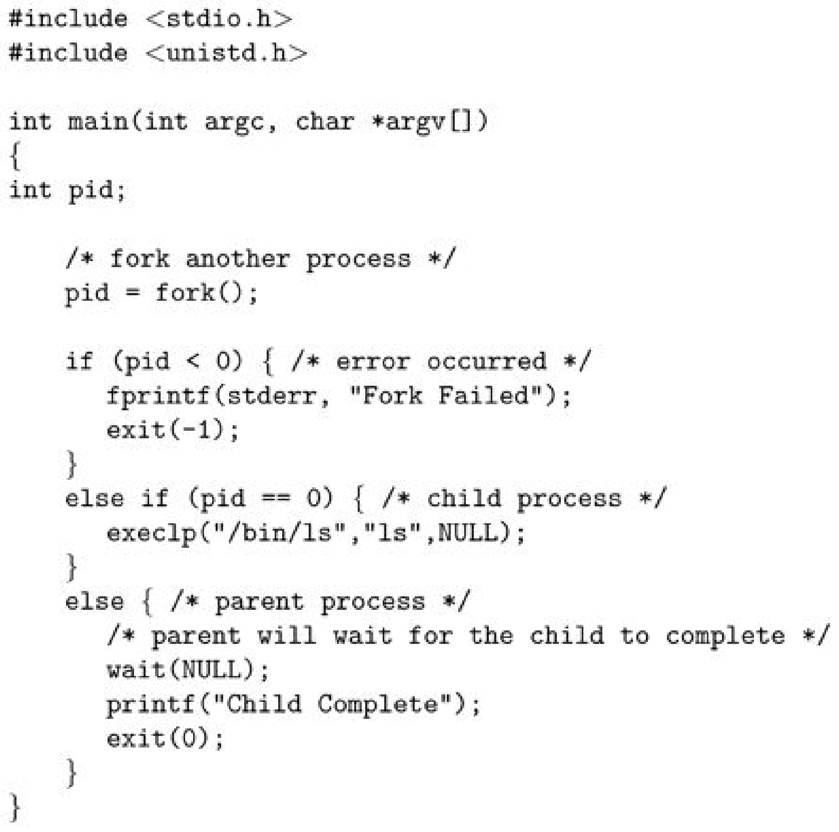
To illustrate these differences, let us consider the UNIX operating system. In UNIX, each process is identified by its **process identifier**, which is a unique integer. A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. 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.

Typically, the exec() system call is used after a fork() system call by one of the two processes 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.

The C program shown in Figure 3.8 illustrates the UNIX system calls previously described. We now have two different processes running a copy of the same program. The value of *pid* for the child process is zero; that for the parent is an integer value greater than zero. The child process 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, the parent process resumes from the call to wait(), where it completes using the exit() system call.

The DEC VMS operating system, in contrast, creates a new process, loads a specified program into that process, and starts it running. The Microsoft Windows NT operating system supports both models: The parent's address space may be duplicated, or the parent may specify the name of a program for the operating system to load into the address space of the new process.

Figure 3.8 C program forking a separate process.



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

int main(int argc, char *argv[])
{
int pid;

   /* fork another process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      exit(-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");
      exit(0);
   }
}
```

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, abort()). Usually, such a system call can be invoked by only 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. 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.

Many systems, including VMS, 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, we consider that, in UNIX, we can terminate a process by using the exit() system call; its parent process may wait for the termination of a child process by using the wait() system call. The wait() system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated. If the parent terminates, however, all its children have assigned as their new parent the *init* process. Thus, the children still have a parent to collect their status and execution statistics.

3.4 Cooperating Processes

The concurrent processes executing 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 (temporary or persistent) 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 these types of resources.

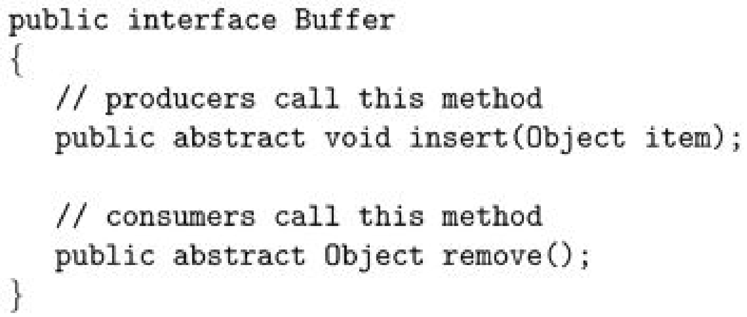
- *Computation speedup:* If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

- *Modularity:* We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

- *Convenience:* Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

To illustrate the concept of cooperating processes, let us 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, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader. The producer-consumer problem 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 file server produces (that is, provides) a file which is consumed (that is, read) by the client requesting the file. 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. 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. In this situation, the consumer must wait until an item is produced.

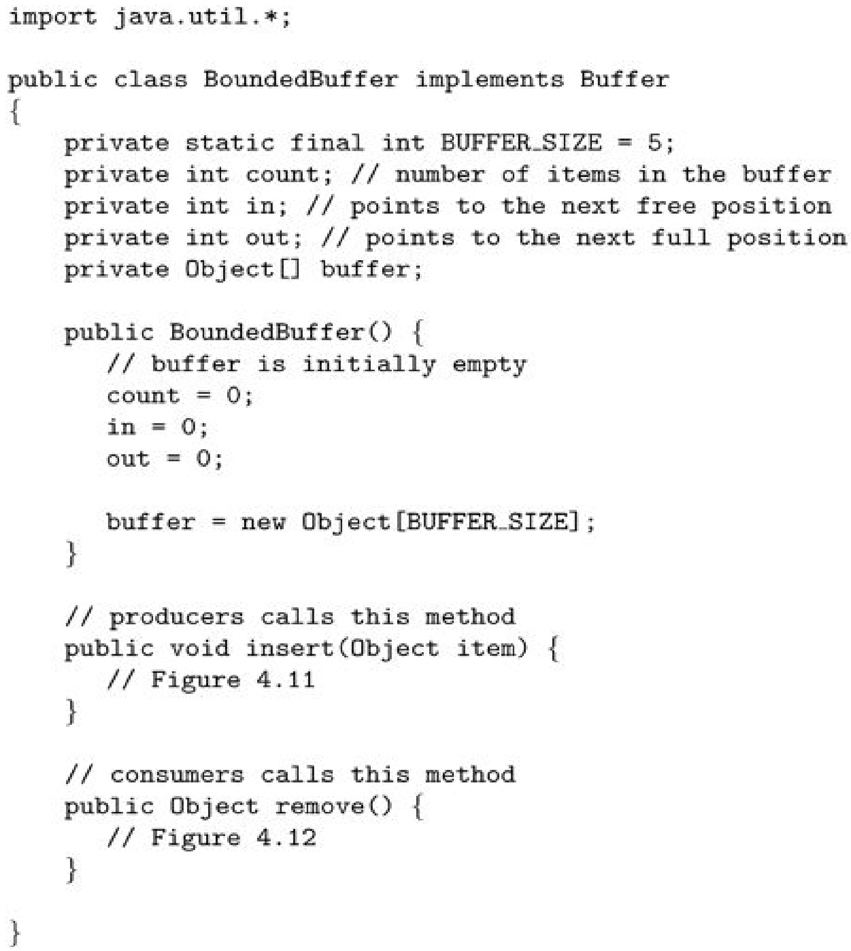
Figure 3.9 Interface for buffer implementations.



```
public interface Buffer
{
   // producers call this method
   public abstract void insert(Object item);

   // consumers call this method
   public abstract Object remove();
}
```

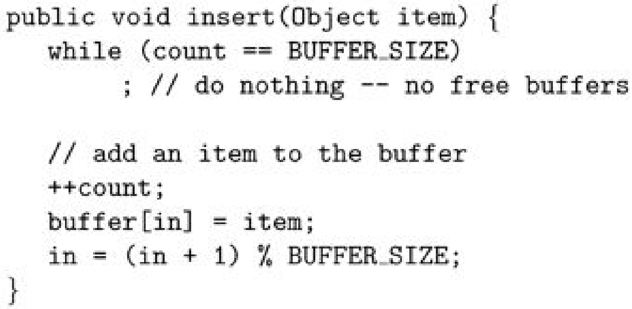
Solutions to the producer-consumer problem may implement the Buffer interface shown in Figure 3.9. The producer process invokes the insert() method when it wishes to enter an item in the buffer, and the consumer calls the remove() method when it wants to consume an item from the buffer. The nature of the buffer—unbounded or bounded—provides a way to further describe the producer-consumer problem. The **unbounded-buffer** producer-consumer problem 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** producer-consumer problem assumes that the buffer size is fixed. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Figure 3.10 Shared-memory solution to the producer-consumer problem.



```
import java.util.*;

public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private int count; // number of items in the buffer
    private int in; // points to the next free position
    private int out; // points to the next full position
    private Object[] buffer;

    public BoundedBuffer() {
       // buffer is initially empty
       count = 0;
       in = 0;
       out = 0;

       buffer = new Object[BUFFER_SIZE];
    }

    // producers calls this method
    public void insert(Object item) {
       // Figure 4.11
    }

    // consumers calls this method
    public Object remove() {
       // Figure 4.12
    }

}
```

Figure 3.11 The insert() method.

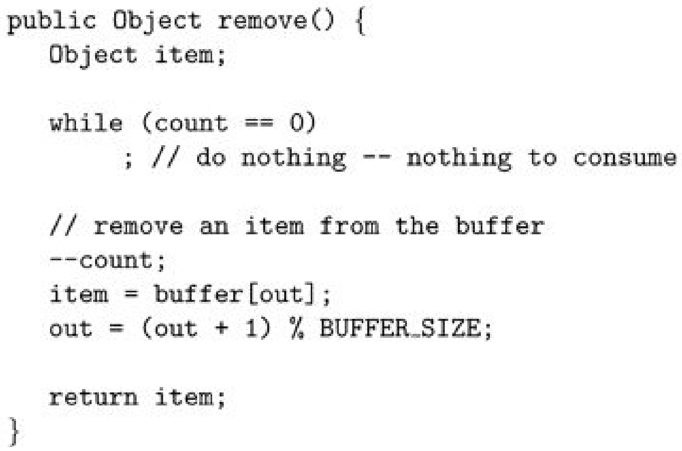


```
public void insert(Object item) {
   while (count == BUFFER_SIZE)
        ; // do nothing -- no free buffers

   // add an item to the buffer
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}
```

The buffer may be either provided by the operating system through the use of an inter process-communication (IPC) facility or explicitly coded by the application programmer with the use of shared memory. Although Java does not provide support for shared memory, we can design a solution to the bounded-buffer problem in Java that emulates shared memory by allowing the producer and consumer processes to share an instance of the Bounded Buffer class (Figure 3.10), which implements the Buffer interface. Such sharing involves passing a reference to an instance of the Bounded Buffer class to the producer and consumer processes.

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. count is the number of items currently in the buffer. The buffer is empty when count == 0 and is full when count == BUFFER_SIZE. Note that both the producer and the consumer will block in the while loop if the buffer is not usable to them.

Figure 3.12 The remove() method.



```
public Object remove() {
   Object item;

   while (count == 0)
        ; // do nothing -- nothing to consume

   // remove an item from the buffer
   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   return item;
}
```

3.5 Inter process Communication

In Section 3.4, we showed how cooperating processes can communicate in a shared memory environment. The scheme requires that these processes share a common buffer pool and that the code for implementing the buffer 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 an **interprocess communication (IPC)** facility.

IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. IPC is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected with a network. An example is a chat program used on the World Wide Web. IPC is best provided by a message-passing system, and message systems can be defined in many different ways. Here, we look at different design issues and present a Java solution to the producer-consumer problem that uses message passing.

3.5.1 Message-Passing System

The function of a message system is to allow processes to communicate with one another without the need to resort to shared data. An IPC facility provides at least the two operations send(message) and receive(message). Messages sent by a process can be of either fixed or variable 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 through 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, 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

3.5.2 Naming

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

3.5.2.1 Direct 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 and the receiver processes have to name the other to communicate. A variant of this scheme employs *asymmetry* in addressing. 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 those involving a level of indirection, as described next.

3.5.2.2 Indirect Communication

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. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox, however.

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, there may be a number of different links, 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 *P*2 and *P*3 each execute a receive() from *A*. Which process will receive the message sent by *P*1? The answer depends on the scheme that we choose:

- Allow a link to be associated with at most two processes.

- 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 also may define an algorithm for selecting which process will receive the message (that is, round robin). 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 (who can only receive messages through this mailbox) and the user of the mailbox (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who 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.

3.5.3 Synchronization

Communication between processes takes place by calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **non-blocking**—also known as **synchronous** and **asynchronous**.

- **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.

- **Non-blocking send:** The sending process sends the message and resumes operation.

- **Blocking receive:** The receiver blocks until a message is available.

- **Non-blocking receive:** The receiver retrieves either a valid message or a null.

Different combinations of send() and receive() are possible. When both the send() and receive() are blocking, we have a **rendezvous** between the sender

and the receiver. Note that the concepts of synchronous and asynchronous occur frequently in operating-system I/O algorithms, as will be seen throughout this text.

3.5.4 Buffering

Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, there are three ways to implement such a queue:

- **Zero capacity:** The queue has maximum length 0; 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 latter 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 has a finite capacity, however. If the link is full, the sender must block until space is available in the queue.

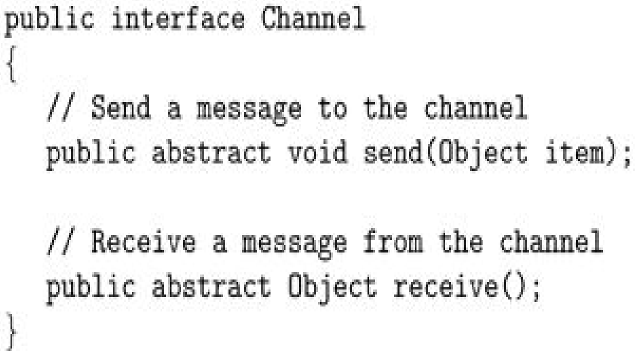
- **Unbounded capacity:** The queue has potentially infinite length; thus, any number of messages can wait in it. The sender never blocks. The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as automatic buffering.

3.5.5 Producer-Consumer Example

We can now present a solution to the producer-consumer problem that uses message passing. Our solution will implement the Channel interface shown in Figure 3.13. The producer and consumer will communicate indirectly using the shared mailbox illustrated in Figure 3.14.

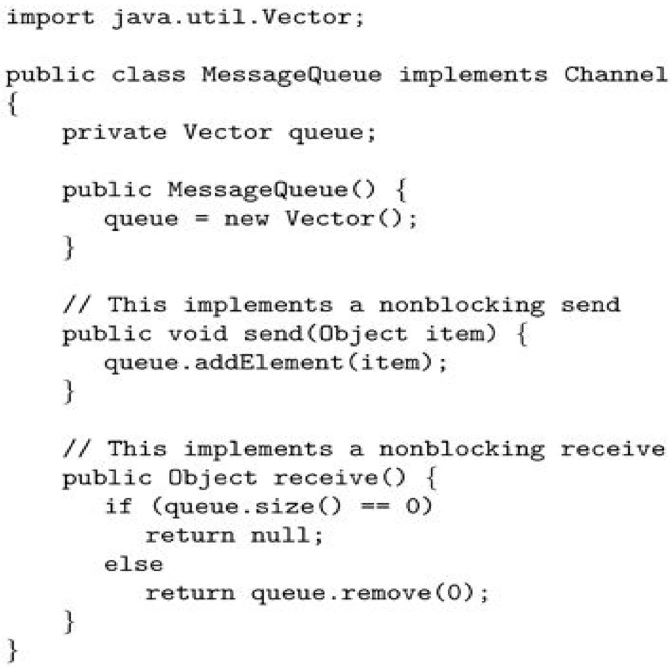
The buffer is implemented using the java.util. Vector class, meaning that it will be a buffer of unbounded capacity. Also note that both the send() and receive() methods are non-blocking. When the producer generates an item, it places that item in the mailbox via the send() method. The code for the producer is shown in Figure 3.15.

Figure 3.13 Interface for message passing.



```
public interface Channel
{
   // Send a message to the channel
   public abstract void send(Object item);

   // Receive a message from the channel
   public abstract Object receive();
}
```

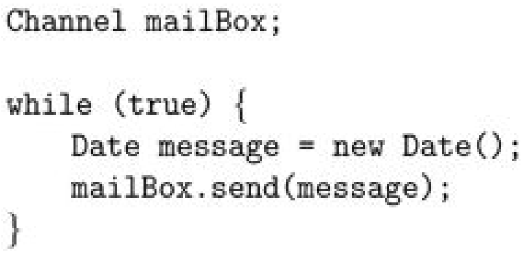
Figure 3.14 Mailbox for message passing.



```
import java.util.Vector;

public class MessageQueue implements Channel
{
    private Vector queue;

    public MessageQueue() {
       queue = new Vector();
    }

    // This implements a nonblocking send
    public void send(Object item) {
       queue.addElement(item);
    }

    // This implements a nonblocking receive
    public Object receive() {
       if (queue.size() == 0)
          return null;
       else
          return queue.remove(0);
    }
}
```

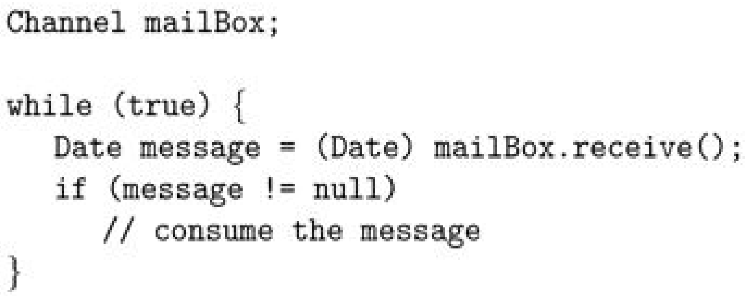
The consumer obtains an item from the mailbox using the receive() method. Because receive() is non-blocking, the consumer must evaluate the value of the Object returned from receive(). If it is null, the mailbox is empty. The code for the consumer is shown in Figure 3.16.

Figure 3.15 The producer process.



```
Channel mailBox;

while (true) {
    Date message = new Date();
    mailBox.send(message);
}
```

Figure 3.16 The consumer process.



```
Channel mailBox;

while (true) {
   Date message = (Date) mailBox.receive();
   if (message != null)
      // consume the message
}
```

3.5.6 An Example: Mach

As an example of a message-based operating system, we next consider the Mach operating system, developed at Carnegie Mellon University.. The Mach kernel supports the creation and destruction of multiple tasks, which are similar to processes but have multiple threads of control. Most communication in Mach—including most of the system calls and all intertask information—is carried out by *messages*.

Messages are sent to and received from mailboxes, called *ports* in Mach. Even system calls are made by messages. When a task is created, two special mailboxes—the Kernel mailbox and the Notify mailbox—are also created. The Kernel mailbox is used by the kernel to communicate with the task. The kernel sends notification of event occurrences to the Notify port. Only three system calls

are needed for message transfer. The msg_send call sends a message to a mailbox. A message is received via msg_receive. Remote procedure calls (RPCs) are executed via msg_rpc, which sends a message and waits for exactly one return message from the sender. In this way, RPC models a typical subroutine procedure call but can work between systems.

The port_allocate system call creates a new mailbox and allocates space for its queue of messages. The maximum size of the message queue defaults to eight messages. The task that creates the mailbox is that mailbox's owner. The owner also is given receive access to the mailbox. Only one task at a time can either own or receive from a mailbox, but these rights can be sent to other tasks if desired. The mailbox has an initially empty queue of messages. As messages are sent to the mailbox, the messages are copied into the mailbox. All messages have the same priority. Mach guarantees that multiple messages from the same sender are queued in first-in, first-out (FIFO) order but does not guarantee an absolute ordering. For instance, messages from two senders may be queued in any order.

The messages themselves consist of a fixed-length header, followed by a variablelength data portion. The header includes the length of the message and two mailbox names. When a message is sent, one mailbox name is the mailbox to which the message is being sent. Commonly, the sending thread expects a reply; the mailbox name of the sender is passed on to the receiving task, which may use it as a "return address" to send messages back.

The variable part of a message is a list of typed data items. Each entry in the list has a type, size, and value. The type of the objects specified in the message is important, since operating-system-defined objects—such as the ownership or receive access rights, task states, and memory segments—may be sent in messages. The send and receive operations themselves are flexible. For instance, when a message is sent to a mailbox, the mailbox may be full. If the mailbox is not full, the message is copied to the mailbox and the sending thread continues. If the mailbox is full, the sending thread has four options:

1. Wait indefinitely until there is room in the mailbox.

2. Wait at most *n* milliseconds.

3. Do not wait at all, but rather return immediately.

4. Temporarily cache a message. One message can be given to the operating system to keep, even though the mailbox to which it is being sent is full. When the message can be put in the mailbox, a message is sent back to the sender; only

one such message to a full mailbox can be pending at any time for a given sending thread.

The final option is meant for server tasks, such as a line-printer driver. After finishing a request, these tasks may need to send a one-time reply to the task that had requested service; but they must also continue with other service requests, even if the reply mailbox for a client is full.

The receive operation must specify from which mailbox or mailbox set to receive a message. A mailbox set is a collection of mailboxes, as declared by the task, which can be grouped together and treated as one mailbox for the purposes of the task. Threads in a task can receive from only a mailbox or mailbox set for which that task has receive access. A port_status system call returns the number of messages in a given mailbox. The receive operation attempts to receive from (1) any mailbox in a mailbox set or (2) a specific (named) mailbox. If no message is waiting to be received, the receiving thread may either wait at most *n* milliseconds, or not wait at all.

The Mach system was especially designed for distributed systems, but Mach is also suitable for single-processor systems. The major problem with message systems has generally been poor performance caused by double copying of messages; the message is copied first from the sender to the mailbox and then from the mailbox to the receiver. The Mach message system attempts to avoid double-copy operations by using virtual-memory management techniques. Essentially, Mach maps the address space containing the sender's message into the receiver's address space. The message itself is never actually copied. This message-management technique provides a large performance boost but works for only intrasystem messages.

3.5.7 An Example: Windows XP

The Windows XP operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows XP provides support for multiple operating environments, or *subsystems*, with which application programs communicate via a message-passing mechanism. The application programs can be considered to be clients of the Windows XP subsystem server.

The message-passing facility in Windows XP is called the **local procedure-call (LPC)** facility. The LPC in Windows XP communicates between two processes that are on the same machine. It is similar to the standard RPC mechanism that is widely used, but it is optimized for and specific to Windows XP. Like Mach, Windows XP uses a port object to establish and maintain a connection between two processes. Every client that calls a subsystem needs a communication channel, which is provided by a port object and is never inherited. Windows XP uses two types of ports: connection ports and communication ports. They are really the same but are given different names according to how they are used. Connection ports are named *objects* and are visible to all processes; they give applications a way to set up a communication channel. This communication works as follows:

- The client opens a handle to the subsystem's connection port object.
- The client sends a connection request.
- The server creates two private communication ports and returns the handle to one of them to the client.
- The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Windows XP uses two types of message-passing techniques over a port that the client specifies when it establishes the channel. The simplest, which is used for small messages, uses the port's message queue as intermediate storage and copies the message from one process to the other. Under this method, messages of up to 256 bytes can be sent.

If a client needs to send a larger message, it passes the message through a section object (or shared memory). The client has to decide, when it sets up the channel, whether or not it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Likewise, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about that section object. This method is more complicated than the first method, but it avoids the data copying. In both cases, a callback mechanism can be used when either the client or the server cannot respond immediately to a request. The callback mechanism allows them to perform asynchronous message handling.

3.6 Communication in Client-Server Systems

In this section, we explore three other strategies for communication in client-server systems: sockets, remote procedure calls (RPCs), and Java's remote method invocation (RMI).

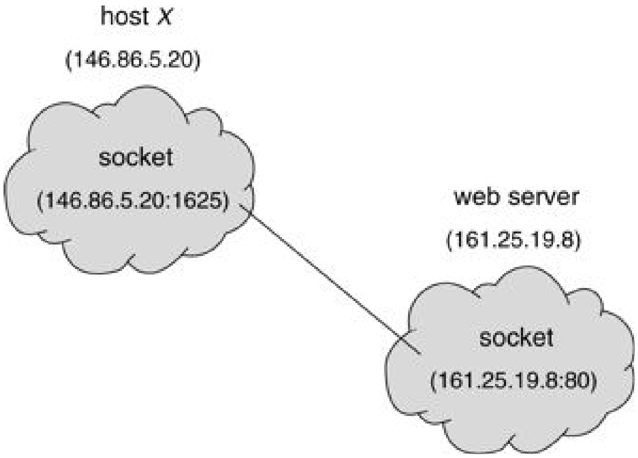
3.6.1 Sockets

A **socket** is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client-server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. Servers implementing specific services (such as telnet, ftp, and http) listen to well known ports (a telnet server listens to port 23, an ftp server listens to port 21, and a web, or http, server listens to port 80). All ports below 1024 are considered *well known*; we can use them to implement standard services.

When a client process initiates a request for a connection, it is assigned a port by the host computer. This port is some arbitrary number greater than 1024. For example, if a client on host *X* with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host *X* may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host *X* and (161.25.19.8:80) on the web server. This situation is illustrated in Figure 3.17. The packets traveling between the hosts are delivered to the appropriate process, based on the destination port number.

All connections must be unique. Therefore, if another process also on host *X* wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

Figure 3.17 Communication using sockets.

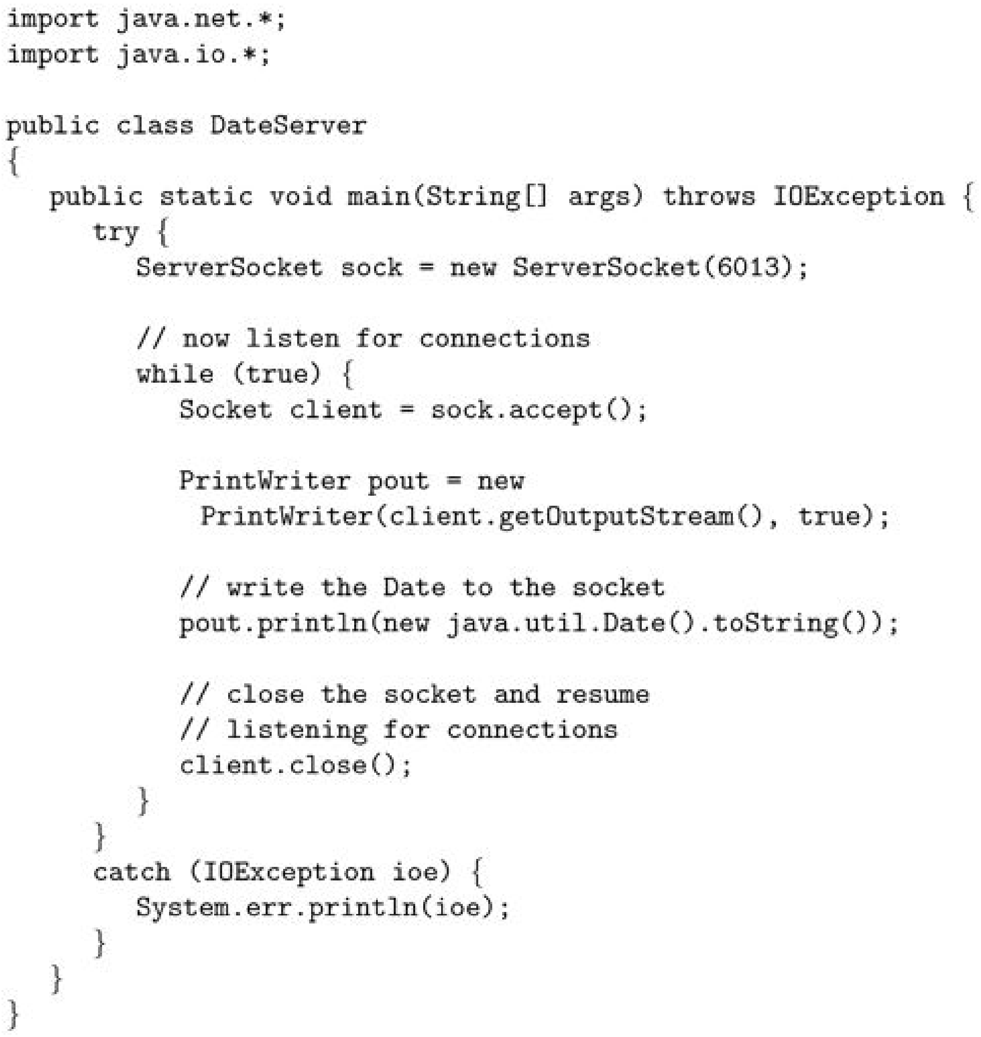


Java provides three different types of sockets. **Connection-oriented (TCP) sockets** are implemented with the Socket class. **Connectionless (UDP) sockets** use the DatagramSocket class. Finally, the MulticastSocket class is a subclass of the DatagramSocket class. A multicast socket allows data to be sent to multiple recipients.

Our example describes a date server that uses connection-oriented TCP sockets. The operation allows clients to request the current date and time from the server. The server listens to port 6013, although the port could be any arbitrary number greater than 1024. When a connection is received, the server returns the date and time to the client.

The date server is shown in Figure 3.18. The server creates a ServerSocket that specifies it will listen to port 6013. The server then begins listening to the port with the accept() method. The server blocks on the accept() method waiting for a client to request a connection. When a connection request is received, accept() returns a socket that the server can use to communicate with the client. The details illustrating how the server communicates with the socket are as follows. The server first establishes a PrintWriter object that it will use to communicate with the client. A PrintWriter object allows the server to write to the socket using the routine print() and println() methods for output. The server process sends the date to the client, calling the method println(). Once it has written the date to the socket, the server closes the socket to the client and resumes listening for more requests.

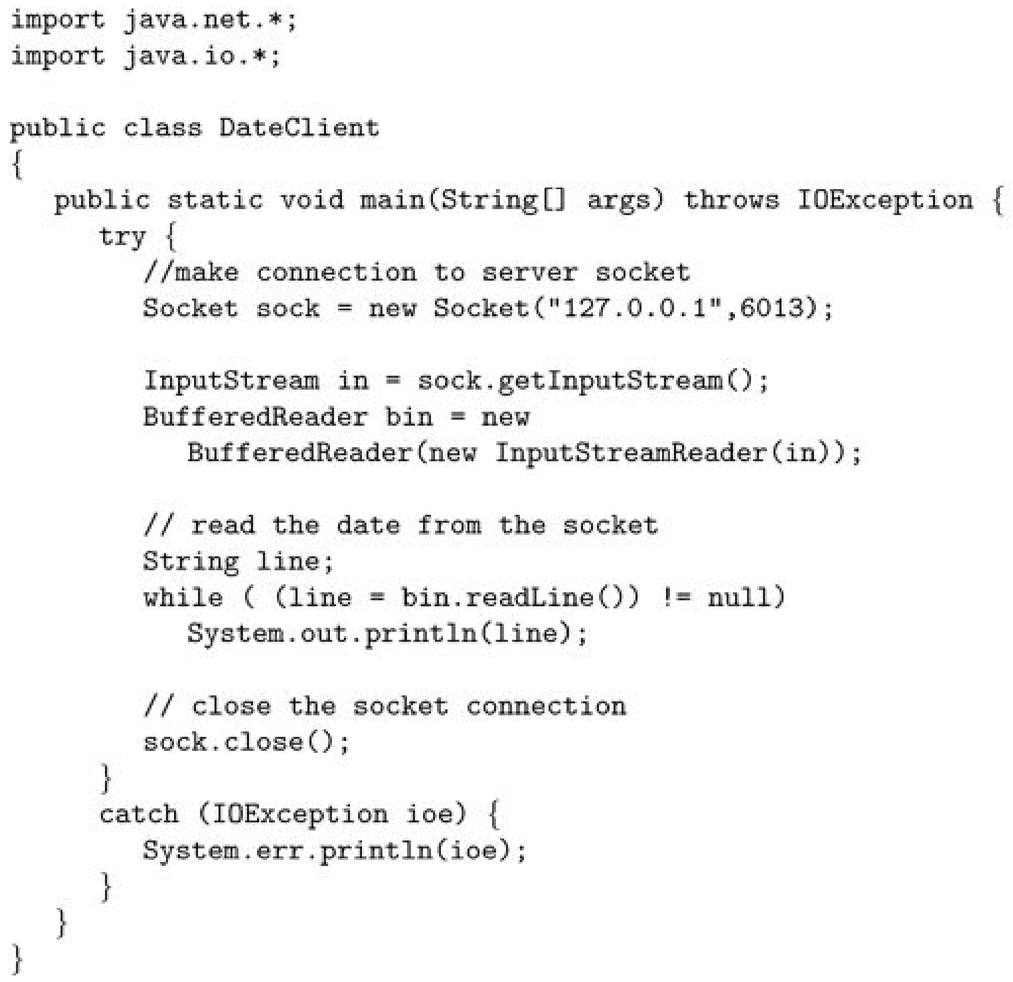
Figure 3.18 Date server.



```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) throws IOException {
      try {
         ServerSocket sock = new ServerSocket(6013);

         // now listen for connections
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            // write the Date to the socket
            pout.println(new java.util.Date().toString());

            // close the socket and resume
            // listening for connections
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

A client communicates with the server by creating a socket and connecting to the port the server is listening on. We implement such a client in the Java program shown in Figure 3.19. The client creates a Socket and requests a connection with the server at IP address 127.0.0.1 on port 6013. Once the connection is made, the client can read from the socket using normal stream I/O statements. After it has received the date from the server, the client closes the socket and exits. The IP address 127.0.0.1 is a special IP address known as the **loopback**. When a computer refers to IP address 127.0.0.1, it is referring to itself. This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol. The IP address 127.0.0.1 could be replaced with the IP address of another host running the date server. In addition to using an IP address, an actual host name, such as **www.westminstercollege.edu**, can be used as well.

Figure 3.19 Date client.



```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) throws IOException {
      try {
         //make connection to server socket
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         // read the date from the socket
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         // close the socket connection
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Communication using sockets—although common and efficient—is considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data.

3.6.2 Remote Procedure Calls

One of the most common forms of remote service is the RPC paradigm, The RPC was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. Here, however, because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service. In contrast to the IPC facility, the messages exchanged for RPC communication are well structured and are thus no longer just packets of data. Each message is addressed to an RPC daemon listening to a port on the remote system and contains an identifier of the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message.

A *port* is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports. If a remote process needs a service, it addresses a message to the proper port. For instance, if a system wished to allow other systems to be able to list the current users on it, it would have a daemon supporting such an RPC attached to a port—say, port 3027. Any remote system could obtain the needed information (that is, the list of current users) by sending an RPC message to port 3027 on the server; the data would be received in a reply message.

The semantics of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally. The RPC system hides the details that allow communication to take place by providing a stub on the client side. Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and *marshalls* the parameters. Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique.

One issue that must be dealt with concerns differences in data representation on the client and server machines. Consider the representation of 32-bit integers. Some systems use the high memory address to store the most significant byte (known as *big-endian*), while other systems store the least significant byte at the high memory address (known as *little-endian*). To resolve differences like this, many RPC systems define a machine-independent representation of data. One such representation is known as **external data representation (XDR)**. On the client side, parameter marshalling involves converting the machine-dependent data into XDR before being sent to the server. On the server side, the XDR data is unmarshalled and converted into the machine-dependent representation for the server.

Another important issue is the semantics of a call. Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network errors. One way to address this problem is for the operating system to ensure that messages are acted on *exactly* *once*, rather than *at most once*. Most local procedure calls have this functionality, but it is more difficult to implement.

First, we consider "at most once". This semantic can be assured by attaching to each message a timestamp. The server must keep a history of all the timestamps of messages it has already processed or a history large enough to ensure that repeated messages are detected. Incoming messages that have a timestamp already in the history are ignored. The client can then send a message one or more times and be assured that it only executed once.

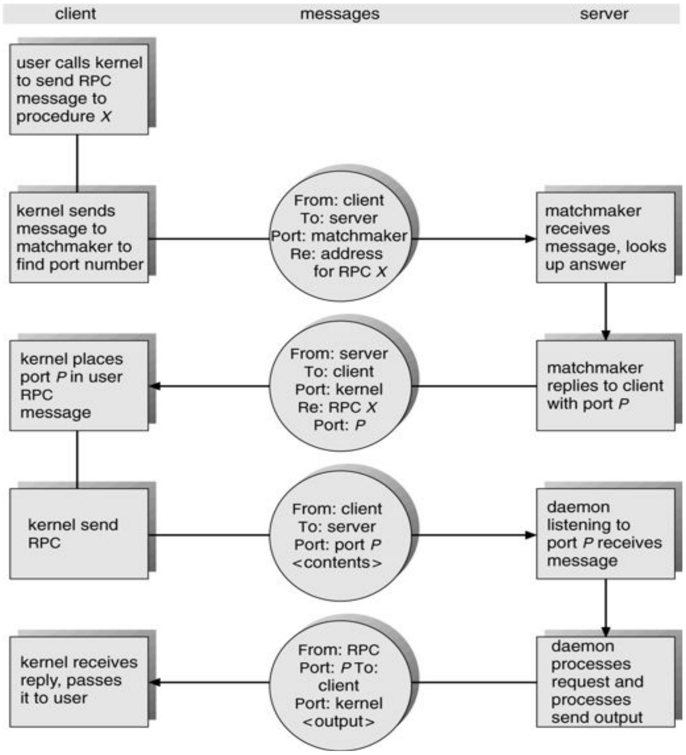
For "exactly once," we need to remove the risk that the server never received the request. To accomplish this, the server must implement the "at most once" protocol described above, as well as acknowledge to the client that the RPC call was received and executed. These "ACK" messages are common throughout networking. The client must resend each RPC call periodically until it receives the "ACK" for that call.

Another important issue concerns the communication between a server and a client. With standard procedure calls, some form of binding takes place during link, load, or execution time, such that a procedure call's name is replaced by the memory address of the procedure call. The RPC scheme requires a similar binding of the client and the server port, but how does a client know the port numbers on the server? Neither system has full information about the other because they do not share memory.

Two approaches are common. First, the binding information may be predetermined, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service. Second, binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a **matchmaker**) daemon on a fixed RPC port. A client then sends a message, containing the name of the RPC, to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls may be sent to that port until the process terminates (or the server crashes).

This method requires the extra overhead of the initial request but is more flexible than the first approach. Figure 3.20 shows a sample interaction. The RPC scheme is useful in implementing a distributed file system. Such a system can be implemented as a set of RPC daemons and clients. The messages are addressed to the DFS port on a server on which a file operation is to take place. The message contains the disk operation to be performed. Disk operations might be read, write, rename, delete, or status, corresponding to the usual file-related system calls. The return message contains any data resulting from that call, which is executed by the DFS daemon on behalf of the client. For instance, a message might contain a request to transfer a whole file to a client or be limited to simple block requests. In the latter case, several such requests might be needed if a whole file is to be transferred.

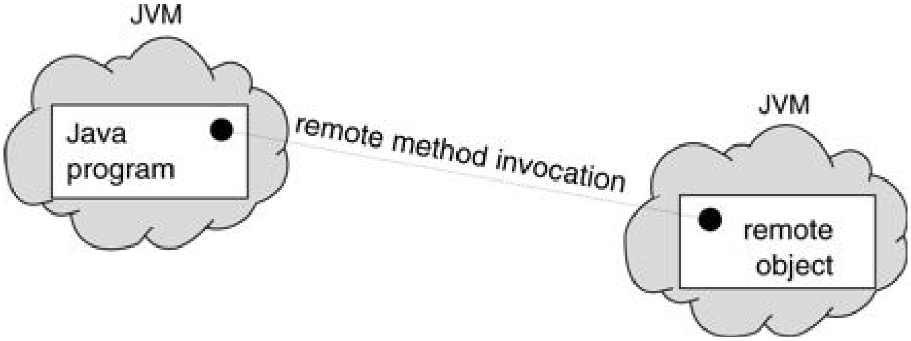
Figure 3.20 Execution of a remote procedure call (RPC).


client
messages
server
user calls kernel to send RPC message to procedure X
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC X
matchmaker receives message, looks up answer
kernel places port P in user RPC message
From: server To: client Port: kernel Re: RPC X Port: P
matchmaker replies to client with port P
kernel send RPC
From: client To: server Port: port P <contents>
daemon listening to port P receives message
kernel receives reply, passes it to user
From: RPC Port: P To: client Port: kernel <output>
daemon processes request and processes send output

3.6.3 Remote Method Invocation

Remote method invocation (RMI) is a Java feature similar to RPCs. RMI allows a thread to invoke a method on a remote object. Objects are considered remote if they reside in a different Java virtual machine (JVM). Therefore, the remote object may be in a different JVM on the same computer or on a remote host connected by a network. This situation is illustrated in Figure 3.21.

Figure 3.21 Remote method invocation.



RMI and RPCs differ in two fundamental ways. First, RPCs support procedural programming whereby only remote procedures or functions may be called. In contrast, RMI is object-based: It supports invocation of methods on remote objects. Second, the parameters to remote procedures are ordinary data structures in RPC; with RMI, it is possible to pass objects as parameters to remote methods. By allowing a Java program to invoke methods on remote objects, RMI makes it possible for users to develop Java applications that are distributed across a network.

To make remote methods transparent to both the client and the server, RMI implements the remote object using stubs and skeletons. A **stub** is a proxy for the remote object; it resides with the client. When a client invokes a remote method, the stub for the remote object is called. This client-side stub is responsible for creating a **parcel** consisting of the name of the method to be invoked on the server and the marshalled parameters for the method. The stub then sends this parcel to the server, where the skeleton for the remote object receives it. The **skeleton** is responsible for unmarshalling the parameters and invoking the desired method on the server. The skeleton then marshals the return value (or exception, if any) into a parcel and returns this parcel to the client. The stub unmarshals the return value and passes it to the client.

Let us look more closely at how this process works. Assume that a client wishes to invoke a method on a remote object server with a signature some Method(Object, Object) that returns a boolean value. The client executes the statement boolean val = server.someMethod(A, B); The call to someMethod() with the parameters *A* and *B* invokes the stub for the remote object. The stub marshals into a parcel the parameters *A* and *B* and the name of the method that is to be invoked on the server, then sends this parcel to the server.

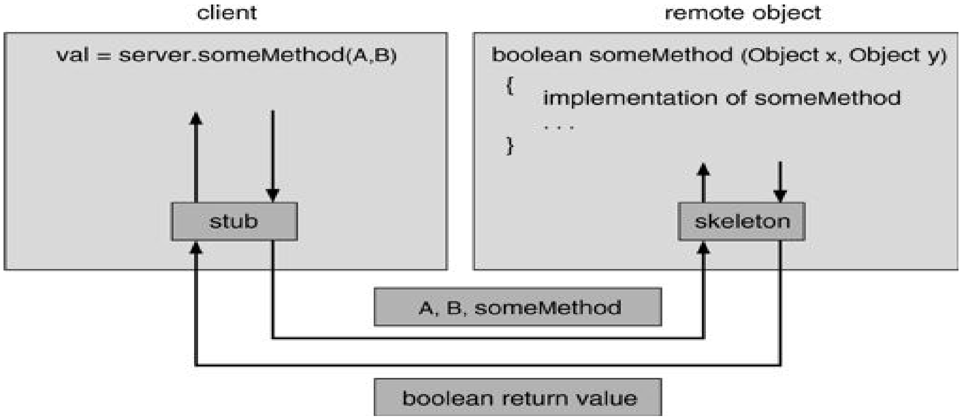
The skeleton on the server unmarshals the parameters and invokes the method someMethod(). The actual implementation of someMethod() resides on the server. Once the method is completed, the skeleton marshals the boolean value returned from someMethod() and sends this value back to the client. The stub unmarshals this return value and passes it to the client. The process is shown in Figure 3.22.

Fortunately, the level of abstraction that RMI provides makes the stubs and skeletons transparent, allowing Java developers to write programs that invoke distributed methods just as they would invoke local methods. It is crucial, however, to understand a few rules about the behavior of parameter passing.

- If the marshalled parameters are **local** (or **nonremote**) objects, they are passed by copy using a technique known as **object serialization**. However, if the parameters are also remote objects, they are passed by reference. In our example, if *A* is a local object and *B* a remote object, *A* is serialized and passed by copy, and *B* is passed by reference. This would in turn allow the server to invoke methods on *B* remotely.

- If local objects are to be passed as parameters to remote objects, they must implement the interface *java.io.Serializable*. Many objects in the core Java API implement *Serializable*, allowing them to be used with RMI. Object serialization allows the state of an object to be written to a byte stream.

Figure 3.22 Marshalling parameters.


client
remote object
val = server.someMethod(A,B)
boolean someMethod (Object x, Object y)
{
implementation of someMethod
. . .
}
stub
skeleton
A, B, someMethod
boolean return value

3.6.3.1 Remote Objects

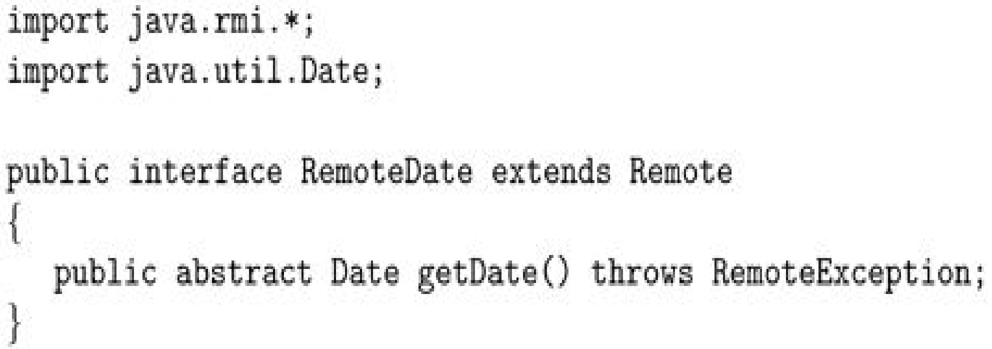
Building a distributed application initially requires defining the necessary remote objects. We define remote objects by first declaring an interface that specifies the methods that can be invoked remotely. In this example of a date server, the remote method will be named getDate() and will return a Date containing the current date. To provide for remote objects, this interface must also extend the cjava.rmi.Remote interface, which identifies objects implementing this interface as being remote. Further, each method declared in the interface must throw the exception java.rmi.RemoteException. For remote objects, we provide the RemoteDate interface shown in Figure 3.23.

The class that defines the remote object must implement the RemoteDate interface (Figure 3.24). In addition to defining the getDate() method, the class must also extend java.rmi.server.UnicastRemoteObject. Extending UnicastRemoteObject allows the creation of a single remote object that listens for network requests using RMI's default scheme of sockets for network communication. This class also includes a main() method. The main() method creates an instance of the object and registers with the RMI registry running on the server with the rebind() method. In this case, the object instance registers itself with the name "DateServer". Also note thatwemust create a default constructor for the RemoteDateImpl class, and it must throw a RemoteException if a communication or network failure prevents RMI from exporting the remote object.

3.6.3.2 Access to the Remote Object

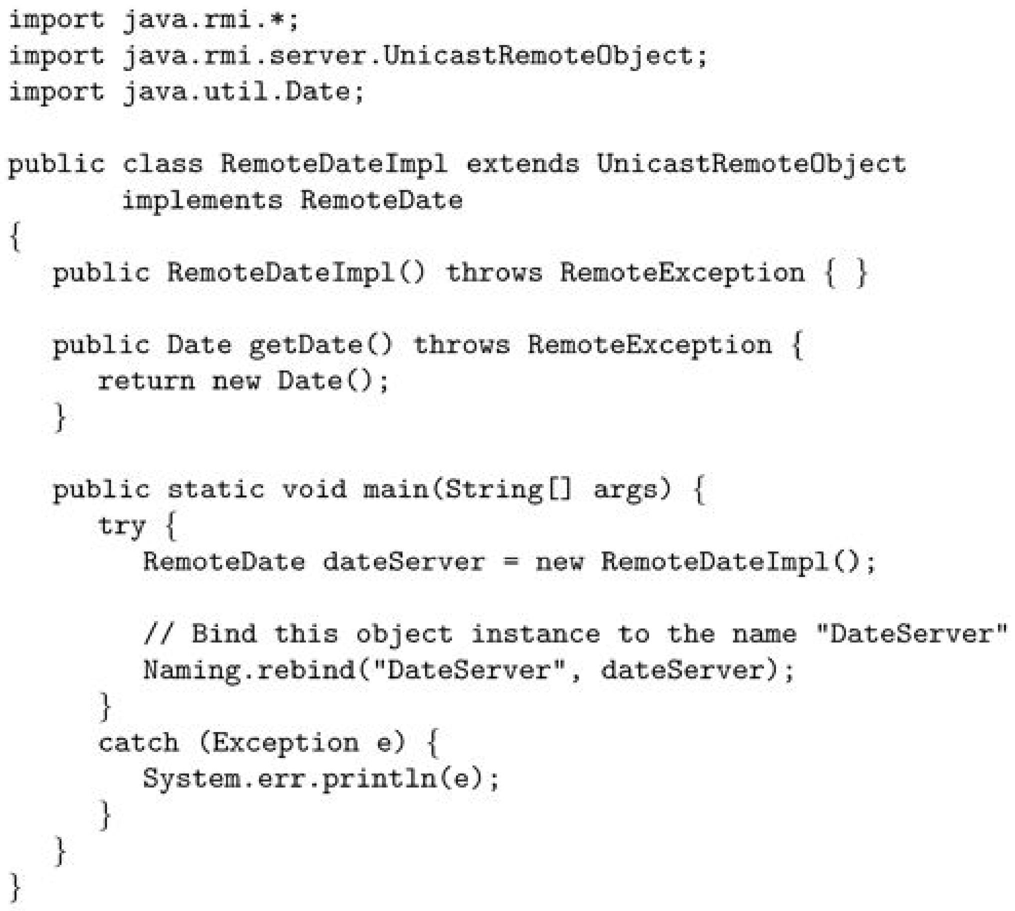
Once an object is registered on the server, a client (as shown in Figure 3.25) can get a proxy reference to this remote object from the RMI registry running on the server by using the static method lookup() in the Naming class. RMI provides a URL-based lookup scheme using the form rmi://host/objectName, where host is the IP name (or address) of the server on which the remote object objectName resides. objectName is the name of the remote object specified by the server in the rebind() method (in this case, DateServer). Once the client has the proxy reference to the remote object, it invokes the remote method getDate(), which returns the current date. Because remote methods—as well as the Naming.lookup() method—can throw exceptions, they must be placed in try-catch blocks.

Figure 3.23 The RemoteDate interface.



```
import java.rmi.*;
import java.util.Date;

public interface RemoteDate extends Remote
{
   public abstract Date getDate() throws RemoteException;
}
```

Figure 3.24 Implementation of the RemoteDate interface.



```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class RemoteDateImpl extends UnicastRemoteObject
      implements RemoteDate
{
   public RemoteDateImpl() throws RemoteException { }

   public Date getDate() throws RemoteException {
      return new Date();
   }

   public static void main(String[] args) {
      try {
         RemoteDate dateServer = new RemoteDateImpl();

         // Bind this object instance to the name "DateServer"
         Naming.rebind("DateServer", dateServer);
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

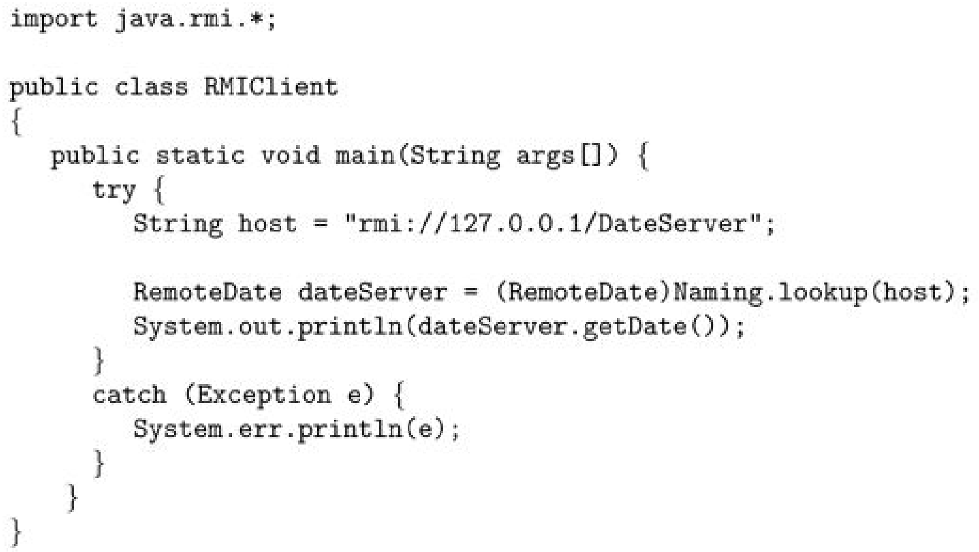
3.6.3.3 Running of the Programs

We now demonstrate the steps necessary to run the example programs. For simplicity, we are assuming that all programs are running on the local host—that is, IP address 127.0.0.1. However, communication is still considered remote, because the client and server programs are each running in their own JVM.

1. *Compile all source files.*

2.*Generate the stub and skeleton.* The user generates the stub and skeleton, using the tool rmic, by entering rmic RemoteDateImpl on the command line; this creates the files RemoteDateImpl_Skel.class and RemoteDateImpl_Stub.class. (If you are running this example on two different computers, make sure that all the class files—including the stub classes—are available on each computer. It is possible to load classes dynamically using RMI, a topic beyond the scope of this text but covered in texts mentioned in the Bibliographical Notes.)

Figure 3.25 The RMI client.



```
import java.rmi.*;

public class RMIClient
{
   public static void main(String args[]) {
      try {
         String host = "rmi://127.0.0.1/DateServer";

         RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
         System.out.println(dateServer.getDate());
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

3. *Start the registry and create the remote object.* To start the registry on UNIX platforms, the user can type

```
rmiregistry &
```

For Windows, the user can type

```
start rmiregistry
```

This command starts the registry with which the remote object will register.

Next, create an instance of the remote object with

```
java RemoteDateImpl
```

This remote object will register using the name DateServer.

4. *Reference the remote object.* The statement

```
java RMIClient
```

is entered on the command line to start the client. This program will get a proxy reference to the remote object named DateServer and invokes the remote method getDate().

3.6.3.4 RMI versus Sockets

Contrast the socket-based client program shown in Figure 3.19 with the client using RMI shown in Figure 3.25. The socket-based client must manage the socket connection, including opening and closing the socket and establishing an InputStream to read from the socket. The design of the client using RMI is much simpler. All it must do is get a proxy for the remote object, which allows it to invoke the remote method getDate() as it would invoke an ordinary local method. This illustrates the appeal of techniques such as RPCs and RMI: They provide developers of distributed systems a communication mechanism allowing them to design distributed programs without incurring the overhead of socket management.

3.7 Summary

A process is a program in execution. As a process executes, it changes state. The state of a process is defined by that process's current activity. Each process may be in one of the following states: new, ready, running, waiting, or terminated. Each process is represented in the operating system by its own process-control block (PCB). A process, when it is not executing, is placed in some waiting queue. There are two major classes of queues in an operating system: I/O request queues and the ready queue. The ready queue contains all the processes that are ready to execute and are waiting for the CPU. Each process is represented by a PCB, and the PCBs can be linked together to form a ready queue. Long-term (job) scheduling is the selection of processes to be allowed to contend for the CPU. Normally, long-term scheduling is heavily influenced by resource-allocation considerations, especially memory management. Short-term (CPU) scheduling is the selection of one process from the ready queue. The processes in most systems can execute concurrently. There are several reasons for allowing concurrent execution: information sharing, computation speedup, modularity, and convenience. Concurrent execution requires a mechanism for process creation and deletion.

The processes executing in the operating system may be either independent processes or cooperating processes. Cooperating processes must have the means to communicate with each other. Principally, communication is achieved through two complementary schemes: shared memory and message systems. The shared-memory method requires communicating processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared-memory system, the responsibility for providing communication rests with the application programmers; the operating system needs to provide only the shared memory. The message-system method allows the processes to exchange messages.

The responsibility for providing communication may rest with the operating system itself. These two schemes are not mutually exclusive and can be used simultaneously within a single operating system. Communication in client-server systems may use (1) sockets, (2) remote procedure calls (RPCs), or (3) Java's remote method invocation (RMI). A socket is defined as an endpoint for communication. A connection between a pair of applications consists of a pair of sockets, one at each end of the communication channel. RPCs are another form of distributed communication. An RPC occurs when a process (or thread) calls a procedure on a remote application. RMI is the Java version of an RPC. RMI allows a thread to invoke a method on a remote object just as it would invoke a method on a local object. The primary distinction between RPCs and RMI is that in RPC data are passed to a remote procedure in the form of an ordinary data structure, whereas RMI allows objects to be passed in remote method calls.

Exercises

3.1. Palm OS provides no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

3.2. Describe the differences among short-term, medium-term, and long-term scheduling.

3.3. The Sun UltraSPARC processor has multiple register sets. Describe the actions of a context switch if the new context is already loaded into one of the register sets. What else must happen if the new context is in memory rather than in a register set and all the register sets are in use?

3.4. Describe the actions taken by a kernel to context-switch between processes.

3.5. What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer levels.

a. Synchronous and asynchronous communication

b. Automatic and explicit buffering

c. Send by copy and send by reference

d. Fixed-sized and variable-sized messages

3.6. Consider the RPC mechanism. Describe the undesirable circumstances that could arise from not enforcing either the "at most once" or "exactly once" semantic. Describe possible uses for a mechanism that has neither of these guarantees.

3.7. Again considering the RPC mechanism, consider the "exactly once" semantic. Does the algorithm for implementing this semantic execute correctly even if the "ACK" message back to the client is lost because of a network problem? Describe the sequence of messages and whether "exactly once" is still preserved.

3.8. Modify the date server shown in Figure 3.18 so that it delivers random one-line fortunes rather than the current date.

3.9. Modify the RMI date server shown in Figure 3.24 so that it delivers random one-line fortunes rather than the current date.

4. THREADS

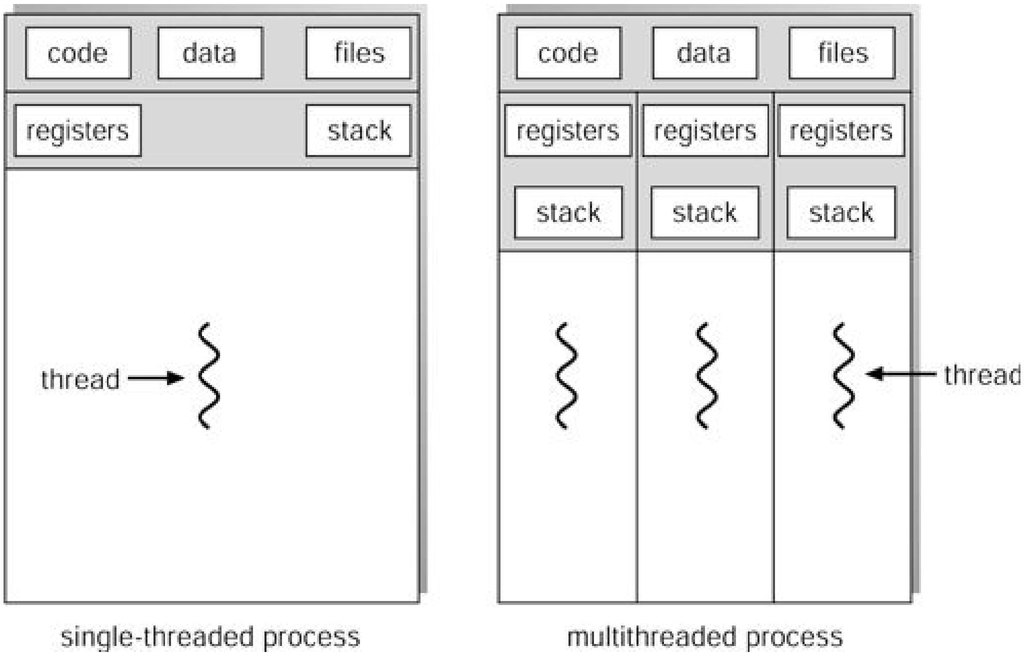
4.1 Overview

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or **heavyweight**) process has a single thread of control. If the process has multiple threads of control, it can do more than one task at a time. Figure 4.1 illustrates the difference between a traditional single-threaded process and a multithreaded process.

4.1.1 Motivation

Many software packages that run on modern desktop PCs are **multithreaded**. An application typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network, for example. A word processor may have a thread for displaying graphics, another thread for reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Figure 4.1 Single- and multithreaded processes.



In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands) of clients concurrently accessing it. If the web server ran as a traditional **singlethreaded** process, it would be able to service only one client at a time. The amount of time that a client might have to wait for its request to be serviced could be enormous.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time-consuming and resource intensive, as was shown in the previous chapter. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient for one process that contains multiple threads to serve the same purpose. This approach would multithread the web-server process. The server would create a separate thread that would listen for client requests; when a request was made, rather than creating another process, it would create another thread to service the request.

Threads also play a vital role in remote procedure call (RPC) systems. Recall from Chapter 3 that RPCs allow interprocess communication by providing a communication mechanism similar to ordinary function or procedure calls. Typically, RPC servers are multithreaded. When a server receives a message, it services the message using a separate thread. This allows the server to service several concurrent requests. RMI systems work similarly. Finally, many operating system kernels are now multithreaded; several threads operate in the kernel, and each thread performs a specific task, such as managing devices or interrupt handling. For example, Solaris creates a set of threads in the kernel specifically for interrupt handling.

4.1.2 Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image is being loaded in another thread.

2. **Resource sharing:** By default, threads share the memory and the resources of the process to which they belong. The benefit of code sharing is that it allows an application to have several different threads of activity within the same address space.

3. **Economy:** Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

4. **Utilization of multiprocessor architectures:** The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.

4.1.3 User and Kernel Threads

Our discussion so far has treated threads in a generic sense. However, support for threads may be provided at either the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Most contemporary operating systems—including Windows XP, Solaris, and Tru64 UNIX (formerly Digital UNIX)—support kernelthreads.

4.1.4 Thread Libraries

A **thread library** provides the programmer an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today: (1) POSIX pthreads, (2) Java, and (3) Win32. An implementation of the POSIX standard may be of either the first or the second type. The Win32 thread library is a kernel-level library. The Java thread API may be implemented by Pthreads or Win32 or possibly another library.

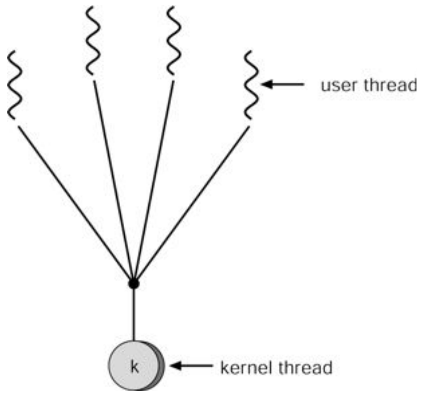
4.2 Multithreading Models

In Section 4.1.3, we distinguished between threads at the user and kernel levels. Ultimately, there must exist a relationship between these two types of structures. In this section, we look at three common ways of establishing this relationship.

4.2.1 Many-to-One Model

The many-to-one model (Figure 4.2) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. **Green threads**—a thread library available for Solaris—uses this model, as does **GNU Portable Threads**.

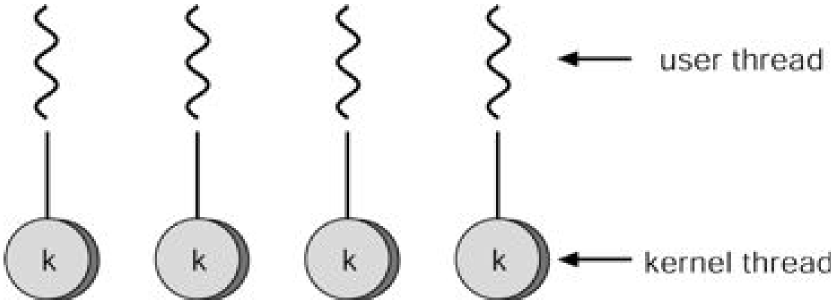
Figure 4.2 Many-to-one model.



4.2.2 One-to-One Model

The one-to-one model (Figure 4.3) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems-including Windows 95/98/NT/2000/XP-implement the one-to-one model.

Figure 4.3 One-to-one model.



4.2.3 Many-to-Many Model

The many-to-many model (Figure 4.4) multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor). Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time. The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create). The many-to-many model suffers from neither of these shortcomings: Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

One popular variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation, sometimes referred to as the *two-level model* (Figure 4.5), is supported by operating systems such as IRIX, HP-UX, and Tru64 UNIX. The Solaris operating system supported the two-level model in versions older than Solaris 9. However, beginning with Solaris 9, this system uses the one-to-one model.

Figure 4. 4 Many-to-many model.

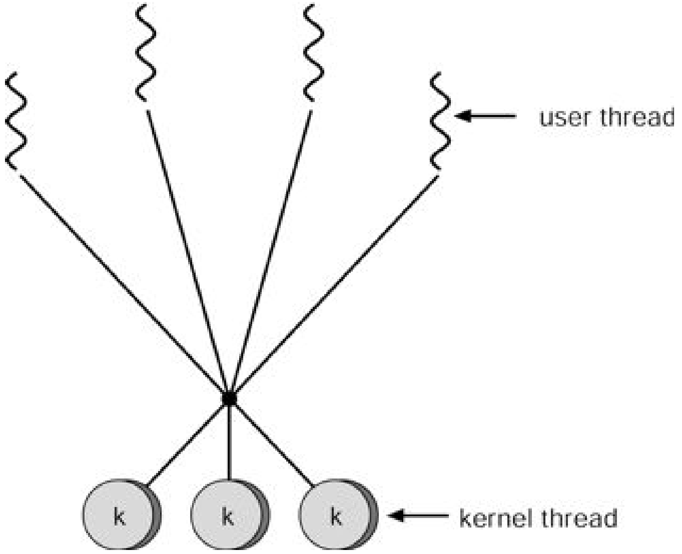
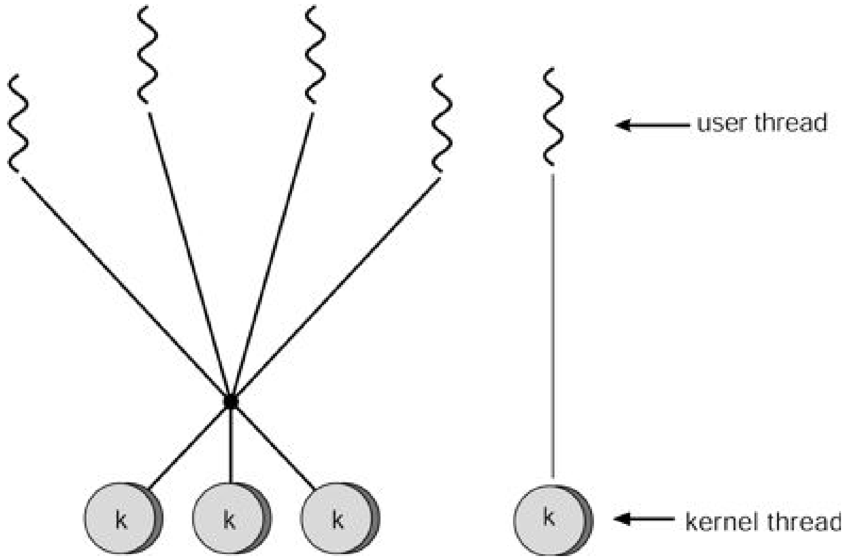


Figure 4. 5 Two-level model.



4.3 Threading Issues

In this section, we discuss some of the issues to consider with multithreaded programs.

4.3.1 The fork() and exec() System Calls

In Chapter 3, we described how the fork() system call is used to create a separate, duplicate process. In a multithreaded program, the semantics of the fork() and exec() system calls change. If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.

The exec() system call typically works in the same way as described in Chapter 3. That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads and LWPs. Which of the two versions of fork() to use depends on the application. If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate. If, however, the separate process does not call exec() after forking, the separate process should duplicate all threads.

4.3.2 Cancellation

Thread cancellation is the task of terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page is loaded using several threads (each image is loaded in a separate thread). When a user presses the *stop* button, all threads loading the page are cancelled. A thread that is to be cancelled is often referred to as the **target thread**.

Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation:** One thread immediately terminates the target thread.

2. **Deferred cancellation:** The target thread can periodically check whether it should terminate, allowing the target thread an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs in situations where resources have been allocated to a cancelled thread or where a thread is cancelled while in the middle of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a cancelled thread but will not reclaim all resources. Therefore, cancelling a thread asynchronously may not free a necessary systemwide resource. With deferred cancellation, in contrast, one thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not. This allows a thread to check whether it should be cancelled at a point when it can be cancelled safely. Pthreads refers to such points as **cancellation points**.

4.3.3 Signal Handling

A **signal** is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source and the reason for the event being signaled. All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.

2. A generated signal is delivered to a process.

3. Once delivered, the signal must be handled.

An example of a synchronous signal includes an illegal memory access or division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation causing the signal (that is the reason they are considered synchronous). When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as <control><C>) and having a timer expire. Typically, an asynchronous signal is sent to another process. Every signal may be *handled* by one of two possible handlers:

1. A default signal handler

2. A user-defined signal handler

Every signal has a **default signal handler** that is run by the kernel when handling that signal. This default action may be overridden by a **user-defined signal handler** function. In this instance, the user-defined function is called to handle the signal rather than the default action. Signals may be handled in different ways. Some signals may be simply ignored (such as changing the size of a window); others may be handled by terminating the program (such as an illegal memory access).

Handling signals in single-threaded programs is straightforward; signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where then should a signal be delivered?

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.

2. Deliver the signal to every thread in the process.

3. Deliver the signal to certain threads in the process.

4. Assign a specific thread to receive all signals for the process.

The method for delivering a signal depends on the type of signal generated. For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process. However, the situation with asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (<control><C>, for example)—should be sent to all threads. Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block. Therefore, in some cases, an asynchronous signal may be delivered to only those threads that are not blocking it. However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it.

Although Windows does not explicitly provide support for signals, they can be emulated using **asynchronous procedure calls (APCs)**. The APC facility allows a user thread to specify a function that is to be called when the user thread receives notification of a particular event. As indicated by its name, an APC is roughly equivalent to an asynchronous signal in UNIX. However, whereas UNIX must contend with how to deal with signals in a multithreaded environment, the APC facility is more straightforward, as an APC is delivered to a particular thread rather than a process.

4.3.4 Thread Pools

In Section 4.1, we mentioned multithreading in a web server. In this situation, whenever the server receives a request, it creates a separate thread to service the request. Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems. The first concerns the amount of time required to create the thread prior to servicing the request, together with the fact that this thread will be discarded once it has completed its work.

The second issue is more problematic: If we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this issue is to use **thread pools**.

The general idea behind a thread pool is to create a number of threads at process startup and place them into a *pool*, where they sit and wait for work. When a server receives a request, it awakens a thread from this pool—if one is available—and passes it the request to service. Once the thread completes its service, it returns to the pool and awaits more work. If the pool contains no available thread, the server waits until one becomes free.

Thread pools offer these benefits:

1. Servicing a request with an existing thread is usually faster than waiting to create a thread.

2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

The number of threads in the pool can be set heuristically based on factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests. More sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns. Such architectures provide the further benefit of having a smaller pool—thereby consuming less memory—when the load on the system is low.

4.3.5 Thread-Specific Data

Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread-specific data**. For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction may be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific data. Most thread libraries— including Win32 and Pthreads—provide some form of support for thread-specific data. Java provides support as well, and we will explore this in Section 4.7.5.

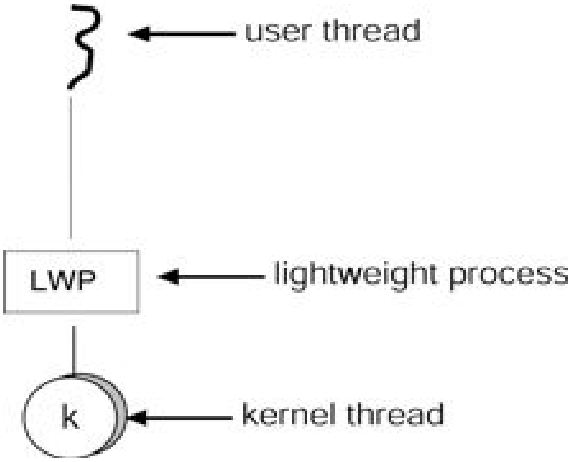
4.3.6 Scheduler Activations

A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models discussed in Section 4.2.3. Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

Many systems implementing either the many-to-many or two-level model place an intermediate data structure between the user and kernel threads. This data structure, typically known as a lightweight process or LWP, is shown in Figure 4.6. To the user-thread library, the LWP appears to be a *virtual processor* on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.

An application may require any number of LWPs to run efficiently. Consider a CPU bound application running on a uniprocessor. In this scenario, only one thread may be running at once, so one LWP is sufficient. An application that is I/O-intensive may require multiple LWPs to execute, however. Typically, an LWP is required for each concurrent blocking system call. Suppose, for example, that five different file-read requests occur simultaneously. Five LWPs are needed, because all could be waiting for I/O completion in the kernel. If a process has only four LWPs, then the fifth request must wait for one of the LWPs to return from the kernel.

Figure 4.6 Lightweight process (LWP.)



One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor. One event that triggers an upcall occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the application.

The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on this new virtual processor. When the event that the blocking thread was waiting on occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor. After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

4.4 Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a *specification* for thread behavior, not an *implementation*. Operating system designers may implement the specification in any way they wish. Numerous systems implement the Pthreads specification, including Solaris, Linux, Tru64 UNIX, and Mac OS X. *Shareware* implementations are available in the public domain for the various Windows operating systems as well. In this section, we introduce some of the Pthreads API as an example of a user-level thread library. We refer to it as a user-level library because no distinct relationship exists between a thread created using the Pthreads API and any associated kernelthreads.

The C program shown in Figure 4.7 demonstrates the basic Pthreads API for constructing a multithreaded program. If you are interested in more details on programming Pthreads, we encourage you to consult the Bibliographical Notes. The program shown in Figure 4.7 creates a separate thread for the summation of a non-negative integer. In a Pthreads program, separate threads begin execution in a specified function. In Figure 4.7, this is the runner() function. When this program begins, a single thread of control begins in main(). After some initialization, main() creates a second thread that begins control in the runner() function. Both threads share the global data sum.

We now provide a more detailed overview of this program. All Pthreads programs must include the pthread.h header file. The statement pthread_t tid declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The pthread_attr_t attr declaration represents the attributes for the thread. We will set the attributes in the function call pthread_attr_init(&attr). Because we did not explicitly set any attributes, we will use the default attributes provided.

A separate thread is created with the pthread_create() function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution—in this case, the runner() function. Last, we pass the integer parameter that was provided on the command line, argv[1].

At this point, the program has two threads: the initial thread in main() and the thread performing the summation in the runner() function. After creating the second thread, the main() thread will wait for the runner() thread to complete by calling the pthread_join() function. The runner() thread will complete when it calls the function pthread_exit(). Once the runner() thread has returned, the main() thread outputs the value of the shared data sum.

4.5 Windows XP Threads

Windows XP implements the Win32 API. The Win32 API is the primary API for the family of Microsoft operating systems (Windows 95/98/NT, Windows 2000 and Windows XP.) Indeed, much of what is mentioned in this section applies to this family of operating systems. A Windows XP application runs as a separate process. Each process may contain one or more threads. Windows XP uses the one-to-one mapping described in Section 4.2.2 where each user-level thread maps to an associated kernel thread. However, Windows XP also provides support for a **fiber** library, which provides the functionality of the

many-to-many model (Section 4.2.3). Every thread belonging to a process can access the virtual address space of the process.

The general components of a thread include:

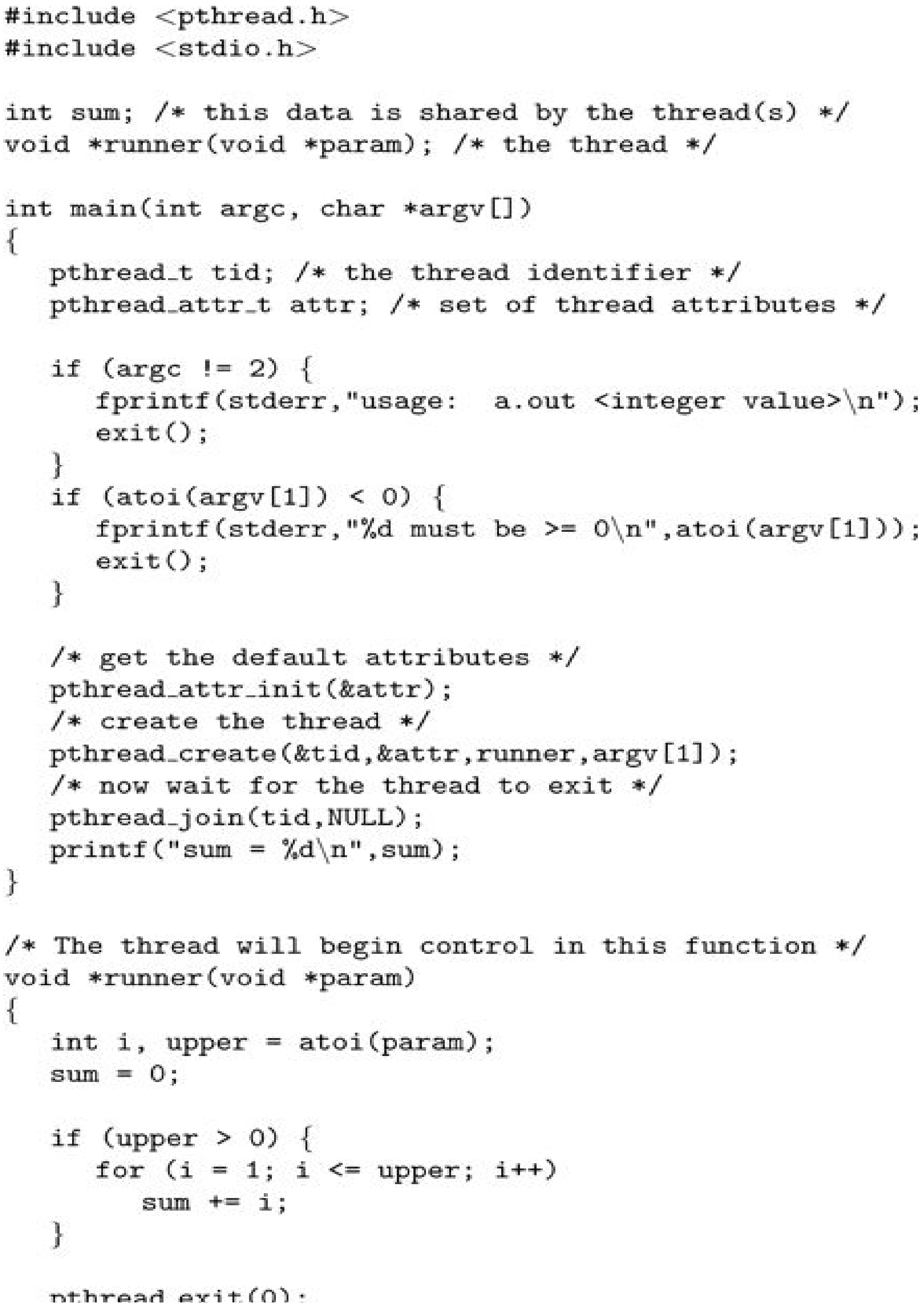
- A thread ID uniquely identifying the thread.

- A register set representing the status of the processor.

- A user stack used when the thread is running is user mode. Similarly, each thread has a kernel stack used when the thread is running in kernel mode.

- A private storage area used by various run-time libraries and dynamic link libraries (DLLs). The register set, stacks, and private storage area are known as the **context** of the thread.

Figure 4.7 Multithreaded C program using the Pthreads API.



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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage:  a.out <integer value>\n");
      exit();
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      exit();
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* now wait for the thread to exit */
   pthread_join(tid,NULL);
   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   if (upper > 0) {
      for (i = 1; i <= upper; i++)
         sum += i;
   }

   pthread_exit(0);
```

The primary data structures of a thread include:

- ETHREAD (executive thread block).
- KTHREAD (kernel thread block).
- TEB (thread environment block).

The key components of the ETHREAD include a pointer to the process to which the thread belongs and the address of the routine in which the thread starts control. The ETHREAD also contains a pointer to the corresponding KTHREAD.

The KTHREAD includes scheduling and synchronization information for the thread. In addition, the KTHREAD includes the kernel stack (used when the thread is running in kernel mode) and a pointer to the TEB.

The ETHREAD and the KTHREAD exist entirely in kernel space; this means only the kernel can access them. The TEB is a user-space data structure that is accessed when the thread is running in user mode. Among other fields, the TEB contains a user mode stack and an array for thread-specific data (which Windows XP terms **thread-local storage**)

4.6 Linux Threads

Linux provides a fork() system call with the traditional functionality of duplicating a process. Linux also provides the clone() system call, which is analogous to creating a thread. clone() behaves much like fork(), except that instead of creating a copy of the calling process, it creates a separate process that shares the address space of the calling process. This sharing of the address space of the parent process enables a cloned task to behave much like a separate thread. The sharing of the address space is allowed because of the way a process is represented in the Linux kernel. A unique kernel data structure exists for each process in the system. However, the data structure, rather than storing the data for the process, contains pointers to other data structures where this data is stored—for example, data structures that represent the list of open files, signal-handling information, and virtual memory. When fork() is invoked, a new process is created along with a *copy* of all the associated data structures of the parent process. A new process is also created when the clone() system call is made. However, rather than copying all data structures, the new process *points* to the data structures of the parent process, thereby allowing the child process to share the memory and other process resources of the parent.

A set of flags is passed as a parameter to the clone() system call. This set of flags is used to indicate how much of the parent process is to be shared with the child. If none of the flags is set, no sharing occurs; and clone() acts just like fork(). If all flags are set, the child process shares everything with the parent. Other combinations of flags allow various levels of sharing between these two extremes. The Linux kernel also creates several kernel threads that are designated for specific tasks; such as memory management.

Interestingly, Linux does not distinguish between processes and threads. In fact, Linux generally uses the term *task*—rather than *process* or *thread*—when referring to a flow of control within a program. Several Pthreads implementations are available for Linux, however; please consult the Bibliography for specifics.

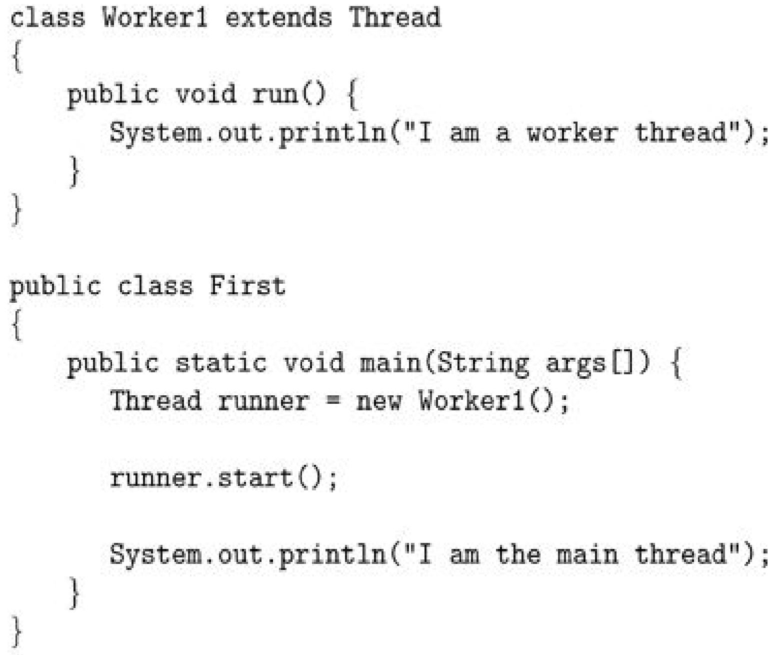
4.7 Java Threads

As we have already seen, support for threads may be provided at the user level with a library such as Pthreads. Furthermore, most operating systems provide support for threads at the kernel level as well. Java is one of a small number of languages that provide support at the language level for the creation and management of threads.

However, because threads are managed by the Java virtual machine (JVM), not by a user-level library or kernel, it is difficult to classify Java threads as either user- or kernel-level. In this section, we present Java threads as an alternative to the strict user- or kernel-level models. We also discuss how a Java thread can be mapped to the underlying kernel thread.

All Java programs comprise at least a single thread of control. Even a simple Java program consisting of only a main() method runs as a single thread in the JVM. In addition, Java provides commands that allow the developer to create and manipulate additional threads of control within the program.

Figure 4.8 Thread creation by extending the Thread class.



```
class Worker1 extends Thread
{
    public void run() {
       System.out.println("I am a worker thread");
    }
}

public class First
{
    public static void main(String args[]) {
       Thread runner = new Worker1();

       runner.start();

       System.out.println("I am the main thread");
    }
}
```

4.7.1 Thread Creation

One way to create a thread explicitly is to create a new class that is derived from the Thread class and to override the run() method of the Thread class. This approach is shown in Figure 4.8. An object of this derived class will run as a separate thread of control in the JVM. However, creating an object that is derived from the Thread class does not specifically create the new thread; rather, it is the start() method that actually creates the new thread. Calling the start() method for the new object (1) allocates memory and initializes a new thread in the JVM and (2) calls the run() method, making the thread eligible to be run by the JVM. (Note: Do not ever call the run() method directly. Call the start() method, and it will call the run() method on your behalf.)

When this program runs, two threads are created by the JVM. The first is the thread associated with the application—the thread that starts execution at the main() method. The second thread is the runner thread, created explicitly with the start() method. The runner thread begins execution in its run() method.

Another option to create a separate thread is to define a class that implements the Runnable interface. The Runnable interface is defined as follows:

```
public interface Runnable
{
        public abstract void run();
}
```

When a class implements Runnable, it must define a run() method. (The Thread class, in addition to defining static and instance methods, also implements the Runnable interface. That explains why a class derived from Thread must define a run() method.) Implementing the Runnable interface is similar to extending the Thread class. The only change is that "extends Thread" is substituted for "implements Runnable":

```
class Worker2 implements Runnable
{
        public void run()
        {
                System.out.println("I am a worker thread.");
        }
}
```

Creating a new thread from a class that implements Runnable is slightly different from creating a thread from a class extending Thread, however. Since the new class does not extend Thread, it does not have access to the static or instance methods—such as the start() method—of the Thread class. However, an object of the Thread class is still needed because it is the start() method that creates a new thread of control. Figure 4.9 shows how threads can be created using the Runnable interface.

In the class Second, a new Thread object is created that is passed a Runnable object in its constructor. When the thread is created with the start() method, the new thread begins execution in the run() method of the Runnable object. Why does Java support two approaches for creating threads? Which approach is more appropriate to use in what situations?

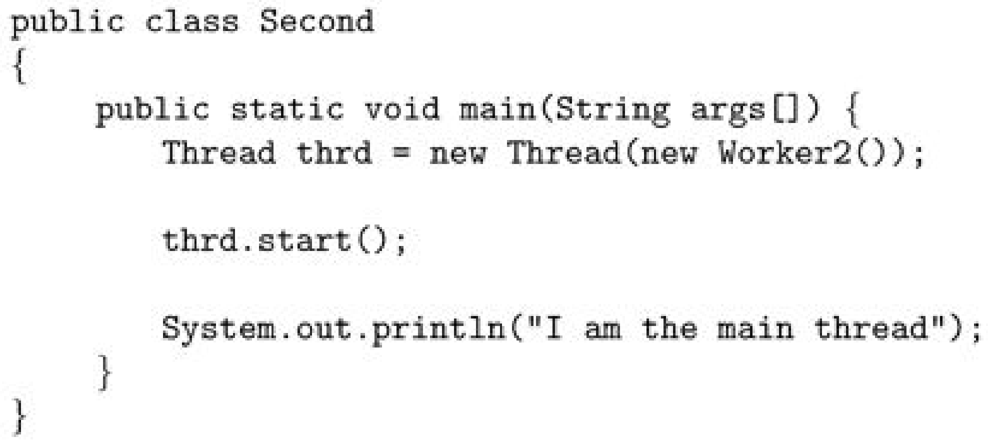
The first question is easy to answer. Since Java does not support multiple inheritance, if a class is already derived from another class, it will not also be able

to extend the Thread class. A good example is that an applet already extends the Applet class. To multithread an applet, you extend the Applet class and implement the Runnable interface:

```
public class ThreadedApplet extends Applet
implements Runnable
{
        . . .
}
```

The answer to the second question is less obvious. Object-oriented purists might say that, unless you are enhancing the Thread class, the class should not be extended. (This point may be moot, however, as many of the methods in the Thread class are defined as final.) We will not attempt to determine the correct answer in this debate. For this text, we adopt the practice of implementing the Runnable interface, as this approach seems more commonly used today.

Figure 4.9 Thread creation implementing the Runnable interface.



```
public class Second
{
    public static void main(String args[]) {
        Thread thrd = new Thread(new Worker2());

        thrd.start();

        System.out.println("I am the main thread");
    }
}
```

4.7.2 Thread States

A Java thread can be in one of four states:

1. *New:* A thread is in this state when an object for the thread is created (that is, the new statement).

2. *Runnable:* Calling the start() method allocates memory for the new thread in the JVM and calls the run() method for the thread object. When a thread's run() method is invoked, the thread moves from the new to the runnable state. A thread in the runnable state is eligible to be run by the JVM. Note that Java does not distinguish between a thread that is eligible to run and a thread that is currently running. A running thread is still in the runnable state.

3. *Blocked:* A thread becomes blocked if it performs a blocking statement—for example, by doing I/O—or if it invokes certain Java Thread methods, such as sleep().

4. *Dead:* A thread moves to the dead state when its run() method terminates.

It is not possible to determine the exact state of a thread, although the isAlive() method returns a boolean value that a program can use to determine whether or not a thread is in the dead state. Figure 4.10 illustrates the different thread states and labels several possible transitions.

4.7.3 Joining Threads

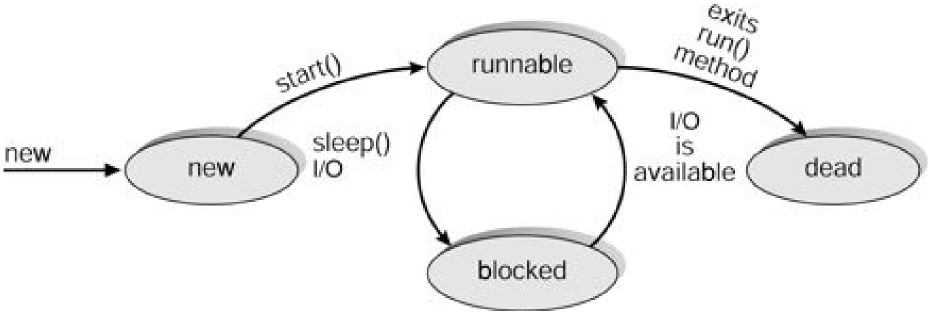
The thread created in the program in Figure 4.9 runs independently of the thread that creates it (the thread associated with the main() method). In situations where the creating thread wants to wait for any threads it has created Java provides the join() method. By invoking join(), the creating thread is able to wait for the run() method of the joining thread to terminate before continuing its operation. The join() method is useful in situations where the creating thread can continue only after a worker thread has completed. For example, assume that thread *A* creates thread *B* and thread *B* will perform calculations that thread *A* requires. In this scenario, thread *A* will join on thread *B*.

We illustrate the join() method by modifying the code in Figure 4.9 as follows:

```
Thread thrd = new Thread(new Worker2());
thrd.start();
try {
thrd.join();
} catch (InterruptedException ie) {}
```

Note that join() can throw an InterruptedException, which we will ignore for now.

Figure 4.10 Java thread states.



4.7.4 Thread Cancellation

In Section 4.3.2, we discussed issues concerning thread cancellation. Java threads can be asynchronously terminated using the stop() method of the Thread class. However, this method was **deprecated**. Deprecated methods are still implemented in the current API; however, their use is discouraged.

It is possible to cancel a thread using deferred cancellation as described in Section 4.3.2. Recall that deferred cancellation works by having the target thread periodically check whether it should terminate. In Java, checking involves use of

the interrupt() method. The Java API defines the interrupt() method for the Thread class. When the interrupt() method is invoked, the **interruption status** of the target thread is set. A thread can periodically check its interruption status by invoking either the interrupted() method or the isInterrupted() method, both of which return true if the interruption status of the target thread is set. (It is important to note that the interrupted() method will clear the interruption status of the target thread, whereas the isInterrupted() method preserves the interruption status.) Figure 4.11 illustrates how deferred cancellation works when the isInterrupted() method is used.

An instance of an InterruptibleThread can be interrupted using the following code:

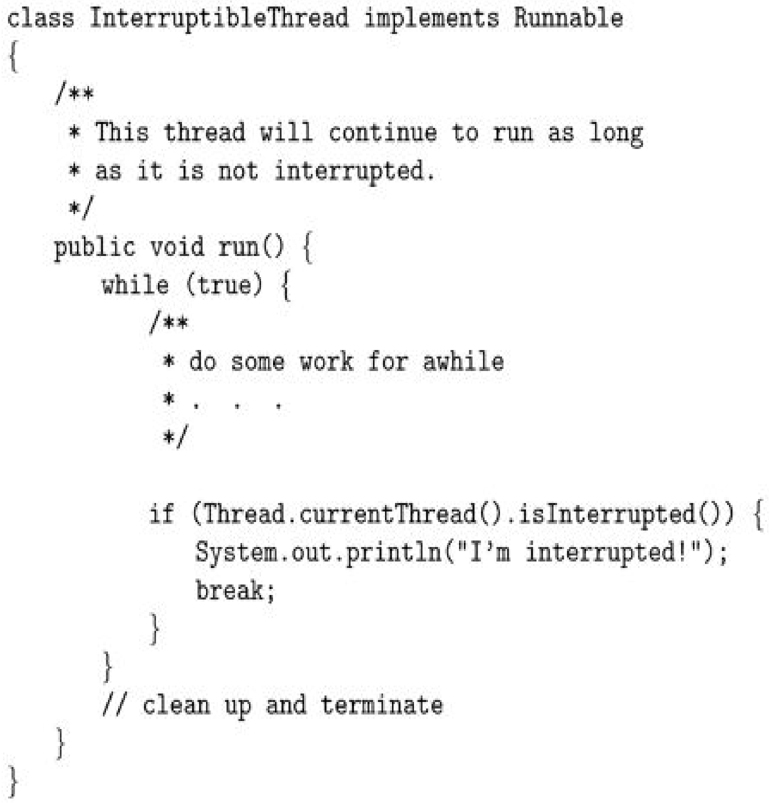
```
Thread thrd = new Thread(new

InterruptibleThread());

thrd.start();

. . .

thrd.interrupt();
```

In this example, the target thread can periodically check its interruption status via the isInterrupted() method and—if it is set—clean up before terminating. Because the InterruptibleThread class does not extend Thread, it cannot directly invoke instance methods in the Thread class. To invoke instance methods in Thread, a program must first invoke the static method currentThread() which returns a Thread object representing the thread that is currently running. This return value can then be used to access instance methods in the Thread class.

It is important to recognize that interrupting a thread via the interrupt() method only sets the interruption status of a thread; it is up to the target thread to periodically check this interruption status. Traditionally, Java does not wake a thread that is blocked in an I/O operation using the java.io package. Any thread blocked doing I/O in this package will not be able to check its interruption status

until the call to I/O is completed. However, the java.nio package introduced in Java 1.4 provides facilities for interrupting a thread that is blocked performing I/O.

Figure 4.11 Deferred cancellation using the isInterrupted() method.

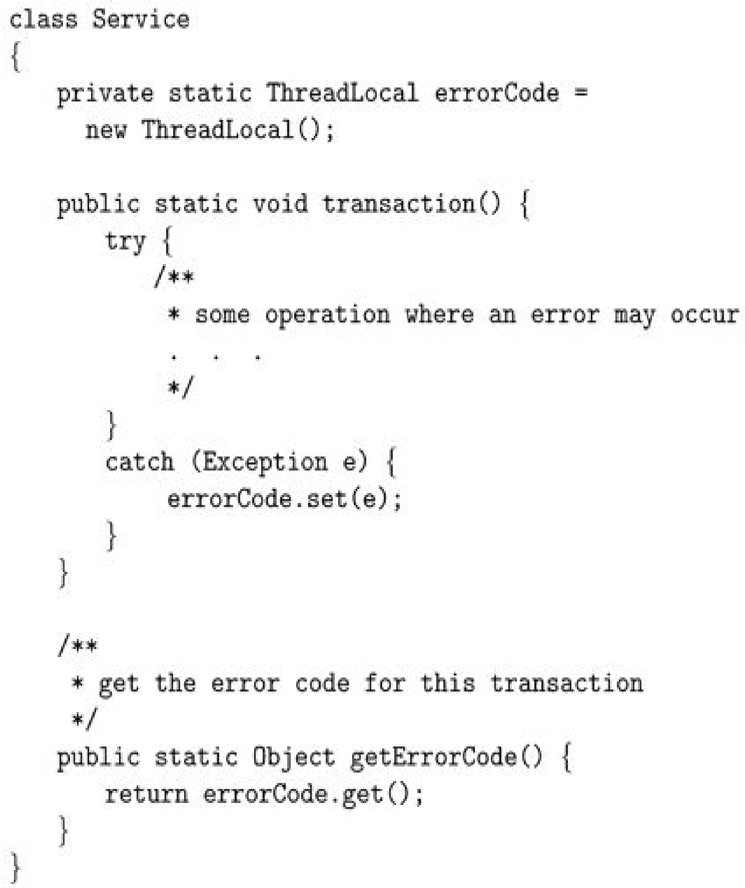


```
class InterruptibleThread implements Runnable
{
   /**
    * This thread will continue to run as long
    * as it is not interrupted.
    */
   public void run() {
      while (true) {
         /**
          * do some work for awhile
          * .  .  .
          */

         if (Thread.currentThread().isInterrupted()) {
            System.out.println("I'm interrupted!");
            break;
         }
      }
      // clean up and terminate
   }
}
```

4.7.5 Thread-Specific Data

In Section 4.3.5, we described thread-specific data, which allow a thread to have its own private copy of data. At first glance, it may appear that Java has no need for thread-specific data, and this viewpoint is partly correct. All that is required for each thread to have its own private data is to create threads by subclassing the Thread class and to declare instance data in this class. This approach works fine when threads are constructed in this way. However, when the developer has no control over the thread-creation process—for example, when a thread pool is being used—then an alternative approach is necessary.

Figure 4.12 Using the ThreadLocal class.



```
class Service
{
    private static ThreadLocal errorCode =
      new ThreadLocal();

    public static void transaction() {
        try {
            /**
             * some operation where an error may occur
             .  .  .
            */
        }
        catch (Exception e) {
            errorCode.set(e);
        }
    }

    /**
     * get the error code for this transaction
     */
    public static Object getErrorCode() {
        return errorCode.get();
    }
}
```

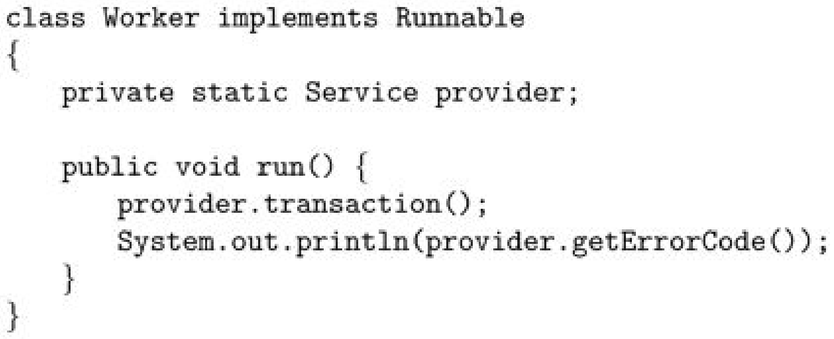
The Java api provides the ThreadLocal class for declaring thread-specific data. ThreadLocal data can be initialized with either the initialValue() method or the set() method, and a thread can inquire as to the value of ThreadLocal data using the get() method. Typically, ThreadLocal data are declared as static. Consider the Service class shown in Figure 4.12, which declares errorCode as ThreadLocal data. The transaction() method in this class can be invoked by any number of threads. If an exception occurs, we assign the exception to errorCode using the set() method of the ThreadLocal class.

Now consider a scenario in which two threads—say, thread 1 and thread2—invoke transaction(). Assume that thread 1 generates exception *A* and thread2 generates exception *B*. The value of errorCode for thread 1 and thread 2 will be *A* and *B*, respectively. Figure 4.13 illustrates how a thread can inquire as to the value of errorCode() after invoking transaction().

4.7.6 The JVM and the Host Operating System

The JVM is typically implemented on top of a host operating system. This setup allows the JVM to hide the implementation details of the underlying operating system and to provide a consistent, abstract environment that allows Java programs to operate on any platform that supports a JVM. The specification for the JVM does not indicate how Java threads are to be mapped to the underlying operating system, instead leaving that decision to the particular implementation of the JVM. For example, the Windows 2000 operating system use the one-to-one model; therefore, each Java thread for a JVM running on these systems maps to a kernel thread. On operating systems that use the many-to-many model (such as Tru64 UNIX), a Java thread is mapped according to the many-to-many model. Solaris initially implemented the JVM using the many to- one model (called **green threads**). Later releases of the JVM were implemented using the many-to-many model. Beginning with Solaris 9, Java threads were mapped using the one-to-one model.

Figure 4.13 Inquiring the value of ThreadLocal data.

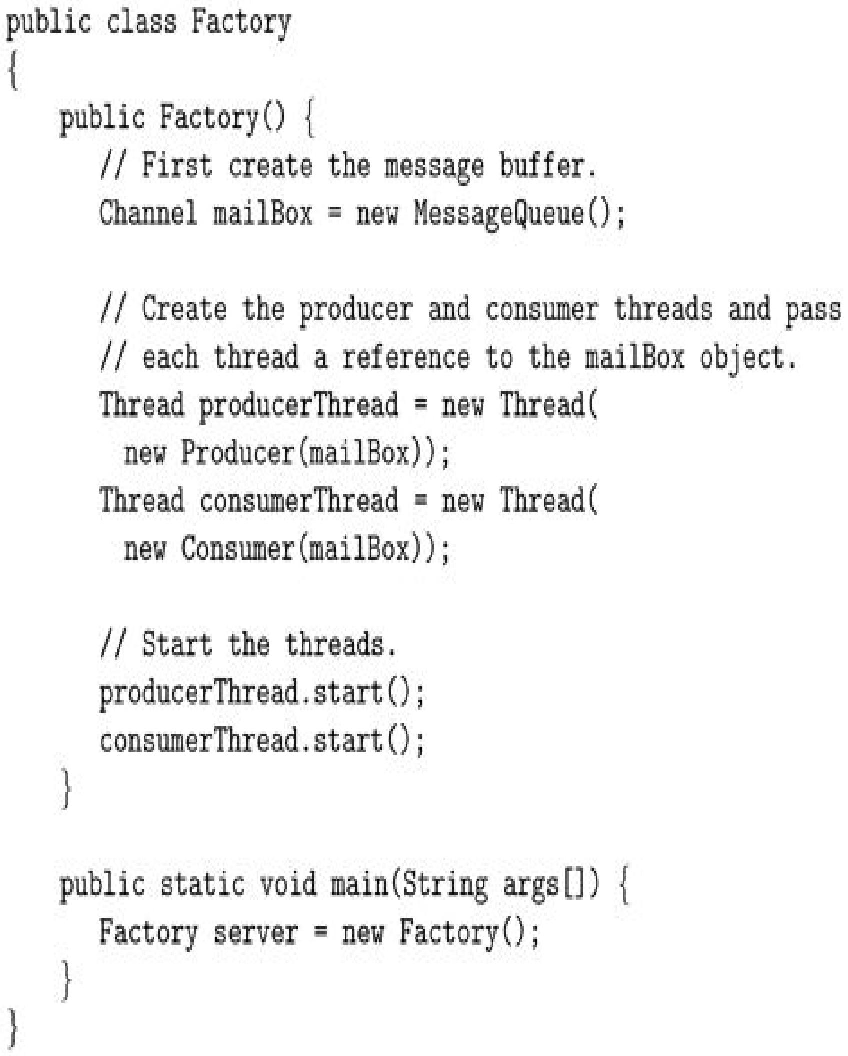


```
class Worker implements Runnable
{
    private static Service provider;

    public void run() {
        provider.transaction();
        System.out.println(provider.getErrorCode());
    }
}
```

4.7.7 Multithreaded Solution Example

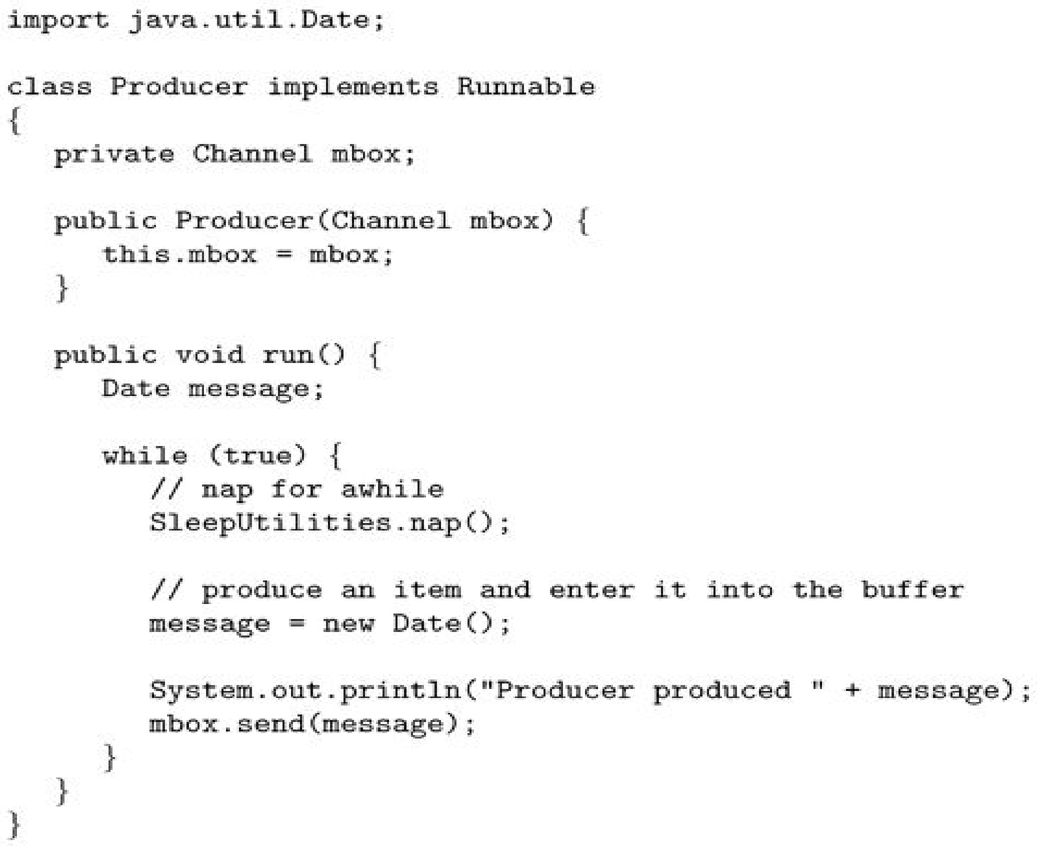
We conclude our discussion of Java threads with a complete multithreaded solution to the producer-consumer problem that uses message passing. The class Factory in Figure 4.14 first creates a mailbox for buffering messages, using the MessageQueue class. It then creates separate producer and consumer threads (Figures 4.15 and 4.16, respectively) and passes each thread a reference to the shared mailbox. The producer thread alternates among sleeping for a while, producing an item, and entering that item into the mailbox. The consumer alternates between sleeping and then retrieving an item from the mailbox and consuming it. Because the receive() method of the MessageQueue class is non-blocking, the consumer must check to see whether the message that it retrieved is null.

Figure 4.14 The class Factory.



```
public class Factory
{
   public Factory() {
      // First create the message buffer.
      Channel mailBox = new MessageQueue();

      // Create the producer and consumer threads and pass
      // each thread a reference to the mailBox object.
      Thread producerThread = new Thread(
        new Producer(mailBox));
      Thread consumerThread = new Thread(
        new Consumer(mailBox));

      // Start the threads.
      producerThread.start();
      consumerThread.start();
   }

   public static void main(String args[]) {
      Factory server = new Factory();
   }
}
```

Figure 4.15 Producer thread.



```
import java.util.Date;

class Producer implements Runnable
{
   private Channel mbox;

   public Producer(Channel mbox) {
      this.mbox = mbox;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // produce an item and enter it into the buffer
         message = new Date();

         System.out.println("Producer produced " + message);
         mbox.send(message);
      }
   }
}
```

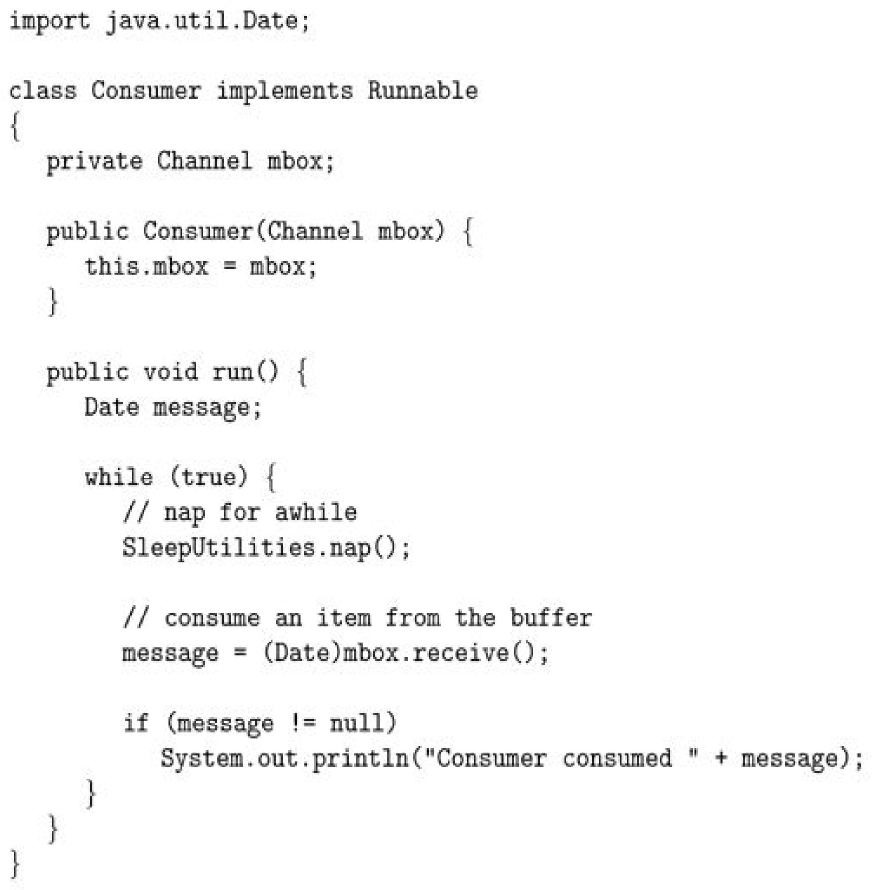
4.8 Summary

A thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and the ability to take advantage of multiprocessor architectures.

User-level threads are threads that are visible to the programmer and are unknown to the kernel. A thread library in user space typically manages user-level threads. The operating-system kernel supports and manages kernel-level threads. In general, userlevel threads are faster to create and manage than are kernel threads. Three different types of models relate user and kernel threads: The many-to-one model maps many user threads to a single kernel thread. The one-to-one model maps each user thread to a corresponding kernel thread. The many-to-many model multiplexes many user threads to a smaller or equal number of kernel threads.

Multithreaded programs introduce many challenges for the programmer, including the semantics of the fork() and exec() system calls. Other issues include thread cancellation, signal handling, and thread-specific data. Most modern operating systems provide kernel support for threads; among these are Windows NT and Windows, Solaris, and Linux. The Pthreads API provides a set of functions to create and manage threads at the user level. Java provides a similar API for supporting threads. However, because Java threads are managed by the JVM and not by a userlevel thread library or kernel, they do not fall under the category of either user- or kernel-level threads.

Figure 4.16 Consumer thread.



```
import java.util.Date;

class Consumer implements Runnable
{
   private Channel mbox;

   public Consumer(Channel mbox) {
      this.mbox = mbox;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // consume an item from the buffer
         message = (Date)mbox.receive();

         if (message != null)
            System.out.println("Consumer consumed " + message);
      }
   }
}
```

Exercises

4.1 Provide two programming examples in which multithreading provides better performance than a single-threaded solution.

4.2 Provide two programming examples in which multithreading does *not* provide better performance than a single-threaded solution.

4.3 What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

4.4. Describe the actions taken by a kernel to context-switch between kernellevel threads.

4.5. Describe the actions taken by a thread library to context-switch between user-level threads.

4.6. What resources are used when a thread is created? How do they differ from those used when a process is created?

4.7. Assume an operating system maps user-level threads to the kernel using the many-to-many model where the mapping is done through LWPs. Furthermore, the system allows the developers to create real-time threads. Is it necessary to bind a real-time thread to an LWP? Explain.

4.8. A Pthreads program that performs the summation function was provided in Section 4.4. Rewrite this program in Java.

4.9. Write a multithreaded Java or Pthreads program that generates the Fibonacci series. This program should work as follows: The user will run the program and will enter on the command line the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers.

4.10 Write a multithreaded Java or Pthreads program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number that the user entered.

4.11 Write a multithreaded Java or Pthreads program that performs matrix multiplication. Specifically, use two matrices, A and B, where A is a matrix with M rows and K columns and matrix B contains K rows and N columns. The *matrix product* of A and B is C, where C contains M rows and N columns. The entry in matrix C for row i column j ($C_{i,j}$) is the sum of the products of elements for row i in matrix A and column j in matrix B. That is,

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} \times B_{n,j}$$

For example, if A were a 3-by-2 matrix and B were a 2-by-3 matrix, element $C_{3,1}$ would be the sum of $A_{3,1} \times B_{1,1}$ and $A_{3,2} \times B_{2,1}$. Calculate each element $C_{i,j}$ in a separate thread. This will involve creating $M \times N$ threads.

UNIT – II

CPU SCHEDULING

Structure

5. INTRODUCTION

CPU scheduling is the basis of multi programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

5.1 Basic Concepts

The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. In a unipocessor system, only one process may run at a time; any other processes must wait until the CPU is free .

The idea of multiprogramming is relatively simple. A process is executed, eventually it has to wait for the completion of some I/O request. In a simple computer system, the CPU would then sit idle; all this waiting time is wasted. With multiprogramming, we try to use this time . Several processes are kept in memory at one time. When 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. Scheduling 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.

5.1.1CPU-I/0 Burst Cycle

The success of CPU scheduling depends on the following 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, then another CPU burst, then another I/O burst, and so on. Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst (Figure 5.1).

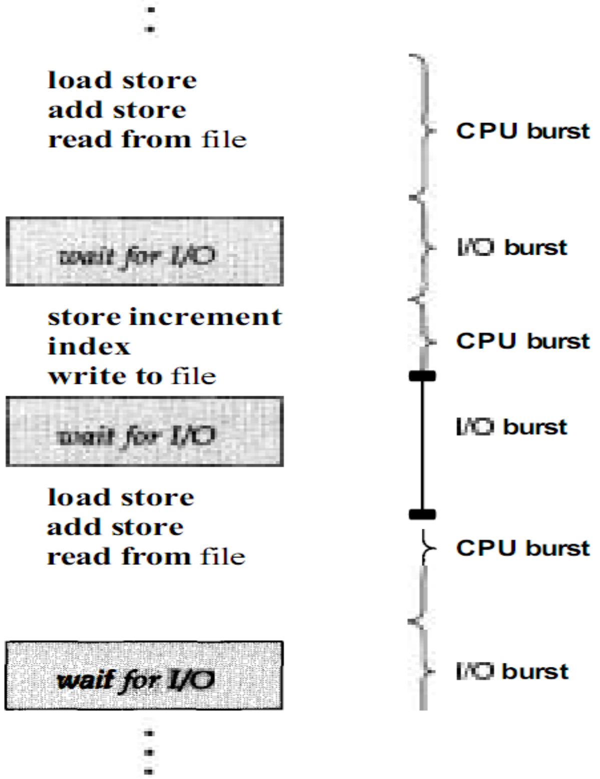


Figure 5.1 Alternating sequence of CPU and I/O burst.

5.1.2 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 from among the processes in memory that are ready to execute, and allocates the CPU to one of them. A ready queue may be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

5.1.3 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, I/O request, or invocation of wait for the termination of one of the child processes)

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, completion of I/O)
4. When a process terminates

In circumstances 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, in circumstance 2 and 3. When scheduling takes place only under circumstances 1 and 4, the scheduling scheme may be nonpreemptive or 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 is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems.

Unfortunately, preemptive scheduling incurs a cost. Consider the case of two processes sharing data. One may be in the midst of updating the data when it is preempted and the second process is run. The second process may try to read the data, which are currently in an inconsistent state. New mechanisms thus are needed to coordinate access to shared data.

Preemption also has an effect on 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 could ensue. Some 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.

5.1.4 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:

- 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, given that 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.

5.1.5 Scheduling Criteria

Different CPU-scheduling algorithms have different properties and 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. The characteristics used for comparison can make a substantial difference in the determination of the best algorithm. The criteria include the following:

CPU utilization: We want to keep the CPU as busy as possible. CPU utilization may 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 used system).

Throughput: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit, called throughput. For long processes, this rate may be 1 process per hour; for short transactions, throughput might be 10 processes per second.

Turnaround time: 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: The measure, called response time, is the amount of time it takes to start responding, but not the time that it takes to output that response. The turnaround time is generally limited by the speed of the output device.

We want to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure.

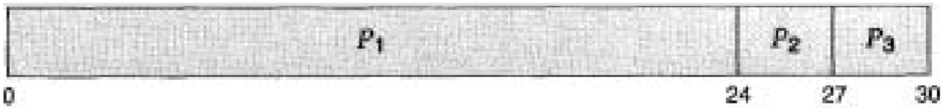
5.2 Scheduling Algorithms

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

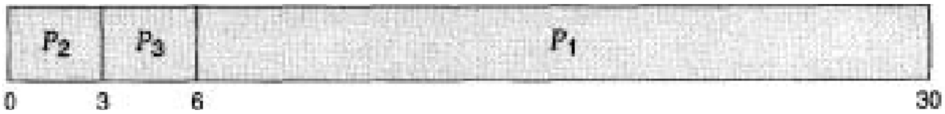
1.2.1 First-Come, First-Served Scheduling: By far the simplest CPU-scheduling algorithm is the first-come first-served (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO 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. The average waiting time under the FCF policy, is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process	**Burst Time**
PI	24
P2	3
P3	3

If the processes arrive in the order PI, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart:



The waiting time is 0 milliseconds for process PI, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P2, P3, P1, 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 a FCFS policy is generally not minimal, and may vary substantially if the process 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 the CPU and hold it. During this time, all the other processes will finish their I/O and 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 very 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. 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 particularly troublesome for time-sharing systems.

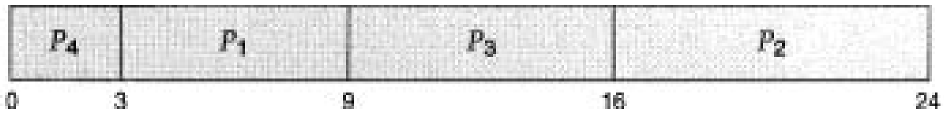
5.2.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the shortest-job-first **(SJF)** scheduling algorithm. This algorithm gives the first importance to the process which has short CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. We use the term SJF because most people and textbooks refer to this type of scheduling discipline as SJF.

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

Process	**Burst Time**
PI	**6**
p2	**8**
p3	***7***
p4	**3**

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



The waiting time is 3 milliseconds for process PI, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process p4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. If we were using the FCFS scheduling scheme, then the average waiting time would be 10.25 milliseconds.

By moving a short process before a long one, the waiting time of the short process decreases more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

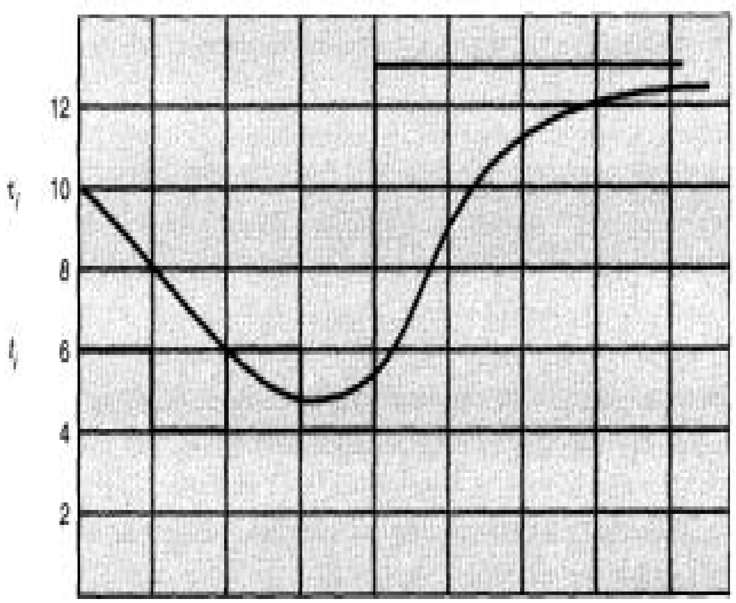
The real difficulty with the SJF algorithm is knowing the length of the next CPU request. users are motivated to estimate the process time limit accurately, since a lower value may mean faster response.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. One approach 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

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t, be the length of the nth CPU burst, and let our predicted value for the next CPU burst. Then, for α ,0 < α < 1, define

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

This formula defines an **exponential average.**



time --->

CPU burst (ti) 6 4 6 4 13 13 13 .. .

"**guess" (T~)** 10 8 **6** 6 5 9 11 12 . . .

5.2 Prediction of the length of the next CPU burst.

5.2.3 priority Scheduling

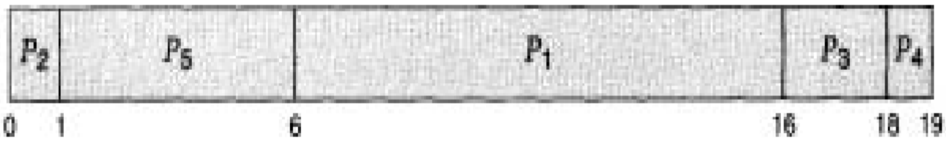
A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

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. Priorities are generally some fixed range of numbers, such as 0 to 7, or 0 to 4,095.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order PI, P2, ..., Ps, with the length of the CPU-burst time 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 that are external to 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 non preemptive. 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 non preemptive 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 priority-scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

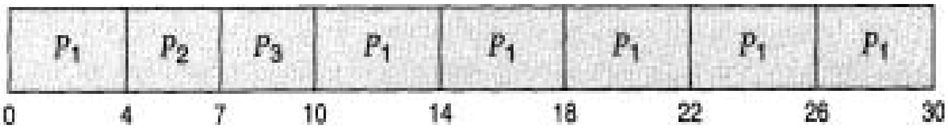
A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

5.2.4 Round-Robin Scheduling

The round-robin **(RR)** scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added 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. 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 1time quantum. To implement RR scheduling, we keep 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. Otherwise, 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, however, is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process	**Burst Time**
PI	**24**
P2	**3**
P3	**3**

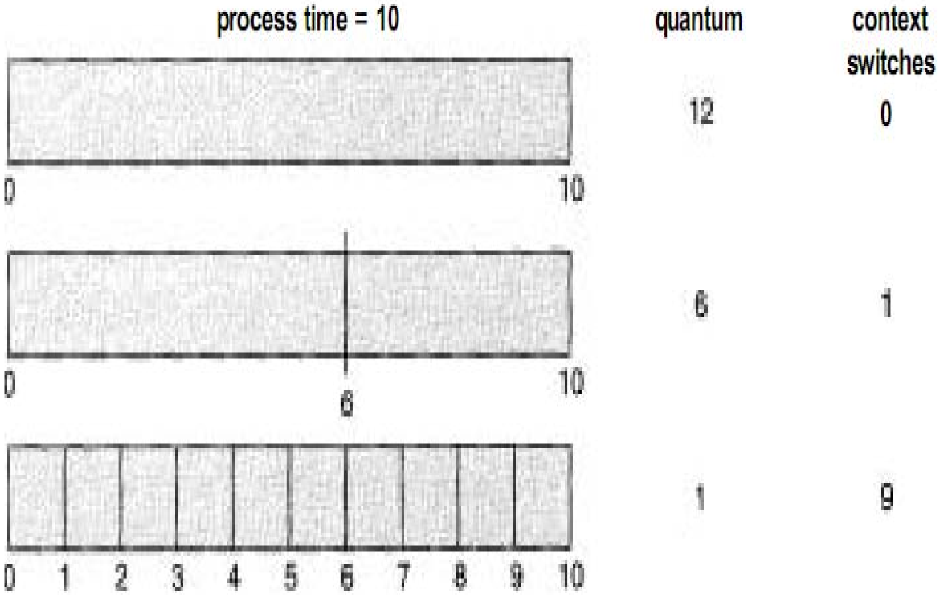
If we use a time quantum of 4 milliseconds, then process P1 gets the first4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Since process P2 does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is



The average waiting time is ***17/3*** = 5.66 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is very large (infinite), the RR policy is the same as the FCFS policy. If the time quantum is very small (say***1*** microsecond), the ***RR*** approach is called processor sharing, and appears (in theory) to the users as though each of *n* processes has its own processor running at ***l/n*** the speed of the real processor.

We need also to consider the effect of context switching on the performance of RR scheduling. Let us assume 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 2quanta, resulting in 1 context switch. If the time quantum is 1 time unit, then 9 context switches will occur, slowing the execution of the process accordingly

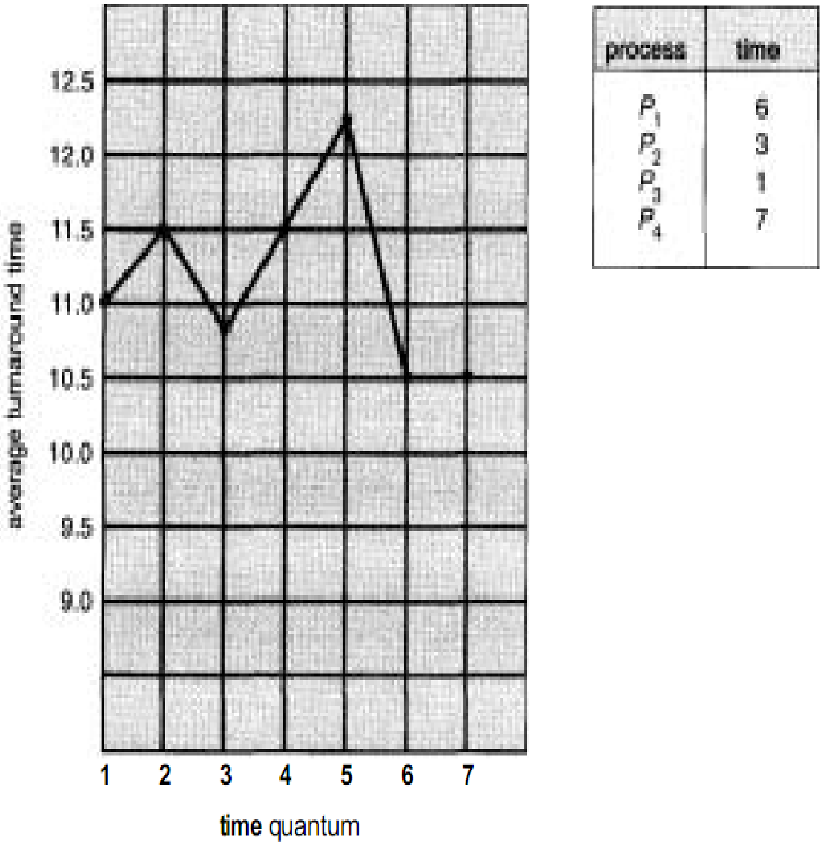


5.3 Showing how a smaller time quantum increases context switches.

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 switch.

Turnaround time also depends on the size of the time quantum. As we can see from Figure the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turn around 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 for a smaller time quantum, since more context switches will be required.



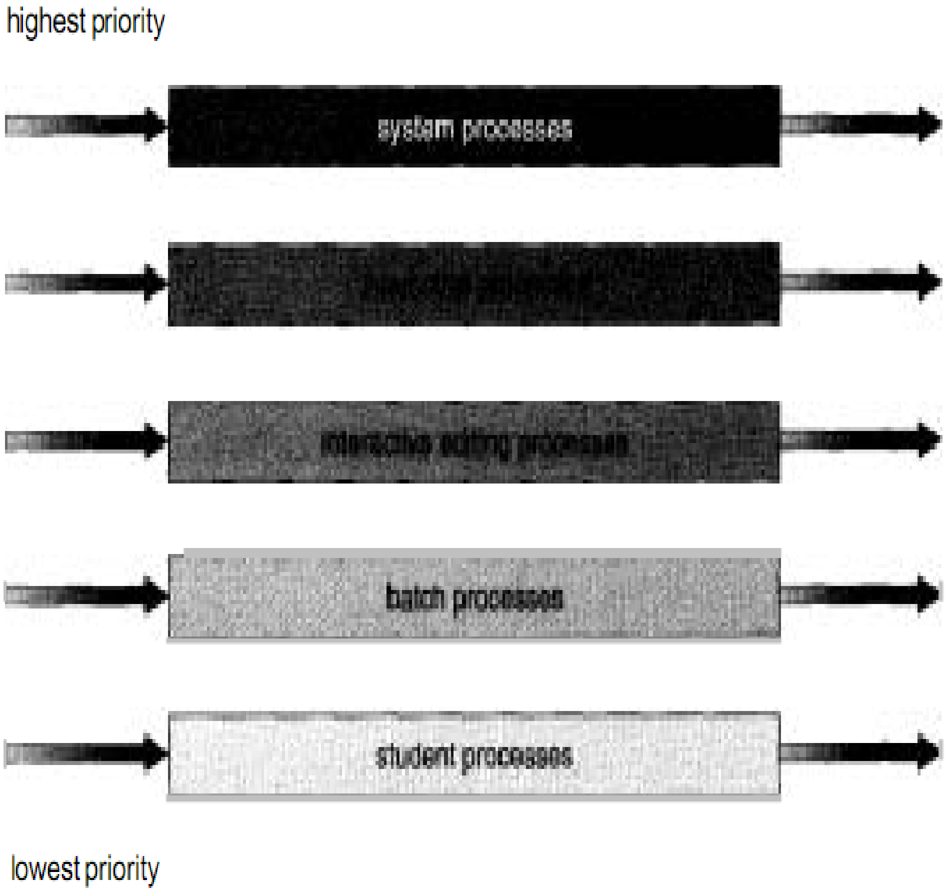
process	time
P_1	6
P_2	3
P_3	1
P_4	7

5.4 Showing how turnaround time varies with the time quantum.

5.2.5 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 common division is made between foreground(or interactive) processes and back-ground (or batch)processes. These two types of processes have different response-time requirements, and so might have different scheduling needs.

A multilevel queue-scheduling algorithm partitions the ready queue into several separate queues .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.



5.5 Multilevel queue scheduling

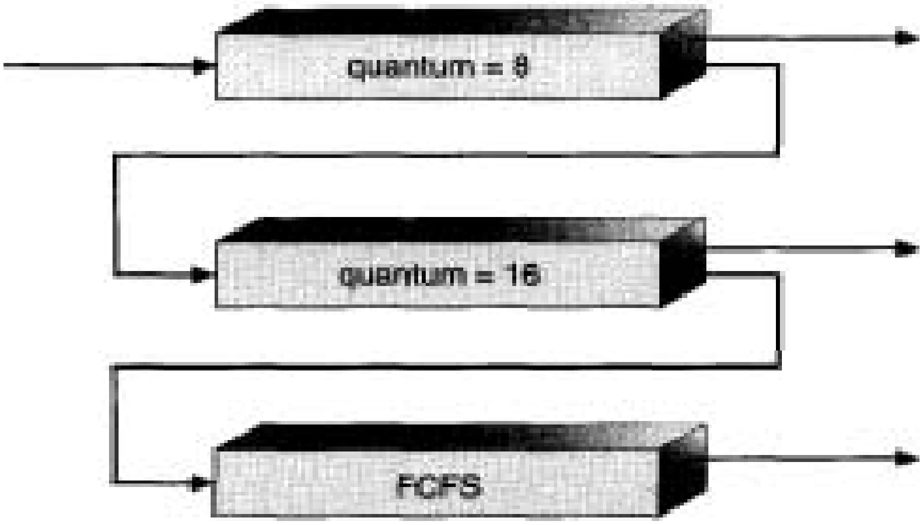
No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time slice between the queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue.

5.2.6 Multilevel Feedback Queue Scheduling

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. 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. Similarly, 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. The scheduler first executes all 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 that arrives for queue 0 will, in turn, preempt a process in queue 1.



5.6 Multilevel feedback queues.

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, only when queues 0 and 1 are empty.

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

5.3 Multiple-Processor Scheduling

Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processor. If multiple CPUs are available, the scheduling problem is correspondingly more complex. Many possibilities have been tried, and, as we saw with single-processor CPU scheduling, there is no one best solution. In the following, we discuss briefly some of the issues concerning multiprocessor scheduling. We concentrate on systems where the processors are identical (or homogeneous) in terms of their functionality; any available processor can then be used to run any processes in the queue. We also assume uniform memory access (UMA).

Even within a homogeneous multiprocessor, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes wishing to use that device must be scheduled to run on that processor, otherwise the device would not be available.

If several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. In this case, however, one processor could be idle, with an empty queue, while another processor was very busy. To prevent this situation, we use a common ready queue. All processes go into one queue and are scheduled onto any available processor.

In such a scheme, one of two scheduling approaches may be used. In one approach, each processor is self-scheduling. Each processor examines the common ready queue and selects a process to execute. If we have multiple processors trying to access and update a common data structure, each processor must be programmed very carefully. We must ensure that two processors do not choose the same process, and that processes are not lost from the queue. The other approach avoids this problem by appointing one processor as scheduler for the other processors, thus creating a master-slave structure.

Some systems carry this structure one step further, by having all scheduling decisions, I/O processing, and other system activities handled by one single processor-the master server. The other processors only execute user code.

This asymmetric multiprocessing is far simpler than symmetric multiprocessing, because only one processor accesses the system data structures, alleviating the need for data sharing. However, it is also not as efficient. I/O-bound processes may bottleneck on the one CPU that is performing all of the operations. Typically, asymmetric multiprocessing is implemented first within an operating system, and is then upgraded to symmetric multiprocessing as the system evolves.

5.4 Real-Time Scheduling

Real-time computing is divided into two types**.** **Hard real-time** systems are required to complete a critical task within a guaranteed amount of time. Generally, a process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O. The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as resource reservation. Such a guarantee requires that the scheduler know exactly how long each type of operating-system function takes to perform, and therefore each operation must be guaranteed to take a maximum amount of time.

Soft real-time computing is less restrictive. It requires that critical processes receive priority over less fortunate ones. Although adding soft real-time functionality to a time-sharing system may cause an unfair allocation of resources and may result in longer delays, or even starvation, for some processes, it is at least possible to achieve. The result is a general-purpose system that can also support multimedia, high-speed interactive graphics, and a variety of tasks that would not function acceptably in an environment that does not support soft real-time computing.

Implementing soft real-time functionality requires careful design of the scheduler and related aspects of the operating system. First, the system must have priority scheduling, and real-time processes must have the highest priority. The priority of real-time processes must not degrade over time, even though the priority of non-real-time processes may. Second, the dispatch latency must be small. The smaller the latency, the faster a real-time process can start executing once it is runable.

To keep dispatch latency low, we need to allow system calls to be preemptible. There are several ways to achieve this goal. One is to insert preemption points in long-duration system calls, that check to see whether a high- priority process needs to be run. If so, a context switch takes place and, when the high-priority process terminates, the interrupted process continues with the system call. Preemption points can be placed at only "safe" locations in the kernel-only where kernel data structures are not being modified. Even with preemption points dispatch latency can be large, because only a few preemption points can be practically added to a kernel.

Another method for dealing with preemption is to make the entire kernel preemptible. So that correct operation is ensured, all kernel data structures must be protected through the use of various synchronization mechanisms . With this method, the kernel can always be preemptible, because any kernel data being updated are protected from modification by the high-priority process. This is the most effective (and complex) method in widespread use; it is used in Solaris 2.

But what happens if the higher-priority process needs to read or modify kernel data currently being accessed by another, lower-priority process? The high-priority process would be waiting for a lower-priority one to finish. This situation is known as priority inversion. In fact, a chain of processes could all be accessing resources that the high-priority process needs. This problem can be solved via the priority-inheritance protocol, in which all these processes (the ones accessing resources that the high-priority process needs) inherit the high priority until they are done with the resource in question. When they are finished, their priority reverts to its original value. In Figure 1.7, we show the makeup of dispatch latency. The conflict phase of dispatch latency has two components:

1. Preemption of any process running in the kernel

2. Release by low-priority processes resources needed by the high-priority process

As an example, in Solaris 2, the dispatch latency with preemption disabled is over 100 milliseconds. However, the dispatch latency with preemption enabled is usually reduced to 2 milliseconds.

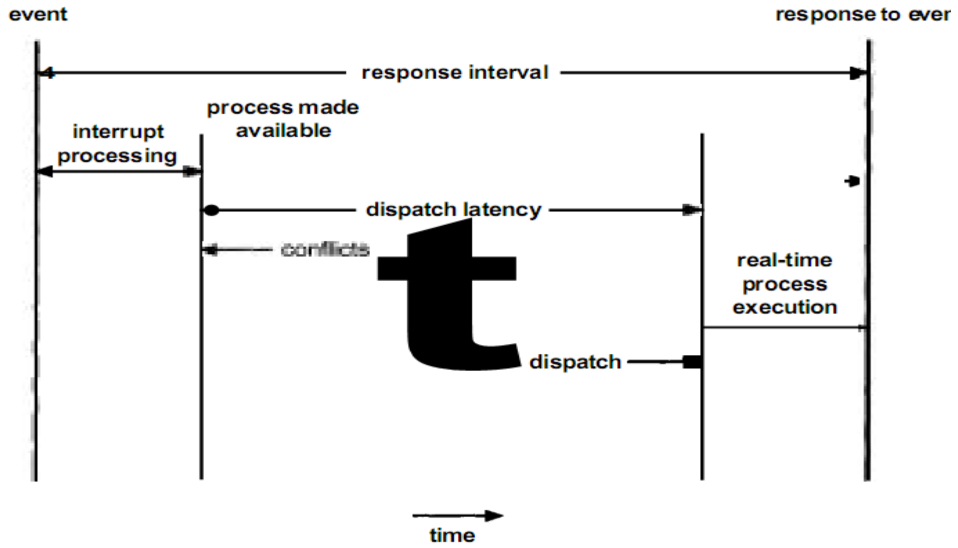


Figure 5.7 Dispatch latency.

5.5 Summary

- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.
- First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes. Shortest job-first (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult because predicting the length of the next CPU burst is difficult.
- Round-robin (RR) scheduling is more appropriate for a time-shared (interactive) system. RR scheduling allocates the CPU to the first process in the ready queue for q time units, where q is the time quantum. After q time units, if the process has not relinquished the CPU, it is preempted and the process is put at the tail of the ready queue. The major problem is the selection of the time quantum.
- The FCFS algorithm is non preemptive; the RR algorithm is preemptive. The SJF and priority algorithms may be either preemptive or non preemptive.
- Multilevel queue algorithms allow different algorithms to be used for various classes of processes. The most common is a foreground interactive queue, which uses RR scheduling, and a background batch queue, which uses FCFS scheduling. Multilevel feedback queues allow processes to move from one queue to another.

Exercises

5.1 A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of n.

5.2 Define the difference between preemptive and non preemptive scheduling. State why strict non preemptive scheduling is unlikely to be used in a computer center.

5.3 Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
p4	1	4
P5	5	2

The processes are assumed to have arrived in the order PI, P2, *P3,* P4, P5, all at time 0.

a. Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a non preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of the scheduling algorithms in part a?

d. Which of the schedules in part a results in the minimal average waiting time (over all processes)?

5.4 What advantage is there in having different time-quantum sizes on different levels of a multilevel queueing system?

5.5 Explain the differences in the degree to which the following scheduling algorithms discriminate in favor of short processes:

a. FCFS

b. RR

c. Multilevel feedback queues

6. PROCESS SYNCHRONIZATION

6.1 Background

Consider the shared-memory solution to the bounded-buffer problem that we presented in lession3 of unit1.Our solution allows at most BUFFER-SIZE - 1 items in the buffer at the same time. Suppose that we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable counter, initialized to 0.counter is incremented every time we add a new item to the buffer, and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```
while (1)
{
/* produce an item in nextproduced */
while (counter == BUFFER-SIZE)
; /* do nothing */
buffer [in] = nextproduced;
in = (in + I) % BUFFER-SIZE;
counter++;
}
```

The code for the consumer process can be modified as follows:

```
while (1) {
while (counter == 0)
; /* do nothing */
nextconsumed = buffer [out] ;
out = (out + 1) % BUFFER-SIZE;
counter--;
/* consume the item in nextconsumed *
```

The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution where the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high level statement is preserved). One such interleaving is

T0: *producer* execute *registerl* = counter *{registerl = 5)*

TI: *producer* execute *register1 = registerl* + 1 *{registerl = 6)*

T2: *consumer* execute *register2* = counter *{register2 = 5)*

T3: *consumer* execute *register2 = register2 - 1 {register2 = 4)*

T4: *producer* execute counter = *registerl {counter = 6)*

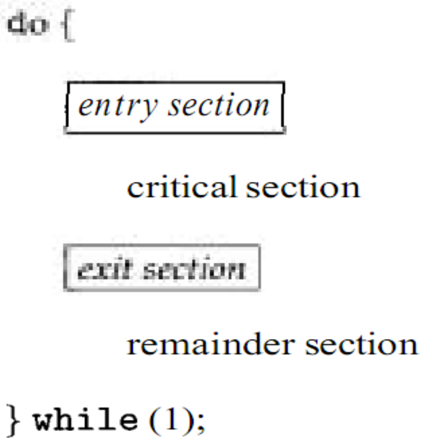
T5: *consumer* execute counter = *register2 {counter = 4)*

Notice that we have arrived at the incorrect state "counter == 4" recording that there are four full buffers, when, in fact, there are five full buffers. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state" counter == 6".

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require some form of synchronization of the processes. A major portion of this chapter is concerned with the issue of process synchronization and coordination.

6.2 The Critical-Section Problem

Consider a system consisting of *n* processes *{Po,P1, ..., P,-1)*. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is *mutually exclusive* in time. The *critical-section* problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the,, remainder section.



```
do {

    entry section

        critical section

    exit section

        remainder section

} while (1);
```

Figure 6.1 General structure of a typical process Pi.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual Exclusion: If process ***Pi*** is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded Waiting: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

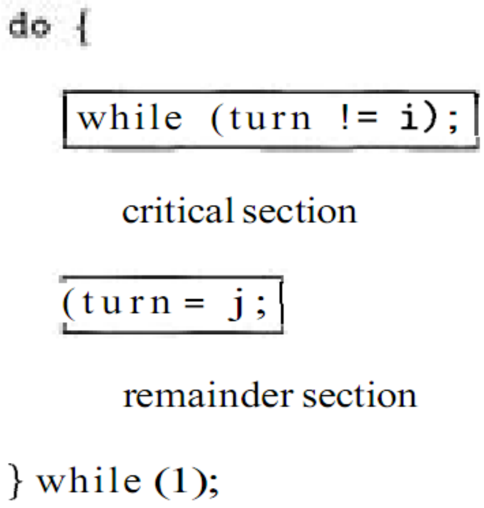
When presenting an algorithm, we define only the variables used for synchronization purposes, and describe only a typical process Pi whose general structure is shown in the above Figure 2.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

6.2.1Two-Process Solutions

In this section, we restrict our attention to algorithms that are applicable to only two processes at a time. The processes are numbered ***Po*** and ***PI.*** For convenience, we use ***Pi*** and ***pj*** to denote the processes.

6.2.1.1Algorithm 1

Our first approach is to let the processes share a common integer variable turn initialized to 0 (or 1). If turn == i, then process ***Pi*** is allowed to execute in its critical section. The structure of process ***Pi*** is shown in the Figure 6.2



```
do {

    while (turn != i);

        critical section

    (turn = j;

        remainder section

} while (1);
```

Figure 6.2 The structure of process Pi in algorithm 1.

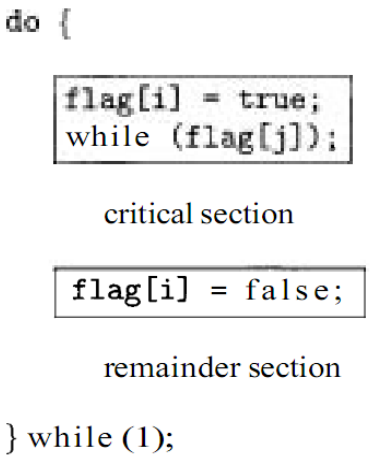
This solution ensures that only one process at a time can be in its critical section. However, it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical section. For example, if turn == 0 and ***PI*** is ready to enter its critical section, ***PI*** cannot do so, even though ***Po*** may be in its remainder section.

6.2.1.2 Algorithm 2

The problem with algorithm 1 is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter its critical section. To remedy this problem, we can replace the variable turn with the following array:

Boolean f lag[2] ;

The elements of the array are initialized to false. If f lag[i] is true, this value indicates that ***Pi*** is ready to enter the critical section. The structure of process ***Pi*** is shown in the Figure 6.3**.**



```
do {

    flag[i] = true;
    while (flag[j]);

        critical section

    flag[i] = false;

        remainder section

} while (1);
```

Figure 6.3 The structure of process Pi in algorithm 2.

In this algorithm, process ***Pi*** first sets f lag[i] to be true, signaling that it is ready to enter its critical section. Then, ***Pi*** checks to verify that process ***Pj*** is not also ready to enter its critical section. If ***Pj*** were ready, then **Pi** would wait

until ***Pj*** had indicated that it no longer needed to be in the critical section (that is, until flag [j I was false).At this point, ***Pi*** would enter the critical section. On exiting the critical section, Pi would set flag [i] to be false, allowing the other process (if it is waiting) to enter its critical section.

In this solution, the mutual-exclusion requirement is satisfied. Unfortunately, the progress requirement is not met. To illustrate this problem, we consider the following execution sequence:

To: Po sets flag [OI = true

TI: P1setsflagCl1 = true

Now Po and PI are looping forever in their respective while statements. This algorithm is crucially dependent on the exact timing of the two processes.

Note that switching the order of the instructions for setting flag [i], and testing the value of a flag [j] , will not solve our problem. Rather, we will have a situation where it is possible for both processes to be in the critical section at the same time, violating the mutual-exclusion requirement.

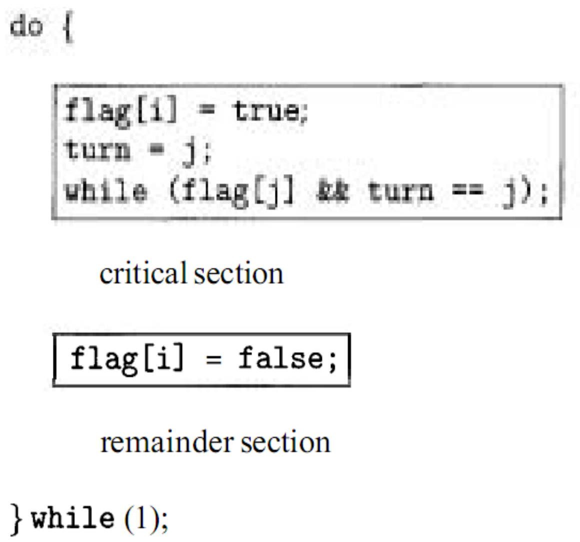
6.2.1.3 Algorithm 3

By combining the key ideas of algorithm 1 and algorithm 2, we obtain a correct solution to the critical-section problem, where all three requirements are met. The processes share two variables:

boolean flag [21;

int turn;

Initially f l a g [O] = f l a g [I] = false, and the value of turn is immaterial (but is either *0* or 1). The structure of process Pi is shown in Figure 6.4.



```
do {

    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (1);
```

Figure 6.4 The structure of process Pi in algorithm 3.

To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur, but will be overwritten immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved,

2. The progress requirement is satisfied,

3. The bounded-waiting requirement is met.

To prove property 1, we note that each Pi, enters its critical section only if either flag [j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then f lag[0]== flag[1]==true. These two observations imply that Po and PI could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1, but cannot be both. Hence, one of the processes-say Pj-must have successfully executed the while statement, Whereas Pi had to execute at least one additional statement ("turn == j").However, since, at that time, flag == true, and turn == j, and this condition will persist as long as Pi is in its critical section, the result follows: Mutual exclusion is preserved.

6.2.2 Multiple-Process Solutions

We have seen that algorithm 3 solves the critical-section problem for two processes. Now let us develop an algorithm for solving the critical-section problem for n processes. This algorithm is known as the bakery algorithm, and it is based on a scheduling algorithm commonly used in bakeries, ice-cream stores, deli counters, motor-vehicle registries, and other locations where order must be made out of chaos.

On entering the store, each customer receives a number. The customer with the lowest number is served next. Unfortunately, the bakery algorithm cannot guarantee that two processes (customers) do not receive the same number. In the case of a tie, the process with the lowest name is served first. That is, if Pi and Pj receive the same number and if i < j, then Pi is served first. Since process names are unique and totally ordered, our algorithm is completely deterministic,

The common data structures are

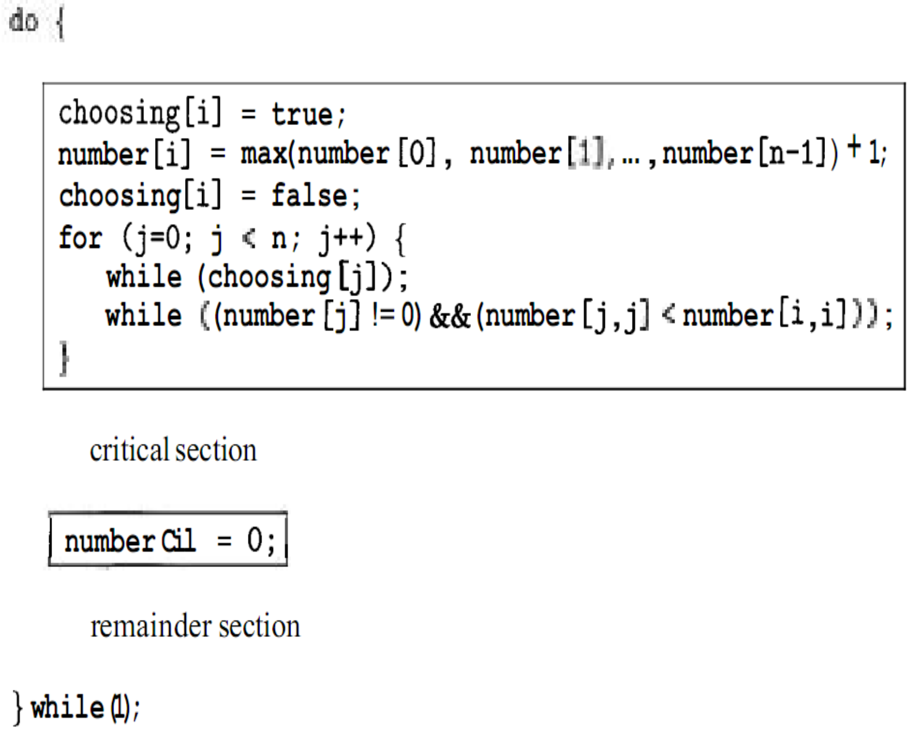
boolean choosing [n] ;

int number [n] ;

Initially, these data structures are initialized to false and 0, respectively. For convenience, we define the following notation:

(a,b) < (c,d) if a < c or if a == c and b < d.

a max(ao, ..., a,-l) is a number, k, such that k 2 ai for i = 0, ..., n - 1. The structure of process Pi, used in the bakery algorithm, is shown in Figure 6.5. To prove that the bakery algorithm is correct, we need first to show that, if Pi is in its critical section and Pk (k != i) has already chosen its number k != 0, then (number [i] , i) < (number [k] , k).



```
do {

    choosing[i] = true;
    number[i] = max(number [0], number[1],...,number[n-1]) + 1;
    choosing[i] = false;
    for (j=0; j < n; j++) {
       while (choosing[j]);
       while ((number [j] != 0) && (number [j,j] < number[i,i]));
    }

       critical section

    number Cil = 0;

       remainder section

} while (1);
```

6.5 The structure of process Pi in the bakery algorithm.

Given this result, it is now simple to show that mutual exclusion is observed. Indeed, consider Pi in its critical section and Pk trying to enter the Pk critical section. When process Pk executes the second while statement for j == i, it finds that

Number[i] != 0

(number [i] , i) < (number [k] , k).

Thus, it continues looping in the while statement until Pi leaves the Pi critical section.

6.3 Synchronization Hardware

As with other aspects of software, hardware features can make the programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems, and show how they can be used effectively in solving the critical-section problem.

The critical-section problem could be solved simply in a uni-processor environment if we could forbid interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. Unfortunately, this solution is not feasible in a multiprocessor environment.

Many machines therefore provide special hardware instructions that allow us either to test and modify the content of a word, or to swap the contents of two words, atomically-that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, let us abstract the main concepts behind these types of instructions. The TestAndSet instruction can be defined as shown in Figure 6.6. The important characteristic is that this instruction is executed atomically. Thus, if two TestAndSet instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

```
boolean TestAndSet(boo1ean &target) {
        boolean rv = target;
        target = true;
       return rv;
```

Figure 6.6 The definition of the TestAndSet instruction.

If the machine supports the TestAndSet instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process Pi is shown in Figure 6.7.

```
Do{
while (TestAndSet (lock)) ;
          critical section
     lock = false;
           remainder section
     } while (1);
```

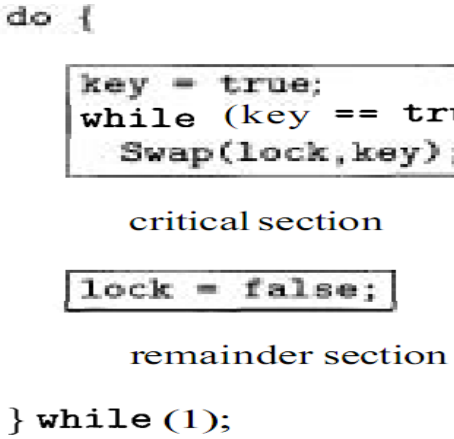
Figure 6.7 Mutual-exclusion implementation with TestAndSet.

The Swap instruction, defined as shown in Figure 6.8, operates on the contents of two words; like the TestAndSet instruction, it is executed atomically.

```
void  Swap(boo1ean &a, boolean  &b) {
            boolean  temp  =  a;
            a  =  b;
             b  =  temp;
```

Figure 6.8 The definition of the Swap instruction.

If the machine supports the Swap instruction, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized to false. In addition, each process also has a local Boolean variable key. The structure of process Pi is shown in Figure 6.9.



```
do {
     key = true;
     while (key == true)
       Swap(lock,key);

        critical section

     lock = false;

        remainder section

} while (1);
```

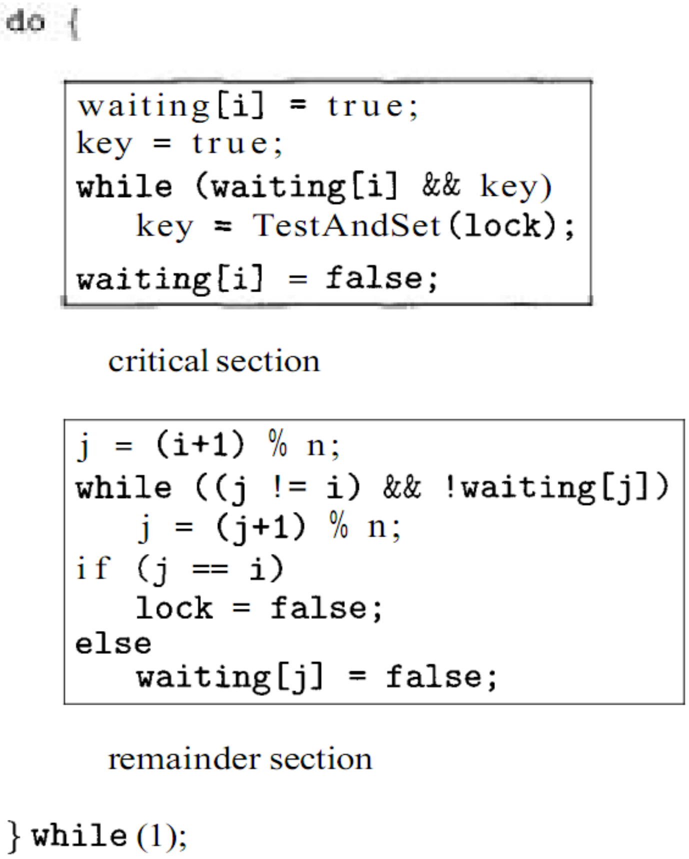
Figure 6.9 Mutual-exclusion implementation with the Swap instruction.

These algorithms do not satisfy the bounded-waiting requirement. We present an algorithm that uses the TestAndSet instruction in Figure 6.10. This algorithm satisfies all the critical-section requirements. The common data structures are

boolean waiting[n] ;

boolean lock;

These data structures are initialized to false. To prove that the mutual-exclusion requirement is met, we note that process Pi can enter its critical section only if either waitingCi1 == false or key == false. The value of key can become false only if the TestAndSet is executed. The first process to execute the TestAndSet will find key == false; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting [i] is set to false, maintaining the mutual- exclusion requirement.



```
do {

    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
       key = TestAndSet(lock);
    waiting[i] = false;

       critical section

    j = (i+1) % n;
    while ((j != i) && !waiting[j])
       j = (j+1) % n;
    if (j == i)
       lock = false;
    else
       waiting[j] = false;

       remainder section

} while (1);
```

Figure 6.10 Bounded-waiting mutual exclusion with TestAndSet.

To prove the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false, or sets waiting [j] to false. Both allow a process that is waiting to enter its critical section to proceed.

To prove the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n - 1,O, ..., i - 1). It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns.

6.4 Semaphores

The solutions to the critical-section problem presented in the previous Section are not easy to generalize to more complex problems. To overcome this difficulty, we can use a synchronization tool called a semaphore. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait and signal. These operations were originally termed P and V The classical definition of wait in pseudocode is

```
wait(S) {
while (S <= 0)
```

```
    ; //  no-op
    S--;
```

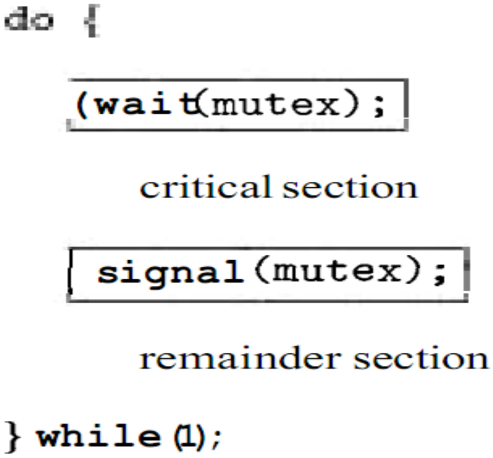
The classical definitions of signal in pseudocode is

```
signal(S){
S++
}
```

Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the wait (S), the testing of the integer value of S (S <= O), and its possible modification (S--), must also be executed without interruption.

6.4.1 Usage

We can use semaphores to deal with the n-process critical-section problem. The n processes share a semaphore, mutex (standing for mutual exclusion), initialized to 1. Each process Pi is organized as shown in Figure 6.11.



```
do {
     (wait(mutex);
          critical section
     signal(mutex);
          remainder section
} while (1);
```

Figure 6.11 Mutual-exclusion implementation with semaphores.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: PI with a statement S1 and P2 with a statement S2. Suppose that we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements

```
s1;
  signal  (synch) ;
```

in process PI, and the statements

```
wait (synch) ;
  s2;
```

in process P2. Because synch is initialized to 0, P2 will execute S2 only after PI has invoked signal (synch), which is after S1.

6.4.2 Implementation

The main disadvantage of the mutual-exclusion solutions of the previous Section, and of the semaphore definition given here, is that they all require busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** (because the process "spins" while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful.

To overcome the need for busy waiting, we can modify the definition of the wait and signal semaphore operations. When a process executes the wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation. The process is restarted by a wakeup operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as a "C" struct:

```
typedef struct  {
            int value  ;
            struct process *L;
             } semaphore;
```

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore, it is added to the list of processes. A signal operation removes one process from the list of waiting processes and awakens that process.

The wait semaphore operation can now be defined as

```
void wait(semaphore  S) {
         S.value--;
       if  (S.value <  0) {
       add this process to S  .  L;
```

```
          block( ) ;
                }
          }
```

The signal semaphore operation can now be defined as

```
     void signal(semaphore S) {
          S.value++;
          if (S.value <= 0) {
          remove a process P from S . L ;
          wakeup (P ) ;
                    }
                }
```

The block operation suspends the process that invokes it. The wakeup(P1 operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

The critical aspect of semaphores is that they are executed atomically. We must guarantee that no two processes can execute wait and signal operations on the same semaphore at the same time. This situation is a critical-section problem, and can be solved in either of two ways.

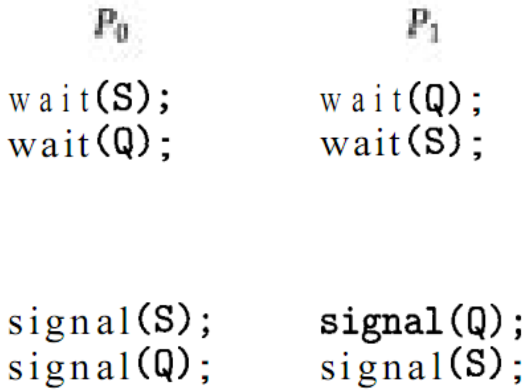
In a uniprocessor environment (that is, where only one CPU exists), we can simply inhibit interrupts during the time the wait and signal operations are executing.

In a multiprocessor environment, inhibiting interrupts does not work. Instructions from different processes (running on different processors) may be interleaved in some arbitrary way.

6.4.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal operation. When such a state is reached, these processes are said to be deadlocked**.**

To illustrate this, we consider a system consisting of two processes, Po and PI, each accessing two semaphores, S and Q, set to the value 1:



```
P0              P1

wait(S);        wait(Q);
wait(Q);        wait(S);



signal(S);      signal(Q);
signal(Q);      signal(S);
```

6.4.4 Binary Semaphores

Suppose that Po executes wait (S) , and then P1 executes wait (Q). When Po executes wait (41, it must wait until PI executes signal (4). Similarly, when P1 executes wait (S), it must wait until Po executes signal (S) . Since these signal operations cannot be executed, Po and P1 are deadlocked.

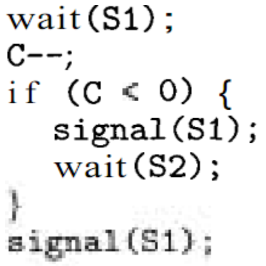
The semaphore construct described in the previous sections is commonly known as a counting semaphore, since its integer value can range over an unrestricted domain. A binary semaphore is a semaphore with an integer value that can range only between 0 and 1. A binary semaphore can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture. We will now show how a counting semaphore can be implemented using binary semaphores.

Let S be a counting semaphore. To implement it in terms of binary semaphores we need the following data structures:

```
binary-semaphore S1, S2;
  int C;
```

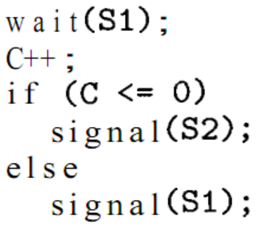
Initially S1 = 1, S2 = 0, and the value of integer C is set to the initial value of the counting semaphore S.

The wait operation on the counting semaphore S can be implemented as follows:



```
wait(S1);
C--;
if (C < 0) {
   signal(S1);
   wait(S2);
}
signal(S1);
```

The signal operation on the counting semaphore S can be implemented as follows:



```
wait(S1);
C++;
if (C <= 0)
   signal(S2);
else
   signal(S1);
```

6.5 Classic Problems of Synchronization

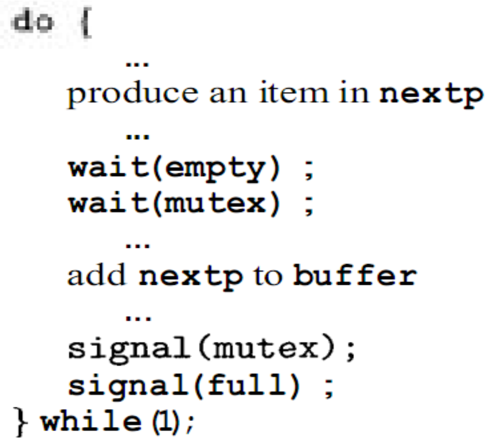
In this section, we present a number of different synchronization problems as examples for a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. Semaphores are used for synchronization in our solutions.

6.5.1 The Bounded-Buffer Problem

The bounded-buffer problem was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. We assume that the pool consists of n buffers, each capable

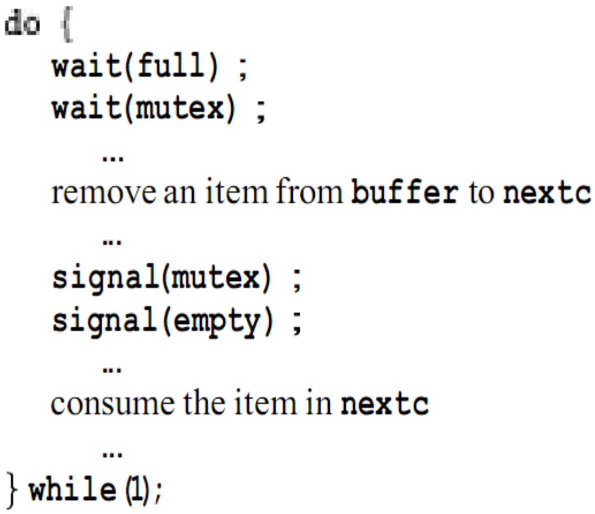
of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process is shown in Figure 6.12; the code for the consumer process is shown in Figure 6.13. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer, or as the consumer producing empty buffers for the producer.



```
do {
      ...
    produce an item in nextp
      ...
    wait(empty) ;
    wait(mutex) ;
      ...
    add nextp to buffer
      ...
    signal(mutex);
    signal(full) ;
} while (1);
```

Figure 6.12 The structure of the producer process.



```
do {
   wait(full) ;
   wait(mutex) ;
      ...
   remove an item from buffer to nextc
      ...
   signal(mutex) ;
   signal(empty) ;
      ...
   consume the item in nextc
      ...
} while (1);
```

Figure 6.13 The structure of the consumer process.

6.5.2 The Readers- Writers Problem

A data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (that is, to read and write) the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as readers, and to the rest as writers. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the shared object simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to as the readers-writers problem.

The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we present a solution to the first readers-writers problem.

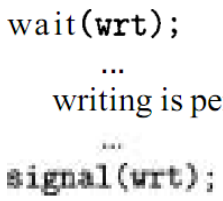
In the solution to the first readers-writers problem, the reader processes share the following data structures:

semaphore mutex, wrt;

```
int  readcount;
```

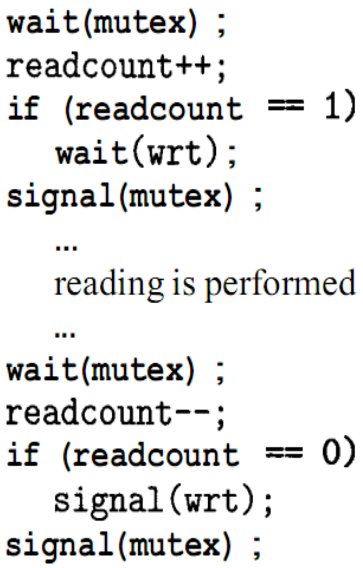
The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both the reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 6.14; the code for a reader process is shown in Figure 6.15. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n - 1 readers are queued on mutex. Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.



```
wait(wrt);
      ...
   writing is performed
      ...
signal(wrt);
```

Figure 6.14 The structure of a writer process.



```
wait(mutex) ;
readcount++;
if (readcount == 1)
   wait(wrt);
signal(mutex) ;
   ...
   reading is performed
   ...
wait(mutex) ;
readcount--;
if (readcount == 0)
   signal(wrt);
signal(mutex) ;
```

Figure 6.15 The structure of a reader process.

6.5.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 6.16). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

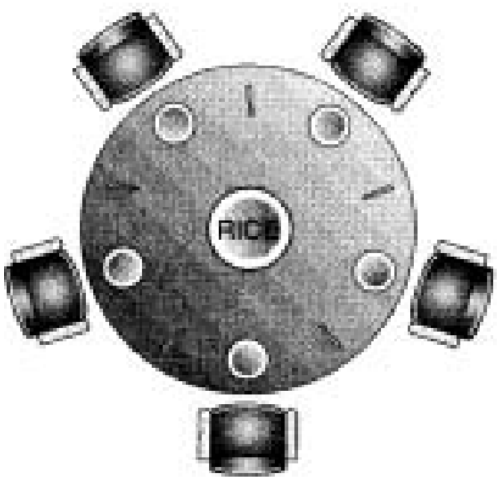


Figure 6.16 The situation of the dining philosophers.

One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5] ;

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown in Figure 6.17.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock. Suppose that all five philosophers become hungry simultaneously, and each grabs her left chopstick. All the elements of chopstick will now be

equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever. we present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.

- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).

- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

6.6 Critical Regions

Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place, and these sequences do not always occur.

Unfortunately, such timing errors can still occur with the use of semaphores. To illustrate how, let us review the solution to the critical-section problem using semaphores. All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait (mutex) before entering the critical section, and signal (mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

Let us examine the various difficulties that may result. Note that these difficulties will arise if even a single process is not well behaved. This situation may be the result of an honest programming error or of an uncooperative programmer.

- Suppose that a process interchanges the order in which the wait and signal operations on the semaphore mutex are executed, resulting in the following execution:

```
                    signal  (muted  ;
                               ...
                         critical section
                              . . .
                         wait (mutex)  ;
```

In this situation, several processes may be executing in their critical section simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces signal (mutex) with wait (mutex) . That is, it executes

```
                    Wait(mutex)
                              . . .
                        critical section
```

```
. . .
wait (mutex) ;
```

In this case, a deadlock will occur.

- Suppose that a process omits the wait (mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when semaphores are used incorrectly to solve the critical-section problem.

To deal with the type of errors we have outlined, a number of high-level language constructs have been introduced. In this section, we describe one fundamental high-level synchronization construct the **critical region** (sometimes referred to as conditional critical region). In the previous Section , we present another fundamental synchronization construct the **monitor**. In our presentation of these two constructs, we assume that a process consists of some local data, and a sequential program that can operate on the data. The local data can be accessed by only the sequential program that is encapsulated within the same process. That is, one process cannot directly access the local data of another process. Processes can, however, share global data.

The critical-region high-level language synchronization construct requires that a variable v of type T, which is to be shared among many processes, be declared as

```
v: shared T;
```

The variable v can be accessed only inside a region statement of the following form: region v when B do S; This construct means that, while statement S is being executed, no other process can access the variable v. The expression B is a Boolean expression that governs the access to the critical region. When a process tries to enter the critical-section region, the Boolean expression B is evaluated. If the expression is true, statement S is executed. If it is false, the process relinquishes the mutual exclusion and is delayed until B becomes true and no other process is in the region associated with v. Thus, if the two statements,

```
region v when (true) S1;
region v when (true) S2;
```

are executed concurrently in distinct sequential processes, the result will be equivalent to the sequential execution "S1 followed by S2" or "S2 followed by Sl."

The critical-region construct guards against certain simple errors associated with the semaphore solution to the critical-section problem that may be made by a programmer.

The critical-region construct can be effectively used to solve certain general synchronization problems. To illustrate, let us code the bounded-buffer scheme. The buffer space and its pointers are encapsulated in

```
struct buffer {
item pool [n] ;
int count, in, out;
};
```

The producer process inserts a new item nextp into the shared buffer by executing region buffer

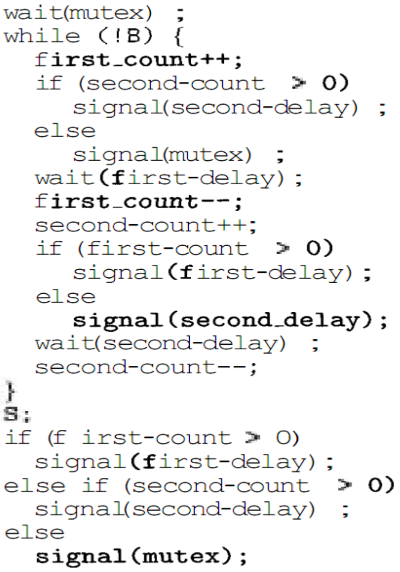
```
when  (count  <  n) {
pool  [in] =  nextp;
in =  (in+l) %  n;
count++  ;
         }
```

The consumer process removes an item from the shared buffer and puts it in nextc by executing region buffer

```
when  (count  >  0)  {
nextc =  pool[out] ;
out =  (out+l) %  n;
count--  ;
}
```

Let us illustrate how the conditional critical region could be implemented by a compiler. With each shared variable, the following variables are associated: semaphore mutex, first-delay, second-delay; int first-count, second-count; The semaphore mutex is initialized to 1; the semaphores first delay and second-delay are initialized to 0. The integers first -count and second count are initialized to 0.

Mutually exclusive access to the critical section is provided by mutex. If a process cannot enter the critical section because the Boolean condition B is false, it initially waits on the first-delay semaphore. A process waiting on the first -delay semaphore is eventually moved to the second-delay semaphore before it is allowed to reevaluate its Boolean condition B. We keep track of the number of processes waiting on first -delay and second-delay, with first-count and second-count respectively.



```
wait(mutex)  ;
while (!B) {
  first_count++;
  if (second-count  > 0)
     signal(second-delay) ;
  else
     signal(mutex)  ;
  wait(first-delay);
  first_count--;
  second-count++;
  if (first-count  > 0)
     signal(first-delay);
  else
     signal(second_delay);
  wait(second-delay)  ;
  second-count--;
}
S;
if (f irst-count > O)
  signal(first-delay);
else if (second-count  > 0)
  signal(second-delay)  ;
else
  signal(mutex);
```

Figure 6.18 Implementation of the conditional-region construct.

When a process leaves the critical section, it may have changed the value of some Boolean condition B that prevented another process from entering the critical section. Accordingly, we must trace through the queue of processes waiting on first-delay and second-delay (in that order) allowing each process to test its Boolean condition. When a process tests its Boolean condition (during this trace), it may discover that the latter now evaluates to the value true. In this case, the process enters its critical section. Otherwise, the

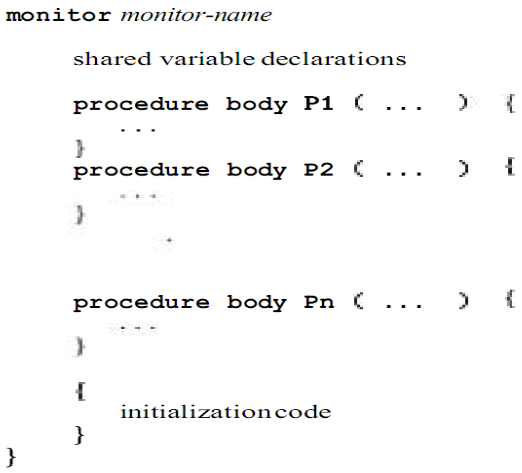
process must wait again on the first-delay and second-delay semaphores, as described previously. Accordingly, for a shared variable x, the statement

region x when (B) S;

can be implemented as shown in Figure 6.18. Note that this implementation requires the reevaluation of the expression B for any waiting processes every time a process leaves the critical region. If several processes are delayed waiting for their respective Boolean expressions to become true, this reevaluation overhead may result in inefficient code.

6.7 Monitors

Another high-level synchronization construct is the monitor type. A monitor is characterized by a set of programmer-defined operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. The syntax of a monitor is shown in Figure 6.19.



```
monitor monitor-name

      shared variable declarations

      procedure body P1 ( ...  ) {
          . . .
      }
      procedure body P2 ( ...  ) {
          . . .
      }
              .

      procedure body Pn ( ...  ) {
          . . .
      }
      {
          initialization code
      }
}
```

Figure 6.19 Syntax of a monitor.

The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 6.20). However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write her own tailor-made synchronization scheme can define one or more variables of type condition:

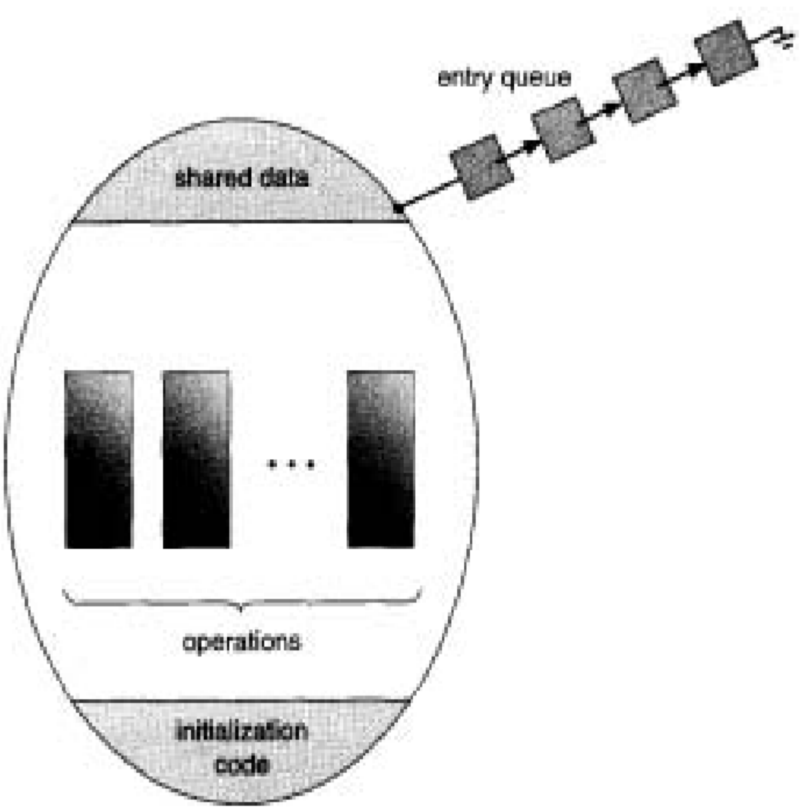


Figure 6.20 Schematic view of a monitor.

condition x, y;

The only operations that can be invoked on a condition variable are wait and signal. The operation

x.wait()

means that the process invoking this operation is suspended until another process invokes

x,signal()

The x . signal (1 operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect; that is, the state of x is as though the operation were never executed (Figure 6.21). Contrast this operation with the signal operation associated with semaphores, which always affects the state of the semaphore.

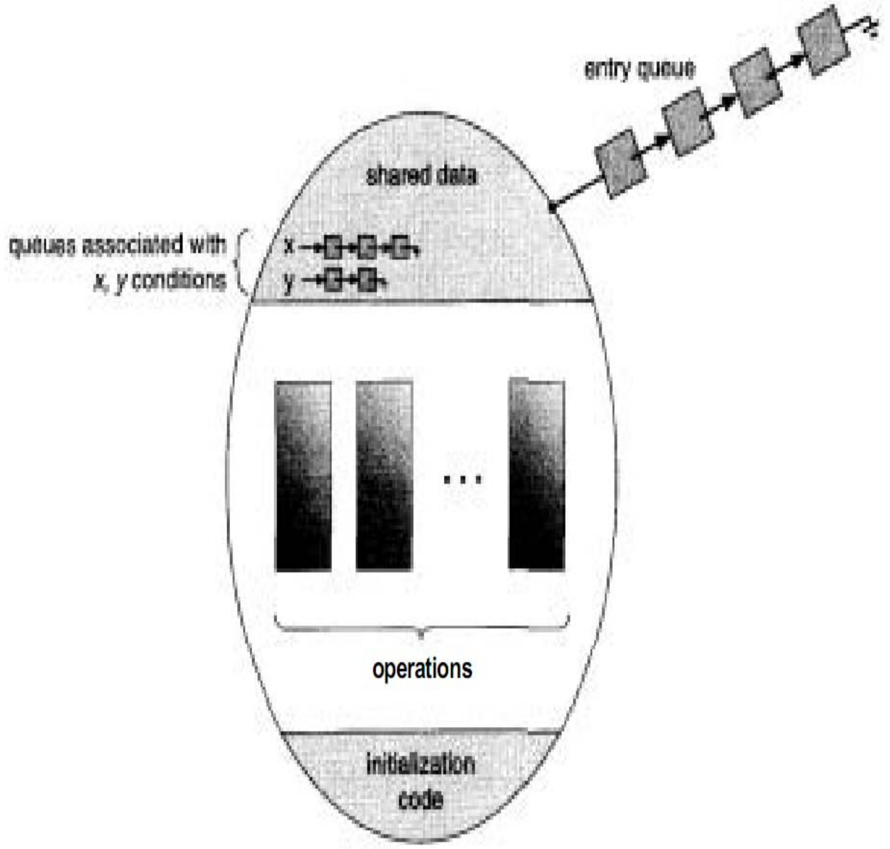


Figure 6.21 Monitor with condition variables.

Now suppose that, when the x . signal () operation is invoked by a process P, there is a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q will be active simultaneously within the monitor. **Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:**

1. P either waits until Q leaves the monitor, or waits for another condition.

2. Q either waits until P leaves the monitor, or waits for another condition.

Let us illustrate these concepts by presenting a deadlock-free solution to the dining-philosophers problem. Recall that a philosopher is allowed to pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which a philosopher may be. For this purpose, we introduce the following data structure:

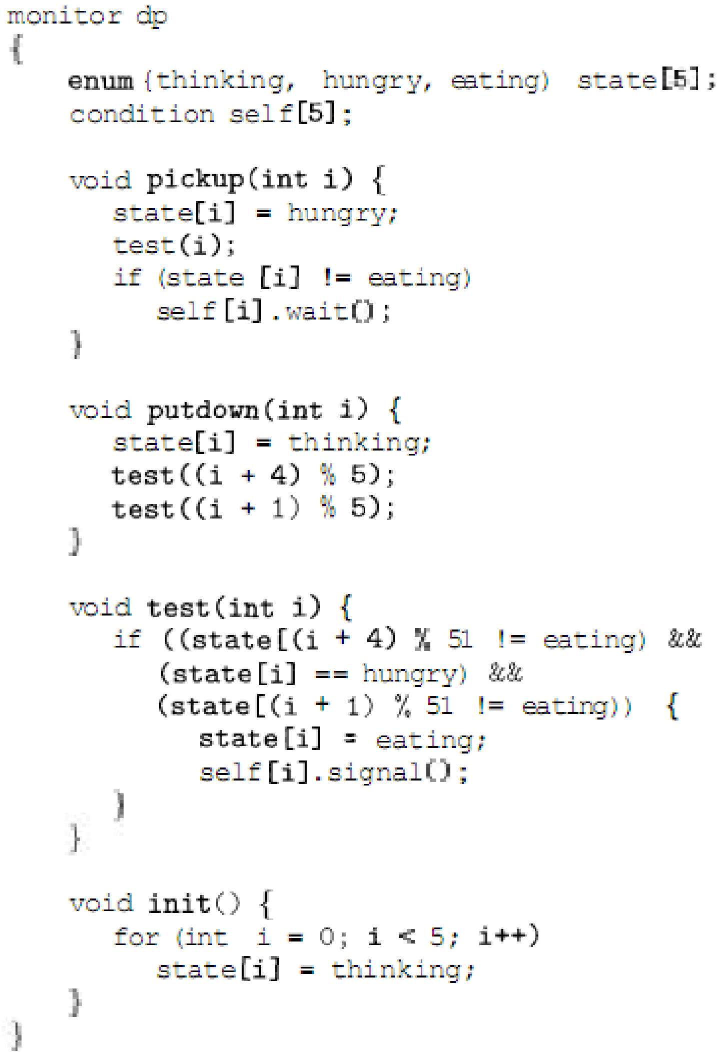
```
enum {thinking, hungry, eating} state[5] ;
```

Philosopher i can set the variable state[i] = eating only if her two neighbors are not eating: (state [(i+4) % 5] ! = eating) and (state[(i+l) % 5] != eating). We also need to declare

condition self C51 ;

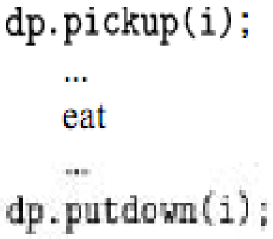
where philosopher i can delay herself when she is hungry, but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosopher problem. The distribution of the chopsticks is controlled by the monitor dp, whose definition is shown in Figure 6.22. Each philosopher, before starting to eat, must invoke the operation pickup. This may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the put- down operation, and may start to think. Thus, philosopher i must invoke the operations pickup and putdown in the following sequence:



```
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];

    void pickup(int i) {
       state[i] = hungry;
       test(i);
       if (state [i] != eating)
          self[i].wait();
    }

    void putdown(int i) {
       state[i] = thinking;
       test((i + 4) % 5);
       test((i + 1) % 5);
    }

    void test(int i) {
       if ((state[(i + 4) % 51 != eating) &&
          (state[i] == hungry) &&
          (state[(i + 1) % 51 != eating)) {
             state[i] = eating;
             self[i].signal();
       }
    }

    void init() {
       for (int i = 0; i < 5; i++)
          state[i] = thinking;
    }
}
```

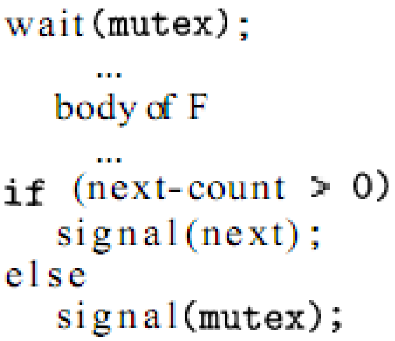
Figure 6.22 A monitor solution to the dining-philosopher problem.



```
dp.pickup(i);
    ...
    eat
    ...
dp.putdown(i);
```

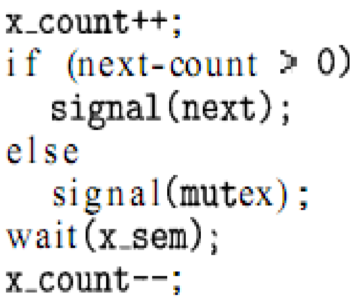
It is easy to show that this solution ensures that no two neighbors are eating simultaneously, and that no deadlocks will occur. We shall now consider a possible implementation of the monitor mechanism using

semaphores. For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait (mutex) before entering the monitor, and must execute signal (mutex) after leaving the monitor. Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0, on which the signaling processes may suspend themselves. An integer variable next-count will also be provided to count the number of processes suspended on next. Thus, each external procedure F will be replaced by



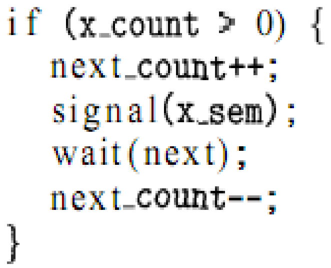
```
wait(mutex);
    ...
   body of F
    ...
if (next-count > 0)
   signal(next);
else
   signal(mutex);
```

Mutual exclusion within a monitor is ensured. We can now describe how condition variables are implemented. For each condition x, we introduce a semaphore x-sem and an integer variable x-count, both initialized to 0. The operation x. wait can now be implemented as



```
x_count++;
if (next-count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

The operation x . signal() can be implemented as



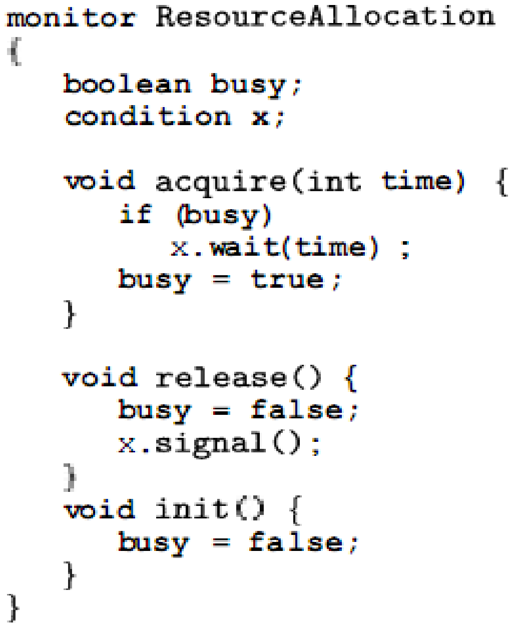
```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition x, and an x.signa1 operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use a FCFS ordering, so that the process waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. For this purpose, the conditional-wait construct can be used; it has the form

x.wait(c)

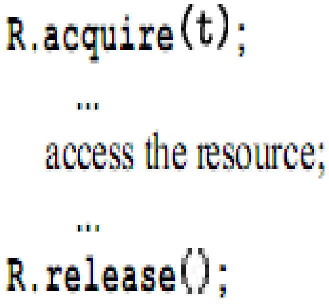
where c is an integer expression that is evaluated when the wait operation is executed. The value of c, which is called a priority number, is then stored with the name of the process that is suspended. When x. signal is executed, the process with the smallest associated priority number is resumed next.

To illustrate this new mechanism, we consider the monitor shown in Figure 6.23, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of its resources, specifies the maximum time it plans to use the resource. The monitor allocates the resource to that process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:



```
monitor ResourceAllocation
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time) ;
      busy = true;
   }

   void release() {
      busy = false;
      x.signal();
   }
   void init() {
      busy = false;
   }
}
```

Figure 6.23 A monitor to allocate a single resource.



```
R.acquire(t);
   ...
  access the resource;
   ...
R.release();
```

where R is an instance of type Resource Allocation.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequences will be observed. In particular,

A process might access the resource without first gaining access permission to that resource.

A process might never release the resource once it has been granted access to that resource..

A process might attempt to release a resource that it never requested.

A process might request the same resource twice (without first releasing that resource).

6.8 Synchronization in Solaris 2

The Solaris 2 operating system was designed to provide real-time computing, be multithreaded, and support multiple processors. To control access to critical sections, Solaris 2 provides adaptive mutexes, condition variables, semaphores, reader-writer locks, and turnstiles. Solaris 2 implements semaphores and condition variables.

An adaptive mutex protects access to every critical data item. On a multiprocessor system, an adaptive mutex starts as a standard semaphore implemented as a spinlock. If the data are locked and therefore already in use, the adaptive mutex does one of two things. If the lock is held by a thread that is currently running on another CPU, the thread spins while waiting for the lock to become available because the thread holding the lock is likely to be done soon. If the thread holding the lock is not currently in run state, the thread blocks, going to sleep until it is awakened by the lock being released. It is put to sleep so that it will avoid spinning when the lock will not be freed reasonably quickly. A lock held by a sleeping thread is likely to be in this category. On a uniprocessor system, the thread holding the lock is never running if the lock is being tested by another thread, because only one thread can run at a time. Therefore, on a uniprocessor system, threads always sleep rather than spin if they encounter a lock. We use the adaptive-mutex method to protect only those data that are accessed by short code segments. That is, a mutex is used if a lock will be held for less than a few hundred instructions. If the code segment is longer than that, spin waiting will be exceedingly inefficient. For longer code segments, condition variables and semaphores are used. If the desired lock is already held, the thread issues a wait and sleeps. When a thread frees the lock, it issues a signal to the next sleeping thread in the queue. The extra cost of putting a thread to sleep and waking it, and of the associated context switches, is less than the cost of wasting several hundred instructions waiting in a spinlock.

The readers-writers locks are used to protect data that are accessed frequently, but usually only in a read-only manner. In these circumstances, readers-writers locks are more efficient than semaphores, because multiple threads may be reading data concurrently, whereas semaphores would always serialize access to the data. Readers-writers locks are relatively expensive to implement, so again they are used on only long sections of code.

Solaris 2 uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or a reader-writer lock. A turnstile is a queue structure containing threads blocked on a lock. For example, if one thread currently owns the lock for a synchronized object, all other threads trying to acquire the lock will block and enter the turnstile for that lock. When the lock is released, the kernel selects a thread from the turnstile as the next owner of the lock. Each synchronized object with at least one thread blocked on the a turnstile with each synchronized object, Solaris 2 gives each kernel thread its own turnstile. The turnstile for the first thread to block on a synchronized object becomes the turnstile for the object itself. Subsequent threads blocking on the lock will be added to this turnstile. When the initial thread ultimately releases the lock, it gains a new turnstile from a list of free turnstiles the kernel maintains.

6.9 Atomic Transactions

The mutual exclusion of critical sections ensures that the critical sections are executed atomically. That is, if two critical sections are executed concurrently, the result is equivalent to their sequential execution in some unknown order. Although this property is useful in many application domains, in many cases

we would like to make sure that a critical section forms a single logical unit of work that either is performed in its entirety or is not performed at all. An example is funds transfer, in which one account is debited and another is credited. Clearly, it is essential for data consistency that either both the credit and debit occur, or that neither occur.

The remainder of this section is related to the field of database systems. Databases are concerned with the storage and retrieval of data, and with the consistency of the data.

6.9.1 System Model

A collection of instructions (or operations) that performs a single logical function is called a transaction. A major issue in processing transactions is the preservation of atomicity despite the possibility of failures within the computer system. In this section, we describe various mechanisms for ensuring transaction atomicity. We do so by first considering an environment where only one transaction can be executing at a time. Then, we consider the case where multiple transactions are active simultaneously. A transaction is a program unit that accesses and possibly updates various data items that may reside on the disk within some files. From our point of view, a transaction is simply a sequence of **read** and **write** operations, terminated by either a **commit** operation or an abort operation. A commit operation signifies that the transaction has terminated its execution successfully, whereas an abort operation signifies that the transaction had to cease its normal execution due to some logical error. A terminated transaction that has completed its execution successfully is committed; otherwise, it is **aborted.** The effect of a committed transaction cannot be undone by abortion of the transaction.

A transaction may also cease its normal execution due to a system failure. In either case, since an aborted transaction may have already modified the various data that it has accessed, the state of these data may not be the same as it would be had the transaction executed atomically. So that the atomicity property is ensured, an aborted transaction must have no effect on the state of the data that it has already modified. Thus, the state of the data accessed by an aborted transaction must be restored to what it was just before the transaction started executing. We say that such a transaction has been rolled back. It is part of the responsibility of the system to ensure this property.

To determine how the system should ensure atomicity, we need first to identify the properties of devices used for storing the various data accessed by the transactions. Various types of storage media are distinguished by their relative speed, capacity, and resilience to failure.

- **Volatile Storage**: Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself and because it is possible to access directly any data item in volatile storage.
- **Nonvolatile Storage**: Information residing in nonvolatile storage usually survives system crashes. Examples of media for such storage are disks and magnetic tapes. Disks are more reliable than is main memory, but are less reliable than are magnetic tapes. Both disks and tapes, however, are subject to failure, which may result in loss of information. Currently, nonvolatile storage is slower than volatile storage by several orders of magnitude, because disk and tape devices are electromechanical and require physical motion to access data.
- **Stable Storage**: Information residing in stable storage is never lost (never should be taken with a grain of salt, since theoretically such absolutes cannot be guaranteed). To implement an approximation of such storage, we need to replicate information in several nonvolatile storage caches (usually disk) with independent failure modes, and to update the information in a controlled manner .

Here, we are concerned only with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

6.9.2 Log-Based Recovery

Transaction Name: The unique name of the transaction that performed the write operation

Data Item Name: The unique name of the data item written

Old Value: The value of the data item prior to the write operation

New Value: The value that the data item will have after the write

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.

Before a transaction Ti starts its execution, the record < Ti starts> is written to the log. During its execution, any write operation by Ti is preceded by the writing of the appropriate new record to the log. When Ti commits, the record < Ti commits> is written to the log.

Because the information in the log is used in reconstructing the state of the data items accessed by the various transactions, we cannot allow the actual update to a data item to take place before the corresponding log record is written out to stable storage. We therefore require that, prior to a write(X)operation being executed, the log records corresponding to X be written onto stable storage.

Using the log, the system can handle any failure that does not result in the loss of information on nonvolatile storage. The recovery algorithm uses two procedures:

undo(T_i), which restores the value of all data updated by transaction Ti to the old values

redo(T_i), which sets the value of all data updated by transaction Ti to the new values

The set of data updated by T_i and their respective old and new values can be found in the log.

The undo and redo operations must be idempotent (that is, multiple executions of an operation have the same result as does one execution) to guarantee correct behavior, even if a failure occurs during the recovery process.

If a transaction T_i aborts, then we can restore the state of the data that it has updated by simply executing undo(T_i). If a system failure occurs, we restore the state of all updated data by consulting the log to determine which transactions need to be redone and which need to be undone. This classification of transactions is accomplished as follows:

- Transaction T_i needs to be undone if the log contains the < T_i starts> record, but does not contain the < T_i commits> record.
- Transaction Ti needs to be redone if the log contains both the < T_i starts> and the < T_i commits> records.

6.9.3 Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to make these determinations. There are two major drawbacks to this approach:

1. The searching process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already actually updated the data that the log says they need to modify. Although redoing the data modifications will cause no harm (due to idempotency), it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce the concept of check-points. During execution, the system maintains the write-ahead log. In addition, the system periodically performs checkpoints that require the following sequence of actions to take place:

1. Output all log records currently residing in volatile storage (usually main memory) onto stable storage.
2. Output all modified data residing in volatile storage to the stable storage.
3. Output a log record <checkpoint> onto stable storage.

The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure. Consider a transaction T_i that committed prior to the checkpoint. The < T_i commits> record appears in the log before the <checkpoint> record. Any modifications made by T_i must have been written to stable storage either prior to the checkpoint, or as part of the checkpoint itself.

Thus, at recovery time, there is no need to perform a redo operation on T_i. This observation allows us to refine our previous recovery algorithm. After a failure has occurred, the recovery routine examines the log to determine the most recent transaction Ti that started executing before the most recent checkpoint took place. It finds such a transaction by searching the log backward to find the first <checkpoint> record, and then finding the subsequent < T_i start > record.

Once transaction Ti has been identified, the redo and undo operations need to be applied to only transaction Ti and all transactions Ti that started executing after transaction Ti. Let us denote these transactions by the set T. The remainder of the log can thus be ignored. The recovery operations that are required are as follows: For all transactions T_k in T such that the record < T_k commits, appears in the log, execute redo(T_k). For all transactions T_k in T that have no < T_k commits> record in the log, execute undo(T_k).

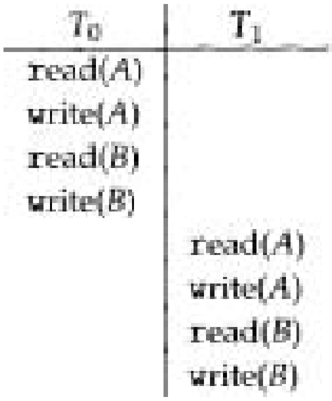
6.9.4 Concurrent Atomic Transactions

Because each transaction is atomic, the concurrent execution of transactions must be equivalent to the case where these transactions executed serially in some arbitrary order. This property called serializability, can be maintained by simply executing each transaction within a critical section. That is, all transactions share a common semaphore mutex, which is initialized to 1. When a transaction starts executing, its first action is to execute wait(mutex). After the transaction either commits or aborts, it executes signal(mutex).

Although this scheme ensures the atomicity of all concurrently executing transactions, it is nevertheless too restrictive. As we shall see, in many cases we can allow transactions to overlap their execution, while maintaining serializability. A number of different concurrency-control algorithms ensure serializability. These are described below.

6.9.4.1 Serializability

Consider a system with two data items A and B that are both read and written by two transactions T_o and T_I. Suppose that these transactions are executed atomically in the order T_o followed by T_I. This execution sequence, which is called a schedule, is represented in Figure 6.24. In schedule 1 of Figure 6.24, the sequence of instruction steps is in chronological order from top to bottom, with instructions of To appearing in the left column and instructions of T_I appearing in the right column.

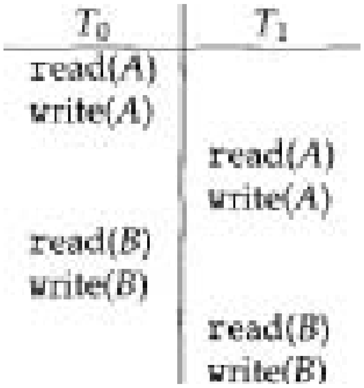


T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure 6.24 Schedule 1: A serial schedule in which To is followed by TI.

A schedule where each transaction is executed atomically is called a serial schedule. Each serial schedule consists of a sequence of instructions from various transactions where the instructions belonging to one single transaction appear together in that schedule. Thus, for a set of n transactions, there exist n! different valid serial schedules. Each serial schedule is correct, because it is equivalent to the atomic execution of the various participating transactions, in some arbitrary order. If we allow the two transactions to overlap their execution, then the resulting schedule is no longer serial.

A nonserial schedule does not necessarily imply that the resulting execution is incorrect (that is, is not equivalent to a serial schedule). To see that this is the case, we need to define the notion of conflicting operations. Consider a schedule S in which there are two consecutive operations O_i and O_j of transactions T_1 and Ti, respectively. We say that Oi and Oi conflict if they access the same data item and at least one of these operations is a write operation. To illustrate the concept of conflicting operations, we consider the nonserial schedule 2 of Figure 6.25. The write(A) operation of To conflicts with the read(A) operation of TI. However, the write(A) operation of T1 does not conflict with the read(B) operation of To, because the two operations access different data items.



T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Let Oi and Oj be consecutive operations of a schedule S. If Oi and Oj are operations of different transactions and Oi and Oj do not conflict, then we can swap the order of Oi and Oi to produce a new schedule S'. We expect S to be equivalent to S', as all operations appear in the same order in both schedules, except for Oi and Oi, whose order does not matter.

Let us illustrate the swapping idea by considering again schedule 2 of Figure 6.25. As the write(A) operation of T1 does not conflict with the read(B) operation of To, we can swap these operations to generate an equivalent schedule. Regardless of the initial system state, both schedules produce the same final system state. Continuing with this procedure of swapping nonconflicting operations, we get:

- Swap the read(B) operation of To with the read(A) operation of TI.
- Swap the write(B) operation of To with the writ e(A) operation of TI.

Swap the write(B) operation of To with the read(A) operation of TI.

The final result of these swaps is schedule 1 in Figure 2.24, which is a serial schedule. Thus, we have shown that schedule 2 is equivalent to a serial schedule. This result implies that, regardless of the initial system state, schedule 2 will produce the same final state as will some serial schedule. If a schedule S can be transformed into a serial schedule S' by a series of swaps of nonconflicting operations, we say that a schedule S is conflict serializable. Thus, schedule 2 is conflict serializable, because it can be transformed into

6.9.4.2 Locking Protocol

One way to ensure serializability is to associate with each data item a lock, and to require that each transaction follow a locking protocol that governs how locks are acquired and released. There are various modes in which a data item can be locked. In this section, we restrict our attention to two modes:

- **Shared:** If a transaction Ti has obtained a shared-mode lock (denoted by S) on data item Q, then Ti can read this item, but cannot write Q.
- **Exclusive:** If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on data item Q, then Ti can both read and write Q.

We require that every transaction request a lock in an appropriate mode on data item Q, depending on the type of operations it will perform on Q.

To access a data item Q, transaction Ti must first lock Q in the appropriate mode. If Q is not currently locked, then the lock is granted, and Ti can now access it. However, if the data item Q is currently locked by some other transaction, then Ti may have to wait. More specifically, suppose that Ti requests an exclusive lock on Q. In this case, Ti must wait until the lock on Q is released. If Ti requests a shared lock on Q, then Ti must wait if Q is locked in exclusive mode. Otherwise, it can obtain the lock and access Q. Notice that this scheme is quite similar to the readers-writers algorithm A transaction may unlock a data item that it had locked at an earlier point. it must, however, hold a lock on a data item as long as it accesses that item.

Moreover, it is not always desirable for a transaction to unlock a data item immediately after its last access of that data item, because serializability may not be ensured. One protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

- **Growing Phase**: A transaction may obtain locks, but may not release any lock.
- **Shrinking Phase:** A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued. The two-phase locking protocol ensures conflict serializability .is possible that, for a set of transactions, there are conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to improve performance over two-phase locking, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data.

6.9.4.3 Timestamp-Based Protocols

In the locking protocols described above, the order between every pair of conflicting transactions is determined at execution time by the first lock that they both request and that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a timestamp- ordering scheme.

With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS(Ti). This timestamp is assigned by the system before the transaction Ti starts execution. If a transaction Ti has been assigned timestamp TS(Ti), and later on a new transaction TJ enters the system, then TS(Ti) < TS(TJ). There are two simple methods for implementing this scheme:

- Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. This method will not work for transactions that occur on separate systems or for processors that do not share a clock.
- Use a logical counter as the timestamp; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. The counter is incremented after a new timestamp is assigned. the serial schedule 1.

The timestamps of the transactions determine the serializability order. Thus, if TS(Ti) < TS(TJ), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction TJ.

To implement this scheme, we associate with each data item Q two timestamp values:

W-timestamp(Q), which denotes the largest timestamp of any transaction that executed writ e(Q) successfully

R-timestamp(Q), which denotes the largest timestamp of any transaction that executed read(Q) successfully

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed. The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

Suppose that transaction Ti issues read(&):

- If TS(Ti) < W-timestamp(), then this state implies that Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.
- If TS(Ti) >=W-timestamp(Q), then the read operation is executed, and

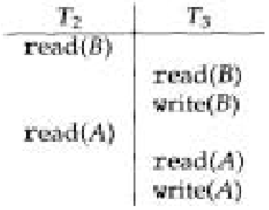
R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

Suppose that transaction Ti issues write(Q):

- If TS(Ti) < R-timestamp(Q), then this state implies that the value of Q that Ti is producing was needed previously and Ti assumed that this value would never be produced. Hence, the write operation is rejected, and Ti is rolled back.
- If TS(Ti) < W-timestamp(Q), then this state implies that Ti is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and Ti is rolled back.
- Otherwise, the write operation is executed.

A transaction Ti, that is rolled back by the concurrency-control scheme as a result of the issuing of either a read or write operation is assigned a new timestamp and is restarted.

To illustrate this protocol, we consider schedule 3 of Figure 2.26 with transactions T2 and T3. We assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3, TS(T2) < TS(T3), and the schedule is possible under the timestamp protocol.



T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Figure 6.26 Schedule 3: A schedule possible under the timestamp protocol.

This execution can also be produced by the two-phase locking protocol. However, some schedules are possible under the two-phase locking protocol but not under the timestamp protocol, and vice versa. The timestamp-ordering protocol ensures conflict serializability. This capability follows from the fact that conflicting operations are processed in timestamp order. The protocol ensures freedom from deadlock, because no transaction ever waits.

6.10 SUMMARY

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided. One solution is to ensure that a critical section code is in use by only one process or thread at a time. Different algorithms exist for solving the critical-section problem, with the assumption that only storage interlock is available.

The main disadvantage of these user-coded solutions is that they all require busy waiting. Semaphores overcome this difficulty. Semaphores can be use to solve various synchronization problems and can be implemented efficiently especially if hardware support for atomic operations is available.

Various different synchronization problems (such as the bounded-buff problem, the readers-writers problem, and the dining-philosophers problem are important mainly because they are examples of a large class of concurrency control problems. These problems are used to test nearly every newly propose synchronization scheme.

EXERCISES

6.1 What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

6.2 Explain why spinlocks are not appropriate for uniprocessor systems yet may be suitable for multiprocessor systems.

6.3 Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

6.4 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, Po and PI, share the following variables:

boolean flag [2]; /* initially false */

int turn;

The structure of process Pi (i == 0 or I), with Pi (j == 1 or 0) being the other process, is shown in Figure 2.27. Prove that the algorithm satisfies all three requirements for the critical- section problem.

6.5 Demonstrate that monitors, conditional critical regions, and semaphores are all equivalent, insofar as the same types of synchronization problems can be implemented with them.

6.6 Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

6.7 Consider a system consisting of processes PI, P2, ..., P,, each of which has a unique priority number. Write a monitor that allocates three identical line printers to these processes, using the priority numbers for deciding the order of allocation.

6.8 Write a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units (ticks). You may assume the existence of a real hardware clock that invokes a procedure tick in your monitor at regular intervals.

6.9 Explain the differences, in terms of cost, among the three storage types: volatile, nonvolatile, and stable.

6.10 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect:

System performance when no failure occurs?

The time it takes to recover from a system crash?

The time it takes to recover from a disk crash?

6.11 Explain the concept of transaction atomicity.

6.12 Show that the two-phase locking protocol ensures conflict serializability.

6.13 Show that some schedules are possible under the two-phase locking protocol but not possible under the timestamp protocol, and vice versa.

7. DEADLOCKS

7.1 Introduction

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 wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock. In this chapter, we describe methods that an operating system can use to prevent or deal with deadlocks.

7.2 System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, files, and I/O devices (such as printers and tape 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.

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: If the request cannot be granted immediately (for example, 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 are system calls. Examples are the request and release device, open and close file, and allocate and free memory system calls.

A set of processes is in a deadlock 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. However, other types of events may result in deadlocks.

To illustrate a deadlock state, we consider a system with three tape drives. Suppose each of three processes holds one of these tape drives. If each process now requests another tape drive, the three processes will be in a deadlock state. Each is waiting for the event "tape drive 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 tape drive. Suppose that process Pi is holding the tape drive and process Pj is holding the printer. If Pi requests the printer and Pj requests the tape drive, a deadlock occurs.

7.3 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 shall describe features that characterize deadlocks.

7.3.1 Necessary Conditions

1. Mutual exclusion: At least one resource must be held in a non sharable 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 {Po, PI, ..., Pn) of waiting processes must exist such that Po is waiting for a resource that is held by PI, PI is waiting for a resource that is held by P2, ..., Pn -1 is waiting for a resource that is held by Pn and Pn is waiting for a resource that is held by Po.

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.

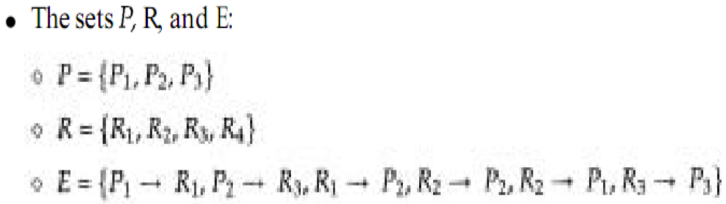
7.3.2 Resource-Allocation Graph

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 = {PI, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm), the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi -> Rj; it signifies that process Pi requested an instance of resource type Ri and is currently waiting for that resource. A directed edge from resource type Ri to process Pi is denoted by Rj -> Pi; it signifies that an instance of resource type Ri has been allocated to process Pi. A directed edge Pi -> Rj is called a request edge; a directed edge Rj -> Pi is called an assignment edge.

Pictorially, we represent each process Pi as a circle, and each resource type Ri as a square. Since resource type Ri may have more than one instance, we represent each such instance as a dot within the square. Note that a request edge points to only the square Xi, whereas an assignment edge must also designate one of the dots in the square. When process Pi requests an instance of resource type Rj, 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, and as a result the assignment edge is deleted.

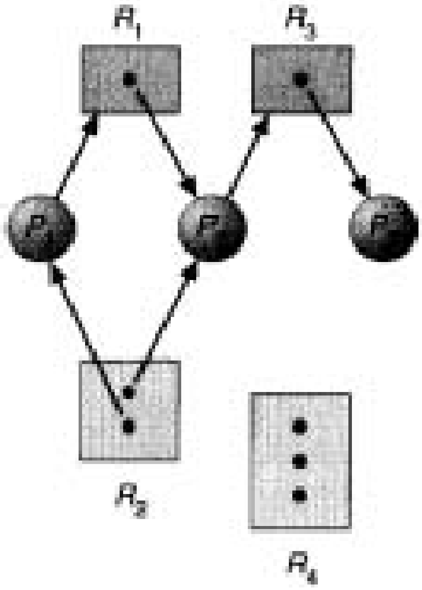
The resource-allocation graph shown in the following Figure depicts the situation.



- The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

- Resource instances:
  - o One instance of resource type R1
  - o Two instances of resource type R2
  - o One instance of resource type R3

- o Three instances of resource type R4



Resource-allocation graph.

- Process states:

o Process PI is holding an instance of resource type R2, and is waiting for an instance of resource type R1.

o Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.

o process P3 is holding an instance of R3.

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.

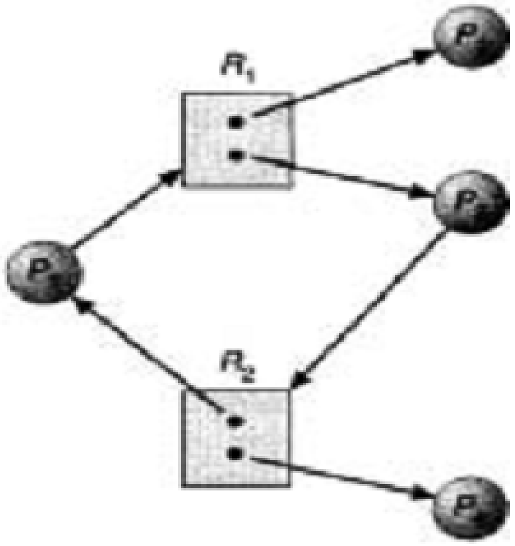
If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. To illustrate this concept, let us return to the resource-allocation graph depicted in the above Figure .Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 -> RL is added to the graph (Below Figure). 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 PI, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3, on the other hand, is waiting for either process PI or process P2 to release resource R2. In addition, process PI is waiting for process P2 to release resource R1.

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

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.



Resource-allocation graph with a cycle but no deadlock.

7.4 Methods for Handling Deadlocks

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 deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover
- We can ignore the problem altogether, and pretend that deadlocks never occur in the system. This solution is used by most operating systems, including UNIX.

To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock avoidance scheme. Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we 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.

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur.

7.5 Deadlock Prevention

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. Let us elaborate on this approach by examining each of the four necessary conditions separately.

Mutual Exclusion

The mutual-exclusion condition must hold for non sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, 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: Some resources are intrinsically non sharable.

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 can be used requires each process to request and be allocated all its resources before it begins execution.

An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a tape drive to a disk file, sorts the disk file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the tape drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the tape drive and disk file. It copies from the tape drive to the disk, then releases both the tape drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

These protocols have two main disadvantages. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.

Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption

The third necessary condition is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration. Let R = {R1, R2, ..., Rn,) be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R -> N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive) = 1,

F(disk drive) = 5,

F(printer) = 12.

We can now consider the following protocol to prevent deadlocks:

Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say Xi. After that, the process can request instances of resource type Ri if and only if F(Rj) > F(Ri). If several instances of the same resource type are needed, a single request for all of them must be issued. For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

Alternatively, we can require that, whenever a process requests an instance of resource type Rj, it has released any resources Ri such that F(Ri) 2 F(Rj). If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of processes involved in the circular wait be {Po, PI, ..., P,), where Pi is waiting for a resource Xi, which is held by process P_{i+1}. (Modulo arithmetic is used on the indexes, so that P, is waiting for a resource R, held by Po.) Then, since process P_{i+1} is holding resource Ri while requesting resource R_{i+1}, we must have F(Ri) < F(R_{i+1}), for all i. But this condition means that F(Ro) < F(R1) < ... < F(Rn) < F(Ro). By transitivity, F(Ro) < F(Ro), which is impossible.

Therefore, there can be no circular wait. Note that the function F should be defined according to the normal order of usage of the resources in a system. For example, since the tape drive is usually needed before the printer, it would be reasonable to define F(tape drive) < F(printer).

7.6 Deadlock Avoidance

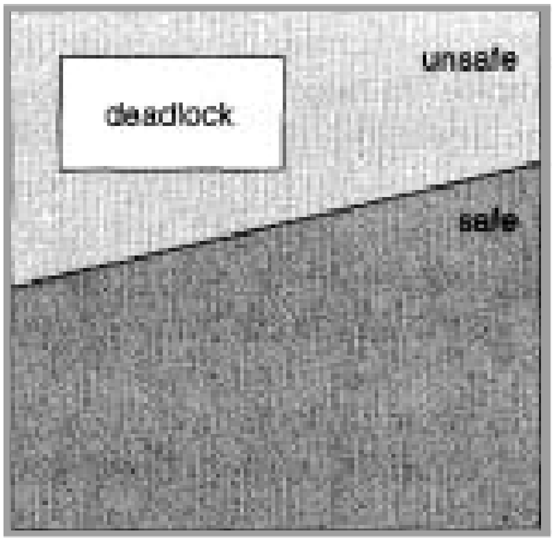
Possible side effects of prevention method, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, we might be told that process P will request first the tape drive, and later the printer, before releasing both resources. Process Q, on the other hand, will request first the printer, and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.

The various algorithms differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This algorithm defines the deadlock-avoidance approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

7.6.1 Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes <PI, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all the Pi, with j < i. In this situation, if he resources that process Pi needs are not immediately available, then Pi can wait until all Pi have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+l can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.



Safe, unsafe, and deadlock state spaces

A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks, however (in the above Figure). An unsafe state may lead to a deadlock.

To illustrate, we consider a system with 12 magnetic tape drives and 3 processes: Po, PI, and P2. Process Po requires 10 tape drives, process PI may need as many as 4, and process P2 may need up to 9 tape drives. Suppose that, at time to, process Po is holding 5 tape drives, process PI is holding 2, and process P2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

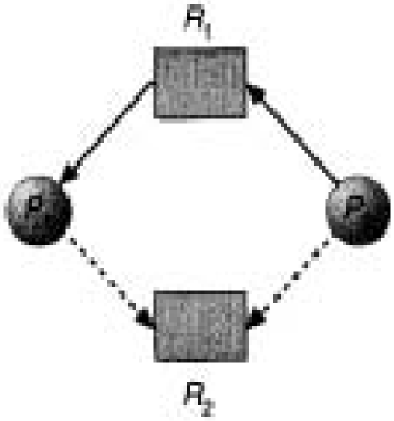
At time to, the system is in a safe state. The sequence <PI, PO, P2> satisfies the safety condition, since process PI can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process Po can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process P2 could get all its tape drives and return them (the system will then have all 12 tape drives available).

A system may go from a safe state to an unsafe state. Suppose that, at time tl, process Pp requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process PI can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process Po is allocated 5 tape drives, but has a maximum of 10, it may then request 5 more tape drives. Since they are unavailable, process Po must wait. Similarly, process P2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.

7.6.2 Resource-Allocation Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource allocation graph can be used for deadlock avoidance.

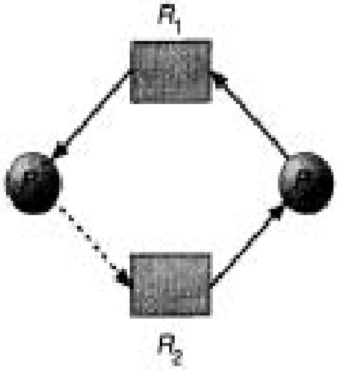
In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge. A claim edge pi->rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line. When process Pi requests resource Rj, the claim edge Pi -> Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi, the assignment edge Rj -> Pi is reconverted to a claim edge Pi -> Xi. We note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge Pi -> Rj to be added to the graph only if all the edges associated with process Pi are claim edges. Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi -> Rj to an assignment edge Rj -> Pi does not result in the formation of a cycle in the resource-allocation graph. Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.



Resource-allocation graph for deadlock avoidance.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.

To illustrate this algorithm, we consider the resource-allocation graph of above Figure . Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph (bellow Figure). A cycle indicates that the system is in an unsafe state. If PI requests R2, and P2 requests R1, then a deadlock will occur.



An unsafe state in a resource-allocation graph.

7.6.3 Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. So here we introduce a new algorithm known as the banker's algorithm, but is less efficient than the resource-allocation graph scheme.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources. Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let n be the number of processes in the system and rn be the number of resource types. We need the following data structures:

Available: A vector of length rn indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type Rj available.

Max: An n x rn matrix defines the maximum demand of each process. If Max[i,j] = k, then process Pi may request at most k instances of resource type Ri.

Allocation: An n x rn matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process Pi is currently allocated k instances of resource type Rj.

Need: An n x rn matrix indicates the remaining resource need of each process. If Need[i,j] = k, then process Pi may need k more instances of resource type Ri to complete its task. Note that Need[i,j] = Max[i,j] - Allocafion[i,j].

7.6.4 Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work := Available and Finish[i] :=false for i = 1,2, ...,n.

2. Find an i such that both

   a. Finish[i] =false

   b. Needi <= Work. If no such i exists, go to step 4.

3. Work := Work + Allocationi

   Finish[i] := true

   go to step 2.

4. If Finish[i] = true for all i, then the system is in a safe state.

This algorithm may require an order of m x n2 operations to decide whether a state is safe.

7.6.5 Resource-Request Algorithm

Let Request i be the request vector for process Pi. If Request;[j] = k, then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1. If Request_i<= Need_i, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If Request_i <=Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

   Available := Available - Request,;

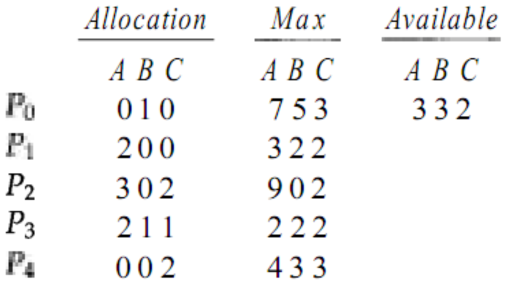
   Allocation_i := Allocation; + Request;;

   Need_i := Need_i - Request_i;

If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Requesti and the old resource-allocation state is restored.

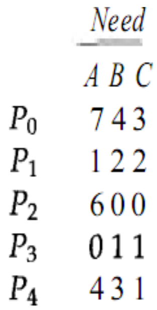
7.6.6 An Illustrative Example

Consider a system with five processes Po through P4 and three resource types A, B, C. Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at time To, the following snapshot of the system has been taken:



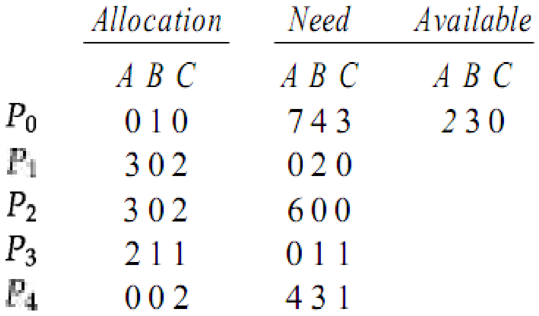
	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix Need is defined to be Max - Allocation and is



	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence <PI, P3, P4, P2, Po> satisfies the safety criteria. Suppose now that process PI requests one additional instance of resource type A and two instances of resource type C, so Request$_i$ = (1,0,2). To decide whether this request can be immediately granted, we first check that Request$_i$ <= Available (that is, (1,0,2) <= (3,3,2)), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:



	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <PI, P3, P4, PO, P2>satisfies our safety requirement. Hence, we can immediately grant the request of process PI. You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available. A request for (0,2,0) by Po cannot be granted, even though the resources are available, since the resulting state is unsafe.

7.7 Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

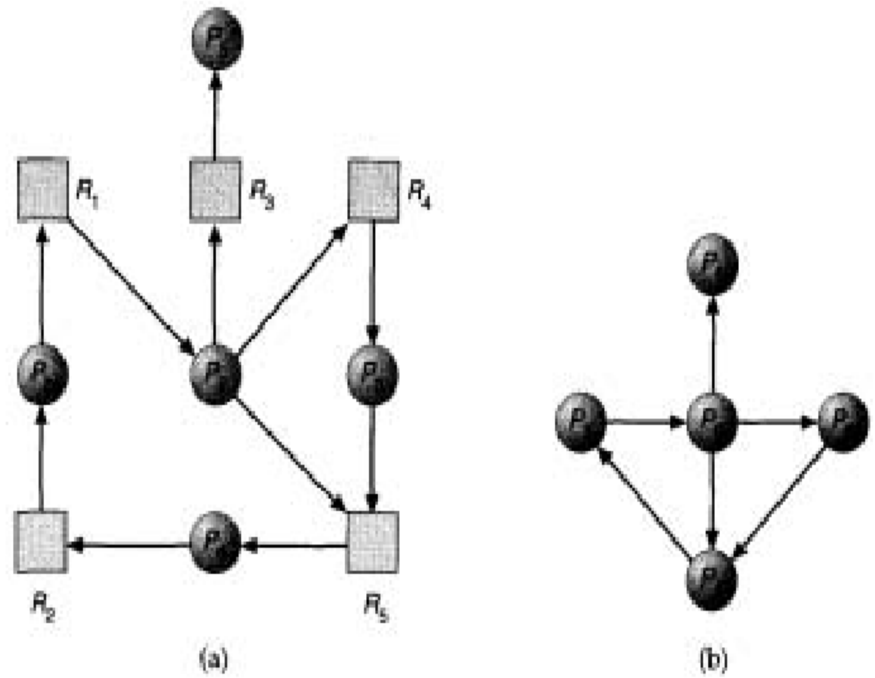
In the following discussion, we elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type.

7.7.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs. An edge Pi to Pj exists in a wait-for graph if and only if the corresponding resource- allocation graph contains two edges Pi -> Rq and Rq-> Pj for some resource Rq. For example, in the following Figure , we present a resource-allocation graph and the corresponding wait-for graph.

As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph.



(a) Resource-allocation graph.

(b) Corresponding wait-for graph.

7.7.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm

- Available: A vector of length m indicates the number of available resources of each type.
- Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process.
- Request: An n x m matrix indicates the current request of each process. If Request[i,j] = k, then process Pi is requesting k more instances of resource type Ri.

The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed. Compare this algorithm with the banker's algorithm

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work := Available. For i = 1, 2, ..., n, if Allocation$_i$ \$0, then Finish[i] :=false; otherwise, Finish[i] := true.

2. Find an index i such that both

   a. Finish[i] =false.

   b. Request$_i$ <=Work. If no such i exists, go to step 4.

3. Work := Work + Allocation$_i$

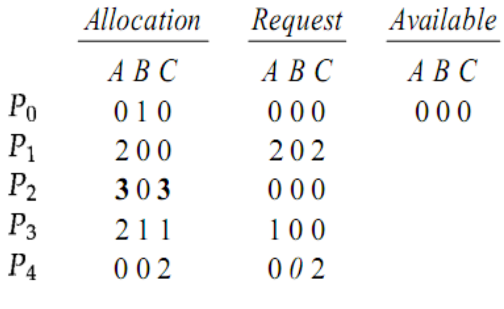
   Finish[i] := true

   go to step 2.

4. If Finish[i] = false, for some i, 1 <= i<= n, then the system is in a deadlock state. Moreover, if Finish[i] =false, then process Pi is deadlocked.

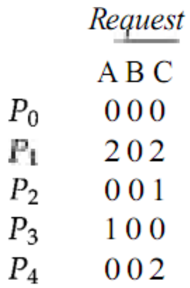
This algorithm requires an order of m x n^2 operations to detect whether the system is in a deadlocked state.

To illustrate this algorithm, we consider a system with five processes Po through P4 and three resource types A, B, C. Resource type A has 7 instances, resource type B has 2 instances, and resource type C has 6 instances. Suppose that, at time To, we have the following resource-allocation state:



	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	**3** 0 **3**	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <Po, P2, P3, PI, P4> will result in Finish[i] = true for all i. Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows:



	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

We claim that the system is now deadlocked.

7.7.3 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How often is a deadlock likely to occur?
2. How many processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes.

Of course, invoking the deadlock-detection algorithm for every request may incur a considerable overhead in computation time. For example, once per hour, or whenever CPU utilization drops below 40 percent.

7.8 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

7.8.1 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

Abort all deadlocked processes: This method clearly will break the dead- lock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.

Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.

If the partial termination method is used, then, given a set of deadlocked processes, we must determine which process (or processes) should be terminated in an attempt to break the deadlock. This determination is a policy decision, similar to CPU-scheduling problems. The question is basically an economic one; we should abort those processes the termination of which will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. Many factors may determine which process is chosen, including:

1. What the priority of the process is

2. How long the process has computed, and how much longer the process will compute before completing its designated task

3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)

4. How many more resources the process needs in order to complete

5. How many processes will need to be terminated

6. Whether the process is interactive or batch

7.8.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim: Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.

2. Rollback: If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However, it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

3. Starvation: How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

7.9 Summary

A deadlock state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. Principally, there are three methods for dealing with deadlocks:

- Use some protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- Allow the system to enter deadlock state, detect it, and then recover.
- Ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.

A deadlock situation may occur if and only if four necessary conditions hold simultaneously in the system: mutual exclusion, hold and wait, no pre- emption, and circular wait. To prevent deadlocks, we ensure that at least one of the necessary conditions never holds.

Another method for avoiding deadlocks that is less stringent than the prevention algorithms is to have a priori information on how each process will be utilizing the resources.

If a system does not employ a protocol to ensure that deadlocks will never occur, then a detection-and-recovery scheme must be employed.

In a system that selects victims for rollback primarily on the basis of cost factors, starvation may occur. As a result, the selected process never completes its designated task.

Exercises

7.1 List three examples of deadlocks that are not related to a computer-system environment.

7.2 Is it possible to have a deadlock involving only one process? Explain your answer.

7.3 Suppose that a system is in an unsafe state. Show that it is possible for the processes to complete their execution without entering a deadlock state.

7.4 In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods (months). Resources break or are replaced, new processes come and go, new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?

a. Increase Available (new resources added)

b. Decrease Available (resource permanently removed from system)

c. Increase Max for one process (the process needs more resources than allowed, it may want more)

d. Decrease Max for one process (the process decides it does not need that many resources)

e. Increase the number of processes

f. Decrease the number of processes

7.5 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

7.6 Consider a system consisting of rn resources of the same type, being shared by n processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock-free if the following two conditions hold:

a. The maximum need of each process is between 1 and rn resources

b. The sum of all maximum needs is less than rn + n

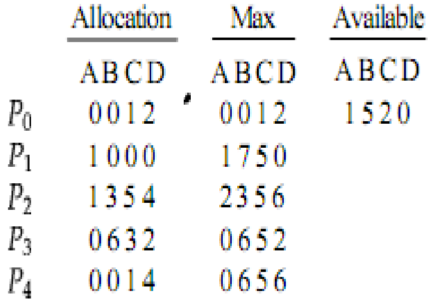
7.7 Consider a computer system that runs 5,000 jobs per month with no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about 10 jobs per deadlock. Each job is worth about $2 (in CPU time), and the jobs terminated tend to be about half-done when they are aborted. A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase in the average execution time per job of about 10 percent. Since the machine currently has 30-percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

a. What are the arguments for installing the deadlock-avoidance algorithm?

b. What are the arguments against installing the deadlock-avoidance b. What are the arguments against installing the deadlock-avoidance algorithm? We can obtain the banker's algorithm for a single resource type from the general banker's algorithm simply by reducing the dimensionality of the various arrays by 1. Show through an example that the multiple-resource- type banker's scheme cannot be implemented by individual application of the single-resource-type scheme to each resource type.

7.8 Can a system detect that some of its processes are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.

7.9 Consider the following snapshot of a system:



	Allocation	Max	Available
	ABCD	ABCD	ABCD
P_0	0012	0012	1520
P_1	1000	1750	
P_2	1354	2356	
P_3	0632	0652	
P_4	0014	0656	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix Need?

b. Is the system in a safe state?

c. If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

7.10 Consider the following resource-allocation policy. Requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked, waiting for resources. If they have the desired resources, then these resources are taken away from them and are given to the requesting process. The vector of resources for which the waiting process is waiting is increased to include the resources that were taken away.

For example, consider a system with three resource types and the vector Available initialized to (4,2,2). If process Po asks for (2,2,1), it gets them. If PI asks for (1,0,1), it gets them. Then, if Po asks for (0,0,1), it is blocked (resource not available). If P2 now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to Po (since Po is blocked). Po's Allocation vector goes down to (1,2,1), and its Need vector goes up to (1,0,1).

a. Can deadlock occur? If so, give an example. If not, which necessary condition cannot occur?

b. Can indefinite blocking occur?

7.11 Suppose that you have coded the deadlock-avoidance safety algorithm and now have been asked to implement the deadlock-detection algorithm. Can you do so by simply using the safety algorithm code and redefining Maxi = Waiting_i + Allocation_i, where Waiting_i is a vector specifying the resources process i is waiting for, and Allocation_i is as defined in Section 7.6? Explain your answer.

8. MEMORY MANAGEMENT

8.1 Introduction

Memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter.

After the instruction has been executed on the operands, results may be stored back in memory. 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, and so on) or what they are for (instructions or data).

Accordingly, we can ignore how a memory address is generated by a program. We are interested in only the sequence of memory addresses generated by the running program.

8.1.1 Address Binding

Usually, a program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The collection of processes on the disk that is waiting to be brought into memory for execution forms the input queue.

The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available. Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process does not need to be 00000.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process does not need to be 00000. This arrangement affects the addresses that the user program can use. In most cases, a user program will go through several steps-some of which may be optional-before being executed (Figure 2.1). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as count). A compiler will typically **bind** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind these relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

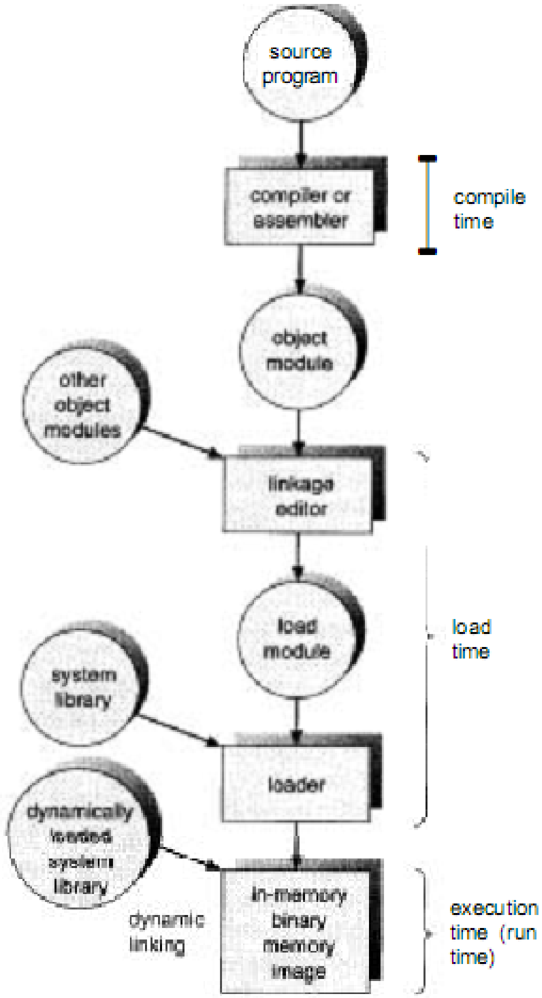


Figure 8.1 Multistep processing of a user program.

Compile time: If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know a priori that a user process resides starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

Load time: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only to reload the user code to incorporate this changed value.

Execution time: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

8.1.2 Logical- Versus Physical-Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit-that is, the one loaded into the memory-address register of the memory-is commonly referred to as a **physical address**.

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. we usually refer to the logical address as a virtual address. The set of all logical addresses generated by a program is a logical-address space; the set of all physical addresses corresponding to these logical addresses is a physical-address space. Thus, in the execution-time address-binding scheme, the logical- and physical-address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).

As illustrated in Figure 2.2, this method requires hardware support. he base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

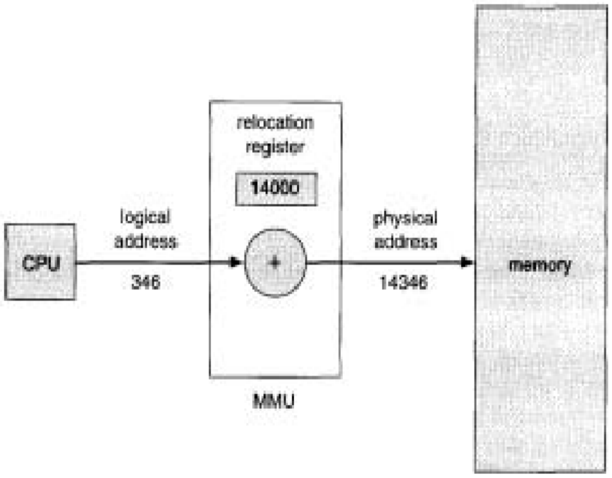


Figure 8.2 Dynamic relocation using a relocation register.

The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, compare it to other addresses all as the number 346. The memory-mapping hardware converts logical addresses into physical addresses.

8.1.3 Dynamic Loading

The entire program and data of a process must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then, control is passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method.

8.1.4 Dynamic Linking and Shared Libraries

Figure 8.1 also shows dynamically linked libraries. Some operating systems support only static linking, in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. With dynamic linking, a stub is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine, or how to load the library if the routine is not already present.

When this stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine, and executes the routine. Thus, the next time that that code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code.

This feature can be extended to library updates (such. as bug fixes). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such 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. only programs that are compiled with the new library version are affected by the 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.

8.1.5 overlay

To enable a process to be larger than the amount of memory allocated to it, we can use overlays. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space occupied previously by instructions that are no longer needed.

As an example, consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code. We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table, and common support routines used by both pass 1 and pass 2. Assume that the sizes of these components are as follows:

Pass 1	70 KB
Pass 2	80 KB
Symbol table	20 KB
Common routines	30 KB

To load everything at once, we would require 200 KB of memory. If only 150 KB is available, we cannot run our process. However, notice that pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays: Overlay A is the symbol table, common routines, and pass 1, and overlay B is the symbol table, common routines, and pass 2. We add an overlay driver (10 KB) and start with overlay A in memory. When we finish pass 1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay A, and then transfers control to pass 2. Overlay A needs only 120 KB, whereas overlay B needs 130 KB (Figure 2.3). We can now run our assembler in the 150 KB of memory. It will load somewhat faster because fewer data need to be transferred before execution starts. However, it will run somewhat slower, due to the extra 1/0 to read the code for overlay B over the code for overlay A.

The code for overlay A and the code for overlay B are kept on disk as absolute memory images, and are read by the overlay driver as needed. Special relocation and linking algorithms are needed to construct the overlays. As in dynamic loading, overlays do not require any special support from the operating system. The programmer, on the other hand, must design and program the overlay structure properly. For these reasons, the use of overlays is currently limited to microcomputer.

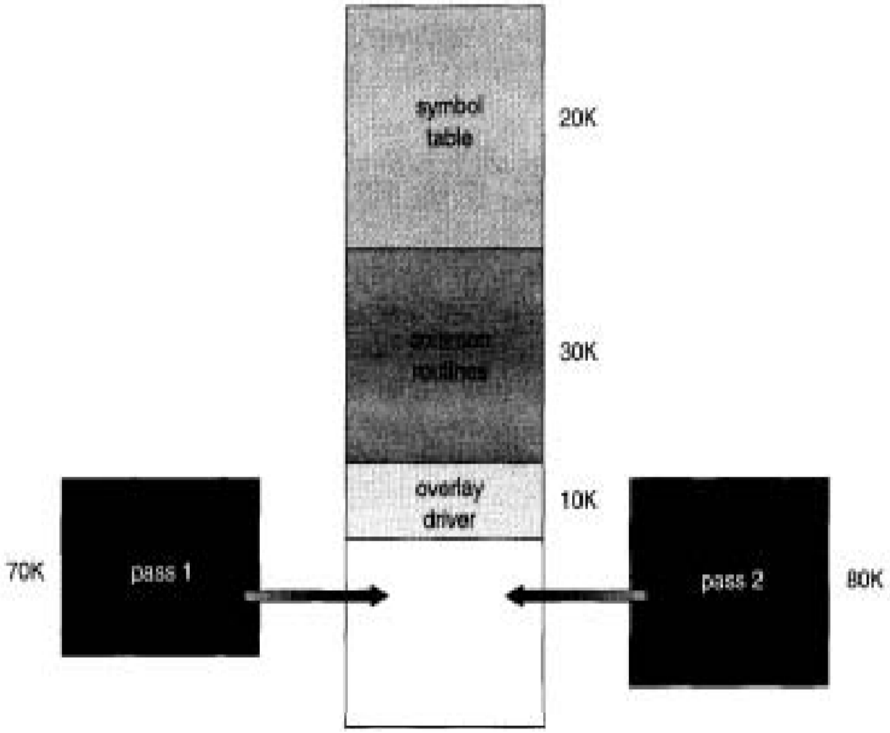


Figure 8.3 Overlays for a two-pass assembler.

8.2 Swapping

A process needs to 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. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed (Figure 8.4). In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.

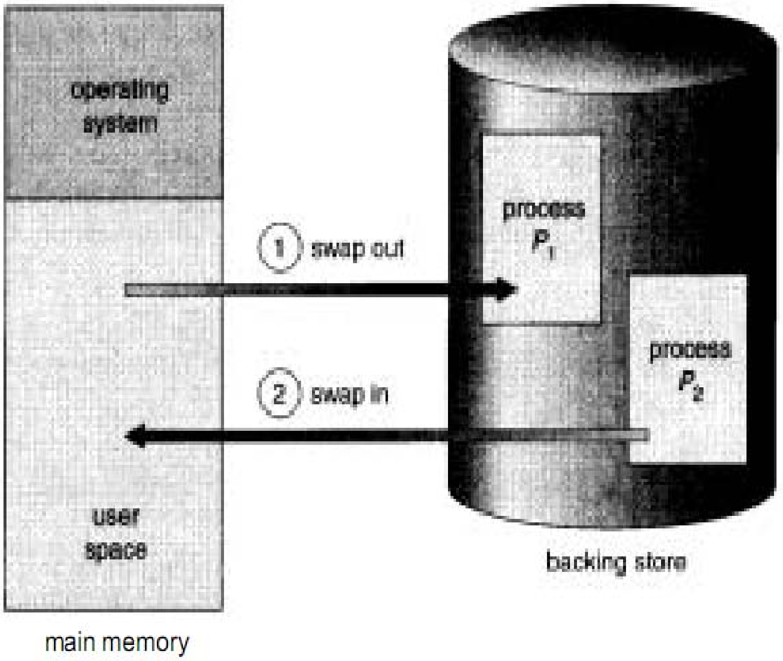


Figure 8.4 Swapping of two processes using a disk as a backing store.

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called roll out, roll in.

Swapping requires a backing store. The backing store is commonly a fast disk. It must be large 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 not, and 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 as normal 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 us assume that the user process is of size 1 282 Chapter 9 Memory Management MB and the backing store is a standard hard disk with a transfer rate of 5 MB per second.

The actual transfer of the 1 MB process to or from memory takes 1000 KB/5000 KB per second = 1 / 5 second = 200 milliseconds.

Assuming that no head seeks are necessary and an average latency of 8 milliseconds, the swap time takes 208 milliseconds. Since we must both swap out and swap in, the total swap time is then about 416 milliseconds. For efficient CPU utilization, we want our execution time for each process to be long relative to the swap time. Thus, in a round-robin CPU-scheduling algorithm, for example, the time quantum should be substantially larger than 0.416 seconds.

8.3 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate different parts of the main memory in the most efficient way possible.

The memory is usually divided into two partitions: one for the resident operating system, and one for the user processes. We may 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.

8.3.1 Memory Protection

Before discussing memory allocation, we must discuss the issue of memory protection protecting the operating system from user processes, and protecting user processes from one another. We can provide this protection by using a relocation register, as discussed in previous Section. 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). With relocation and limit registers, each logical address must be less than 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 8.5).

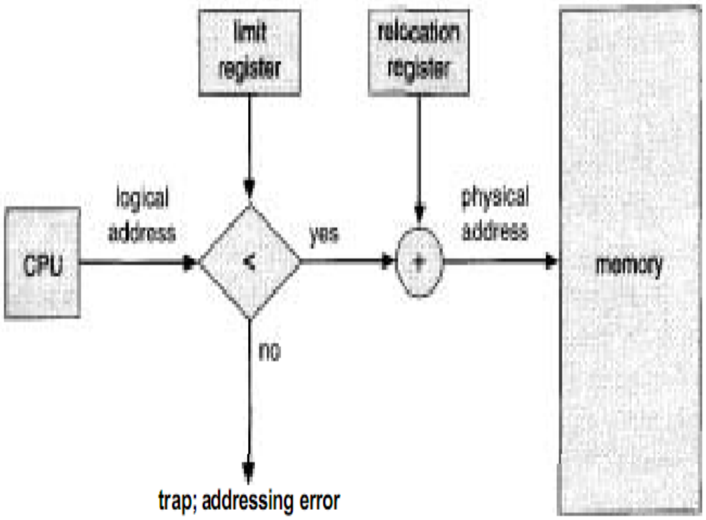


Figure 8.5 Hardware support for relocation and limit registers.

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 the 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.

8.3.2 Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods for memory allocation 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 0S/360 operating system (called MFT); it is no longer in use. The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in a batch environment.

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 as one large block of available memory, a hole. When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.

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 the CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

In general, a set of holes, of various sizes, is scattered throughout memory at any given time. When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large, it is split into two: 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.

This procedure is a particular instance of the general dynamic storage- allocation problem, which is how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The set of holes is searched

to determine which hole is best to allocate. The first-fit, best-fit, and worst-fit strategies are the most common ones 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 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 kept 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.

These algorithms, however, 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 enough total memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of small holes.

The selection of the first-fit versus best-fit strategies can affect the amount of fragmentation.

8.3.3 Fragmentation

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 is to break the physical memory into fixed-sized blocks, and allocate memory in unit of block sizes. 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-memory that is internal to a partition but is not being used.

One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents to place all free memory together in one large block. Compaction is not always possible. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction 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 simply 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 a process to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. Two complementary techniques achieve this solution: paging and segmentation which we are discussed in the next two sections.

8.4. Paging

Paging is a memory-management scheme that permits the physical-address space of a process to be noncontiguous. Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The fragmentation problems discussed in connection with main memory are also prevalent with backing store, except that access is much slower, so compaction is impossible. Because of its advantages over the previous methods, paging in its various forms is commonly used in most operating systems.

8.4.1 Basic Method

Physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken 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 the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

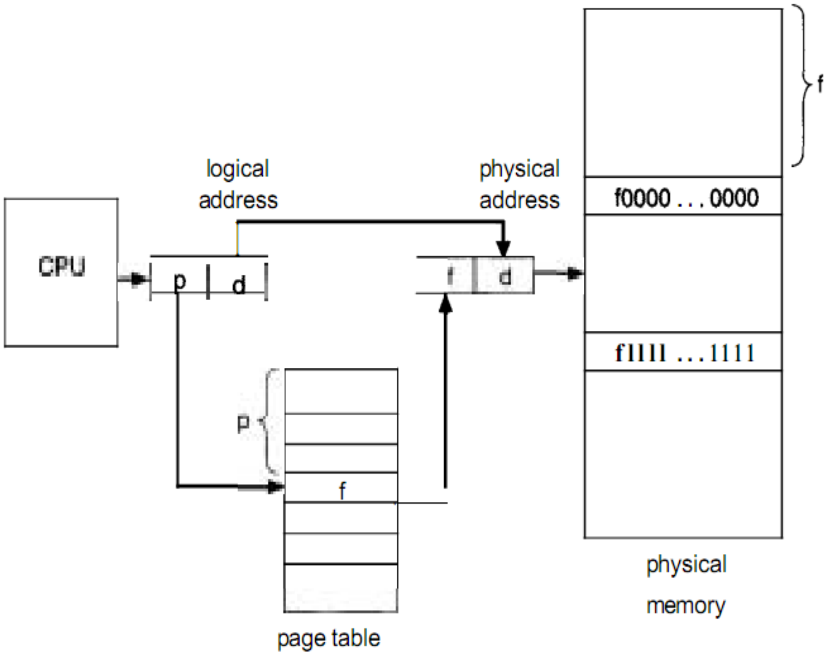


Figure 8.6 Paging hardware.

The hardware support for paging is illustrated in Figure 8.6. Every address generated by the CPU is divided into two parts: a page number (p) and a page 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 8.7.

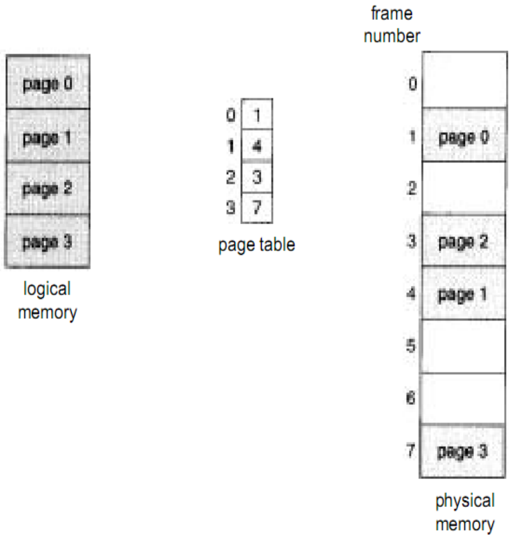
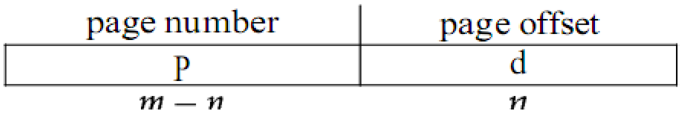


Figure 8.7 Paging model of logical and physical memory

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.

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 8.8. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user'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 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 x 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 x 4) + 0). Logical address 13 maps to physical address 9.

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.

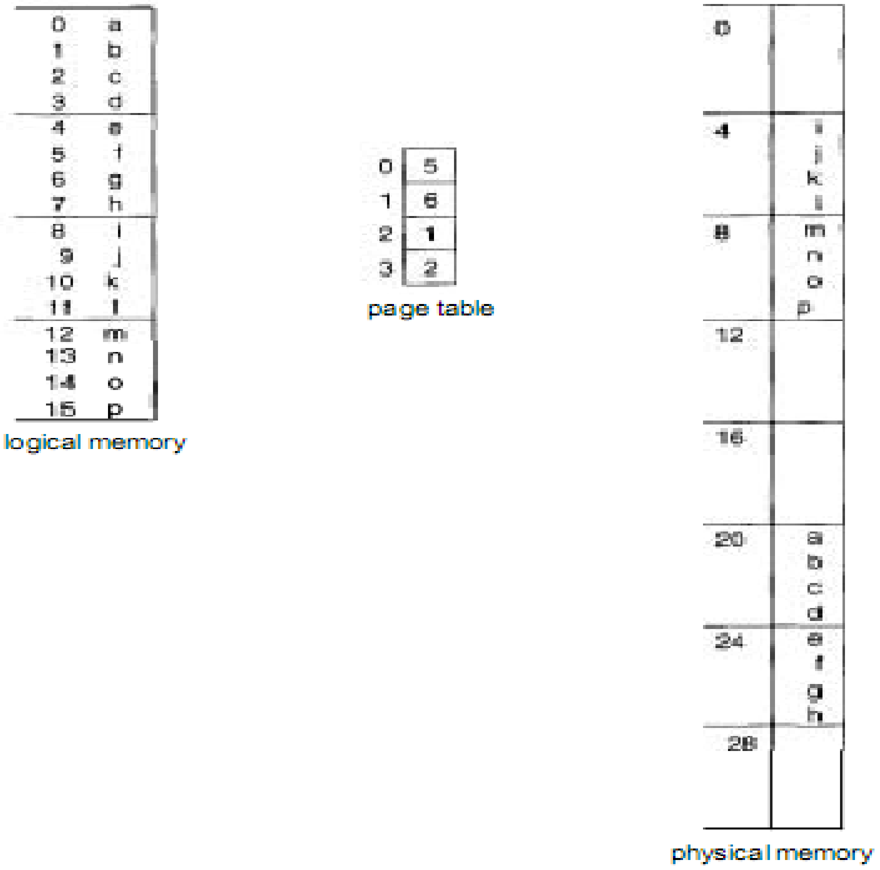


Figure 8.8 Paging example for a 32-byte memory with 4-byte pages.

If the memory requirements of a process do not happen to fall on page boundaries, the last frame allocated may not be completely full. For example, if pages are 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes. It would be allocated 36 frames, resulting in an internal fragmentation of 2048 - 1086 = 962 bytes. In the worst case, a process would need n pages plus one byte. It would be allocated n + 1 frames, resulting in an internal fragmentation of almost an entire frame.

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, and its frame number is put into the page table, and so on (Figure 8.9).

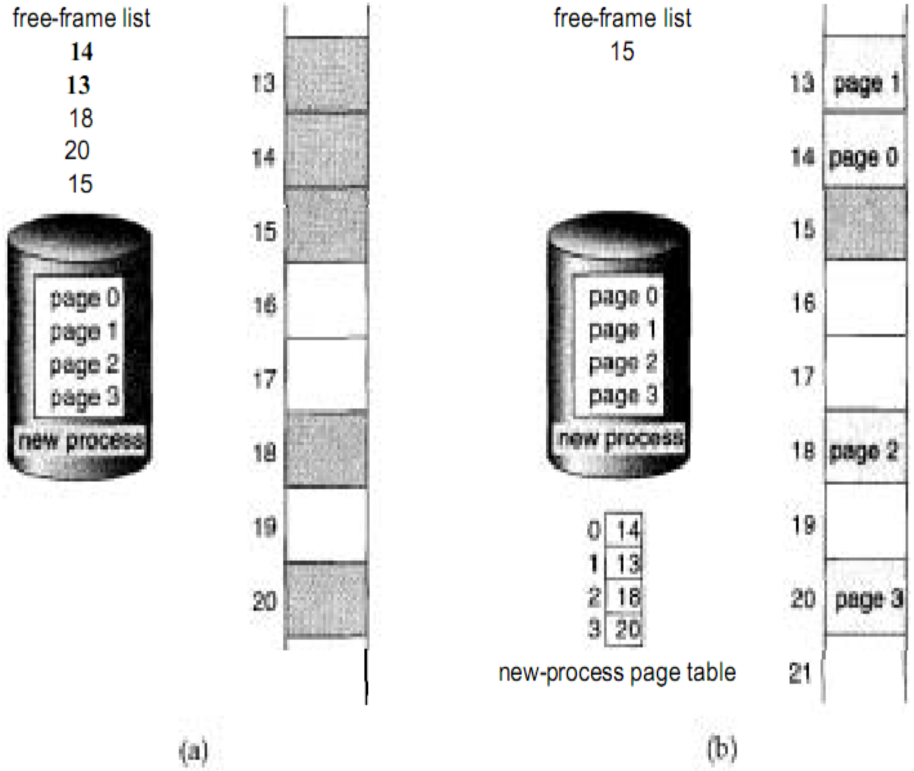


Figure 8.9 Free frames. (a) Before allocation. (b) After allocation.

An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views that memory as one single contiguous 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 user'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 user and is controlled by the operating system.

Advantages:

- No external fragmentation**.**

Disadvantages:

- Lack of users view of memory.
- Small amount of internal fragmentation.

8.4.2 Hardware Support

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 register.

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 translation lookaside buffer (TLB). The TLB s 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, it is compared with all keys simultaneously. If the item is found, he corresponding value field is returned. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

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. The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.

The page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 8.10). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are often wired down.

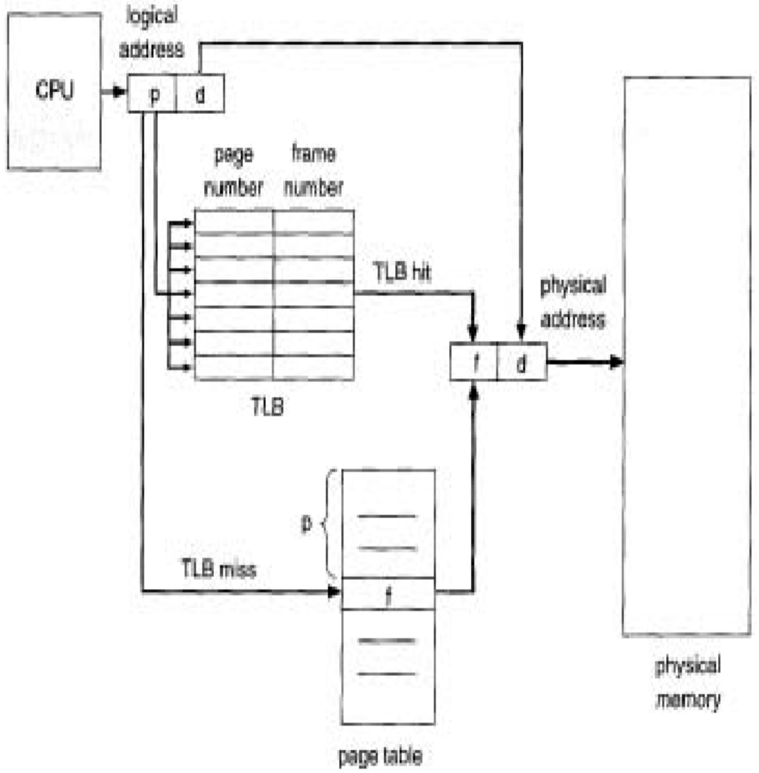


Figure 8.10 Paging hardware with TLB.

Some TLBs store address-space identifiers (ASIDs) in each entry of the TLB. 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 the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, they are 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, every time a new page table is selected (for instance, 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, there could be old entries in the TLB that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that a particular page number is found in the TLB is called the **hit ratio**. An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB, and 100 nanoseconds to access memory, then a mapped- memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB (20 nanoseconds), 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 220 nanoseconds. To find **the effective memory-access time**, we must weigh each case by its probability:

effective access time = 0.80 x 120 + 0.20 x 220

= 140 nanoseconds.

In this example, we suffer a 40-percent slowdown in memory access time (f100 to 140 nanoseconds).

For a 98-percent hit ratio, we have

effective access time = 0.98 x 120 + 0.02 x 220

= 122 nanoseconds.

This increased hit rate produces only a 22-percent slowdown in access time.

8.4.3 Protection

Memory protection in a paged environment is accomplished by protection bits that are 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).

One more bit is generally attached to each entry in the page table: a valid-invalid bit. When this bit is set to "valid," this value indicates that the associated page is in the process' logical-address space, and is thus a legal (or valid) page. If the bit is set to "invalid," this value indicates that the page is not in the process' logical-address space. Illegal addresses are trapped by using the valid- invalid bit. The operating system sets this bit for each page to allow or disallow accesses to that page. For example, in a system with a 14-bit address space (0 to 16383), we may have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we get the situation shown in Figure 8.11. 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, finds that the valid- invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

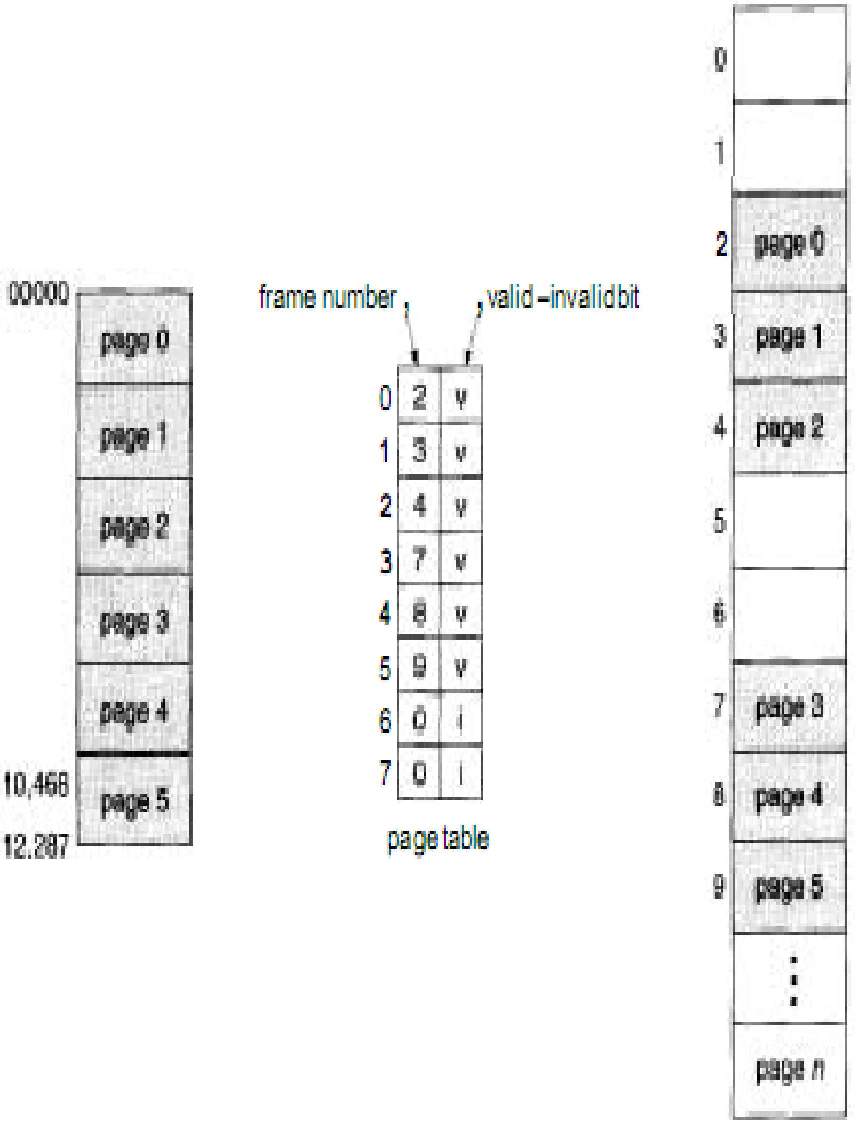


Figure 8.11 Valid (v) or invalid (i) bit in a page table.

Because the program extends to only 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 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.

8.4.4 Structure of the Page Table

8.4.4.1 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 (212), 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. There are several ways to accomplish this division.

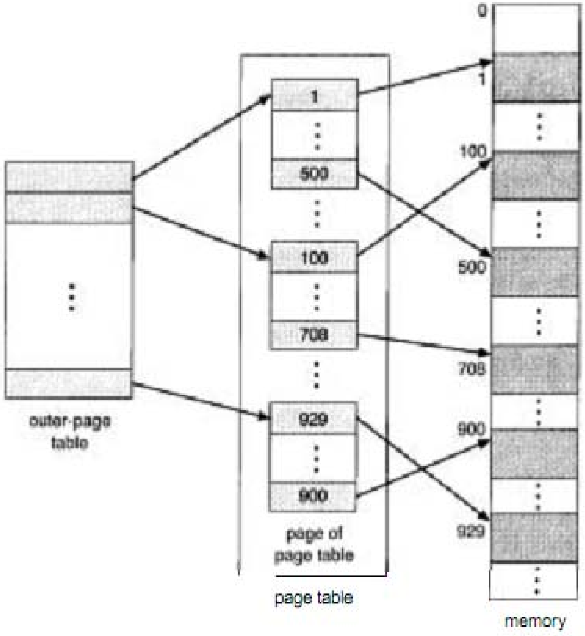


Figure 8.12 A two-level page-table scheme.

One way is to use a two-level paging algorithm, in which the page table itself is also paged (Figure 8.12). Remember our example to our 32-bit machine with 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 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 pl is an index into the outer page table and p2 is the displacement within the page of the outer page table. The address-translation method for this architecture is shown in Figure 8.13. Because address translation works from the outer page table inwards, this scheme is also known as a forward-mapped page table.

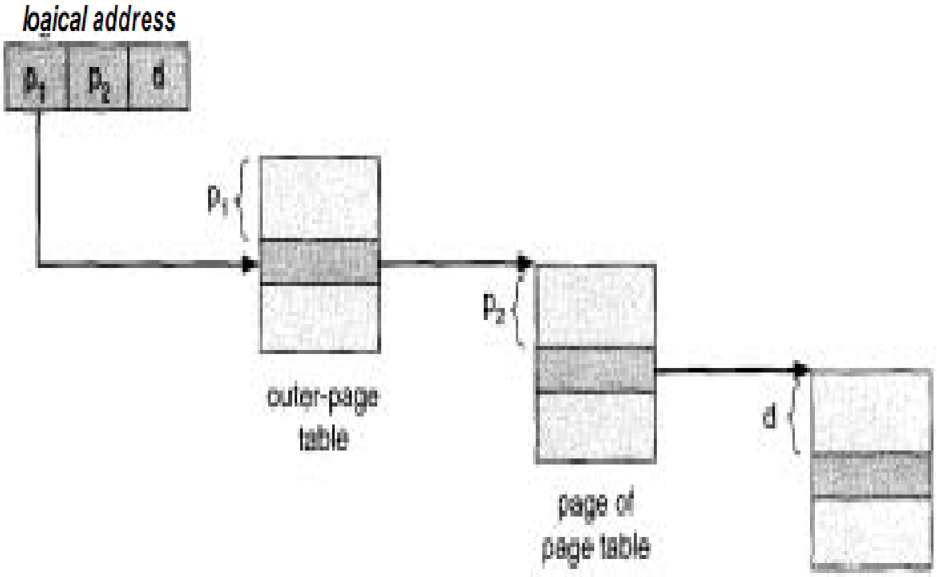


Figure 8.13 Address translation for a two-level 32-bit paging architecture.

For a system with a 64-bit logical-address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let us suppose that the page size in such a system is 4 KB (2^{12}). In this case, the page table will consist of up to 252 entries. If we use a two-level paging scheme, then the inner page tables could conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses would look like:

outer page	inner page	offset
p1	*p2*	*d*
42	10	12

The outer page table will consist of 2^{42} entries, or 2^{44} bytes. The obvious method 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. 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); a 64-bit address space is still daunting:

2nd outer page	outer page	inner page	offset
p1	*p2*	*p3*	d
32	10	10	12

The outer page table is still 2^{34} bytes large.

The next step would be a four-level paging scheme, where the second- level outer page table itself is also paged.

8.4.4.2 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: (a) the virtual page number, (b) the value of he mapped page frame, and (c) a pointer to the next element in the linked list.

he algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared to field (a) in the first element in the linked list. If there is a match, the corresponding page frame (field (b)) 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 8.14.

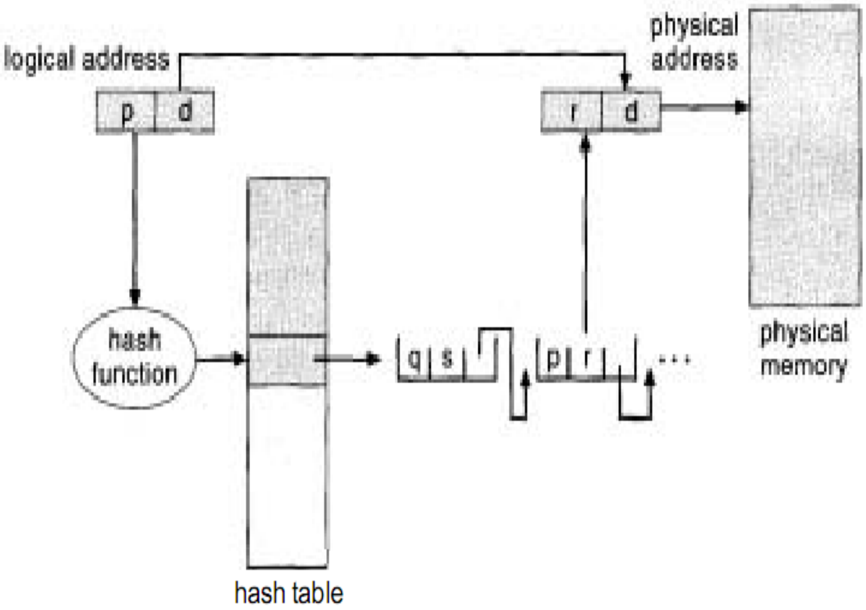


Figure 8.14 Hashed page table.

8.4.4.3 Inverted Page Table

Usually, each process has a page table associated with it. 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, 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, which is required just to keep track of how the 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 that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure 8.15 shows the operation of an inverted page table.

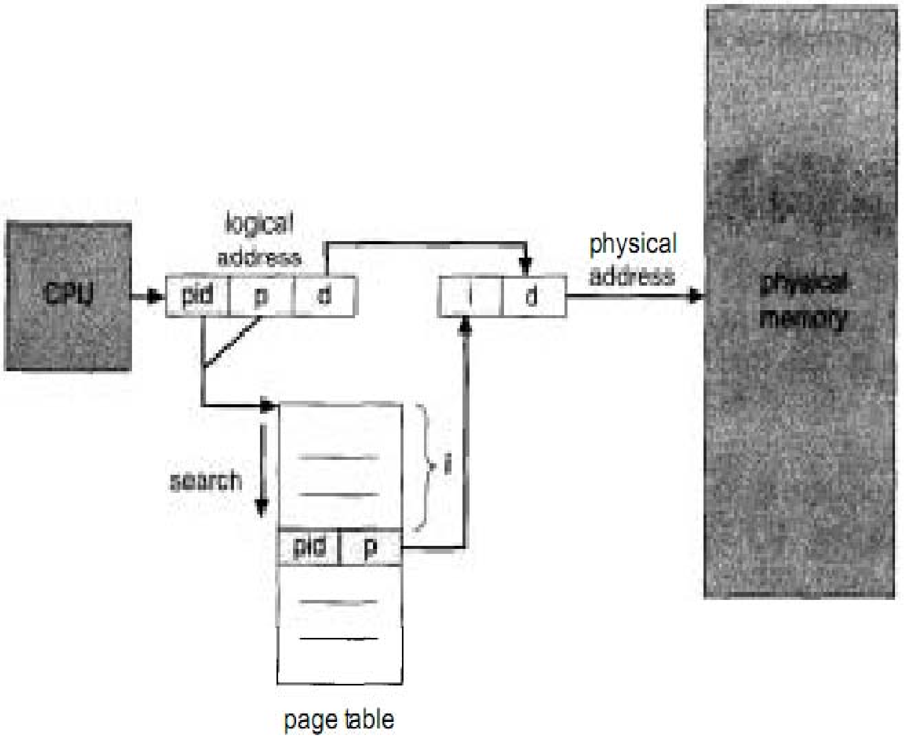


Figure 8.15 Inverted page table.

To illustrate this method, we describe a simplified version of the implementation of the inverted page table used in 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 a physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match. This search would take far too long. To alleviate this problem, we use a hash table as described in Section 8.4.4.2 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. To improve performance, recall that the TLB is searched first, before the hash table is consulted.

8.4.5 Shared Pages

Another 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 would need 8,000 KB to support the 40 users. If the code is reentrant code, however, it can be shared, as shown in Figure 8.16. Here we see a three-page editor-each page of size 50 KB; the large page size is used to simplify the figure-being shared among three processes. Each process has its own data page. Reentrant code (or pure code) is non-self-modifying code. If the code is reentrant, then it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process' execution. The data for two different processes will, of course, vary for each process.

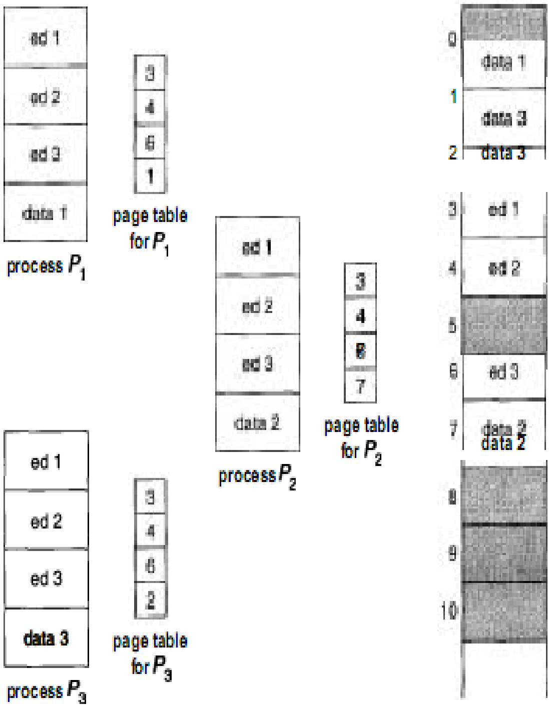


Figure 8.16 Sharing of code in a paging environment.

Only one copy of the editor needs to 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 left to the correctness of the code; the operating system should enforce this property.

8.5 Segmentation

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory. The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. The mapping allows differentiation between logical memory and physical memory.

8.5.1 Basic Method

Users think the memory as a linear array of bytes, some containing instructions and others containing data? Most people would say no. Rather, users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments (Figure 8.17).

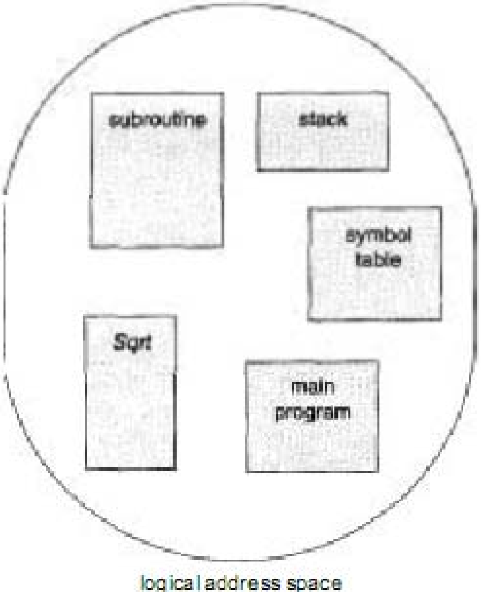


Figure 8.17 User's view of a program.

Consider how you think of a program when you are writing it. You think of it as a main program with a set of subroutines, procedures, functions, or modules. There may also be various data structures: tables, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. You talk about "the symbol table," "function Sqrt," "the main program," without caring what addresses in memory these elements occupy. You are not concerned with whether the symbol table is stored before or after the Sqrt function. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment 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 seventeenth entry in the symbol table, the fifth instruction of the Sqrt function, and so on.

Segmentation is a memory-management scheme that supports this user view of memory. A logical-address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specified only a single address, which was partitioned by the hardware into a page number and an offset, all invisible to the programmer.)

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, the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A Pascal compiler might create separate segments for the following:

1. the global variables;

2. the procedure call stack, to store parameters and return addresses;

3. the code portion of each procedure or function;

4. the local variables of each procedure and function.

8.5.2 Hardware

Although the user 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 addresses. This mapping is affected by a segment table. Each entry of 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, whereas the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure 8.18. 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 into 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). If this 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.

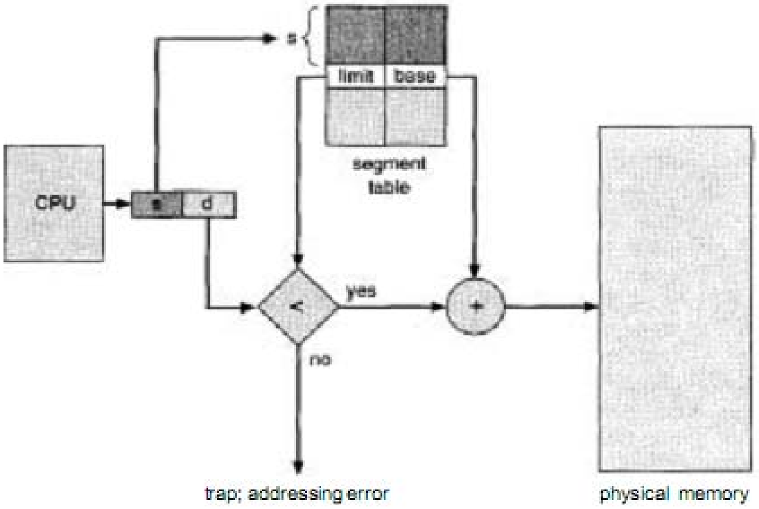
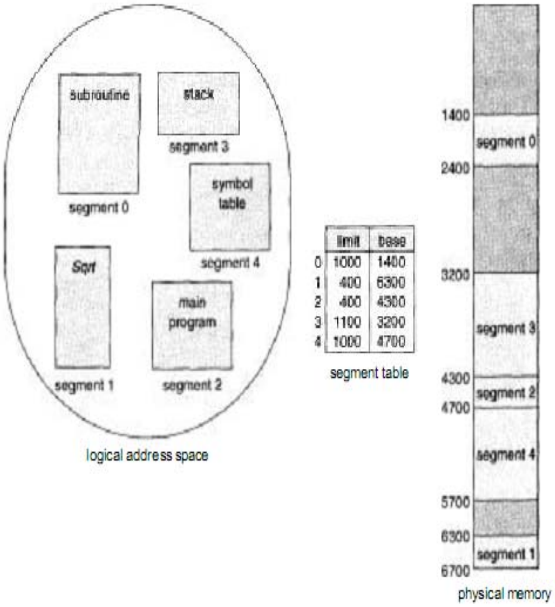


Figure 8.18Segmentation hardware.

As an example, consider the situation shown in Figure 8.19. 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.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

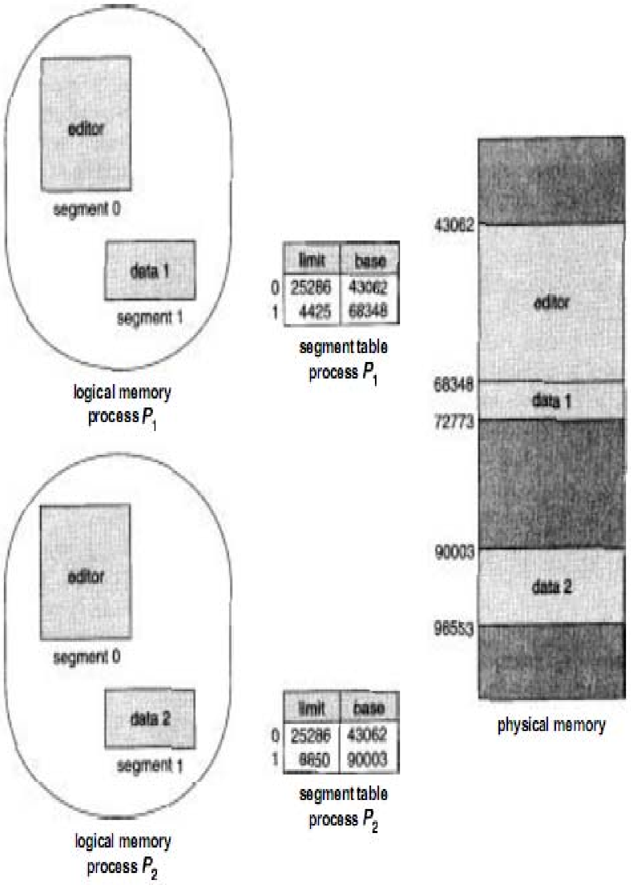
segment table

Figure 8.19 Example of segmentation.

8.5.3 Protection and Sharing

Advantage of segmentation is the association of protection with the segments. Because the segments represent a semantically defined portion of the program, it is likely that all entries in the segment will be used the same way. Hence, some segments are instructions, whereas other segments are data. In a modern architecture, instructions are non-self-modifying, so instruction segments can be defined as read only or execute only. The memory- mapping hardware will check the protection bits associated with each segment- table entry to prevent illegal accesses to memory, such as attempts to write into a read-only segment, or to use an execute-only segment as data. By placing an array in its own segment, the memory-management hardware will automatically check that array indexes are legal and do not stray outside the array boundaries. Thus, many common program errors will be detected by the hardware before they can cause serious damage.

Another advantage of segmentation involves the sharing of code or data. Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU. Segments are shared when entries in the segment tables of two different processes point to the same physical location (Figure 8.20).



	limit	base
0	25286	43062
1	4425	68348

	limit	base
0	25286	43062
1	8650	90003

Figure 8.20 Sharing of segments in a segmented memory system.

The sharing occurs at the segment level. Thus, any information can be shared if it is defined to be a segment. For example, consider the use of a text editor in a time-sharing system. A complete editor might be quite large, composed of many segments. These segments can be shared among all users, limiting the physical memory needed to support editing tasks. Rather than n copies of the editor, we need only one copy. For each user, we still need separate, unique segments to store local variables. These segments, of course, would not be shared.

We can also share only parts of programs. For example, common subroutine packages can be shared among many users if they are defined as sharable, read-only segments.

8.5.4 Fragmentation

The long-term scheduler must find and allocate memory for all the segments of a user program. This situation is similar to paging except that the segments are of variable length; pages are all the same size. Thus, as with the variable-sized partition scheme, memory allocation is a dynamic storage-allocation problem, usually solved with a best-fit or first-fit algorithm.

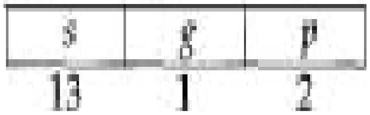
Segmentation may then cause external fragmentation, when all blocks of free memory are too small to accommodate a segment. In this case, the process may simply have to wait until more memory (or at least a larger hole) becomes available, or until compaction creates a larger hole.

How serious a problem is external fragmentation for a segmentation scheme? Would long-term scheduling with compaction help? The answers depend mainly on the average segment size. At one extreme, we could define each process to be one segment. This approach reduces to the variable-sized partition scheme. At the other extreme, every byte could be put in its own segment and relocated separately. This arrangement eliminates external fragmentation altogether; however, every byte would need a base register for its relocation, doubling memory use! Of course, the next logical step-fixed-sized, small segments-is paging.

8.6 Segmentation with Paging

Both paging and segmentation have advantages and disadvantages. In fact, of the two most popular microprocessors now being used, the Motorola 68000 line is designed based on a flat-address space, whereas the Intel 80x86 and Pentium family are based on segmentation. Both are merging memory models toward a mixture of paging and segmentation. We can combine these two methods to improve on each. This combination is best illustrated by the architecture of the Intel 386.

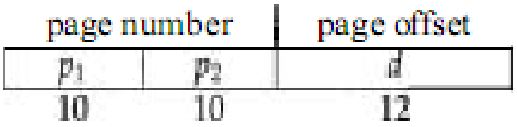
The logical-address space of a process is divided into two partitions. The first partition consists of up to 8 KB segments that are private to that process. The second partition consists of up to 8 KB segments that are shared among all the processes. Information about the first partition is kept in the local descriptor table (LDT), information about the second partition is kept in the global descriptor table (GDT). Each entry in the LDT and GDT consists of 8 bytes, with detailed information about a particular segment including the base location and length of that segment. The logical address is a pair (selector, offset), where the selector is a 16-bit number:



in which s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte (or word) within the segment in question.

The machine has six segment registers, allowing six segments to be addressed at any one time by a process. It has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or GDT. This cache lets the 386 avoid having to read the descriptor from memory for every memory reference. The physical address on the 386 is 32 bits long and is formed as follows. The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question are used to generate a linear address. First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value into a physical address.

As pointed out previously, each segment is paged, and each page is 4 KB. A page table may thus consist of up to 1 million entries. Because 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. The solution adopted in the 386 is to use a two-level paging scheme. The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit page table pointer. The logical address is as follows:



The Intel address translation is shown in more detail in Figure8.21. To improve the efficiency of physical-memory use, Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page- directory entry to indicate whether the table to which the entry is pointing is in memory or on disk. If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.

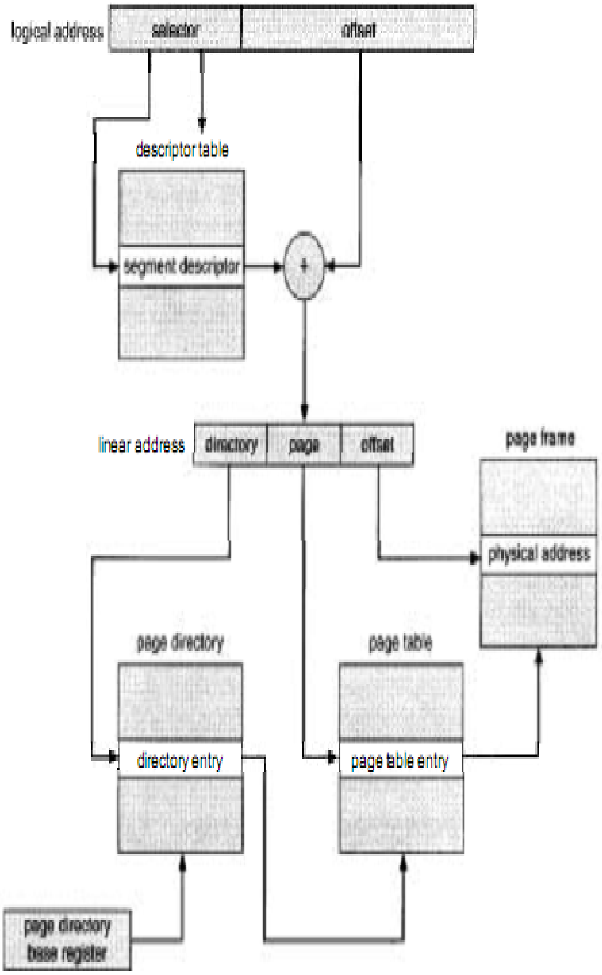


Figure 8.21 Intel 80386 address translation.

8.7 SUMMARY

Memory-management algorithms for multi-programmed operating systems range from the simple single-user system approach to paged segmentation. The greatest determinant of the method used in a particular system is the hard- ware provided. Every memory address generated by the CPU must be checked for legality and possibly mapped to a physical address. The checking cannot be implemented (efficiently) in software. Hence, we are constrained by the hardware available.

The memory-management algorithms discussed (contiguous allocation, paging, segmentation, and combinations of paging and segmentation) differ in many aspects. In comparing different memory-management strategies, you should use the following considerations:

Hardware support: A simple base register or a pair of base and limit registers is sufficient for the single- and multiple-partition schemes, whereas paging and segmentation need mapping tables to define the address map.

Performance: As the memory-management algorithm becomes more complex, the time required to map a logical address to a physical address increases. For the simple systems, we need only to compare or add to the logical address-operations that are fast. Paging and segmentation can be as fast if the table is implemented in fast registers. If the table is in memory, however, user memory accesses can be degraded substantially. A TLB can reduce the performance degradation to an acceptable level.

Fragmentation: A multiprogrammed system will generally perform more efficiently if it has a higher level of multiprogramming. For a given set of processes, we can increase the multiprogramming level only by packing more processes into memory. To accomplish this task, we must reduce memory waste or fragmentation. Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation. Systems with variable-sized allocation units, such as the multiple-partition scheme and segmentation, suffer from external fragmentation.

Relocation: One solution to the external-fragmentation problem is compaction. Compaction involves shifting a program in memory without the program noticing the change. This consideration requires that logical addresses be relocated dynamically, at execution time. If addresses are relocated only at load time, we cannot compact storage.

Swapping: Any algorithm can have swapping added to it. At intervals determined by the operating system, usually dictated by CPU-scheduling policies, processes are copied from main memory to a backing store, and later are copied back to main memory. This scheme allows more processes to be run than can be fit into memory at one time.

Sharing: Another means of increasing the multiprogramming level is to share code and data among different users. Sharing generally requires that either paging or segmentation be used, to provide small packets of information (pages or segments) that can be shared. Sharing is a means of running many processes with a limited amount of memory, but shared programs and data must be designed carefully.

Protection: If paging or segmentation is provided, different sections of a user program can be declared execute only, read only, or read-write. This restriction is necessary with shared code or data, and is generally useful in any case to provide simple run-time checks for common programming errors.

EXERCISES

8.1 Name two differences between logical and physical addresses.

8.2 Explain the difference between internal and external fragmentation.

8.3 Describe the following allocation algorithms:

   a. First fit

   b. Best fit

   c. Worst fit

8.4 When a process is rolled out of memory, it loses its ability to use the CPU (at least for a while). Describe another situation where a process loses its ability to use the CPU, but where the process does not get rolled out.

8.5 Given memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

8.6 Consider a system where a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base-limit register pairs are provided: one for instructions and one for data. The instruction base-limit register pair is automatically set to read only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

8.7 Why are page sizes always powers of 2?

8.8 Consider a logical-address space of eight pages of 1,024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are in the logical address?

b. How many bits are in the physical address?

8.9 On a system with paging, a process cannot access memory that it does not own; why? How could the operating system allow access to other memory? Why should it or should it not?

8.10 Consider a paging system with the page table stored in memory.

a. If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?

b. If we add TLBs, and 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes zero time, if the entry is there.)

8.11 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how you could use this effect to decrease the amount of time needed to copy a large amount of memory from one place to another. What would the effect of updating some byte in the one page be on the other page?

8.12 Why are segmentation and paging sometimes combined into one scheme?

8.13 Describe a mechanism by which one segment could belong to the address space of two different processes.

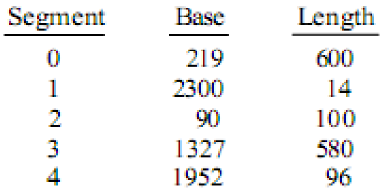
8.14 Explain why it is easier to share a reentrant module using segmentation than it is to do so when pure paging is used.

8.15 Sharing segments among processes without requiring the same segment number is possible in a dynamically linked segmentation system.

a. Define a system that allows static linking and sharing of segments without requiring that the segment numbers be the same.

b. Describe a paging scheme that allows pages to be shared without requiring that the page numbers be the same.

8.16 Consider the following segment table:



Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0430

b. 110

c. 2500

d. 3400

e. 4112

8.17 Consider the Intel address-translation scheme shown in Figure8.21.

a. Describe all the steps that are taken by the Intel 80386 in translating a logical address into a physical address.

b. What are the advantages to the operating system of hardware that provides such complicated memory-translation hardware.

c. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is it not used by every manufacturer?

8.18 In the IBM/370, memory protection is provided through the use of keys. A key is a 4-bit quantity. Each 2 KB block of memory has a key (the storage key) associated with it. The CPU also has a key (the protection key) associated with it. A store operation is allowed only if both keys are equal, or if either is zero. Which of the following memory-management schemes could be used successfully with this hardware?

a. Bare machine

b. Single-user system

c. Multiprogramming with a fixed number of processes

d. Multiprogramming with a variable number of processes

e. Paging

f. Segmentation

UNIT – III

9. VIRTUAL MEMORY

9.1 Background

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is hat programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to easily share files and address spaces, and it provides an efficient mechanism for process creation.

The memory-management algorithms outlined in previous Chapter are necessary because of one basic requirement: The instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Overlays and dynamic loading can help to ease this restriction, but they generally require special precautions and extra work by the programmer. This restriction seems both necessary and reasonable, but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance,

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements. An assembler symbol table may have room for 3,000 symbols, although the average program has less than 200 symbols.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have only recently been used.

Even in those cases where the entire program is needed, it may not all be needed at the same time (such is the case with overlays, for example).

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual-address space, simplifying the programming task. Q
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput, but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and the user.

Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be pro- vided for programmers when only a smaller physical memory is available (Figure 1.1). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available, or about what code can be placed in overlays;

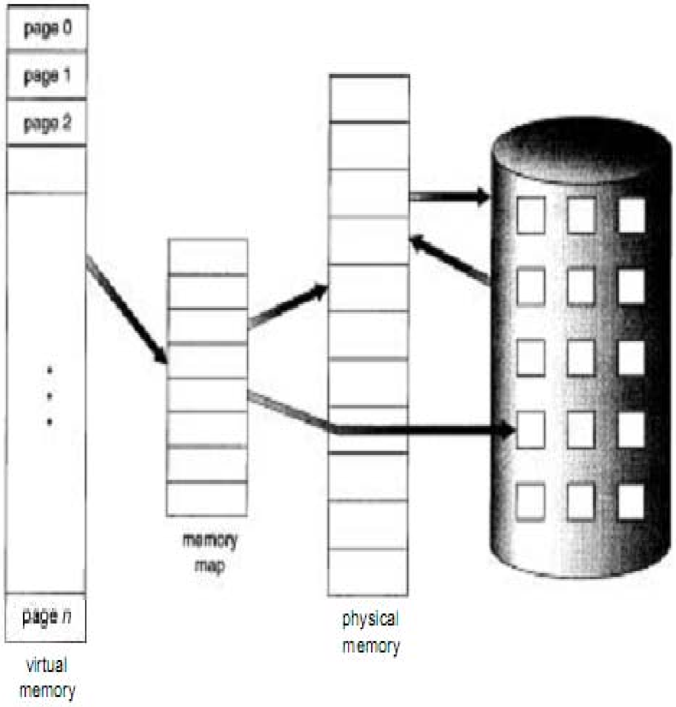


Figure 9.1 Diagram showing virtual memory that is larger than physical memory

In addition to separating logical memory from physical memory, virtual memory also allows files and memory to be shared by several different processes through page sharing. The sharing of pages further allows performance improvements during process creation.

Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Several systems provide a paged segmentation scheme, where segments are broken into pages. Thus, the user view is segmentation, but the operating system can implement this view with demand paging. Demand segmentation can also be used to provide virtual memory.

9.2 Demand Paging

A demand-paging system is similar to a paging system with swapping (Figure 9.2). Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of swap 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.

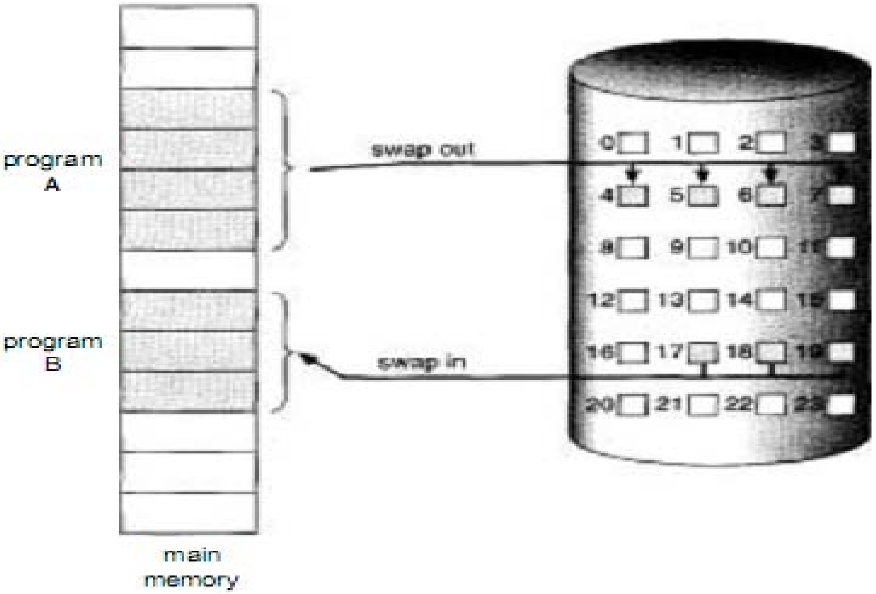


Figure 9.2 Transfer of a paged memory to contiguous disk space.

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 necessary 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 those pages that are in memory and those pages that are on the disk. The valid-invalid bit scheme described in Section 9.4.4 can be used for this purpose. This time, however, when this bit is set to "valid," this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that 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 simply marked invalid, or contains the address of the page on disk. This situation is depicted in Figure 9.3.

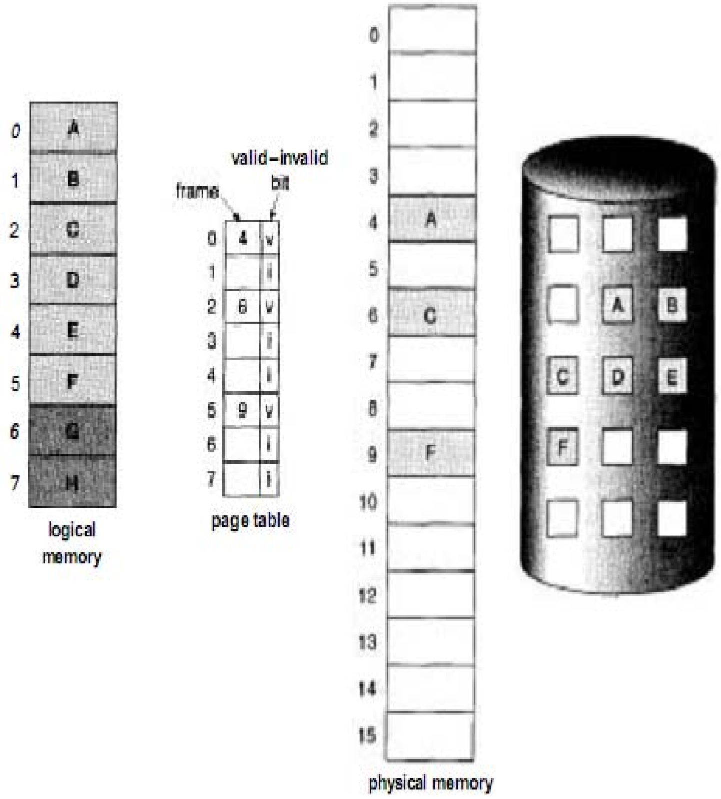


Figure 9.3 Page table when some pages are not in main memory

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 trap. 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 rather than an invalid address error as a result of an attempt to use an illegal memory address. We must therefore correct this oversight. The procedure for handling this page fault is straightforward (Figure 9.4):

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 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 illegal address trap. The process can now access the page as though it had always been in memory.

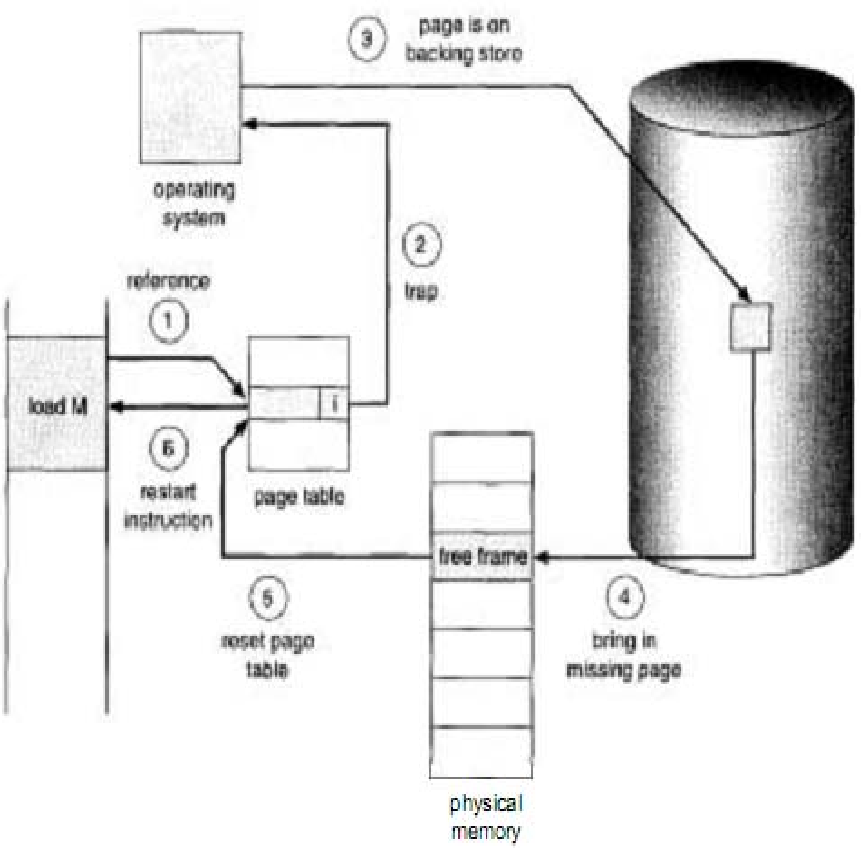


Figure 9.4 Steps in handling a page fault.

It is important to realize that, because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we can restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible. In this way, we are able to execute a process, even though portions of it are not (yet) in memory. When the process tries to access locations that are not in memory, the hardware traps to the operating system (page fault). The operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

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 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.

Performance of Demand Paging

Demand paging can have a significant effect on the performance of a computer

Demand paging can have a significant effect on the performance of a computer system. To see why, let us compute the **effective access time** for a demand- paged memory. For most computer systems, the memory-access time, denoted ma, now 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 ($0 <= p<= 1$). We would expect p to be close to zero; that is, there will be only a few page faults. The **effective access time** is then effective access time = $(1 - p) x ma + p x$ 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. Interrupt from the disk (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, then resume the interrupted instruction.

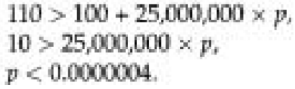
Not all of these steps are necessary in every case. In any case, we are faced with three major components of the page-fault service time:

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

If we take an average page-fault service time of 25 milliseconds and a memory-access time of 100 nanoseconds, then the effective access time in nanoseconds is

$$\begin{aligned}\text{effective access time} &= (1 - p) \times (100) + p\ (25 \text{ milliseconds})\\ &= (1 - p) \times 100 + p \times 25{,}000{,}000\\ &= 100 + 24{,}999{,}900 \times p.\end{aligned}$$

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 25 microseconds. The computer would be slowed down by a factor of 250 because of demand paging! If we want less than 10-percent degradation, we need



$$\begin{aligned}110 &> 100 + 25{,}000{,}000 \times p,\\ 10 &> 25{,}000{,}000 \times p,\\ p &< 0.0000004.\end{aligned}$$

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.

Page Replacement

The page-fault rate has not been a serious problem, because each page has faulted at most once, when it is first referenced. This representation is not strictly accurate. If a process of ten pages actually uses only one-half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had 40 frames, we could run eight processes, rather than the four that could run if each required 10 frames (five of which were never used).

If we increase our degree of multiprogramming, we are over-allocating memory. If we run six processes, each of which is ten pages in size, but actually uses only five pages, we have higher CPU utilization and throughput, with 10 frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for 60 frames, when only 40 are available. Although this situation may be unlikely, it becomes much more likely as we increase the multiprogramming level, so that the average memory usage is close to the available physical memory.

Further, consider that system memory is not used only for holding program pages. Buffers for 1/0 also consume a significant amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allow both user processes and the I/O subsystem to compete for all system memory.

Over-allocation manifests itself as follows. While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this page fault is a genuine one rather than an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds that there are no free frames on the free-frame list: All memory is in use (Figure 9.5).

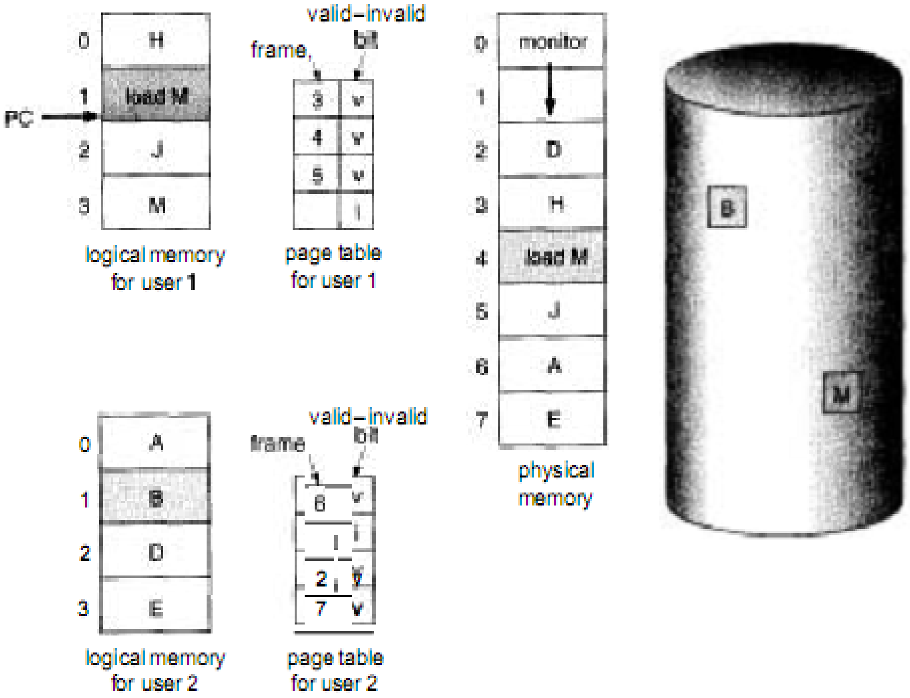


Figure 9.5 Need for page replacement

The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system-paging should be logically transparent to the user. So this option is not the best choice. The operating system could swap out a process, freeing all its frames, and reducing the level of multiprogramming. This option is a good one in certain circumstances.

Basic Scheme

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space, and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 9.6). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

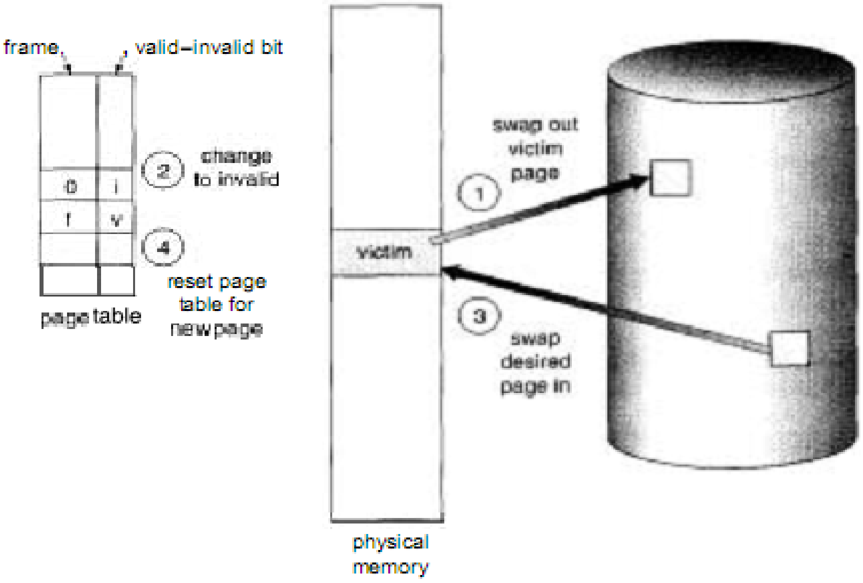


Figure 9.6 Page replacement.

1. Find the location of the desired page on the disk.

2. Find a free frame:

   a. If there is a free frame, use it.

   b. If there is no free frame, use a page-replacement algorithm to select a victim frame.

   c. Write the victim page to the disk; change the page and frame tables accordingly.

3. Read the desired page into the (newly) free frame; change the page and frame tables.

4. Restart the user process.

Notice that, if no frames are free, two page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

We can reduce this overhead by using a modify bit (or dirty bit). Each page or frame may have a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write that page to the disk. If the modify bit is not set, however, the page has not been modified since it was read into memory. Therefore, if the copy of the page on the disk has not been overwritten (by some other page, for example), then we can avoid writing the memory page to the disk: it is already there.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With non-demand paging, user addresses are mapped into physical addresses, so the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a user process of 20 pages, we can execute it in ten frames simply by using demand paging, and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: We must develop a frame-allocation algorithm and a page-replacement algorithm. If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk 1/0 is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a reference string. We can generate reference strings artificially (by a random-number generator, for example) or we can trace a given system and record the address of each memory reference. The latter choice produces a large number of data (on the order of 1 million addresses per second). To reduce the number of data, we use two facts.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address. Second, if we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105,

which, at 100 bytes per page, is reduced to the following reference string

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults, one fault for the first reference to each page. On the other hand, with only one frame available, we would have a replacement with every reference, resulting in 11 faults. In general, we expect a curve such as that in Figure 9.7. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

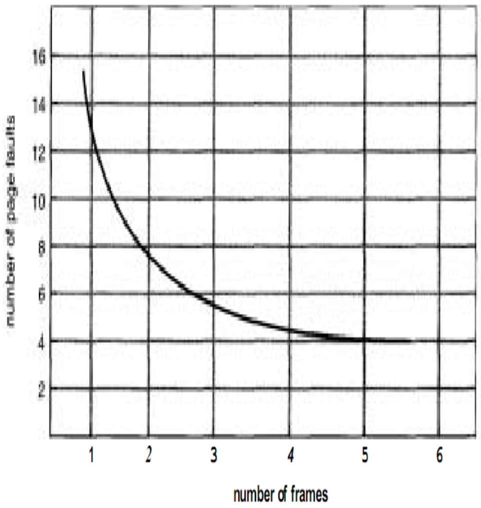


Figure 9.7 Graph of page faults versus the number of frames

To illustrate the page-replacement algorithms, we shall use the reference string

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

for a memory with three frames.

FIFO Page Replacement

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7,0,1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1, and 2) to be brought in. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 9.8. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

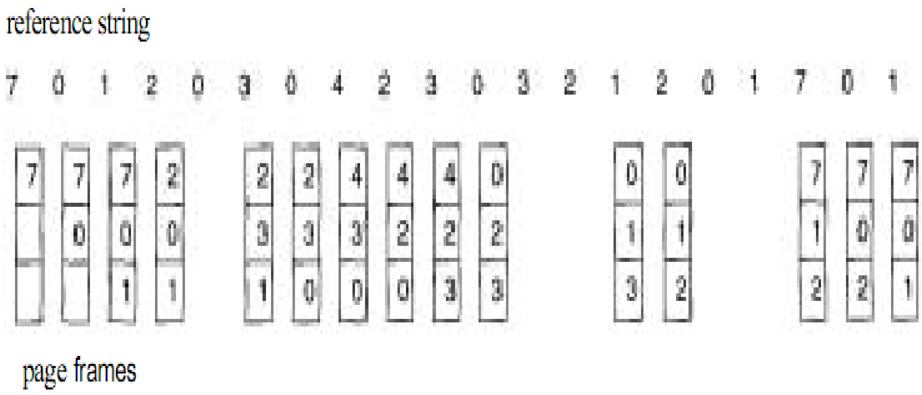


Figure 9.8 FIFO page-replacement algorithm.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution, but does not cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the reference string

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

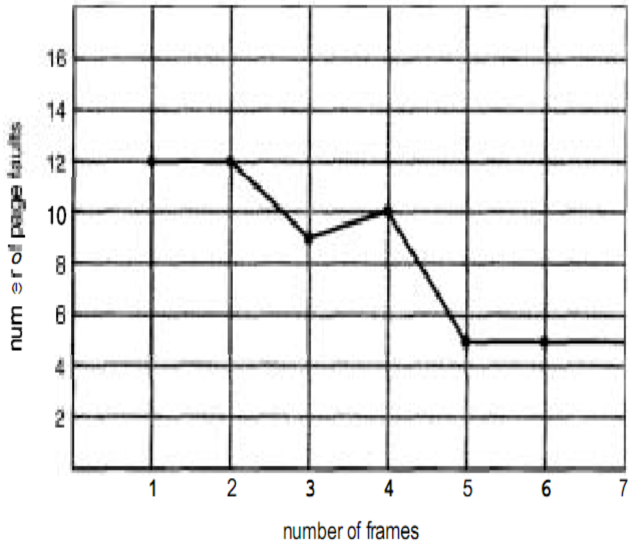


Figure 9.9 Page-fault curve for FIFO replacement on a reference string.

Figure 9.10 shows the curve of page faults versus the number of available frames. We notice that the number of faults for four frames (10) is greater than the number of faults for three frames (nine)! This most unexpected result is known as Belady's anomaly

Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms, and will never suffer from Belady's anomaly. Such an algorithm does exist, and has been called OPT or MIN. It is simply

Replace the page that will not be used
for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page- fault rate for a fixed number of frames. For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 9.10. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with less than nine faults.

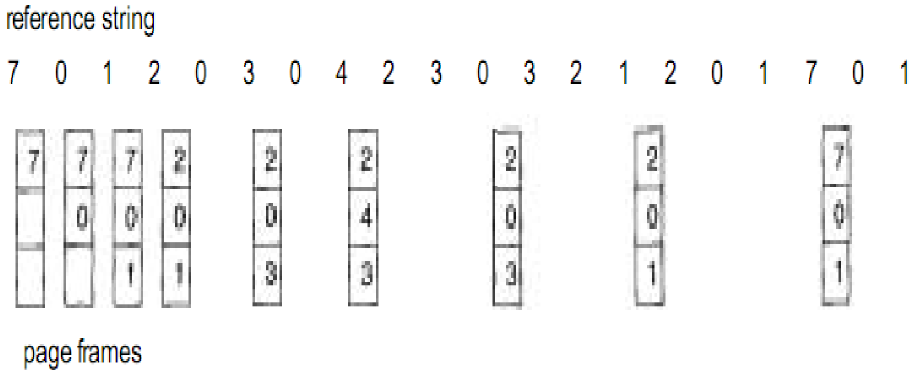


Figure 9.10 Optimal page-replacement algorithm.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU Page Replacement

If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the near future, then we will replace the page that has not been used for the longest period of time (Figure 9.11). This approach is the least-recently-used (LRU) algorithm.

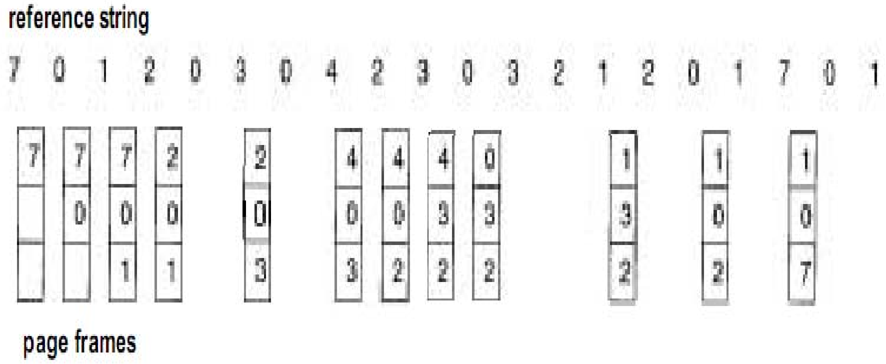


Figure 9.11 LRU page-replacement algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been use for the longest period of time. This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward.

The result of applying LRU replacement to our example reference string is shown in Figure 1.11. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0,3,4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is how to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

Counters:

In the simplest case, we associate with each page-table entry a time-of-use field, and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page, and a write to memory for each memory access. The times must also be maintained when page tables are changed. Overflow of the clock must be considered.

Stack:

Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page (Figure 9.12). Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement. Neither optimal replacement nor LRU replacement suffers from Belady's anomaly.

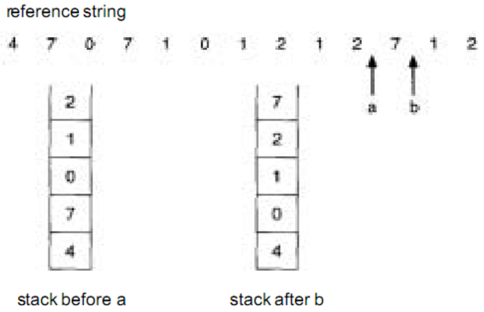


Figure 9.12 Use of a stack to record the most recent page references

LRU Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set, by the hardware, whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a user the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits. We do not know the order of use, but we know which pages were used and which were not used. This partial ordering information leads to many page-replacement algorithms that approximate LRU replacement.

9.4.5.1 Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit. These &bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

A page with a history register value of 11000100 has been used more recently than has one with 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced.

9.4.5.2 Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page. If the reference bit is set to 1, however, we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages are replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance (sometimes referred to as the clock) algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 9.13). Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

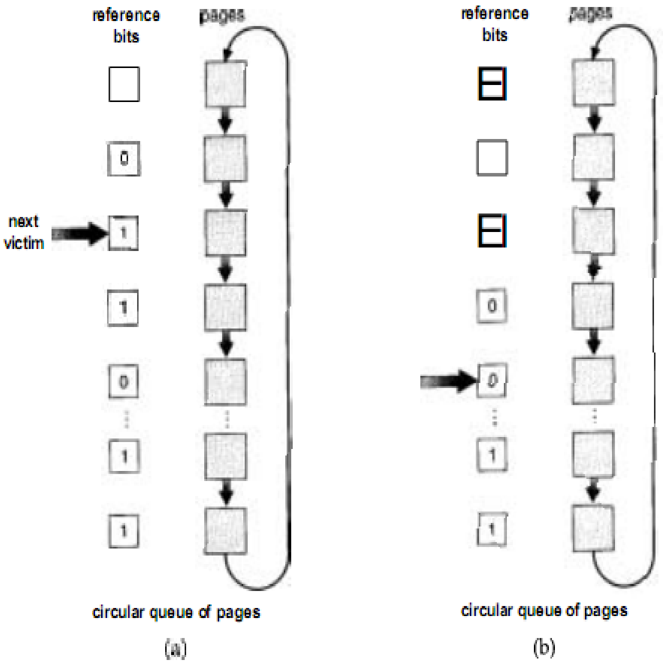


Figure 9.13 Second-chance (clock) page-replacement algorithm

9.4.5.3 Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering both the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:

1. (0,O) neither recently used nor modified-best page to replace

2. (0,l) not recently used but modified-not quite as good, because the page will need to be written out before replacement

3. (1,O) recently used but clean-it probably will be used again soon

4. (1,l) recently used and modified-it probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

When page replacement is called for, each page is in one of these four classes. We use the same scheme as the clock algorithm, but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced.

Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we could keep a counter of the number of references that have been made to each page, and develop the following two schemes.

- **The least frequently used (LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- **The most frequently used (MFU) page-replacement algorithm** is based on the argument that the page with the smallest count was probably just

Page-Buffering Algorithm

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement, and will not need to be written out.

Another modification is to keep a pool of free frames, but to remember which page was in each frame. Since the frame contents are not modified when free-frame pool if it is needed before that frame is reused. No 1/0 is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.

9.5 Allocation of Frames

How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

The simplest case of virtual memory is the single-user system. Consider a single-user system with 128 KB memory composed of pages of size 1 KB. Thus, there are 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page- replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the ninety-fourth, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

There are many variations on this simple strategy. We can require that the operating system allocate all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging. We can try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected, which is then written to the disk as the user process continues to execute.

9.5.1 Minimum Number of Frames

Our strategies for the allocations of frames are constrained in various ways. We cannot allocate more than the total number of available frames (unless there is page sharing). We must also allocate at least a minimum number of frames. Obviously, as the number of frames allocated to each process decreases, the page-fault-rate increases, slowing process execution.

Besides the undesirable performance properties of allocating only a few frames, there is a minimum number of frames that must be allocated. This minimum number is defined by the instruction-set architecture. Remember that, when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, we must have enough frames to hold all the different pages that any single instruction can reference.

For example, consider a machine in which all memory-reference instructions have only one memory address. Thus, we need at least one frame for the instruction and one frame for the memory reference. In addition, if one level indirect addressing is allowed (for example, a load instruction on page 16 can refer to an address on page 0, which is an indirect reference to page 23), then paging requires at least three frames per process. Think about what might happen if a process had only two frames. The minimum number of frames is defined by the given computer architecture.

9.5.2 Allocation Algorithms

The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The leftover three frames could be used as a free-frame buffer pool. This scheme is called **equal allocation**.

An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1 KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than ten frames, so the other 21 are strictly wasted.

To solve this problem, we can use **proportional allocation**. We allocate available memory to each process according to its size. Let the size of the virtual memory for process p_i be S_i, and define

$$S = \sum s_i.$$

Then, if the total number of available frames is m, we allocate a_i frames to process p_i, where ai is approximately

$$a_i = s_i / S \times m.$$

Of course, we must adjust each a_i to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding m.

For proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

$$10/137 \times 62 \approx 4,$$
$$127/137 \times 62 \approx 57.$$

In this way, both processes share the available frames according to their "needs," rather than equally.

In both equal and proportional allocation, of course, the allocation to each process may vary according to the multiprogramming level. If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process. On the other hand, if the multiprogramming level decreases, the frames that had been allocated to the departed process can now be spread over the remaining processes.

Notice that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes.

One approach is to use a proportional allocation scheme where the ratio of frames depends not on the relative sizes of processes, but rather on the priorities of processes, or on a combination of size and priority.

9.5.3 Global Versus Local Allocation

Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: global replacement and local replacement. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

For example, consider an allocation scheme where we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of the low-priority process.

With a local replacement strategy, the number of frames allocated to a process does not change. With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it

One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process, but also on the paging behavior of other processes. Therefore, the same process may perform quite differently (taking 0.5 seconds for one execution and 10.3 seconds for the next execution) due to totally external circumstances. Such is not the case with a local replacement algorithm. Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. For its part, local replacement might hinder a process by not making available to it other, less used pages of memory. Thus, global replacement generally results in greater system throughput, and is therefore the more common method.

9.6 Thrashing

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process' execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.

In fact, look at any process that does not have "enough" frames. Although it is technically possible to reduce the number of allocated frames to the mini- mum, there is some (larger) number of pages in active use. If the process does not have this number of frames, it will quickly page fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again. The process continues to fault, replacing pages for which it then faults and brings back in right away.

This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing.

9.6.1 Cause of Thrashing

Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behavior of early paging systems.

The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages with no regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization, and increases the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults, and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred and system throughput plunges. The page- fault rate increases tremendously. As a result, the effective memory access time increases. No work is getting done, because the processes are spending all their time paging.

This phenomenon is illustrated in Figure 9.14, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

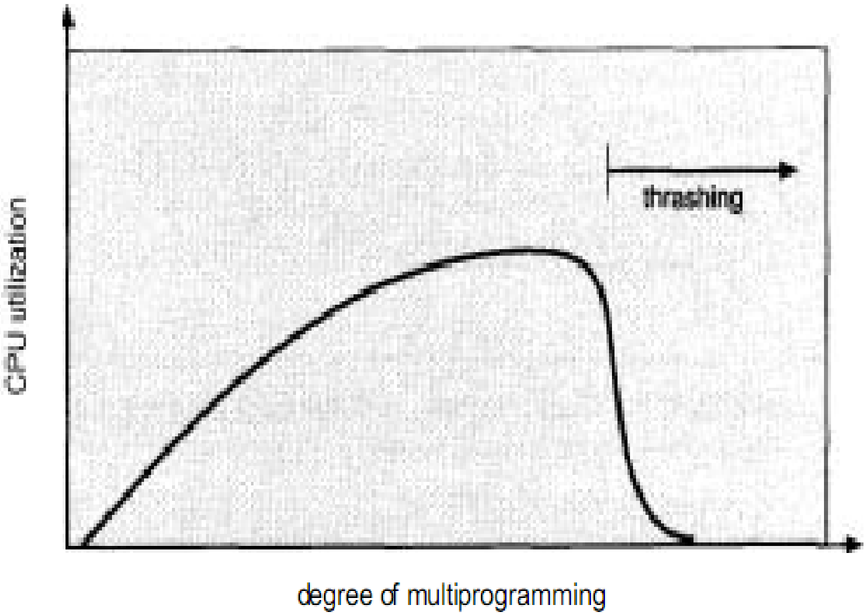
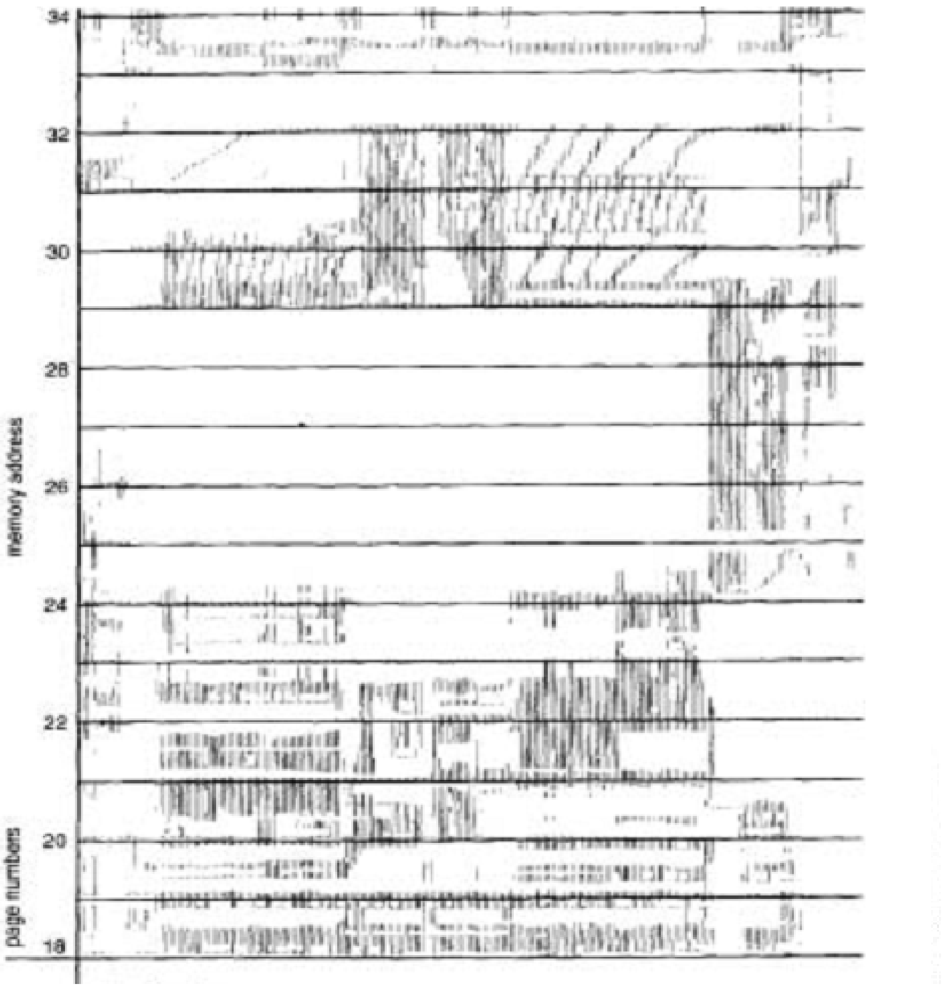


Figure 9.14 Thrashing.

We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm). With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash also. Pages are replaced with regard to the process of which they are a part.

To prevent thrashing, we must provide a process as many frames as it needs. But how do we know how many frames it "needs"? There are several techniques. The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution. The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together (Figure 9.15). A program is generally composed of several different localities, which may overlap.



Execution time-----→

Figure 9.15 Locality in a memory-reference patter

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together (Figure 9.16). A program is generally composed of several different localities, which may overlap.

For example, when a subroutine is called, it defines a new locality. In this locality, memory references are made to the instructions of the subroutine, its local variables, and a subset of the global variables. When the subroutine is exited, the process leaves this locality, since the local variables and instructions of the subroutine are no longer in active use. We may return to this locality later. Thus, we see that localities are defined by the program structure and its data structures.

9.6.2 Working-Set Model

The working-set model is based on the assumption of locality. This model uses a parameter, A, to define the working-set window. The idea is to examine the most recent A page references. The set of pages in the most recent A page references is the working set (Figure 9.16). If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set A time units after its last reference. Thus, the working set is an approximation of the program's locality.

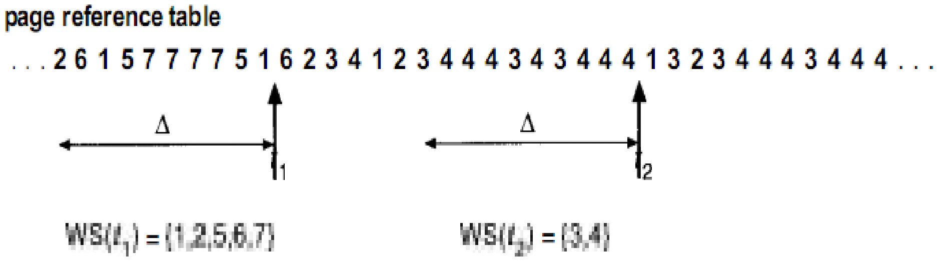


Figure 9.16 Working-set model.

For example, given the sequence of memory references shown in Figure 10.16, if A = 10 memory references, then the working set at time tl is (1, 2, 5, 6,7). By time t2, the working set has changed to {3,4). The accuracy of the working set depends on the selection of A. If A is too small, it will not encompass the entire locality; if A is too large, it may overlap several localities. In the extreme, if A is infinite, the working set is the set of pages touched during the process execution. The most important property of the working set is its size. If we compute the working-set size, WSSi, for each process in the system, we can then consider

$$D = \sum WSS_i,$$

where D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs WSS_i frames. If the total demand is greater than the total number of available frames (D > m), thrashing will occur, because some processes will not have enough frames.

9.6.3 Page-Fault Frequency

The working-set model is successful, and knowledge of the working set can be useful for prepaging but it seems a clumsy way to control thrashing. A strategy that uses the page-fault frequency (PFF) takes a more direct approach. The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Similarly, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate (Figure 9.17). If the actual page-fault rate exceeds the upper limit, we allocate that process another frame; if the page-fault rate falls below the lower limit, we remove a frame from that process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

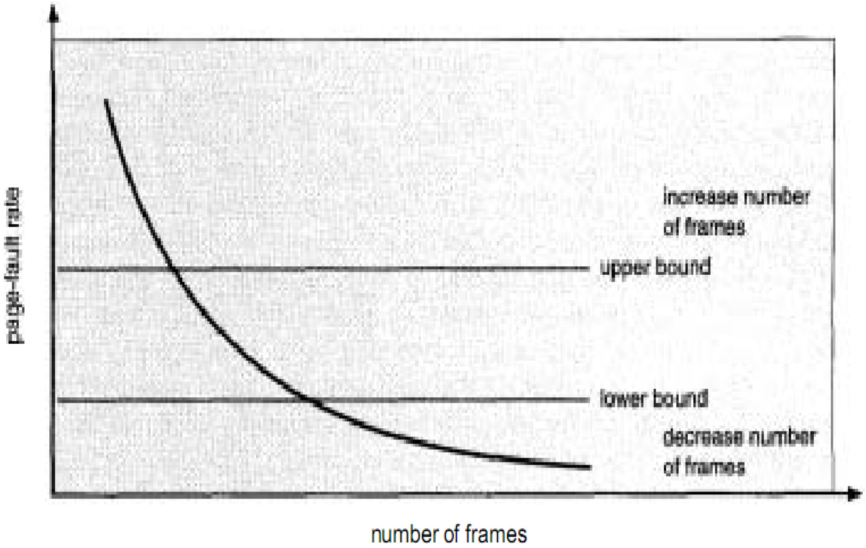


Figure 9.17 Page-fault frequency.

As with the working-set strategy, we may have to suspend a process. If the page-fault rate increases and no free frames are available, we must select some process and suspend it. The freed frames are then distributed to processes with high page-fault rates.

9.7 Other Considerations

The selections of a replacement algorithm and allocation policy are the major decisions that we make for a paging system. There are many other considerations as well.

Prepaging

An obvious property of a pure demand-paging system is the large number of page faults that occur when a process is started. This situation is a result of trying to get the initial locality into memory. The same situation may arise at other times. For instance, when a swapped-out process is restarted, all its pages are on the disk and each must be brought in by its own page fault. Prepaging is an attempt to prevent this high level of initial paging. The strategy is to bring into memory at one time all the pages that will be needed.

In a system using the working-set model, for example, we keep with each process a list of the pages in its working set. If we must suspend a process (due to an I/O wait or a lack of free frames), we remember the working set for that process. When the process is to be resumed (I/O completion or enough free frames), we automatically bring back into memory its entire working set before restarting the process. Prepaging may be an advantage in some cases. The question is simply whether the cost of using prepaging is less than the cost of servicing the corresponding page faults. It may well be the case that many of the pages brought back into memory by prepaging are not used.

Assume that s pages are prepaged and a fraction a of these s pages is actually used (0 < a < 1). The question is whether the cost of the s * a saved page faults is greater or less than the cost of prepaging s * (1 - a) unnecessary pages. If a is close to zero, prepaging loses; if a is close to one, prepaging wins.

Page Size

The designers of an operating system for an existing machine seldom have a choice concerning the page size. However, when new machines are being designed, a decision regarding the best page size must be made. As you might expect, there is no single best page size. Rather, there is a set of factors that support various sizes. Page sizes are invariably powers of 2, generally ranging from 4,096 (2^{12}) to 4,194,304 (2^{22}) bytes.

How do we select a page size? One concern is the size of the page table. For a given virtual-memory space, decreasing the page size increases the number of pages, and hence the size of the page table.

On the other hand, memory is better utilized with smaller pages. If a process is allocated memory starting at location 00000, and continuing until it has as much as it needs, it probably will not end exactly on a page boundary. Thus, a part of the final page must be allocated (because pages are the units of allocation) but is unused (internal fragmentation).

Another problem is the time required to read or write a page. 1/0 time is composed of seek, latency, and transfer times. Transfer time is proportional to the amount transferred (that is, the page size)-a fact that would seem to argue for a small page size.

With a smaller page size, however, total 1/0 should be reduced, since locality will be improved. A smaller page size allows each page to match program locality more accurately.

TLB Reach

Recall the hit ratio for the TLB refers to the percentage of virtual address translations that are resolved in the TLB rather than the page table. Clearly, the hit ratio is related to the number of entries in the TLB and the way to increase the hit ratio is by increasing the number of entries in the TLB.

Related to the hit ratio is a similar metric: the TLB reach. The TLB reach refers to the amount of memory accessible from the TLB and is simply the number of entries multiplied by the page size. Ideally, the working set for a process is stored in the TLB. If not, the process will spend a considerable amount of time resolving memory references in the page table rather than TLB. If we double the number of entries in the TLB, we double the TLB reach. However, for some memory-intensive applications this may still prove insufficient for storing the working set.

Another approach for increasing the TLB reach is by either increasing the size of the page or providing multiple page sizes. If we increase the page size say from 8 KB to 32 KB-we quadruple the TLB reach.

Inverted Page Table

The purpose of inverted page table form of page management was to reduce the amount of physical memory that is needed to track virtual-to-physical address translations. We accomplish this savings by creating a table that has one entry per physical memory page, indexed by the pair <process-id, page-number> .

Because they keep information about which virtual-memory page is stored in each physical frame, inverted page tables reduce the amount of physical memory needed to store this information. However, the inverted page table no longer contains complete information about the logical address space of a process, and that information is required if a referenced page is not currently in memory. Demand paging requires this information to process page faults.

For this information to be available, an external page table (one per process) must be kept. Each such table looks like the traditional per-process page table, containing information on where each virtual page is located.

But do external page tables negate the utility of inverted page tables? Since these tables are referenced only when a page fault occurs, they do not need to be available quickly. Instead, they are themselves paged in and out of memory as necessary. Unfortunately, a page fault may now result in the virtual-memory manager causing another page fault as it pages in the external page table it needs to locate the virtual page on the backing store. This special case requires careful handling in the kernel and a delay in the page-lookup processing.

Program Structure

Demand paging is designed to be transparent to the user program. In many cases, the user is completely unaware of the paged nature of memory. In other cases, however, system performance can be improved if the user (or compiler) has an awareness of the underlying demand paging.

Let's look at a contrived but informative example. Assume that pages are 128 words in size. Consider a Java program whose function is to initialize to 0 each element of a 128 by 128 array. The following code is typical:

```
int A[] [I = new int [I28] [I28] ;

for (int j = 0; j < A.length; j++)

for (int i = 0; i < A.length; i++)

A[i][j] = 0;
```

Notice that the array is stored row major. That is, the array is stored A [0][0],A[0][1]………. , A[0][I27], . . ., A [1][0],A[1][1],…….A[l27][127]. For pages of 128 words, each row takes one page. Thus, the preceding code zeros one word in each page, then another word in each page, and so on. If the operating system allocates less than 128 frames to the entire program, then its execution will result in 128 x 128 = 16,384 page faults. Changing the code to

```
 int A [ ][ ] = new int [I28] [I28] ;

for (int i = 0; i < A.length; i++)

for (int j = 0; j < A.length; j++)

A[i][j] = 0;
```

on the other hand, zeros all the words on one page before starting the next page, reducing the number of page faults to 128.

Careful selection of data structures and programming structures can increase locality and hence lower the page-fault rate and the number of pages in the working set. A stack has good locality, since access is always made to the top.

9.7.6 Interlock

When demand paging is used, we sometimes need to allow some of the pages to be locked in memory. One such situation occurs when 1/0 is done to or from user (virtual) memory. 1/0 is often implemented by a separate 1/0 processor. For example, a magnetic-tape controller is generally given the number of bytes to transfer and a memory address for the buffer (Figure 9.18). When the transfer is complete, the CPU is interrupted.

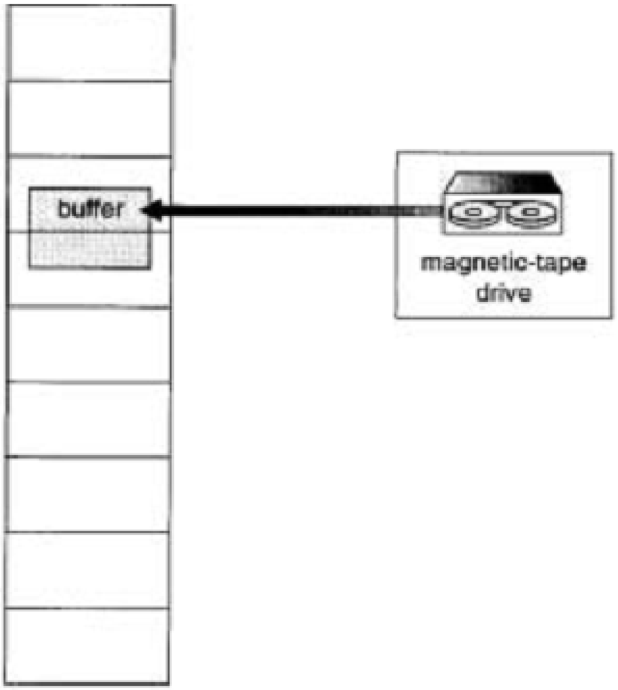


Figure 9.18 The reason why frames used for I/O must be in memory

We must be sure the following sequence of events does not occur: A process issues an 1/0 request, and is put in a queue for that 1/0 device. Meanwhile, the CPU is given to other processes. These processes cause page faults, and, using a global replacement algorithm, one of them replaces the page containing the memory buffer for the waiting process. The pages are paged out. Some time later, when the 1/0 request advances to the head of the device queue, the I/O occurs to the specified address. However, this frame is now being used for a different page belonging to another process.

There are two common solutions to this problem. One solution is never to execute I/O to user memory. Instead, data are always copied between system memory and user memory. 1/0 takes place only between system memory and the I/O device. To write a block on tape, we first copy the block to system memory, and then write it to tape.

This extra copying may result in unacceptably high overhead. Another solution is to allow pages to be locked into memory. A lock bit is associated with every frame. If the frame is locked, it cannot be selected for replacement. Under this approach, to write a block on tape, we lock into memory the pages containing the block. The system can then continue as usual. Locked pages cannot be replaced. When the I/O is complete, the pages are unlocked.

9.7.8 Real-Time Processing

Consider a real-time process or thread, Such a process expects to gain control of the CPU, and to run to completion with a minimum of delays. Virtual memory is the antithesis of real-time computing, because it can introduce unexpected long-term delays in the execution of a process while pages are brought into memory. Therefore, real-time systems almost never have virtual memory.

To solve the page- fault problem, Solaris2 allows a process to tell it which pages are important to that process. In addition to allowing "hints" on page use, the operating system allows privileged users to require pages to be locked into memory.

9.8. SUMMARY

It is desirable to be able to execute a process whose logical address space is larger than the available physical address space. The programmer can make such a process executable by restructuring it using overlays, but doing so is generally a difficult programming task. Virtual memory is a technique to allow a large logical address space to be mapped onto a smaller physical memory. Virtual memory allows extremely large processes to be run, and also allows the degree of multiprogramming to be raised, increasing CPU utilization. Further, it frees application programmers from worrying about memory availability.

Pure demand paging never brings in a page until that page is referenced. The first reference causes a page fault to the operating-system resident monitor. The operating system consults an internal table to determine where the page is located on the backing store. It then finds a free frame and reads the page in from the backing store. The page table is updated to reflect this change, and the instruction that caused the page fault is restarted. This approach allows a process to run even though its entire memory image is not in main memory at once. As long as the page-fault rate is reasonably low, performance is acceptable.

We can use demand paging to reduce the number of frames allocated to a process. This arrangement can increase the degree of multiprogramming (allowing more processes to be available for execution at one time) and-in theory, at least-the CPU utilization of the system. It also allows processes to be run even though their memory requirements exceed the total available physical memory.

Such processes run in virtual memory. If total memory requirements exceed the physical memory, then it may be necessary to replace pages from memory to free frames for new pages. Various page-replacement algorithms are used. FIFO page replacement is easy to program, but suffers from Belady's anomaly. Optimal page replacement requires future knowledge. LRU replacement is an approximation of optimal, but even it may be difficult to implement. Most page-replacement algorithms, such as the second-chance algorithm, are approximations of LRU replacement.

In addition to a page-replacement algorithm, a frame-allocation policy is needed. Allocation can be fixed, suggesting local page replacement, or dynamic, suggesting global replacement. The working-set model assumes that processes execute in localities. The working set is the set of pages in the current locality. Accordingly, each process should be allocated enough frames for its current working set.

If a process does not have enough memory for its working set, it will thrash. Providing enough frames to each process to avoid thrashing may require process swapping and scheduling.

In addition to requiring that we solve the major problems of page replacement and frame allocation, the proper design of a paging system requires that we consider page size, I/O, locking, prepaging

EXERCISES

9.1 Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

9.2 Assume that you have a page-reference string for a process with m frames (initially all empty). The page-reference string has length p; n distinct page numbers occur in it. Answer these questions for any page- replacement algorithms:

a. What is a lower bound on the number of page faults?

b. What is an upper bound on the number of page faults?

9.3 A certain computer provides its users with a virtual-memory space of 232 bytes. The computer has 218 bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how, process creation, program structure, thrashing, and other topics. Virtual memory can be thought of as one level of a hierarchy of storage levels in a computer system. Each level has its own access time, size, and cost parameters. the system establishes the corresponding physical location. Distinguish between software and hardware operations.

9.4 Which of the following programming techniques and structures are "good" for a demand-paged environment? Which are "bad"? Explain your answers.

a. Stack

b. Hashed symbol table

c. Sequential search

d. Binary search

e. Pure code

f. Vector operations

g. Indirection

9.5 Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified, and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds. Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

9.6 Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from "bad" to "perfect" according to their page-fault rate. Separate those algorithms that suffer from Belady's anomaly from those that do not.

a. LRU replacement

b. FIFO replacement

c. Optimal replacement

d. Second-chance replacement

9.7 When virtual memory is implemented in a computing system, it carries certain costs and certain benefits. List those costs and the benefits. It is possible for the costs to exceed the benefits. Explain what measures you can take to ensure that this imbalance does not occur.

9.8 An operating system supports a paged virtual memory, using a central processor with a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1,000 words, and the paging device is a drum that rotates at 3,000 revolutions per minute and transfers one million words per second. The following statistical measurements were obtained from the system: One percent of all instructions executed accessed a page other than the current page. Of the instructions that accessed another page, 80 percent accessed a page already in memory. When a new page was required, the replaced page was modified 50 percent of the time. Calculate the effective instruction time on this system, assuming that the system is running one process only, and that the processor is idle during drum transfers.

9.9 Consider a demand-paging system with the following time-measured

utilizations: CPU utilization 20%

Paging disk 97.7%

Other I/O devices 5%

For each of the following, say whether it will (or is likely to) improve CPU utilization. Explain your answers.

a. Install a faster CPU.

b. Install a bigger paging disk.

c. Increase the degree of multiprogramming.

d. Decrease the degree of multiprogramming.

e. Install more main memory.

f. Install a faster hard disk, or multiple controllers with multiple hard disks.

g. Add prepaging to the page-fetch algorithms.

h. Increase the page size.

9.10 Consider the two-dimensional array A: int A [I [I = new int [I001 [I001 ; where A [O] [O] is stored at location 200, in a paged memory system with pages of size 200. A small process resides in page 0 (locations 0 to 199) for manipulating the A matrix; thus, every instruction fetch will be from page 0. For three page frames, how many page faults are generated by the following array-initialization loops, using LRU replacement, and assuming page frame 1 has the process in it, and the other two are initially empty:

a. for (int j = 0; j < 100; j++)

for (int i = 0; i < 100; i++)

A[i][j] = 0;

b. for (int i = 0; i < 100; i++)

for (int j = 0; j < 100; j++)

A[i] [j] = 0;

9.11 Consider the following page-reference string:

1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, or seven frames? Remember that all frames are initially empty, so your first unique pages will all cost one fault each.

* LRU replacement , * FIFO replacement, * Optimal replacement

9.12 Suppose that you want to use a paging algorithm that requires a reference bit (such as second-chance replacement or working-set model), but the hardware does not provide one. Sketch how you could simulate a reference bit even if one were not provided by the hardware. Calculate the cost of doing so.

9.13 You have devised a new page-replacement algorithm that you think may be optimal. In some contorted test cases, Belady's anomaly occurs. Is the new algorithm optimal? Explain your answer.

9.14 Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discard that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?

9.15 Segmentation is similar to paging, but uses variable-sized "pages." Define two segment-replacement algorithms based on FIFO and LRU page-replacement schemes. Remember that, since segments are not the same size, the segment that is chosen to be replaced may not be big enough to leave enough consecutive locations for the needed segment. Consider strategies for systems where segments cannot be relocated, and those for systems where they can.

9.16 A page-replacement algorithm should minimize the number of page faults. We can do this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages that are associated with that frame. Then, to replace a page, we search for the page frame with the smallest counter.

a. Define a page-replacement algorithm using this basic idea. Specifically address the problems of:

i. what the initial value of the counters is,
ii. when counters are increased,
iii. when counters are decreased,
iv. how the page to be replaced is selected.

b. How many page faults occur for your algorithm for the following reference string, for four page frames?

1,2,3,4,5,3,4,1,6,7,8,7,8,9,7,8,9,5,4,5,4,2.

c. What is the minimum number of page faults for an optimal page replacement strategy for the reference string in part b with four page frames?

9.17 Consider a demand-paging system with a paging disk that has an aver- age access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference, if the page-table entry is in the associative memory. Assume that 80 percent of the accesses are in the associative memory, and that, of the remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

9.18 Consider a demand-paged computer system where the degree of multi-programming is currently fixed at four. The system was recently measured to determine utilization of CPU and the paging disk. The results are one of the following alternatives. For each case, what is happening? Can you increase the degree of multiprogramming to increase the CPU utilization? Is the paging helping in improving performance?

a. CPU utilization, 13 percent; disk utilization, 97 percent

b. CPU utilization, 87 percent; disk utilization, 3 percent

c. CPU utilization, 13 percent; disk utilization, 3 percent

9.19 We have an operating system for a machine that uses base and limit registers, but we have modified the machine to provide a page table. Can we set up the page tables to simulate base and limit registers? How can we do so, or why can we not do so?

9.20 What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

9.21 Write a program that implements the FIFO and LRU page-replacement algorithms presented in this chapter. First, generate a random page- reference string where page numbers range from 0 to 9. Apply the random page-reference string to each algorithm and record the number of page faults incurred by each algorithm. Implement the replacement algorithms so that the number of page frames can vary from 1 to 7. Assume that demand paging is used.

10. FILE SYSTEM INTERFACE

10.1 Introduction

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.

10.2 File Concept

Computers can store information on several different 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 information storage. 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 through power failures and 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 programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined structure according to its type. A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker. An executable file is a series of code sections that the loader can bring into memory and execute.

10.3 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.

10.3.1 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.

The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write 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 is depicted in Figure 10.1. Sequential access is based on a tape model of a file, and works as well on sequential-access devices as it does on random-access ones.

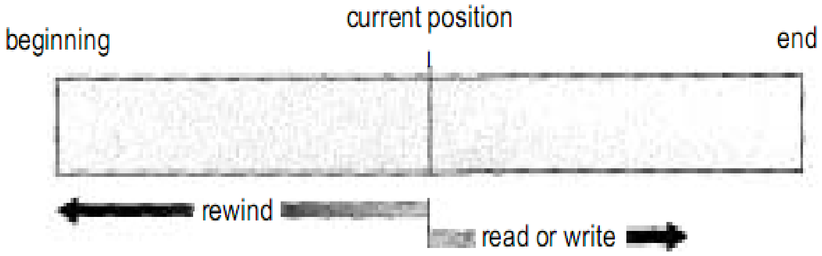


Figure 10.1 Sequential-access file.

10.3.2 Direct Access

Another method is direct access (or relative access). 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. A direct-access file allows arbitrary blocks to be read or written. 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.

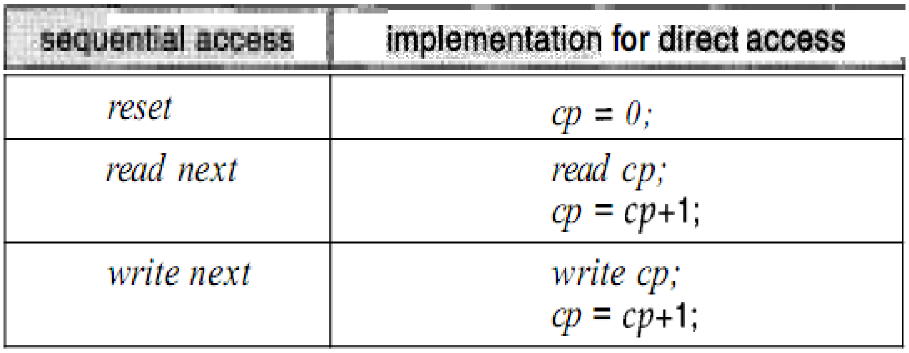
Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer, and then read that block directly to provide the desired information.

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 to n, where n is the block number. Then, to effect a read n, we would position to 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 actual absolute disk address of the block 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), and helps to prevent the user from accessing portions of the file system that may not be part of his file. Some systems start their relative block numbers at 0; others start at 1.

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. However, it is easy to simulate sequential access on a direct-access file. If we simply keep a variable cp that defines our current position, then we can simulate sequential file operations, as shown in Figure 10.2. On the other hand, it is extremely inefficient and clumsy to simulate a direct access file on a sequential-access file.



sequential access	implementation for direct access
reset	*cp = 0;*

read next	*read cp;*
	cp = cp+1;

write next	*write cp;*
	cp = cp+1;

Figure 10.3 Simulation of sequential access on a direct-access file..

10.3.3 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 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 (binary) search the index. From this search, we would know 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 would contain pointers to secondary index files, which would 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 10.3 shows a similar situation as implemented by VMS index and relative files.

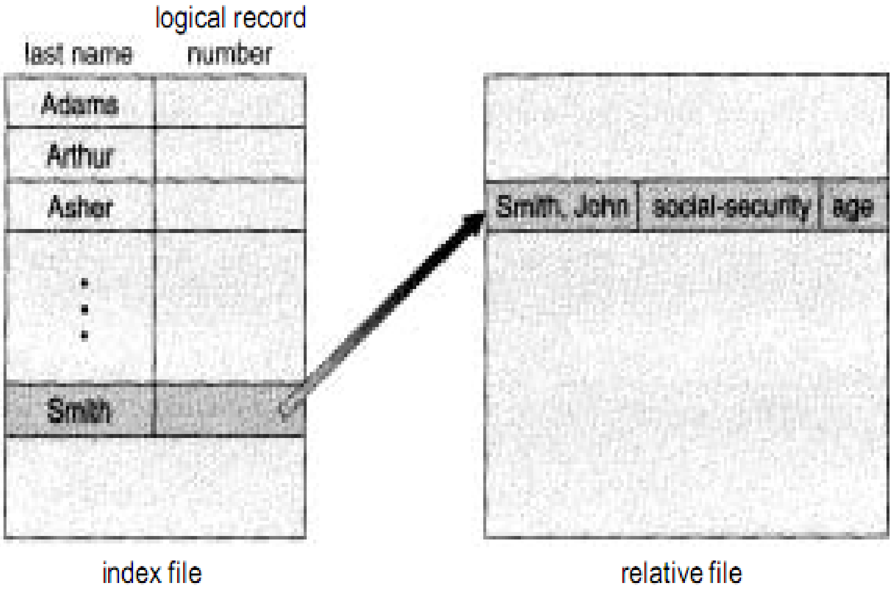


Figure 10.3 Example of index and relative files.

10.4 Directory Structure

The file systems of computers can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts. First, disks are split into one or more partitions, also known as minidisks in the IBM world or volumes in the PC and Macintosh arenas. Typically, each disk on a system contains at least one partition, which is a low-level structure in which files and directories reside.

Second, each partition contains information about files within it. This information is kept in entries in a device directory or volume table of contents. The device directory (more commonly known simply as a directory) records information-such as name, location, size, and type-for all files on that partition. Figure 10.4 shows the typical file-system organization.

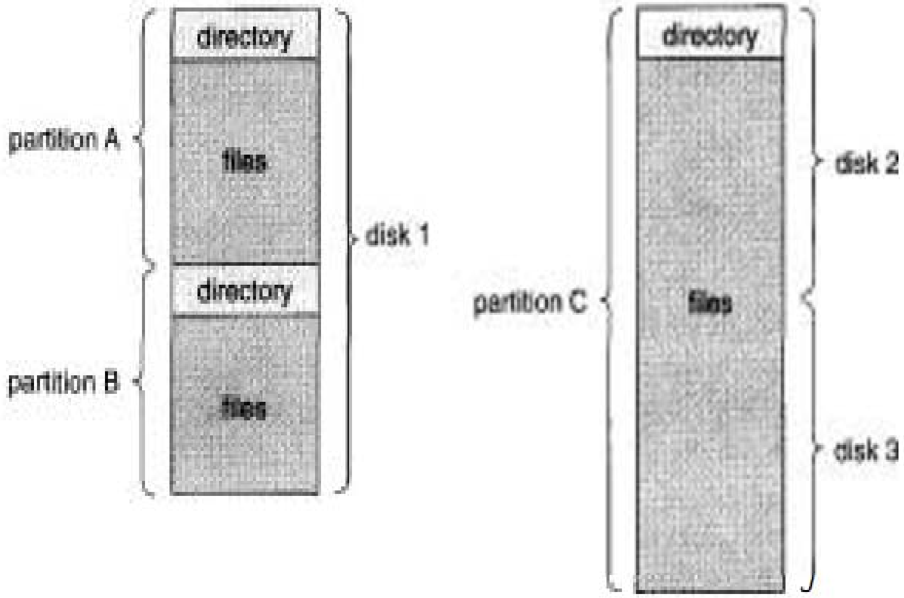


Figure 10.4 A typical file-system organization.

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. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

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 between 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 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, the name must be changeable 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. This saving often consists of copying all files to magnetic tape. This technique provides a backup copy in case of system failure or if the file is simply no longer in use. In this case, the file can be copied to tape, and the disk space of that file released for reuse by another file.

10.4.1 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 10.5).

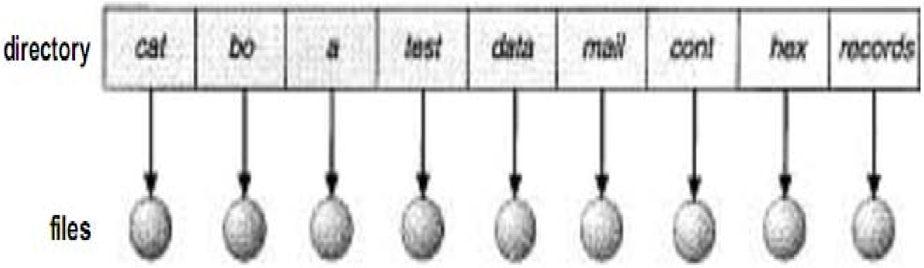


Figure 10.5 Single-level directory.

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 their data file test, then the unique-name rule is violated. For example, in one programming class, 23 students called the program for their second assignment prog2; another 11 called it assign2. Although file names are generally selected to reflect the content of the file, they are often limited in length. The MS-DOS operating system allows only 11-character file names; UNIX allows 255 characters.

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. In such an environment, keeping track of so many files is a daunting task.

10.4.2 Two-Level Directory

A single-level directory often leads to confusion of file names between different users. The standard solution is to create a separate directory for each user. In the two-level directory structure, each user has her own user file directory (UFD). Each UFD has a similar structure, but 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 10.6).

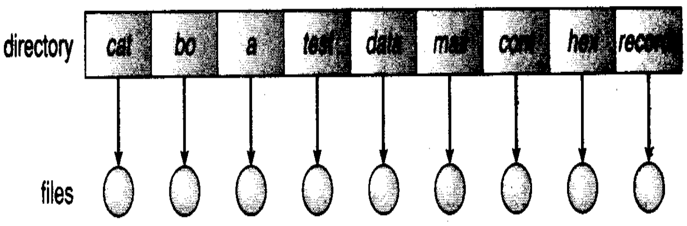


Figure 10.6

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.

The user directories themselves must be created and deleted as necessary.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. This 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 UPDs. 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,* she can simply refer to *test.* To access the file named *test of* user B (with directory-entry name *userb),* however, she might have to refer to */userb/test.* 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 MS-DOS 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. For instance, in VMS, the file *login.com* might be specified as: *u:[sst.jdeck]togin.com;l,* 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 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 case 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. As the directory system is denned presently, 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 requite 5 MB, then supporting 12 users would require *5 x 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 MS-DOS. Systems can also be designed so that each user has his own search path.

10.3.5 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 10.7). 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.

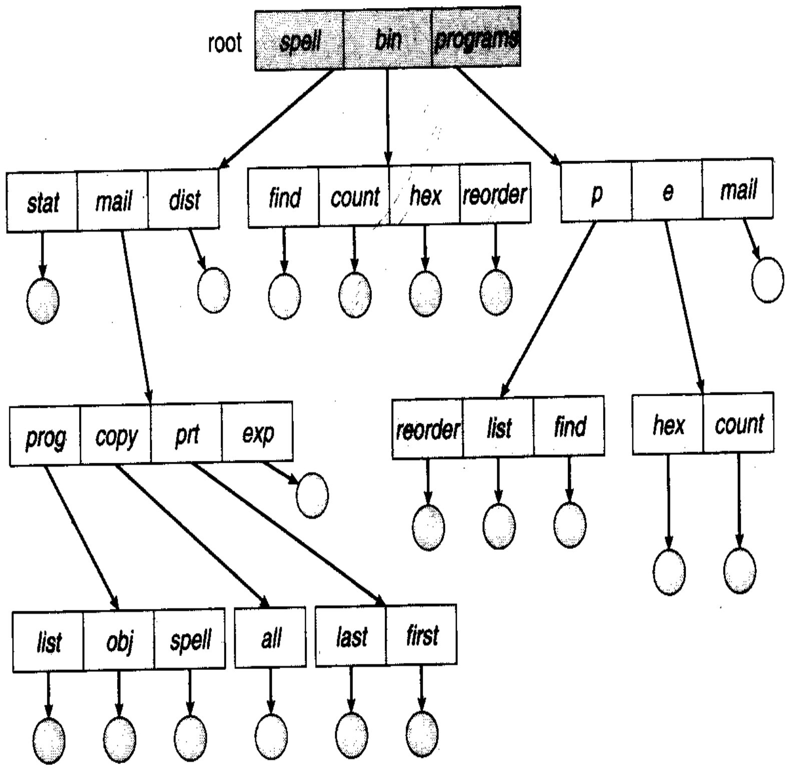


Figure 10.7 Tree-structured directory structure.

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 either specify a path name or change the current directory to be the directory holding that file. To change the directory, a system call is provided that takes the directory name as a parameter and uses it to redefine the current directory.

The initial current directory of the login shell of a user is designated when the user logs in. The operating system searches the user account file o 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.

Path names can be of two types: absolute path names or relative path names. 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 10.8, if the current directory is root/spell/mail, then the relative path name prtlfirst refers to the same file as does the absolute path name root/spell/rnail/prtlfivst.

Allowing the user to define his own subdirectories permits him to impose a structure on his 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 structure is how to handle the deletion of a directory. If a directory is empty, its entry in its containing directory 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, such as MS-DOS, 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 may result in a substantial amount of work.

An alternative approach, such as that taken by the UNIX rm command, is to provide the option that, 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 more dangerous, because an entire directory structure may be removed with one command. If that command were issued in error, a large number of files and directories would need to be restored from backup tapes.

With a tree-structured directory system, users can access, in addition to their files, the files of other users. For example, user B can access files of user A by specifying their path names. User B can specify either an absolute or a relative path name. Alternatively, user B could change her current directory to be user A's directory, and access the files by their file names. Some systems also allow users to define their own search paths. In this case, user B could define her search path to be (1) her local directory, (2) the system file directory, and (3) user A's directory, in that order. As long as the name of a file of user A did not conflict with the name of a local file or system file, it could be referred to simply by its name.

A path to a file in a tree-structured directory can be longer than that in a two-level directory. To allow users to access programs without having to remember these long paths, the Macintosh operating system automates the search for executable programs. It maintains a file, called the Desktop File, containing the name and location of all executable programs it has seen. When a new hard disk or floppy disk is added to the system, or the network is accessed, the operating system traverses the directory structure, searching for executable programs on the device and recording the pertinent information. This mechanism supports the double-click execution functionality described previously. A double-click on a file causes its creator attribute to be read, and the Desktop File to be searched for a match. Once the match is found, the appropriate executable program is started with the clicked-on file as its input. The Microsoft Windows family of operating systems (95,95, NT, 2000) maintains an extended two-level directory structure, with devices and partitions assigned a drive letter.

10.4.4 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. The common subdirectory should be shared. A shared directory or file will exist in the file system in two (or more) places at once.

A tree structure prohibits the sharing of files or directories. An acyclic graph allows directories to have shared subdirectories and files (Figure 10.8)The same file or subdirectory may be in two different directories. An acyclic graph, that is, a graph with no cycles, is a natural generalization of the tree structured directory scheme.

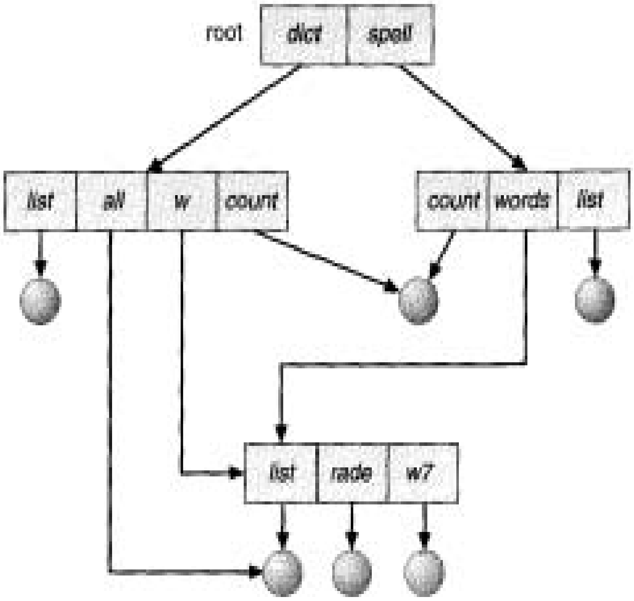


Figure 10.8 Acyclic-graph directory structure.

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 may be put into one directory. The UFDs of all the team members would each contain this directory of shared files as a subdirectory. Even when there is a single user, his file organization may require that some files be put into 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 or subdirectory

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. A 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 if the file is modified.

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 does not need to 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 also, but unless a list of the associated links is kept with each file, this search can be expensive.

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. 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.

10.4.5 General Graph Directory

One 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 to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure (Figure 10.9).

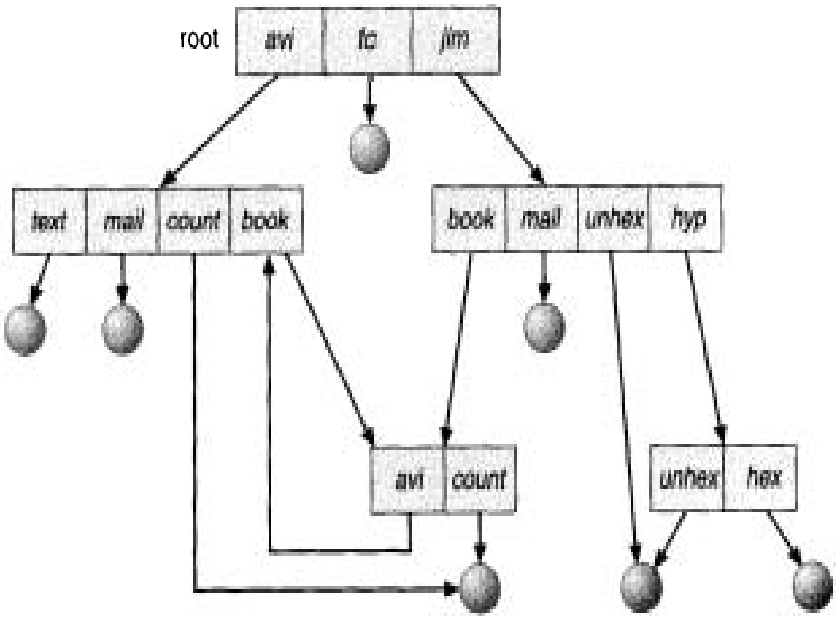


Figure 10.9 General graph directory.

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 is arbitrarily to limit 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. As with acyclic-graph directory structures, a value zero 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 be nonzero, 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.

10.5 Protection

When information is kept in a computer system, we want to keep it safe from physical damage (reliability) and improper access (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 soft- ware can also cause file contents to be lost.

10.5.1 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, or 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.

10.5.2 Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Various 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 the user name 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 needs to 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.

As an example, 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. 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 appropriate feedback.)

To achieve such a protection, we must create a new group, say text, with members Jim, Dawn, and Jill. The name of the group text must be then associated with the file book, and the access right must be set in accordance with the policy we have outlined.

10.5.3 Other Protection Approaches

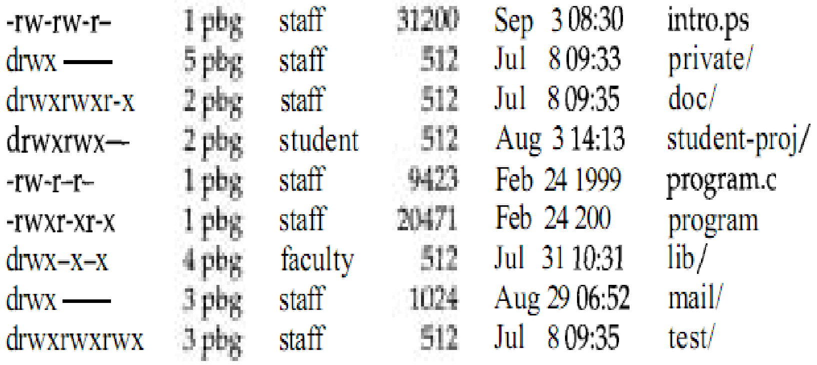
Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a pass- word, access to each file can be controlled by a password. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file to only those users who know the password. This scheme, however, has several disadvantages. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Secondly, if only one password is used for all the files, then, once it is discovered, all files are accessible. Some systems (for example, TOPS-20) allow a user to associate a password with a subdirectory, rather than with an individual file, to deal with this problem.

Limited file protection is also currently available on single user systems, such as MS-DOS and Macintosh operating system. These operating systems, when originally designed, essentially ignored the protection problem. However, since these systems are now being placed on networks where file sharing and communication are necessary, protection mechanisms must be retrofitted into the operating system. Designing a feature into a new operating system is almost always easier than adding a feature to an existing one. Such updates are usually less effective and are not seamless.

In a multilevel directory structure, we need to protect not only individual files, but also collections of files in a subdirectory; 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.

10.5.4 An Example: UNIX

In the UNIX system, directory protection is handled similarly to file protection. That is, associated with each subdirectory are three fields-owner, group, and universe- each consisting of the 3 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.



```
-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 pbg   student     512  Aug 3 14:13    student-proj/
-rw-r-r-       1 pbg   staff      9423  Feb 24 1999    program.c
-rwxr-xr-x     1 pbg   staff     20471  Feb 24 200     program
drwx-x-x       4 pbg   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/
```

Figure 10.12 A sample directory listing.

A sample directory listing from a UNIX environment is shown in Figure10.12. The first field describes the file or directory's protection. 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 unit of bytes, the creation date, and finally the file's name (with optional extension).

10.6 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 (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 tape or disk. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to block 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 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. Each user has her own directory, containing her own files. The directory lists the files by name, and includes such information as 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 his files. Acyclic-graph directory structures allow subdirectories and files to be shared, 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.

Since files are the main information-storage mechanism in most computer systems, file protection is needed. Access to files can be controlled separately for each type of access-read, write, execute, append, delete, list directory, and so on. File protection can be provided by passwords, by access lists, or by special ad hoc techniques.

EXERCISES

10.1 Consider a file system where a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?

10.2 Some systems automatically delete all user files when a user logs off or a job terminates, unless the user explicitly requests that they be kept; other systems keep all files unless the user explicitly deletes them. Discuss the relative merits of each approach.

10.3 Could you simulate a multilevel directory structure with a single-level directory structure in which arbitrarily long names can be used? If your answer is yes, explain how you can do so, and contrast this scheme with the multilevel directory scheme. If your answer is no, explain what prevents your simulation's success. How would your answer change if file names were limited to seven characters?

10.4 Explain the purpose of the open and close operations.

10.5 Some systems automatically open a file when it is referenced for the first time, and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme as compared to the more traditional one, where the user has to open and close the file explicitly.

10.6 Give an example of an application in which data in a file should be accessed in the following order:

a. Sequentially

b. Randomly

10.7 In some systems, a subdirectory can be read and written by an authorized user, just as ordinary files can be.

a. Describe the protection problems that could arise.

b. Suggest a scheme for dealing with each of the protection problems you named in part a.

10.8 Consider a system that supports 5000 users. Suppose that you want to allow 4990 of these users to be able to access one file.

a. How would you specify this protection scheme in UNIX?

b. Could you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by UNIX?

11. FILE SYSTEM STRUCTURE

11.1 File-System Structure

Disks provide the bulk of secondary storage on which a file system is maintained. They have two characteristics that make them a convenient medium for storing multiple files:

1. They can be rewritten in place; it is possible to read a block from the disk, to modify the block, and to write it back into the same place.

2. They can access directly any given block of information on the disk. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.

To provide an efficient and convenient access to the disk, the operating system imposes one or more file systems to allow the data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in Figure 11.1 is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

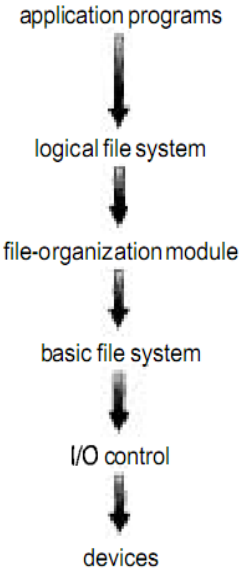


Figure 11.1 Layered file system.

The lowest level, ***the I/O control***, consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123." Its output consists of low- level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the 1/0 controller's memory to tell the controller on which device location to act and what actions to take.

The ***basic file system needs*** only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, sector 10).

The ***file-organization module*** knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer. Each file's logical blocks are numbered from 0 (or 1) through N, whereas the physical blocks containing the data usually do not match the logical numbers, so a translation is needed to locate each block. The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

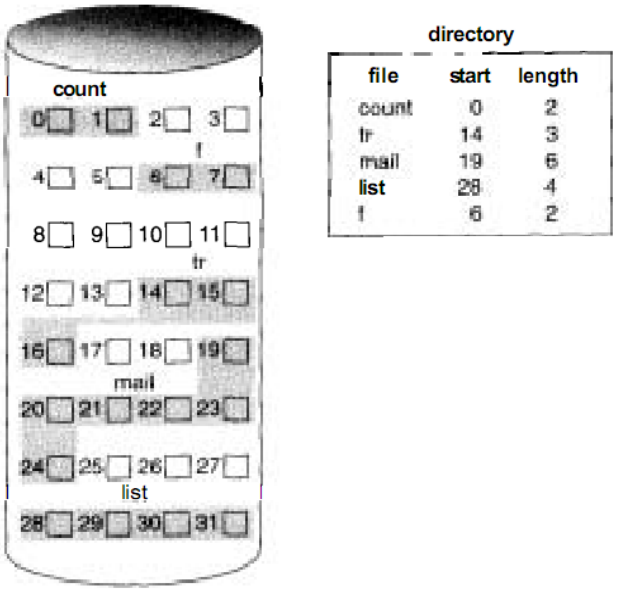
Finally, ***the logical file system*** manages metadata information. Metadata includes all of the file-system structure, excluding the actual data (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file control blocks. A file control block (FCB) contains information about the file, including ownership, permissions, and location of the file contents.

11.2 Allocation Methods

The direct-access nature of disks allows us flexibility in the implementation of files. In almost every case, many files will be stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Some systems support all three.

11.2.1 Contiguous Allocation

The contiguous-allocation method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block b + 1 after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), it is only one track. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed.



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Figure 11.2 Contiguous allocation of disk space.

Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2, ..., b + n - 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file (Figure 11.2).

Advantages:

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block b + i. Thus, both sequential and direct access can be supported by contiguous allocation.

Disadvantages:

Contiguous allocation has some problems, however. One difficulty is finding space for a new file. The implementation of the free-space-management system.

The contiguous disk-space-allocation problem can be seen to be a particular application of the general dynamic storage-allocation, which is how to satisfy a request of size n from a list of free holes. First fit and best fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization. Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster.

These algorithms suffer from the problem of external fragmentation. AS files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It request; storage is fragmented into a number of holes, no one of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created?. If we allocate too little space to a file, we may find that the file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place. Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then run the program again. These repeated runs may be costly.

The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions may be repeated as long as space exists, although it can be time-consuming.

To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme, in which a contiguous chunk of space is allocated initially, and then, when that amount is not large enough, another chunk of contiguous space, an extent, is added to the initial allocation. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.

11.2.2 Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25 (Figure 11.3). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

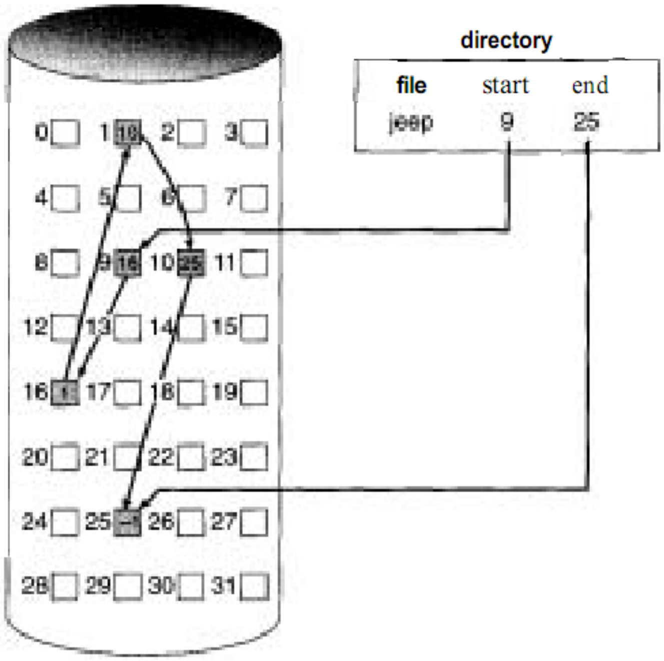


Figure 11.3 Linked allocation of disk space.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to nil (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free block to be found via the free-space-management system, and this new block is then written to, and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file does not need to be declared when that file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

Disadvantages:

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the ith block of a file, we must start at the beginning of that file, and follow the pointers until we get to the ith block. Each access to a pointer requires a disk read, and sometimes a disk seek.

Consequently, it is inefficient to support a direct-access capability for linked allocation files.

Another disadvantage to linked allocation is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it would otherwise.

The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate the clusters rather than blocks. For instance, the file system may define a cluster as 4 blocks, and operate on the disk in only cluster units. Pointers then use a much smaller percentage of the file's disk space. This method allows the logical-to-physical block mapping to remain simple, but improves disk throughput (fewer disk head seeks) and decreases the space needed for block allocation and free-list management. The cost of this approach is an increase in internal fragmentation, because more space is wasted if a cluster is partially full than when a block is partially full. Clusters can be used to improve the disk-access time for many other algorithms, so they are used in most operating systems.

Yet another problem of linked allocation is reliability. Since the files are linked together by pointers scattered all over the disk, consider what would happen if a pointer were lost or damaged. A bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could result in linking into the free-space list or into another file. Partial solutions are to use doubly linked lists or to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

An important variation on the linked allocation method is the use of a file-allocation table (FAT). This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each partition is set aside to contain the table. The table has one entry for each disk block, and is indexed by block number. The FAT is used much as is a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number then contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry, and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure of Figure 3.4 for a file consisting of disk blocks 217,618, and 339.

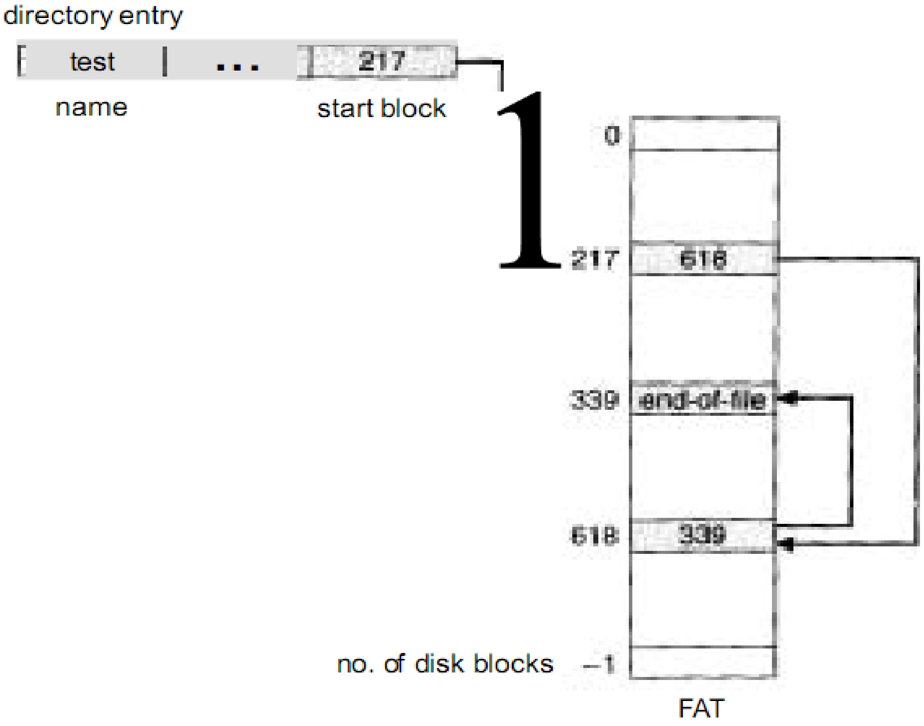


Figure 11.4 File-allocation table

The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the partition to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

11.2.3 Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order, Indexed allocation solves this problem by bringing all the pointers together into one location: the index block. Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the ith block of the file. The directory contains the address of the index block (Figure 11.5). To read the ith block, we use the pointer in the ith index-block entry to find and read the desired block.

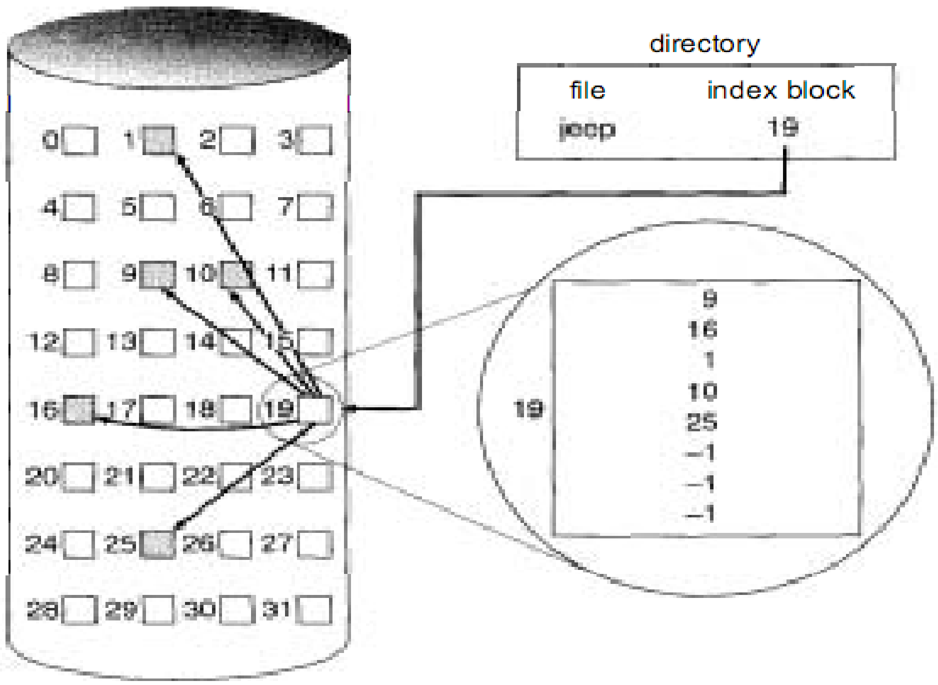


Figure 11.5 Indexed allocation of disk space

When the file is created, all pointers in the index block are set to nil. When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space.

Disadvantages:

Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block (one or two pointers). With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-varnil.

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue:

Linked scheme: An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we may link together several index blocks. For example, an index block might contain a small header giving the name of the file, and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is nil (for a small file) or is a pointer to another index block (for a large file).

Multilevel index: A variant of the linked representation is to use a first- level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block, and that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size. With 4,096-byte blocks, we could store 1,024 4-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks, which allows a file of up to 4 GB.

Combined scheme: Another alternative, used in the UFS, is to keep the first, say, 15 pointers of the index block in the file's ***inode***. The first 12 of these pointers point to ***direct blocks***; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small (no more than 12 blocks) files do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data may be accessed directly. The next 3 pointers point to indirect blocks. The first indirect block pointer is the address of a ***single indirect block***. The single indirect block is an index block, containing not data, but rather the addresses of blocks that do contain data. Then there is a ***double indirect block*** pointer, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer would contain the address of a ***triple indirect block***. An inode is shown in Figure 11.6.

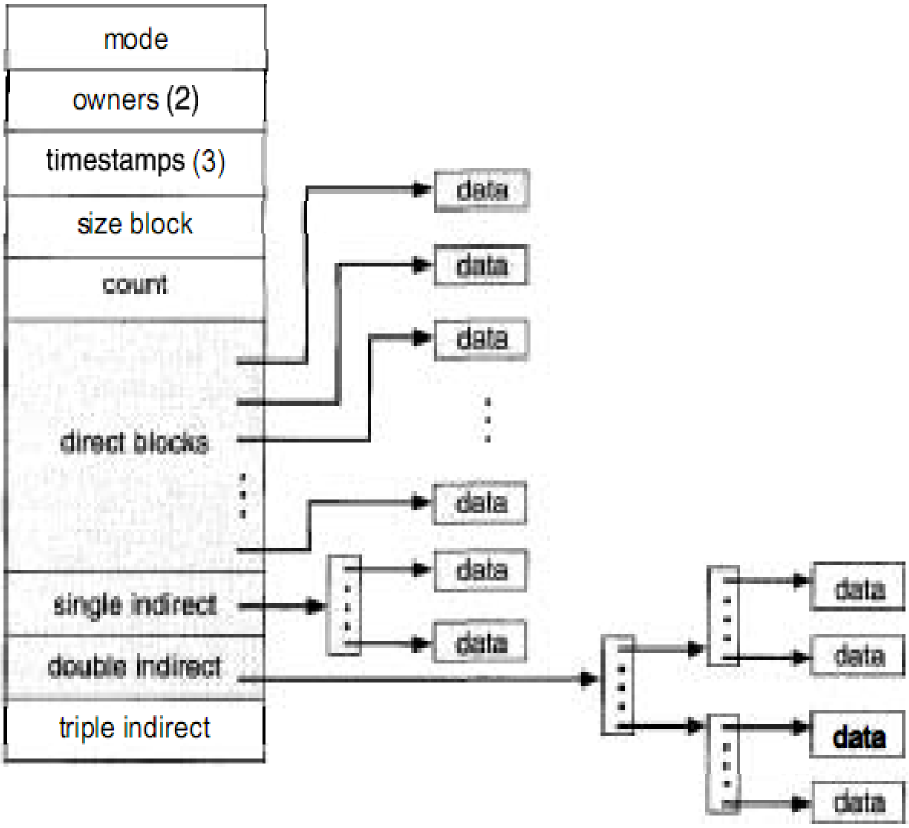


Figure 11.6 The UNIX inode

Indexed-allocation schemes suffer from some of the same performance problems as does linked allocation. Specifically, the index blocks can be cached in memory, but the data blocks may be spread all over a partition.

11.2.4 Performance

The allocation methods that we have discussed vary in their storage efficiency and data-block access times. Both are important criteria in selecting the proper method or methods for an operating system to implement.

Before selecting an allocation method, we need to determine how the systems will be used. A system with mostly sequential access should use a method different from that for a system with mostly random access. For any type of access, contiguous allocation requires only one access to get a disk block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk address of the ith block (or the next block) and read it directly.

For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access; for direct access, however, an access to the ith block might require i disk reads. This problem indicates why linked allocation should not be used for an application requiring direct access.

As a result, some systems support direct-access files by using contiguous allocation and sequential access by linked allocation. For these systems, the type of access to be made must be declared when the file is created. A file created for sequential access will be linked and cannot be used for direct access. A file created for direct access will be contiguous and can support both direct access and sequential access, but its maximum length must be declared when it is created. In this case, the operating system must have appropriate data structures and algorithms to support both allocation methods. Files can be converted from one type to another by the creation of a new file of the desired type, into which the contents of the old file are copied. The old file may then be deleted, and the new file renamed.

Indexed allocation is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block. For a two-level index, two index-block reads might be necessary. For an extremely large file, accessing a block near the end of the file would require reading in all the index blocks to follow the pointer chain before the needed data block finally could be read. Thus, the performance of indexed allocation depends on the index structure, on the size of the file, and on the position of the block desired.

Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for small files (up to three or four blocks), and automatically switching to an indexed allocation if the file grows large. Since most files are small, and contiguous allocation is efficient for small files, average performance can be quite good.

11.3 Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks only allow one write to any given sector, and thus such reuse is not physically possible.) To keep track of free disk space, the system maintains a free-space list. The free-space list records all free disk blocks-those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list.

11.3.1 Bit Vector

Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

For example, consider a disk where blocks 2, 3,4,5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free- space bit map would be

001111001111110001100000011100000 ...

The main advantage of this approach is its relatively simplicity and efficiency in finding the first free block, or n consecutive free blocks on the disk. Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose. For example, the Intel family starting with the 80386 and the Motorola family starting with the 68020 (processors that have powered PCs and Macintosh systems, respectively) have instructions that return the offset in a word of the first bit with the value 1. In fact, the Apple Macintosh operating system uses the bit-vector method to allocate disk space. To find the first free block, the Macintosh operating system checks sequentially each word in the bit map to see whether that value is not 0, since a 0-valued word has all 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

(number of bits per word) x (number of 0-value words) + offset of first 1 bit.

11.3.2 Linked List

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. In our example (Section 3.3.1), we would keep a pointer to block 2, as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on (Figure 11.7).

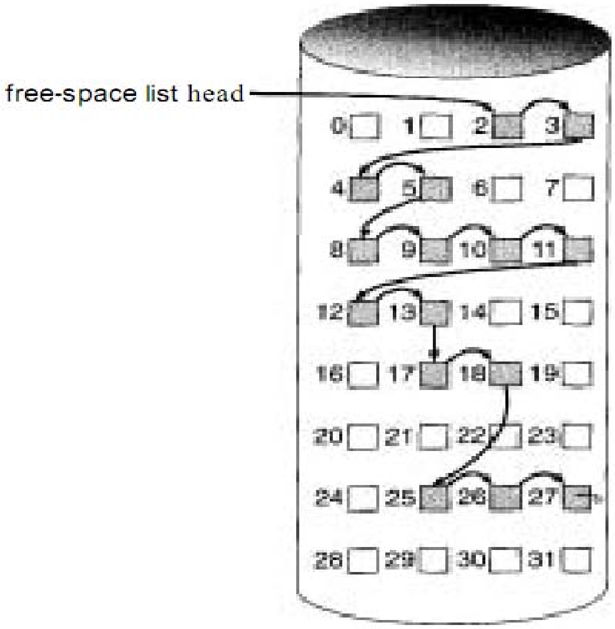


Figure 11.7 Linked free space list on disk.

However, this scheme is not efficient; to traverse the list, we must read each block, which requires substantial 1/0 time.

Fortunately, traversing the free list is not a frequent action. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed.

11.3.3 Grouping

A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first n-1 of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly, unlike in the standard linked-list approach.

11.3.4 Counting

Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number n of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

11.4 Directory Implementation

The selection of directory-allocation and directory-management algorithms has a large effect on the efficiency, performance, and reliability of the file system. Therefore, you need to understand the tradeoffs involved in these algorithms.

11.4.1 Linear List

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. A linear list of directory entries requires a linear search to find a particular entry. This method is simple to program but time-consuming to execute. To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file, then release the space allocated to it. To reuse the directory entry, we can do one of several things. We can mark the entry as unused (by assigning it a special name, such as an all-blank name, or with a used-unused bit in each entry), or we can attach it to a list of free directory entries. A third alternative is to copy the last entry in the directory into the freed location, and to decrease the length of the directory. A linked list can also be used to decrease the time to delete a file.

The real disadvantage of a linear list of directory entries is the linear search to find a file. Directory information is used frequently, and users would notice a slow implementation of access to it. In fact, many operating systems implement a software cache to store the most recently used directory information. A cache hit avoids constantly rereading the information from disk. A sorted list allows a binary search and decreases the average search time. However, the requirement that the list must be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory. A more sophisticated tree data structure, such as a B-tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

11.4.2 Hash Table

Another data structure that has been used for a file directory is a hash table. In this method, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions-situations where two file names hash to the same location. The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63, for instance, by using the remainder of a division by 64. If we later try to create a 65th file, we must enlarge the directory hash table-say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values. Alternately, a chained-overflow hash table can be used. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list. Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries, but this is likely to be much faster than a linear search through the entire directory.

11.5 Efficiency and Performance

Now that we have discussed the block-allocation and directory-management options, we can further consider their effect on performance and efficient disk use. Disks tend to be a major bottleneck in system performance, since they are the slowest main computer component. In this section, we discuss a variety of techniques used to improve the efficiency and performance of secondary storage.

11.5.1 Efficiency

The efficient use of disk space is heavily dependent on the disk allocation and directory algorithms in use. For instance, UNIX inodes are preallocated on a partition. Even an "empty" disk has a percentage of its space lost to inodes. However, by preallocating the inodes and spreading them across the partition, we improve the file system's performance. This improved performance is a result of the UNIX allocation and free-space algorithms, which try to keep a file's data blocks near that file's inode block to reduce seek time.

As another example, let us reconsider the clustering scheme, which aids in file-seek and file-transfer performance at the cost of internal fragmentation. To reduce this fragmentation, BSD UNIX varies the cluster size as a file grows. Large clusters are used where they can be filled, and small clusters are used for small files and the last cluster of a file.

The types of data normally kept in a file's directory (or inode) entry also require consideration. Commonly, a "last write date" is recorded to supply information to the user and to determine whether the file needs to be backed up. Some systems also keep a "last access date," so that a user can determine when the file was last read. The result of keeping this information is that, whenever the file is read, a field in the directory structure must be written to. This change requires the block to be read into memory, a section changed, and the block written back out to disk, because operations on disks occur only in block (or cluster) chunks. So, any time a file is opened for reading, its directory entry must be read and written as well. This requirement can be inefficient for frequently accessed files, so we must weigh its benefit against its performance cost when designing a file system. Generally, every data item associated with a file needs to be considered for its effect on efficiency and performance.

As an example, consider how efficiency is affected by the size of the pointers used to access data. Most systems use either 16- or 32-bit pointers throughout the operating system. These pointer sizes limit the length of a file to either 2^{16} (64 KB) or 2^{32} bytes (4 GB). Some systems implement 64-bit pointers to increase this limit to 2^{64} bytes, which is a very large number indeed. However, 64-bit pointers take more space to store, and in turn make the allocation and free- space-management methods (linked lists, indexes, and so on) use more disk space.

One of the difficulties in choosing a pointer size, or indeed any fixed allocation size within an operating system, is planning for the effects of changing technology. Consider that the IBM PC XT had a 10-MB hard drive and an MS-DOS file system that could support only 32 MB. (Each FAT entry was 12 bits, pointing to an 8-KB cluster.) As disk capacities increased, larger disks had to be split into 32-MB partitions, because the file system could not track blocks beyond 32 MB. As hard disks of over 100-MB capacities became common, the disk data structures and algorithms in MS-DOS had to be modified to allow larger file systems. (Each FAT entry was expanded to 16 bits, and later to 32 bits.) The initial file-system decisions were made for efficiency reasons; however, with the advent of MS-DOS Version 4, millions of computer users were inconvenienced when they had to switch to the new, larger file system.

As another example, consider the evolution of Sun's Solaris operating system. Originally, many data structures were of fixed lengths, allocated at system startup. These structures included the process table and the open-file table. When the process table became full, no more processes could be created. When the file table became full, no more files could be opened. The system would fail to provide services to the users. These table sizes could be increased only by recompiling the kernel and rebooting the system. Since the release of Solaris 2, almost all kernel structures are allocated dynamically, eliminating these artificial limits on system performance.

11.5.2 Performance

Once the basic file-system algorithms are selected, we can still improve performance in several ways. As noted in Chapter 2, most disk controllers include local memory to form an on-board cache that is sufficiently large to store entire tracks at a time. Once a seek is performed, the track is read into the disk cache starting at the sector under the disk head (alleviating latency time). The disk controller then transfers any sector requests to the operating system. Once blocks make it from the disk controller into main memory, the operating system may cache the blocks there.

Some systems maintain a separate section of main memory for a disk cache, where blocks are kept under the assumption that they will be used again shortly. Other systems cache file data using a page cache. The page cache uses virtual-memory techniques to cache file data as pages rather than as file- system-oriented blocks. Caching file data using virtual addresses is far more efficient than caching through physical disk blocks. Several systems, including Solaris, some new Linux releases, and Windows NT and 2000, use page caching to cache both process pages and file data. This is known as unified virtual memory.

Some versions of UNIX provide a unified buffer cache. Consider the two alternatives of opening and accessing a file. One approach is to use memory mapping, the second is to use the standard system calls read and write. Without a unified buffer cache, we have a situation similar to Figure 11.8.

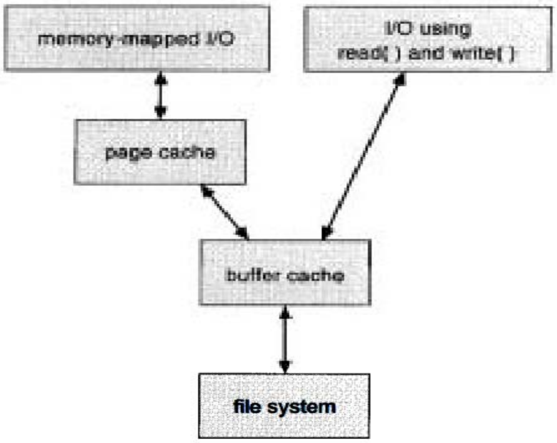


Figure 11.8 I/0 without a unified buffer cache

In this instance, the read and write system calls go through the buffer cache. The memory mapping call requires using two caches-the page cache and buffer cache. A memory mapping proceeds by reading in disk blocks from the file system and storing them in the buffer cache. Because the virtual memory system cannot interface with the buffer cache, the contents of the file in the buffer cache must be copied into the page cache. This situation is known as double caching and requires caching file-system data twice. Not only is it wasteful of memory, but it wastes significant CPU and 1/0 cycles due to the extra data movement within system memory. Also, inconsistencies between the two caches can result in corrupt files. By providing a unified buffer cache, both memory mapping and the read and write system calls use the same page cache. This has the benefit of avoiding double caching and it allows the virtual memory system to manage file-system data. The unified buffer cache is shown in figure 11.9.

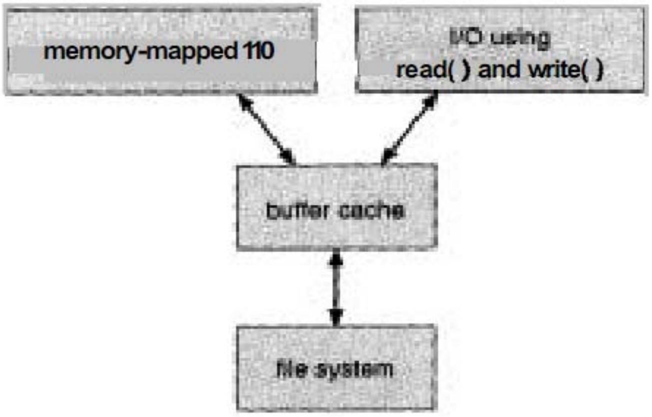


Figure 11.9 I/O using a unified buffer cache

Regardless of whether we are caching disk blocks or pages, LRU seems a reasonable general-purpose algorithm for block or page replacement. However, the evolution of the Solaris page-caching algorithms reveals the difficulty in choosing an algorithm. Solaris allows processes and the page cache to share unused memory.

Prior to Solaris 2.5.1, there was no distinction between allocating pages to a process or the page cache. As a result, a system performing many I/O operations uses most of the available memory for caching pages. Because of the high rates of I/O, the page scanner reclaims pages from processes-rather than the page cache-when free memory runs low. Solaris 2.6 and Solaris 7 optionally implemented priority paging, in which the page scanner gives priority to process pages over the page cache. Solaris 8 added a fixed limit between process pages and file-system page cache, preventing either from forcing the other out of memory.

The page cache, the file system, and the disk drivers have some interesting interactions. When data are written to a disk file, the pages are buffered in the cache, and the disk driver sorts its output queue according to disk address. These two actions allow the disk driver to minimize disk-head seeks and to write data at times optimized for disk rotation. Unless synchronous writes are required, a process writing to disk simply writes into the cache, and the system asynchronously writes the data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much nearer to asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for large transfers, counter to intuition.

Synchronous writes occur in the order in which the disk subsystem receives them, and the writes are not buffered. Thus the calling routine must wait for the data to reach the disk drive before it can proceed. Asynchronous writes are done the majority of the time. In an asynchronous write the data is stored in the cache and returns control to the caller. Metadata writes, among others, can be synchronous. Operating systems frequently include a flag in the open system call to allow a process to request that writes be performed synchronously. For example, databases use this feature for atomic transactions, to assure that data reaches stable storage in the required order.

Some systems optimize their page cache by using different replacement algorithms, depending on the access type of the file. A file being read or written sequentially should not have its pages replaced in LRU order, because the most recently used page will be used last, or perhaps never again. Instead, sequential access may be optimized by techniques known as free-behind and read-ahead. Free-behind removes a page from the buffer as soon as the next page is requested. The previous pages are not likely to be used again and waste buffer space. With read-ahead, a requested page and several subsequent pages are read and cached. These pages are likely to be requested after the current page is processed. Retrieving this data from the disk in one transfer and caching it saves a considerable amount of time. A track cache on the controller does not eliminate the need for read-ahead on a multiprogrammed system, because of the high latency and overhead of many small transfers from the track cache to main memory.

Another method of using main memory to improve performance is common on PCs. A section of memory is set aside and treated as a virtual disk (or RAM disk). In this case, a RAM-disk device driver accepts all the standard disk operations but performs those operations on the memory section, instead of on a disk. All disk operations can then be executed on this RAM disk and, except for the lightning-fast speed, users will no notice a difference. Unfortunately, RAM disks are useful only for temporary storage, since a power failure or a reboot of the system will usually erase them. Commonly, temporary files such as intermediate compiler files are stored there.

The difference between a RAM disk and a disk cache is that the contents of the RAM disk are totally user controlled, whereas those of the disk cache are under the control of the operating system. For instance, a RAM disk will stay empty until the user (or programs, at a user's direction) creates files there. Figure 11.10 shows the possible caching locations in a system.

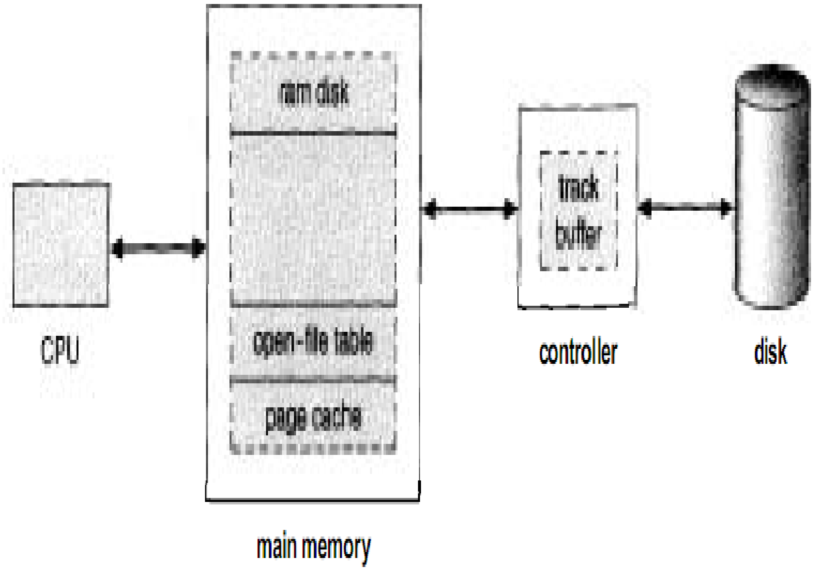


Figure 11.10 Various disk-caching locations

11.6 Recovery

Since files and directories are kept both in main memory and on disk, care must taken to ensure that system failure does not result in loss of data or in data inconsistency.

11.6.1 Consistency Checking

The part of the directory information is kept in main memory (or cache)to speed up access. The directory information in main memory is generally more up to date than is the corresponding information on the disk, because the write of cached directory information to disk does not necessarily occur as soon as the update takes place.

Consider the possible effect of a computer crash. In this case, the table of opened files is generally lost, and with it any changes in the directories of opened files. This event can leave the file system in an inconsistent state: The actual state of some files is not as described in the directory structure. Frequently, a special program is run at reboot time to check for and correct disk inconsistencies.

The **consistency checker** compares the data in the directory structure with the data blocks on disk, and tries to fix any inconsistencies it finds. The allocation and free-space-management algorithms dictate what types of problems the checker can find, and how successful it will be in fixing them.

11.6.2 Backup and Restore

Because magnetic disks sometimes fail, care must be taken to ensure that the data are not lost forever. To this end, system programs can be used to back up data from disk to another storage device, such as a floppy disk, magnetic tape, or optical disk. Recovery from the loss of an individual file, or of an entire disk, may then be a matter of restoring the data from backup.

To minimize the copying needed, we can use information from each file's directory entry. For instance, if the backup program knows when the last backup of a file was done, and the file's last write date in the directory indicates that the file has not changed since that date, then the file does not need to be copied again. A typical backup schedule may then be as follows:

Day 1: Copy to a backup medium all files from the disk. This is called a full backup.

Day 2: Copy to another medium all files changed since day 1. This is an incremental backup.

Day 3: Copy to another medium all files changed since day 2.

Day N: Copy to another medium all files changed since day N- 1. Then go back to Day 1.

The new cycle can have its backup written over the previous set, or onto a new set of backup media. In this manner, we can restore an entire disk by starting restores with the full backup, and continuing through each of the incremental backups. Of course, the larger N is, the more tapes or disks need to be read for a complete restore. An added advantage of this backup cycle is that we can restore any file accidentally deleted during the cycle by retrieving the deleted file from the backup of the previous day. The length of the cycle is a compromise between the amount of backup medium needed and the number of days back from which a restore can be done.

A user may notice that a particular file is missing or corrupted long after the damage was done. For this reason, we usually plan to take a full backup from time to time that will be saved "forever," rather than reusing that backup medium. It is a good idea to store these permanent backups far away from the regular backups to protect against hazard, such as a fire that destroys the computer and all the backups too. And if the backup cycle reuses media, one must take care not to reuse the media too many times-if the media wear out, it might not be possible to restore any data from the backups.

11.7 SUMMARY

The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently. The most common secondary-storage medium is the disk.

Physical disks may be segmented into partitions to control media use and to allow multiple, possibly varying, file systems per spindle. These file systems are mounted onto a logical file system architecture to make them available for use. File systems are often implemented in a layered or modular structure. The lower levels deal with the physical properties of storage devices. Upper levels deal with symbolic file names and logical properties of files. Intermediate levels map the logical file concepts into physical device properties.

The various files can be allocated space on the disk in three ways: through contiguous, linked, or indexed allocation. Contiguous allocation can suffer from external fragmentation. Direct access is very inefficient with linked allocation. Indexed allocation may require substantial overhead for its index block. These algorithms can be optimized in many ways. Contiguous space may be enlarged through extents to increase flexibility and to decrease external fragmentation. Indexed allocation can be done in clusters of multiple blocks to increase throughput and to reduce the number of index entries needed. Indexing in large clusters is similar to contiguous allocation with extents.

Free-space allocation methods also influence the efficiency of use of disk space, the performance of the file system, and the reliability of secondary storage. The methods used include bit vectors and linked lists. Optimizations include grouping, counting, and the FAT, which places the linked list in one contiguous area.

The directory-management routines must consider efficiency, performance, and reliability. A hash table is the most frequently used method; it is fast and efficient. Unfortunately, damage to the table or a system crash could result in the directory information not corresponding to the disk's contents. A consistency checker-a systems program such as fsck in UNIX, or chkdsk in MS-DOS-can be used to repair the damage. Operating-system backup tools allow disk data to be copied to tape, to recover from data or even disk loss due to hardware failure, operating system bug, or user error.

EXERCISES

11.1 Consider a system where free space is kept in a free-space list.

a. Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.

b. Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.

11.2 What problems could occur if a system allowed a file system to be mounted simultaneously at more than one location?

11.3 Why must the bit map for file allocation be kept on mass storage, rather than in main memory?

11.4 Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?

11.5 Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:

a. How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)

b. If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?

11.6 One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution is to define a file structure consisting of an initial contiguous area (of a specified size). If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.

11.7 Fragmentation on a storage device could be eliminated by recompaction of the information. Typical disk devices do not have relocation or base registers (such as are used when memory is to be compacted), so how can we relocate files? Give three reasons why recompacting and relocation of files are often avoided.

11.8 How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?

11.9 In what situations would using memory as a RAM disk be more useful than using it as a disk cache?

11.10 Why is it advantageous to the user for an operating system to dynamically allocate its internal tables? What are the penalties to the operating system for doing so?

11.11 Explain why logging metadata updates ensures recovery of a file system after a file system crash.

11.12 Consider the following backup scheme:

Day 1: Copy to a backup medium all files from the disk.

Day 2: Copy to another medium all files changed since day 1.

Day 3: Copy to another medium all files changed since day 1.

UNIT - IV

12 I/O SYSTEMS

12.1 Overview

The control of devices connected to the computer is a major concern of operating system designers. Because I/O devices vary so widely in their function and speed (consider a mouse, a hard disk, and a CD-ROM jukebox), a variety of methods is needed to control them. These methods form the *I/O subsystem* of the kernel, which separates the rest of the kernel from the complexity of managing I/O devices. I/O-device technology exhibits two conflicting trends. On one hand, we see increasing standardization of software and hardware interfaces. This trend helps us to incorporate improved device generations into existing computers and operating systems.

On the other hand, we see an increasingly broad variety of I/O devices. Some new devices are so unlike previous devices that it is a challenge to incorporate them into our computers and operating systems. This challenge is met by a combination of hardware and software techniques. The basic I/O hardware elements, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices. To encapsulate the details and oddities of different devices, the kernel of an operating system is structured to use device-driver modules. The **device drivers** present a uniform device-access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

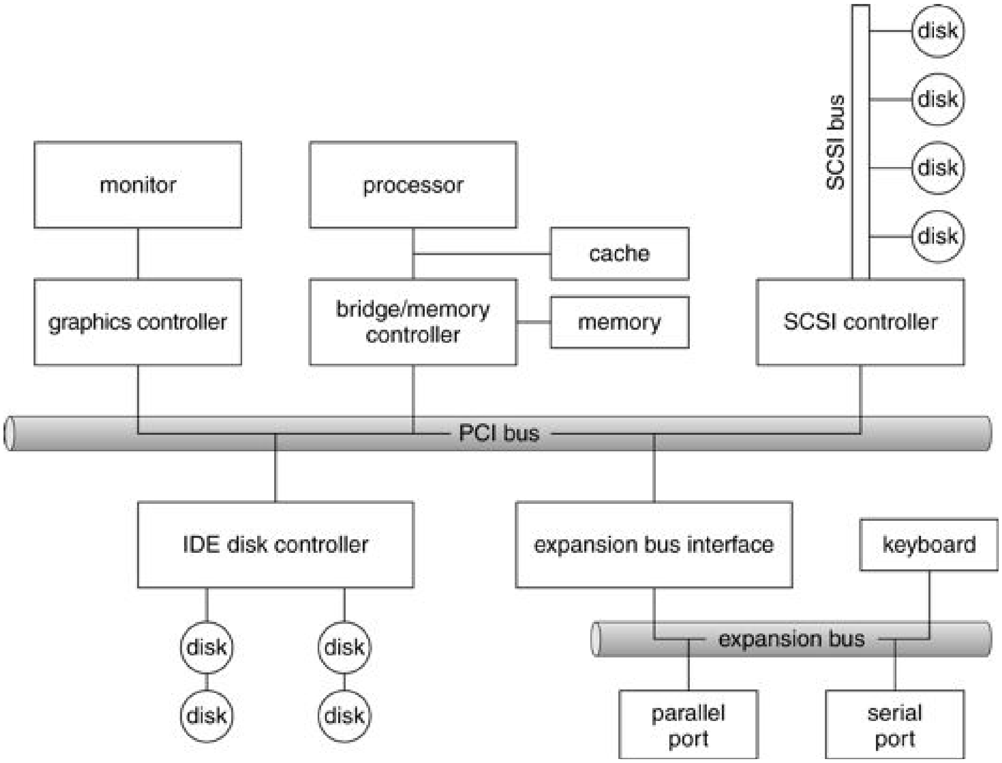
12.2 I/O Hardware

Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks, tapes), transmission devices (network cards, modems), and human-interface devices (screen, keyboard, mouse). Other devices are more specialized, such as the steering of a military fighter jet or a space shuttle. In these aircraft, a human gives input to the flight computer via a joystick, and the computer sends output commands that cause motors to move rudders, flaps, and thrusters. Despite the incredible variety of I/O devices, we need only a few concepts to understand how the devices are attached, and how the software can control the hardware.

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point (or port), for example, a serial port. If one or more devices use a common set of wires, the connection is called a *bus*. A bus is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When device *A* has a cable that plugs into device *B*, and device *B* has a cable that plugs into device *C*, and device *C* plugs into a port on the computer, this arrangement is called a daisy chain. A daisy chain usually operates as a bus.

Buses are used widely in computer architecture. Figure 12.1 shows a typical PC bus structure. This figure shows a PCI bus (the common PC system bus) that connects the processor-memory subsystem to the fast devices, and an expansion bus that connects relatively slow devices such as the keyboard and serial and parallel ports. In the upper-right portion of the figure, four disks are connected together on a SCSI bus plugged into a SCSI controller.

Figure 12.1 A typical PC bus structure.



A controller is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a host adapter) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers. If you look at a disk drive, you will see a circuit board attached to one side. This board is the disk controller. It implements the disk side of the protocol for some kind of connection, SCSI or IDE, for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.

How can the processor give commands and data to a controller to accomplish an I/O transfer? The short answer is that the controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way that this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, the device controller can support memory-mapped I/O. In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers.

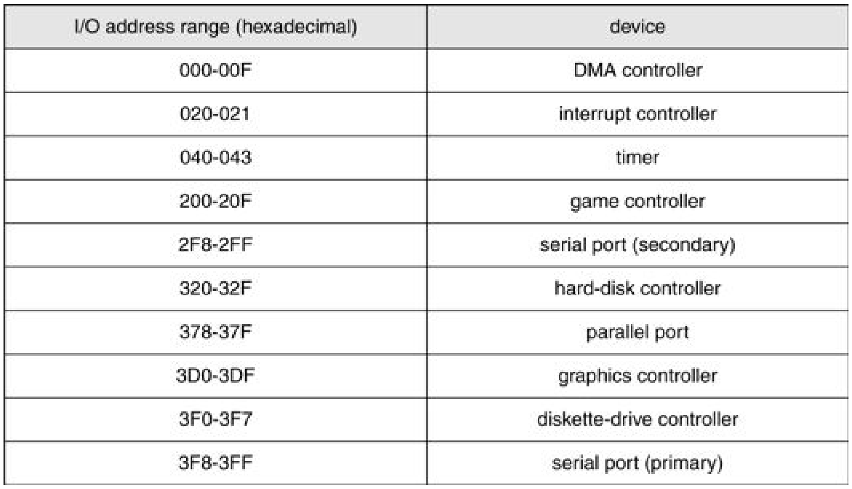
Some systems use both techniques. For instance, PCs use I/O instructions to control some devices and memory-mapped I/O to control others. Figure 12.2 shows the usual PC I/O port addresses. The graphics controller has I/O ports for basic control operations, but the controller has a large memory-mapped region to hold screen contents. The process sends output to the screen by writing data into the memory-mapped region. The controller generates the screen image based on the contents of this memory. This technique is simple to use. Moreover, writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions. But the ease of writing to a memory-mapped I/O controller is offset by a disadvantage. Because a common type of software fault is a write through an incorrect pointer to an unintended region of memory, a memory-mapped device register is vulnerable to accidental modification.

Of course, protected memory helps to reduce this risk. An I/O port typically consists of four registers, called the *status, control, data-in*, and *data-out* registers.

• The *status* register contains bits that can be read by the host. These bits indicate states such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether there has been a device error.

• The *control* register can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex communication, another enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.

Figure 12.2 Device I/O port locations on PCs (partial).



I/O address range (hexadecimal)	device
000-00F	DMA controller
020-021	interrupt controller
040-043	timer
200-20F	game controller
2F8-2FF	serial port (secondary)
320-32F	hard-disk controller
378-37F	parallel port
3D0-3DF	graphics controller
3F0-3F7	diskette-drive controller
3F8-3FF	serial port (primary)

- The *data-in* register is read by the host to get input.
- The *data-out* register is written by the host to send output.

The data registers are typically 1 to 4 bytes. Some controllers have FIFO chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

12.2.1 Polling

The complete protocol for interaction between the host and a controller can be intricate, but the basic *handshaking* notion is simple. We explain handshaking by an example. We assume that 2 bits are used to coordinate the producer-consumer relationship between the controller and the host. The controller indicates its state through the *busy* bit in the *status* register. (Recall that to *set* a bit means to write a 1 into the bit, and to *clear* a bit means to write a 0 into it.) The controller sets the *busy* bit when it is busy working, and clears the *busy* bit when it is ready to accept the next command. The host signals its wishes via the *command-ready* bit in the *command* register. The host sets the *command-ready* bit when a command is available for the controller to execute. For this example, the host writes output through a port, coordinating with the controller by handshaking as follows.

1. The host repeatedly reads the *busy* bit until that bit becomes clear.
2. The host sets the *write* bit in the *command* register and writes a byte into the *data-out* register.
3. The host sets the *command-ready* bit.
4. When the controller notices that the *command-ready* bit is set, it sets the *busy* bit.
5. The controller reads the command register and sees the write command. It reads the *data-out* register to get the byte, and does the I/O to the device.
6. The controller clears the *command-ready* bit, clears the *error* bit in the status register to indicate that the device I/O succeeded, and clears the *busy* bit to indicate that it is finished.

This loop is repeated for each byte.

In step 1, the host is busy-waiting or polling: It is in a loop, reading the *status* register over and over until the *busy* bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task. How then does the host know when the controller has become idle? For some devices, the host must service the device quickly, or data will be lost. For instance, when data are streaming in on a serial port or from a keyboard, the small buffer on the controller will overflow and data will be lost if the host waits too long before returning to read the bytes.

In many computer architectures, three CPU-instruction cycles are sufficient to poll a device: *read* a device register, *logical-and* to extract a status bit, and *branch* if not zero. Clearly, the basic polling operation is efficient. But polling becomes inefficient when it is attempted repeatedly, yet rarely finds a device to be ready for service, while other useful CPU processing remains undone. In such instances, it may be more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, rather than to require the CPU to poll repeatedly for an I/O completion. The hardware mechanism that enables a device to notify the CPU is called an **interrupt**.

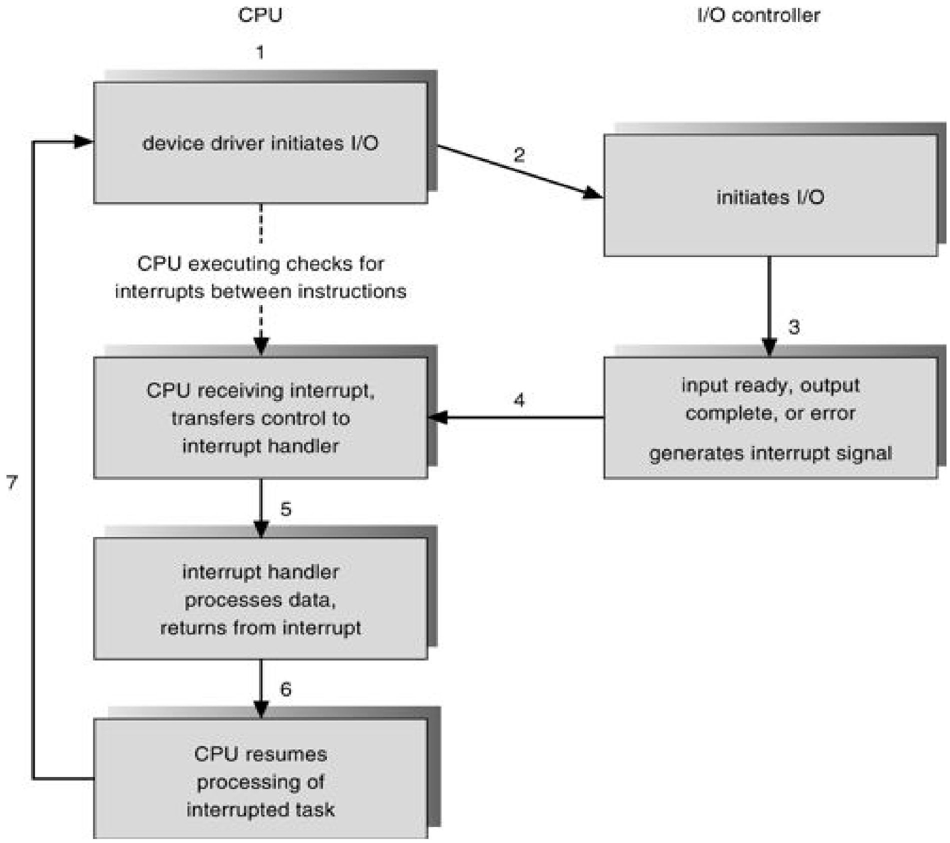
12.2.2 Interrupts

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU saves a small amount of state, such as the current value of the instruction pointer, and jumps to the **interrupt-handler** routine at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller *raises* an interrupt by asserting a signal on the interrupt request line, the CPU *catches* the interrupt and *dispatches* to the interrupt handler, and the handler *clears* the interrupt by servicing the device. Figure 12.3 summarizes the interrupt-driven I/O cycle.

This basic interrupt mechanism enables the CPU to respond to an asynchronous event, such as a device controller becoming ready for service. In a modern operating system, we need more sophisticated interrupt-handling features. First, we need the ability to defer interrupt handling during critical processing. Second, we need an efficient way to dispatch to the proper interrupt handler for a device, without first polling all the devices to see which one raised the interrupt. Third, we need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts, and can respond with the appropriate degree of urgency. In modern computer hardware, these three features are provided by the CPU and by the **interrupt-controller** hardware. Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable**: It can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

The interrupt mechanism accepts an **address**—a number that selects a specific interrupt-handling routine from a small set. In most architectures, this address is an offset in a table called the **interrupt vector**. This vector contains the memory addresses of specialized interrupt handlers. The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use the technique of **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request.

Figure 12.3 Interrupt-driven I/O cycle.

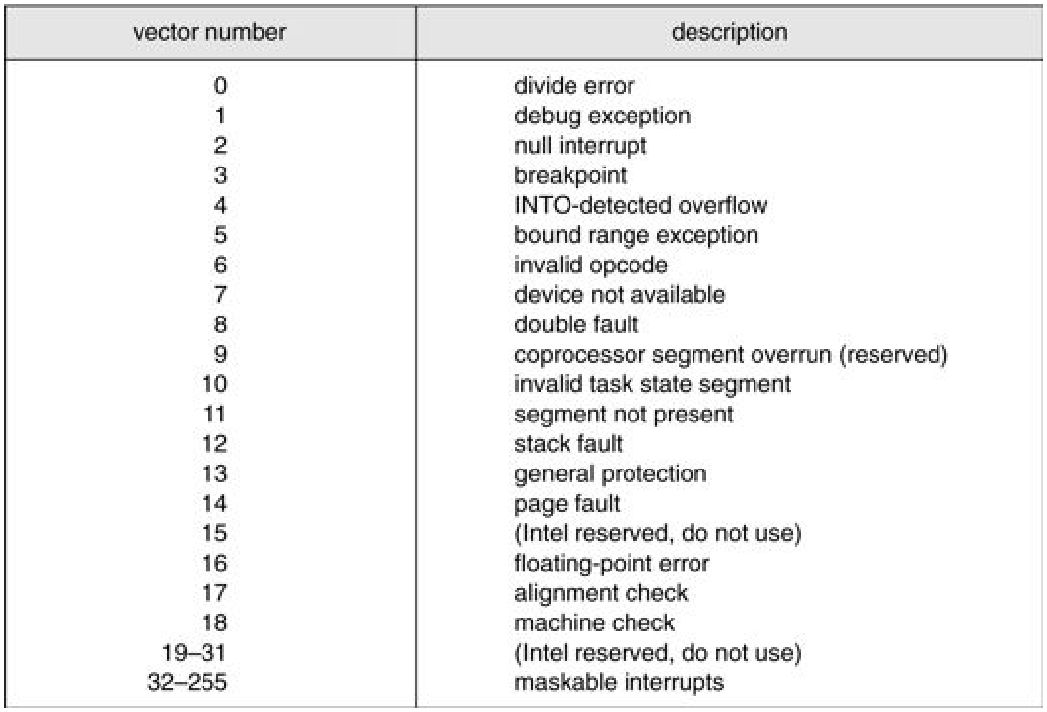

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error
generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of a dispatching to a single interrupt handler. Figure 12.4 illustrates the design of the interrupt vector for the Intel Pentium processor. The events from 0 to 31, which are non maskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of **interrupt priority levels**. This mechanism enables the CPU to defer the handling of low-priority interrupts without masking off all interrupts, and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt. A modern operating system interacts with the interrupt mechanism in several ways. At boot time, the operating system probes the hardware buses to determine what devices are present, and installs the corresponding interrupt handlers into the interrupt vector. During I/O, the various device controllers raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected. The interrupt mechanism is also used to handle a wide variety of **exceptions**, such as dividing by zero, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode.

The events that trigger interrupts have a common property: They are occurrences that induce the CPU to execute an urgent, self-contained routine. An operating system has other good uses for an efficient hardware mechanism that saves a small amount of processor state, and then calls a privileged routine in the kernel. For example, many operating systems use the interrupt mechanism for virtual-memory paging. A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt.

Figure 12.4 Intel Pentium processor event-vector table.



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Another example is found in the implementation of system calls. A *system call* is a function called by an application to invoke a kernel service. The system call checks the arguments given by the application, builds a data structure to convey the arguments to the kernel, and then executes a special instruction called a software interrupt (or a trap). This instruction has an operand that identifies the desired kernel service. When the system call executes the trap instruction, the interrupt hardware saves the state of the user code, switches to supervisor mode, and dispatches to the kernel routine that implements the requested service. The trap is given a relatively low interrupt priority compared to those assigned to device interrupts—executing a system call on behalf of an application is less urgent than servicing a device controller before its FIFO queue overflows and loses data.

Interrupts can also be used to manage the flow of control within the kernel. For example, consider the processing required to complete a disk read. One step is to copy data from kernel space to the user buffer. This copying is time consuming but not urgent—it should not block other high-priority interrupt handling. Another step is to start the next pending I/O for that disk drive. This step has higher priority: If the disks are to be used efficiently, we need to start the next I/O as soon as the previous one completes. Consequently, a *pair* of interrupt handlers implements the kernel code that completes a disk read. The high-priority handler records the I/O status, clears the device interrupt, starts the next pending I/O, and raises a low priority interrupt to complete the work. Later, when the CPU is not occupied with high-priority work, the low-priority interrupt will be dispatched.

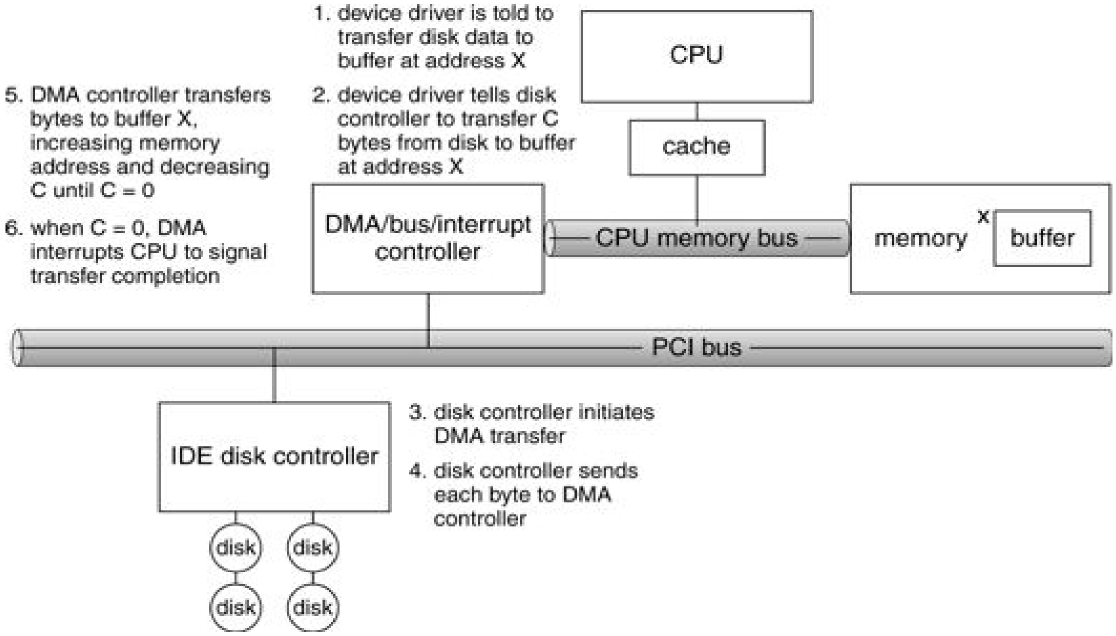
The corresponding handler completes the user-level I/O by copying data from kernel buffers to the application space, and then by calling the scheduler to place the application on the ready queue. A threaded kernel architecture is well suited to implement multiple interrupt priorities and to enforce the precedence of interrupt handling over background processing in kernel and application routines. We illustrate this point with the Solaris kernel. In Solaris, interrupt handlers are executed as kernel threads. A range of high priorities is reserved for these threads. These priorities give interrupt handlers precedence over application code and kernel housekeeping, and implement the priority relationships among interrupt handlers. The priorities cause the Solaris thread scheduler to preempt low-priority interrupt handlers in favor of higher priority ones, and the threaded implementation enables multiprocessor hardware to run several interrupt handlers concurrently. We describe the interrupt architecture of UNIX and WindowsXP in Appendices A and 21, respectively. In summary, interrupts are used throughout modern operating systems to handle asynchronous events and to trap to supervisor-mode routines in the kernel. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Device controllers, hardware faults, and system calls all raise interrupts to trigger kernel routines. Because interrupts are used so heavily for time sensitive processing, efficient interrupt handling is required for good system performance.

12.2.3 Direct Memory Access

For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor to watch status bits and to feed data into a controller register 1 byte at a time—a process termed **programmed I/O (PIO)**. Many computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a **direct-memory-access (DMA)** controller. To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred. The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. A simple DMA controller is a standard component in PCs, and **bus-mastering** I/O boards for the PC usually contain their own high-speed DMA hardware. Handshaking between the DMA controller and the device controller is performed via a pair of wires called DMA-request and DMA-acknowledge. The device controller places a signal on the DMA-request wire when a word of data is available for transfer.

This signal causes the DMA controller to seize the memory bus, to place the desired address on the memory-address wires, and to place a signal on the DMA-acknowledge wire. When the device controller receives the DMAacknowledge signal, it transfers the word of data to memory, and removes the DMA-request signal. When the entire transfer is finished, the DMA controller interrupts the CPU.

Figure 12.5 Steps in a DMA transfer.


1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
CPU
cache
DMA/bus/interrupt controller
CPU memory bus
memory
x
buffer
PCI bus
IDE disk controller
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
disk
disk
disk
disk

This process is depicted in Figure 12.5. When the DMA controller seizes the memory bus, the CPU is momentarily prevented from accessing main memory, although it can still access data items in its primary and secondary cache. Although this **cycle stealing** can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance. Some computer architectures use physical memory addresses for DMA, but others perform **direct virtual-memory access (DVMA)**, using virtual addresses that undergo virtual-to physical-memory address translation. DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.

On protected-mode kernels, the operating system generally prevents processes from issuing device commands directly. This discipline protects data from access-control violations, and also protects the system from erroneous use of device controllers that could cause a system crash. Instead, the operating system exports functions that a sufficiently privileged process can use to access low-level operations on the underlying hardware. On kernels without memory protection, processes can access device controllers directly. This direct access can be used to obtain high performance, since it can avoid kernel communication, context switches, and layers of kernel software. Unfortunately, it interferes with system security and stability. The trend in general-purpose operating systems is to protect memory and devices, so that the system can try to guard against erroneous or malicious applications.

Although the hardware aspects of I/O are complex when considered at the level of detail of electronics-hardware designers, the concepts that we have just described are sufficient to understand many I/O aspects of operating systems. Let's review the main concepts:

- A bus
- A controller
- An I/O port and its registers
- The handshaking relationship between the host and a device controller
- The execution of this handshaking in a polling loop or via interrupts
- The offloading of this work to a DMA controller for large transfers

We gave a basic example of the handshaking that takes place between a device controller and the host in Section 13.2. In reality, the wide variety of available devices poses a problem for operating-system implementers. Each kind of device has its own set of capabilities, control-bit definitions, and protocol for interacting with the host—and they are all different. How can the operating system be designed so that new devices can be attached to the computer without the operating system being rewritten? Also, when the devices vary so widely, how can the operating system give a convenient, uniform I/O interface to applications?

12.3 Application I/O Interface

In this section, we discuss structuring techniques and interfaces for the operating system that enable I/O devices to be treated in a standard, uniform way. We explain, for instance, how an application can open a file on a disk without knowing what kind of disk it is, and how new disks and other devices can be added to a computer without the operating system being disrupted.

Like other complex software-engineering problems, the approach here involves abstraction, encapsulation, and software layering. Specifically, we can abstract away the detailed differences in I/O devices by identifying a few general kinds. Each general kind is accessed through a standardized set of functions—an interface. The differences are encapsulated in kernel modules called device drivers that internally are custom tailored to each device, but that export one of the standard interfaces. Figure 12.6 illustrates how the I/O-related portions of the kernel are structured in software layers.

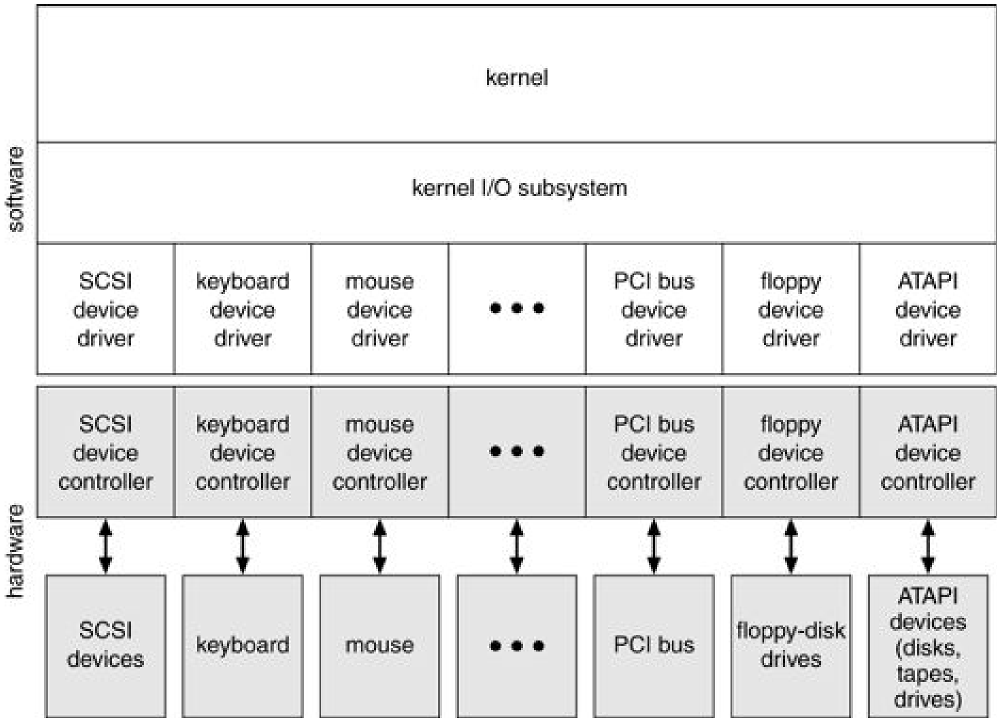
The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, much as the I/O system calls encapsulate the behavior of devices in a few generic classes that hide hardware differences from applications. Making the I/O subsystem independent of the hardware simplifies the job of the operating-system developer. It also benefits the hardware manufacturers. They either design new devices to be compatible with an existing host controller interface (such as SCSI-2), or they write device drivers to interface the new hardware to popular operating systems. Thus, new peripherals can be attached to a computer without waiting for the operating-system vendor to develop support code.

Unfortunately for device-hardware manufacturers, each type of operating system has its own standards for the device-driver interface. A given device may ship with multiple device drivers—for instance, drivers for MS-DOS, Windows 95/98, Windows NT/2000, and Solaris. Devices vary in many dimensions, as illustrated in Figure 12.7.

• **Character-stream or block:** A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.

• **Sequential or random-access:** A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.

Figure 12.6 A kernel I/O structure.



• **Synchronous or asynchronous:** A synchronous device is one that performs data transfers with predictable response times. An asynchronous device exhibits irregular or unpredictable response times.

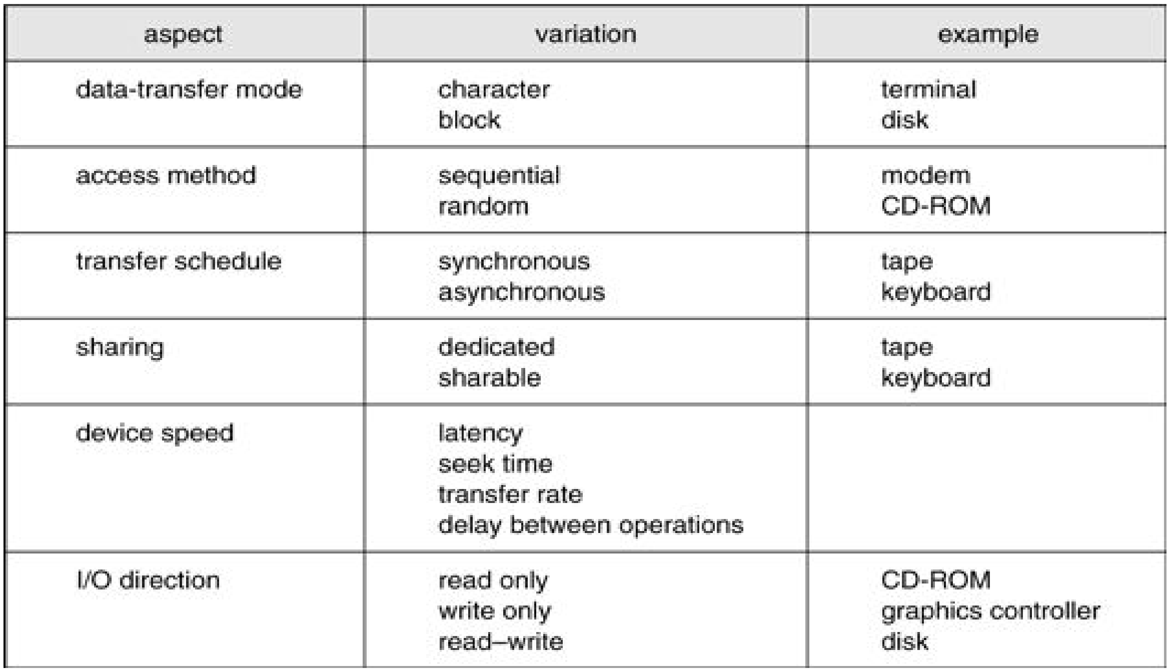
• **Sharable or dedicated:** A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.

• **Speed of operation:** Device speeds range from a few bytes per second to a few gigabytes per second.

• **Read-write, read only, or write only:** Some devices perform both input and output, but others support only one data direction.

For the purpose of application access, many of these differences are hidden by the operating system, and the devices are grouped into a few conventional types. The resulting styles of device access have been found to be useful and broadly applicable. Although the exact system calls may differ across operating systems, the device categories are fairly standard.

Figure 12.7 Characteristics of I/O devices.



aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

The major access conventions include block I/O, character-stream I/O, memory-mapped file access, and network sockets. Operating systems also provide special system calls to access a few additional devices, such as a time-of-day clock and a timer. Some operating systems provide a set of system calls for graphical display, video, and audio devices. Most operating systems also have an escape (or back door) that transparently passes arbitrary commands from an application to a device driver. In UNIX, this system call is ioctl() (for I/O control). The ioctl() system call enables an application to access any functionality that can be implemented by any device driver, without the need to invent a new system call. The ioctl()system call has three arguments.

The first is a file descriptor that connects the application to the driver by referring to a hardware device managed by that driver. The second is an integer that selects one of the commands implemented in the driver. The third is a pointer to an arbitrary data structure in memory, thus enabling the application and driver to communicate any necessary control information or data.

12.3.1 Block and Character Devices

The **block-device** interface captures all the aspects necessary for accessing disk drives and other block-oriented devices. The expectation is that the device understands commands such as read() and write(), and, if it is a random access device, it has a seek() command to specify which block to transfer next. Applications normally access such a device through a file-system interface. The operating system itself, and special applications such as database-management systems, may prefer to access a block device as a simple linear array of blocks. This mode of access is sometimes called **raw I/O**. We can see that read(), write(), and seek() capture the essential behaviors of block-storage devices, so that applications are insulated from the low-level differences among those devices.

Memory-mapped file access can be layered on top of block-device drivers. Rather than offering read and write operations, a memory-mapped interface provides access to disk storage via an array of bytes in main memory. The system call that maps a file into memory returns the virtual-memory address of an array of characters that contains a copy of the file. The actual data transfers are performed only when needed to satisfy access to the memory image. Because the transfers are handled by the same mechanism as that used for demand-paged virtual-memory access, memory-mapped I/O is efficient. Memory mapping is also convenient for programmers—access to a memory-mapped file is as simple as reading and writing to memory. Operating systems that offer virtual memory commonly use the mapping interface for kernel services. For instance, to execute a program, the operating system maps the executable into memory, and then transfers control to the entry address of the executable. The mapping interface is also commonly used for kernel access to swap space on disk.

A keyboard is an example of a device that is accessed through a **character-stream** interface. The basic system calls in this interface enable an application to get() or put() one character. On top of this interface, libraries can be built that offer lineat-a-time access, with buffering and editing services (for example, when a user types a backspace, the preceding character is removed from the input stream). This style of access is convenient for input devices such as keyboards, mice, and modems, which produce data for input "spontaneously"—that is, at times that cannot necessarily be predicted by the application. This access style is also good for output devices such as printers or audio boards, which naturally fit the concept of a linear stream of bytes.

12.3.2 Network Devices

Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the read()-write()-seek() interface used for disks. One interface available in many operating systems, including UNIX and Windows NT, is the network socket interface. Think of a wall socket for electricity: Any electrical appliance can be plugged in.

By analogy, the system calls in the socket interface enable an application to create a socket, to connect a local socket to a remote address (which plugs this application into a socket created by another application), to listen for any remote application to plug into the local socket, and to send and receive packets over the connection. To support the implementation of servers, the socket interface also provides a function called select() that manages a set of sockets. A call to select() returns information about which sockets have a packet waiting to be received, and which sockets have room to accept a packet to be sent. The use of select() eliminates the polling and busy waiting that would otherwise be necessary for network I/O.

These functions encapsulate the essential behaviors of networks, greatly facilitating the creation of distributed applications that can use any underlying network hardware and protocol stack. Many other approaches to inter process communication and network communication

have been implemented. For instance, Windows NT provides one interface to the network interface card, and a second interface to the network protocols. In UNIX, which has a long history as a proving ground for network technology, we find half-duplex pipes, full-duplex FIFOs, full-duplex STREAMS, message queues, and sockets.

12.3.3 Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

- Give the current time

- Give the elapsed time

- Set a timer to trigger operation *X* at time *T*

These functions are used heavily by the operating system, and also by time sensitive applications. Unfortunately, the system calls that implement these functions are not standardized across operating systems. The hardware to measure elapsed time and to trigger operations is called a programmable interval timer. It can be set to wait a certain amount of time and then to generate an interrupt. It can be set to do this operation once, or to repeat the process, to generate periodic interrupts. The scheduler uses this mechanism to generate an interrupt that will preempt a process at the end of its time slice.

The disk I/O subsystem uses it to invoke the flushing of dirty cache buffers to disk periodically, and the network subsystem uses it to cancel operations that are proceeding too slowly because of network congestion or failures. The operating system may also provide an interface for user processes to use timers. The operating system can support more timer requests than the number of timer hardware channels by simulating virtual clocks. To do so, the kernel (or the timer device driver) maintains a list of interrupts wanted by its own routines and by user requests, sorted in earliest-time-first order. It sets the timer for the earliest time. When the timer interrupts, the kernel signals the requester, and reloads the timer with the next earliest time.

On many computers, the interrupt rate generated by the ticking of the hardware clock is between 18 and 60 ticks per second. This resolution is coarse, since a modern computer can execute hundreds of millions of instructions per second. The precision of triggers is limited by the coarse resolution of the timer, together with the overhead of maintaining virtual clocks. And, if the timer ticks are used to maintain the system time-of-day clock, the system clock can drift. In most computers, the hardware clock is constructed from a high-frequency counter. In some computers, the value of this counter can be read from a device register, in which case the counter can be considered to be a high-resolution clock. Although this clock does not generate interrupts, it offers accurate measurements of time intervals.

12.3.4 Blocking and Non-blocking I/O

Another aspect of the system-call interface relates to the choice between blocking I/O and non-blocking (or asynchronous) I/O. When an application issues a blocking system call, the execution of the application is suspended. The application is moved from the operating system's run queue to a wait queue. After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call. The physical actions performed by I/O devices are generally asynchronous—they take a varying or unpredictable amount of time. Nevertheless, most operating systems use blocking system calls for the application interface, because blocking application code is easier to understand than non-blocking application code.

Some user-level processes need non-blocking I/O. One example is a user interface that receives keyboard and mouse input while processing and displaying data on the screen. Another example is a video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display. One way that an application writer can overlap execution with I/O is to write a multithreaded application. Some threads can perform blocking system calls, while others continue executing.

The Solaris developers used this technique to implement a user-level library for asynchronous I/O, freeing the application writer from that task. Some operating systems provide non-blocking I/O system calls. A non-blocking call does not halt the execution of the application for an extended time. Instead, it returns quickly, with a return value that indicates how many bytes were transferred.

An alternative to a non-blocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete. The application continues to execute its code. The completion of the I/O at some future time is communicated to the application, either through the setting of some variable in the address space of the application, or through the triggering of a signal or software interrupt or a call-back routine that is executed outside the linear control flow of the application. The difference between non-blocking and asynchronous system calls is that a non-blocking read() returns immediately with whatever data are available—the full number of bytes requested, fewer, or none at all. An asynchronous read() call requests a transfer that will be performed in its entirety, but that will complete at some future time.

A good example of non-blocking behavior is the select() system call for network sockets. This system call takes an argument that specifies a maximum waiting time. By setting it to 0, an application can poll for network activity without blocking. But using select() introduces extra overhead, because the select() call only checks whether I/O is possible. For a data transfer, select() must be followed by some kind of read() or write() command. A variation of this approach, found in Mach, is a blocking multiple-read call. It specifies desired reads for several devices in one system call, and returns as soon as any one of them completes.

12.4 Kernel I/O Subsystem

Kernels provide many services related to I/O. Several services—scheduling, buffering, caching, spooling, device reservation, and error handling—are provided by the kernel's I/O subsystem and build on the hardware and device-driver infrastructure.

12.4.1 I/O Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete.

Here is a simple example to illustrate the opportunity. Suppose that a disk arm is near the beginning of a disk, and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3 requests one in the middle of the disk.

The operating system can reduce the distance that the disk arm travels by serving the applications in order 2, 3, 1. Rearranging the order of service in this way is the essence of I/O scheduling. Operating-system developers implement scheduling by maintaining a queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests. For instance, requests from the virtual memory subsystem may take priority over application requests. Several scheduling algorithms for disk I/O are detailed in the next Section.

One way that the I/O subsystem improves the efficiency of the computer is by scheduling I/O operations. Another way is by using storage space in main memory or on disk, via techniques called buffering, caching, and spooling.

12.4.2 Buffering

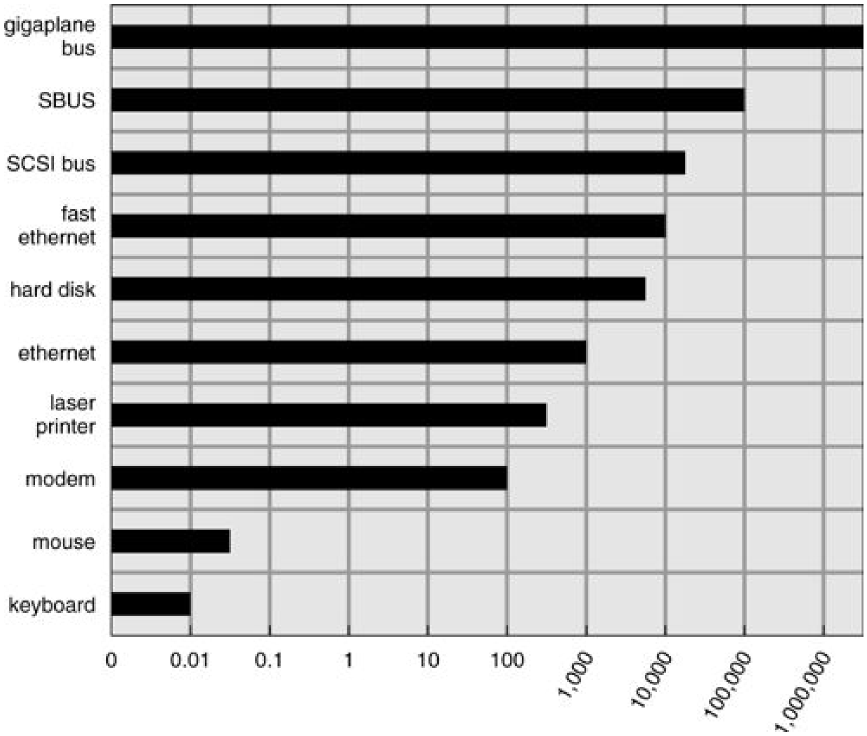
A **buffer** is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk. So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation. Since the disk write is not instantaneous and the modem still needs a place to store additional incoming data, two buffers are used.

After the modem fills the first buffer, the disk write is requested. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This **double buffering** decouples the producer of data from the consumer, thus relaxing timing requirements between them. The need for this decoupling is illustrated in Figure 12.8, which lists the enormous differences in device speeds for typical computer hardware.

A second use of buffering is to adapt between devices that have different data transfer sizes. Such disparities are especially common in computer networking, where buffers are used widely for fragmentation and reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

A third use of buffering is to support copy semantics for application I/O. An example will clarify the meaning of "copy semantics." Suppose that an application has a buffer of data that it wishes to write to disk. It calls the write() system call, providing a pointer to the buffer and an integer specifying the number of bytes to write. After the system call returns, what happens if the application changes the contents of the buffer? With **copy semantics**, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application's buffer.

Figure 12.8 Sun Enterprise 6000 device-transfer rates(logarithmic).



A simple way that the operating system can guarantee copy semantics is for the write() system call to copy the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect. Copying of data between kernel buffers and application data space is common in operating systems, despite the overhead that this operation introduces, because of the clean semantics. The same effect can be obtained more efficiently by clever use of virtual-memory mapping and copy-on-write page protection.

12.4.3 Caching

A **cache** is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, just holds a copy on faster storage of an item that resides elsewhere.

Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes. For instance, to preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data. These buffers are also used as a cache, to improve the I/O efficiency for files that are shared by applications or that are being written and reread rapidly. When the kernel receives a file I/O request, the kernel first accesses the buffer cache to see whether that region of the file is already available in main memory.

If so, a physical disk I/O can be avoided or deferred. Also, disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules. This strategy of delaying writes to improve I/O efficiency is discussed, in the context of remote file access, in the next Section.

12.4.4 Spooling and Device Reservation

A **spool** is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application's output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer.

The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in-kernel thread. In either case, the operating system provides a control interface that enables users and system administrators to display the queue, to remove unwanted jobs before those jobs print, to suspend printing while the printer is serviced, and so on.

Some devices, such as tape drives and printers, cannot usefully multiplex the I/O requests of multiple concurrent applications. Spooling is one way that operating systems can coordinate concurrent output. Another way to deal with concurrent device access is to provide explicit facilities for coordination. Some operating systems (including VMS) provide support for exclusive device access, by enabling a process to allocate an idle device, and to de-allocate that device when it is no longer needed. Other operating systems enforce a limit of one open file handle to such a device.

Many operating systems provide functions that enable processes to coordinate exclusive access among themselves. For instance, Windows NT provides system calls to wait until a device object becomes available. It also has a parameter to the open() system call that declares the types of access to be permitted to other concurrent threads. On these systems, it is up to the applications to avoid deadlock.

12.4.5 Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical glitch. Devices and I/O transfers can fail in many ways, either for transient reasons, such as a network becoming overloaded, or for "permanent" reasons, such as a disk controller becoming defective. Operating systems can often compensate effectively for transient failures. For instance, a disk read() failure results in a read() retry, and a network send() error results in a resend(), if the protocol so specifies. Unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.

As a general rule, an I/O system call will return 1 bit of information about the status of the call, signifying either success or failure. In the UNIX operating system, an additional integer variable named errno is used to return an error code—one of about 100 values—indicating the general nature of the failure (for example, argument out of range, bad pointer, or file not open). By contrast, some hardware can provide highly detailed error information, although many current operating systems are not designed to convey this information to the application.

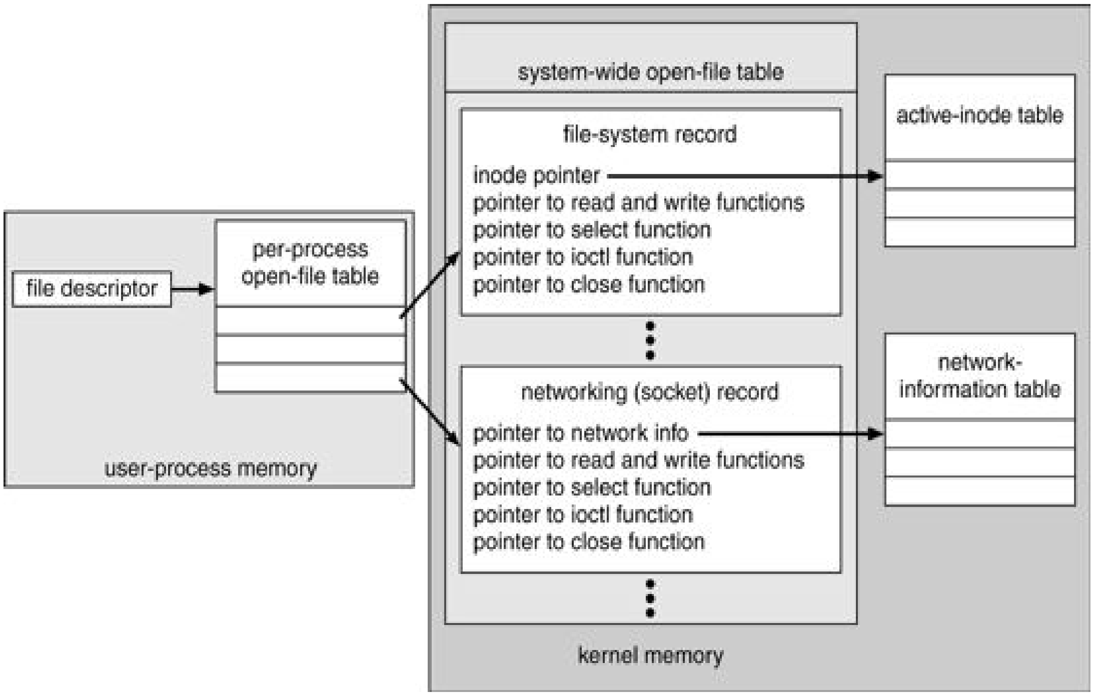
For instance, a failure of a SCSI device is reported by the SCSI protocol in terms of a sense key that identifies the general nature of the failure, such as a hardware error or an illegal request; an additional sense code that states the category of failure, such as a bad command parameter or a self-test failure; and an additional sense-code qualifier that gives even more detail, such as which command parameter was in error, or which hardware subsystem failed its self-test. Further, many SCSI devices maintain internal pages of error-log information that can be requested by the host, but that seldom are.

12.4.6 Kernel Data Structures

The kernel needs to keep state information about the use of I/O components. It does so through a variety of in-kernel data structures, such as the open-file table structure. The kernel uses many similar structures to track network connections, character-device communications, and other I/O activities. UNIX provides file-system access to a variety of entities, such as user files, raw devices, and the address spaces of processes. Although each of these entities supports a read() operation, the semantics differ. For instance, to read a user file, the kernel needs to probe the buffer cache before deciding whether to perform a disk I/O. To read a raw disk, the kernel needs to ensure that the request size is a multiple of the disk sector size, and is aligned on a sector boundary. To read a process image, it is merely necessary to copy data from memory. UNIX encapsulates these differences within a uniform structure by using an object-oriented technique.

The open-file record, shown in Figure 12.9, contains a dispatch table that holds pointers to the appropriate routines, depending on the type of file. Some operating systems use object-oriented methods even more extensively. For instance, Windows NT uses a message-passing implementation for I/O. An I/O request is converted into a message that is sent through the kernel to the I/O manager and then to the device driver, each of which may change the message contents. For output, the message contains the data to be written. For input, the message contains a buffer to receive the data. The message-passing approach can add overhead, by comparison with procedural techniques that use shared data structures, but it simplifies the structure and design of the I/O system, and adds flexibility.

Figure 12.9 UNIX I/O kernel structure.


system-wide open-file table
file-system record
inode pointer
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
active-inode table
per-process open-file table
file descriptor
networking (socket) record
pointer to network info
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
network-information table
user-process memory
kernel memory

In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel. The I/O subsystem supervises

- The management of the name space for files and devices
- Access control to files and devices
- Operation control (for example, a modem cannot seek())
- File system space allocation
- Device allocation
- Buffering, caching, and spooling
- I/O scheduling

- Device-status monitoring, error handling, and failure recovery
- Device-driver configuration and initialization

The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers.

12.5 Transforming I/O to Hardware Operations

Earlier, we described the handshaking between a device driver and a device controller, but we did not explain how the operating system connects an application request to a set of network wires or to a specific disk sector. Let us consider the example of reading a file from disk. The application refers to the data by a file name. Within a disk, the file system maps from the file name through the file-system directories to obtain the space allocation of the file. For instance, in MS-DOS, the name maps to a number that indicates an entry in the file-access table, and that table entry tells which disk blocks are allocated to the file.

In UNIX, the name maps to an inode number, and the corresponding inode contains the space-allocation information. How is the connection made from the file name to the disk controller (the hardware port address or the memory-mapped controller registers)? First, we consider MS-DOS, a relatively simple operating system. The first part of an MS-DOS file name, preceding the colon, is a string that identifies a specific hardware device. For example, *c:* is the first part of every file name on the primary hard disk. The fact that *c:* represents the primary hard disk is built into the operating system; *c:* is mapped to a specific port address through a device table. Because of the colon separator, the device name space is separate from the file-system name space within each device.

This separation makes it easy for the operating system to associate extra functionality with each device. For instance, it is easy to invoke spooling on any files written to the printer. If, instead, the device name space is incorporated in the regular file-system name space, as it is in UNIX, the normal file-system name services are provided automatically. If the file system provides ownership and access control to all file names, then devices have owners and access control. Since files are stored on devices, such an interface provides access to the I/O system at two levels. Names can be used to access the devices themselves, or to access the files stored on the devices.

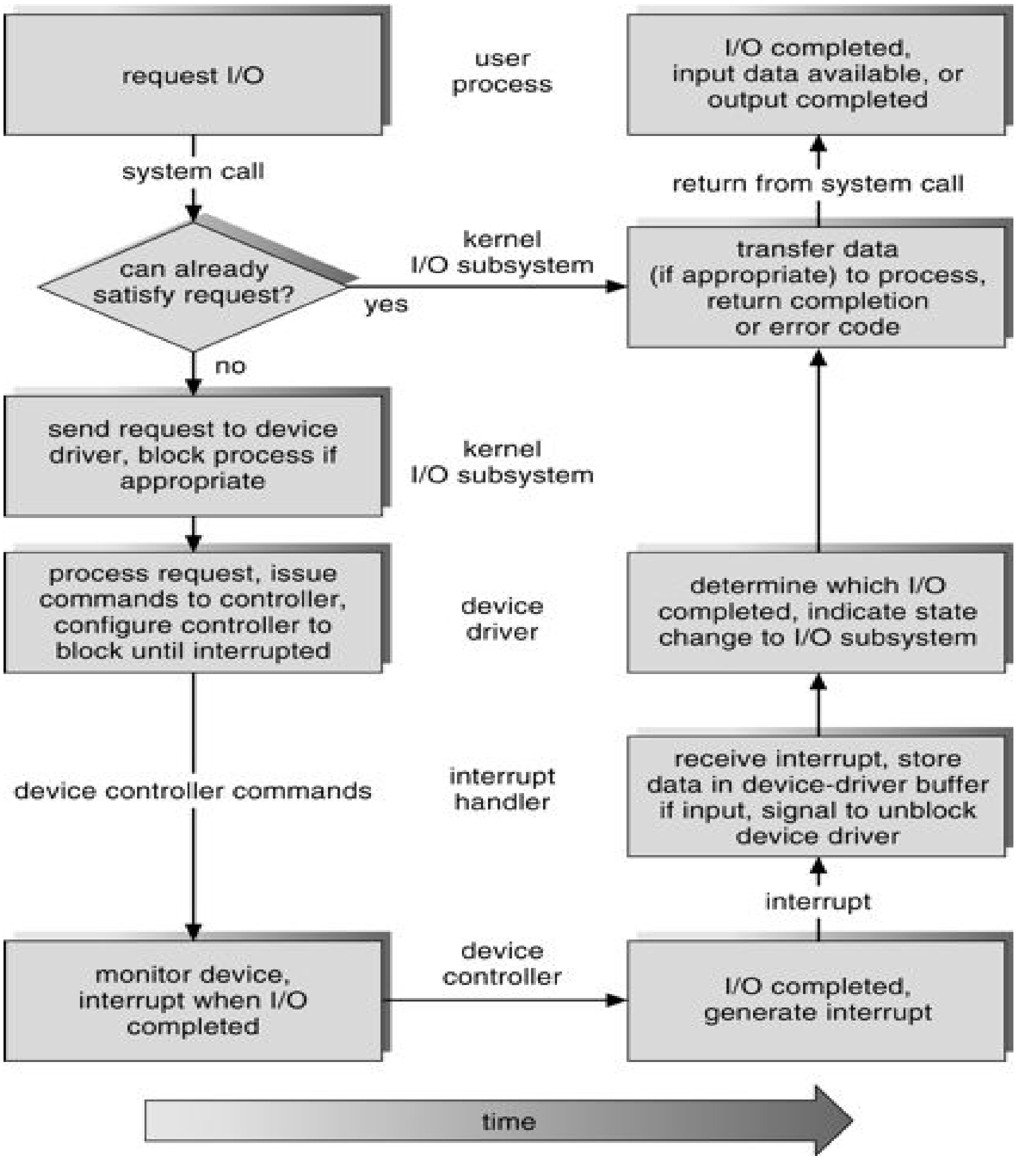
UNIX represents device names in the regular file-system name space. Unlike an MSDOS file name, which has the colon separator, a UNIX path name has no clear separation of the device portion. In fact, no part of the path name is the name of a device. UNIX has a mount table that associates prefixes of path names with specific device names. To resolve a path name, UNIX looks up the name in the mount table to find the longest matching prefix; the corresponding entry in the mount table gives the device name. This device name also has the form of a name in the file-system name space. When UNIX looks up this name in the file-system directory structures, instead of finding an inode number, UNIX finds a <*major, minor*> device number.

The major device number identifies a device driver that should be called to handle I/O to this device. The minor device number is passed to the device driver to index into a device table. The corresponding device-table entry gives the port address or the memory mapped address of the device controller. Modern operating systems obtain significant flexibility from the multiple stages of lookup tables in the path between a request and a physical device controller. The mechanisms that pass requests between applications and drivers are general. Thus, we can introduce new devices and drivers into a computer without recompiling the kernel. In fact, some operating systems have the ability to load device drivers on demand.

At boot time, the system first probes the hardware buses to determine what devices are present, and then the system loads in the necessary drivers, either immediately, or when first required by an I/O request. Now we describe the typical lifecycle of a blocking read request, as depicted in Figure 12.10. The figure suggests that an I/O operation requires a great many steps that together consume a tremendous number of CPU cycles.

1. A process issues a blocking read() system call to a file descriptor of a file that has been *open*ed previously.
2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process and the I/O request is completed.
3. Otherwise, a physical I/O needs to be performed, so the process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or via an in-kernel message.
4. The device driver allocates kernel buffer space to receive the data, and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device control registers.
5. The device controller operates the device hardware to perform the data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
7. The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
8. The device driver receives the signal, determines which I/O request completed, determines the request's status, and signals the kernel I/O subsystem that the request has been completed.
9. The kernel transfers data or return codes to the address space of the requesting process, and moves the process from the wait queue back to the ready queue.
10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

Figure 12.10 The life cycle of an I/O request.


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
kernel I/O subsystem
yes
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

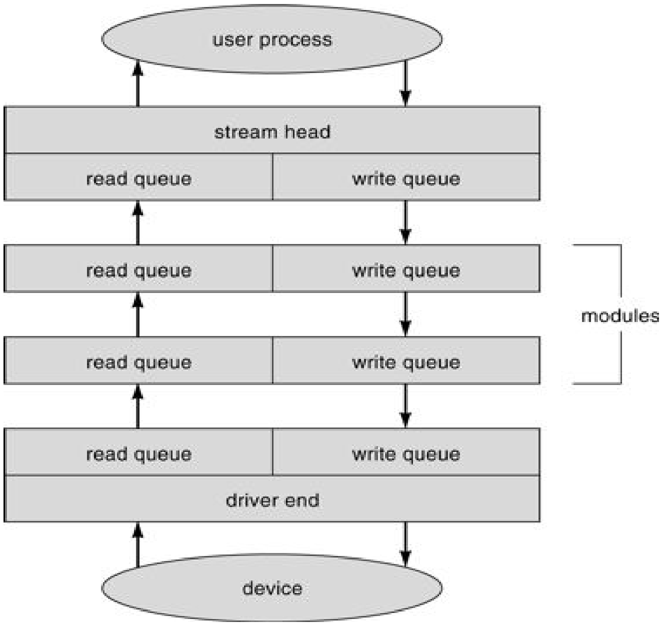
12.6 STREAMS

UNIX System V has an interesting mechanism, called **STREAMS**, that enables an application to assemble pipelines of driver code dynamically. A stream is a full duplex connection between a device driver and a user-level process. It consists of a stream head that interfaces with the user process, a driver end that controls the device, and zero or more stream modules between them. The stream head, the driver end, and each module contain a pair of queues—a read queue and a write queue.

Message passing is used to transfer data between queues. The STREAMS structure is shown in Figure 12.11. Modules provide the functionality of STREAMS processing and they are *pushed* onto a stream using the ioctl() system call. For example, a process can open a serial-port device via a stream, and can push on a module to handle input editing. Because messages are exchanged between queues in adjacent modules, a queue in one module may overflow an adjacent queue. To prevent this from occurring, a queue may support flow control. Without flow control, a queue accepts all messages and immediately sends them on to the queue in the adjacent module without buffering them. A queue supporting flow control buffers messages and does not accept messages without sufficient buffer space.

Flow control is supported by exchanging control messages between queues in adjacent modules. A user process writes data to a device using either the write() or putmsg()system calls. The write() system call writes raw data to the stream whereas putmsg() allows the user process to specify a message.

Figure 13.11 The STREAMS structure.



Regardless of the system call used by the user process, the stream head copies the data into a message and delivers it to the queue for the next module in line. This copying of messages continues until the message is copied to the driver end and hence the device. Similarly, the user process reads data from the stream head using either the read() or getmsg() system calls. If read() is used, the stream head gets a message from its adjacent queue and returns ordinary data (an unstructured byte stream) to the process. If getmsg() is used, a message is returned to the process.

STREAMS I/O is asynchronous (or non-blocking) with the exception of when the user process communicates with the stream head. When writing to the stream, the user process will block, assuming the next queue uses flow control, until there is room to copy the message. Likewise, the user process will block when reading from the stream until data is available. The driver end is similar to a stream head or a module in that it has a read and write queue. However, the driver end must respond to interrupts such as one triggered when a frame is ready to be read from a network.

Unlike the stream head that may block if it is unable to copy a message to the next queue in line, the driver end must handle all incoming data. Drivers must support flow control as well. However, if a device's buffer is full, a device typically resorts to dropping incoming messages. Consider a network card whose input buffer is full. The network card must simply drop further messages until there is ample buffer space to store incoming messages. The benefit of using STREAMS is that it provides a framework to a modular and incremental approach to writing device drivers and network protocols.

Modules may be used by different STREAMS and hence by different devices. For example, a networking module may be used by both an Ethernet network card and a token ring network card. Furthermore, rather than treating character device I/O as an unstructured byte stream, STREAMS allow support for message boundaries and control information between modules. Support for STREAMS is widespread among most UNIX variants and it is the preferred method for writing protocols and device drivers. For example, in System V UNIX and Solaris, the socket mechanism is implemented using STREAMS.

12.7 Performance

I/O is a major factor in system performance. It places heavy demands on the CPU to execute device-driver code and to schedule processes fairly and efficiently as they block and unblock. The resulting context switches stress the CPU and its hardware caches. I/O also exposes any inefficiencies in the interrupt-handling mechanisms in the kernel, and I/O loads down the memory bus during data copy between controllers and physical memory, and again during copies between kernel buffers and application data space. Coping gracefully with all these demands is one of the major concerns of a computer architect. Although modern computers can handle many thousands of interrupts per second, interrupt handling is a relatively expensive task: Each interrupt causes the system to perform a state change, to execute the interrupt handler, and then to restore state.

Programmed I/O can be more efficient than interrupt-driven I/O, if the number of cycles spent busy-waiting is not excessive. An I/O completion typically unblocks a process, leading to the full overhead of a context switch. Network traffic can also cause a high context-switch rate. Consider, for instance, a remote login from one machine to another. Each character typed on the local machine must be transported to the remote machine. On the local machine, the character is typed; a keyboard interrupt is generated; and the character is passed through the interrupt handler to the device driver, to the kernel, and then to the user process.

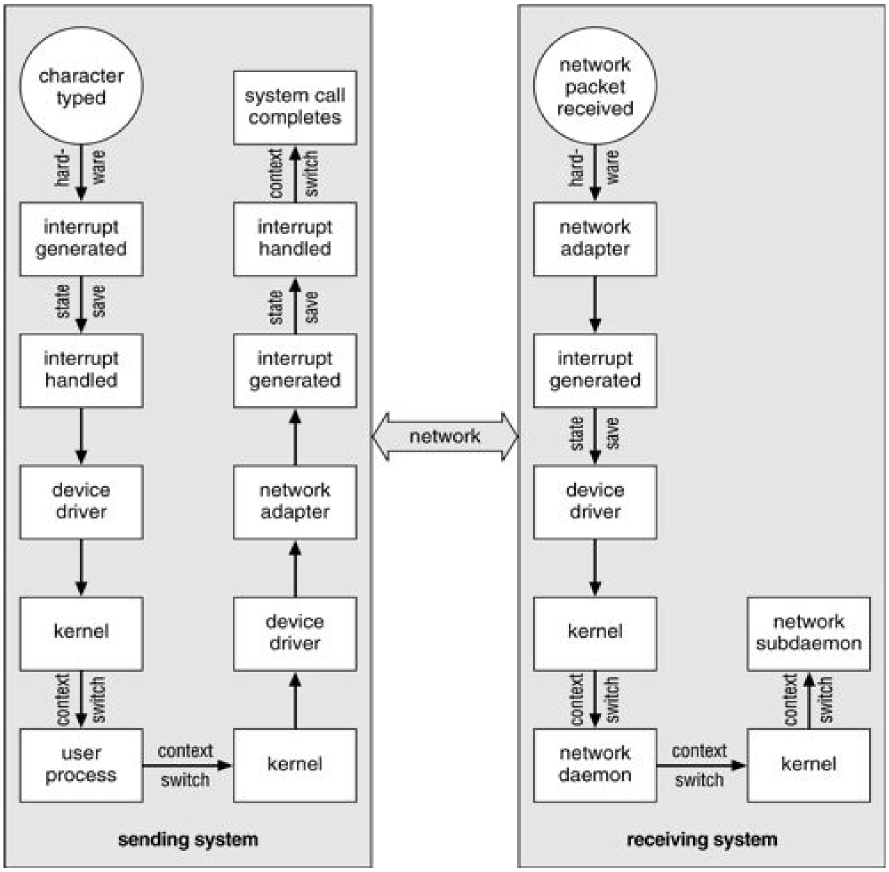
The user process issues a network I/O system call to send the character to the remote machine. The character then flows into the local kernel, through the network layers that construct a network packet, and into the network device driver. The network device driver transfers the packet to the network controller, which sends the character and generates an interrupt. The interrupt is passed back up through the kernel to cause the network I/O system call to complete.

Now, the remote system's network hardware receives the packet, and an interrupt is generated. The character is unpacked from the network protocols and is given to the appropriate network daemon. The network daemon identifies which remote login session is involved, and passes the packet to the appropriate sub-daemon for that session. Throughout this flow there are context switches and state switches (Figure12.12). Usually, the receiver echoes the character back to the sender; that approach doubles the work.

The Solaris developers re-implemented the telnet daemon using in-kernel threads to eliminate the context switches involved in moving each character between daemons and the kernel. Sun estimates that this improvement increased the maximum number of network logins from a few hundred to a few thousand on a large server.

Other systems use separate front-end processors for terminal I/O, to reduce the interrupt burden on the main CPU. For instance, a terminal concentrator can multiplex the traffic from hundreds of remote terminals into one port on a large computer. An I/O channel is a dedicated, special-purpose CPU found in mainframes and in other high-end systems. The job of a channel is to offload I/O work from the main CPU. The idea is that the channels keep the data flowing smoothly, while the main CPU remains free to process the data. Like the device controllers and DMA controllers found in smaller computers, a channel can process more general and sophisticated programs, so channels can be tuned for particular workloads.

Figure 12.12 Inter-computer communications.



We can employ several principles to improve the efficiency of I/O:

- Reduce the number of context switches.

- Reduce the number of times that data must be copied in memory while passing between device and application.

- Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy-waiting can be minimized).

- Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.

- Move processing primitives into hardware, to allow their operation in device controllers concurrent with the CPU and bus operation.

- Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.

Devices vary greatly in complexity. For instance, a mouse is simple. The mouse movements and button clicks are converted into numeric values that are passed from hardware, through the mouse device driver, to the application. By contrast, the functionality provided by the NT disk device driver is complex. It not only manages individual disks but also implements RAID arrays. To do so, it converts an application's read or write request into a coordinated set of disk I/O operations.

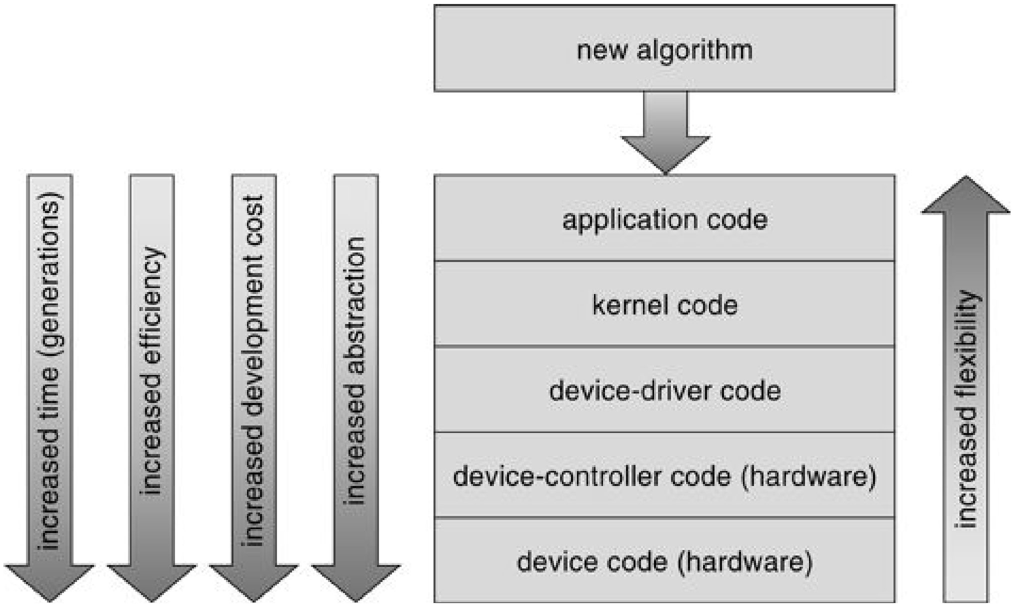
Moreover, it implements sophisticated error-handling and data-recovery algorithms, and takes many steps to optimize disk performance, because of the importance of secondary-storage performance to overall system performance. Where should the I/O functionality be implemented—in the device hardware, in the device driver, or in application software? Sometimes we observe the progression depicted in Figure 12.13.

- Initially, we implement experimental I/O algorithms at the application level, because application code is flexible, and application bugs are unlikely to cause system crashes. Furthermore, by developing code at the application level, we avoid the need to reboot or reload device drivers after every change to the code. An application-level implementation can be inefficient, however, because of the overhead of context switches, and because the application cannot take advantage of internal kernel data structures and kernel functionality (such as efficient in-kernel messaging, threading, and locking).

- When an application-level algorithm has demonstrated its worth, we may re-implement it in the kernel. This can improve the performance, but the development effort is more challenging, because an operating-system kernel is a large, complex software system. Moreover, an in-kernel implementation must be thoroughly debugged to avoid data corruption and system crashes.

- The highest performance may be obtained by a specialized implementation in hardware, either in the device or in the controller. The disadvantages of a hardware implementation include the difficulty and expense of making further improvements or of fixing bugs, the increased development time (months rather than days), and the decreased flexibility. For instance, a hardware RAID controller may not provide any means for the kernel to influence the order or location of individual block reads and writes, even if the kernel has special information about the workload that would enable the kernel to improve the I/O performance.

Figure 12.13 Device-functionality progression.



12.8 SUMMARY

The basic hardware elements involved in I/O are buses, device controllers, and the devices themselves. The work of moving data between devices and main memory is performed by the CPU as programmed I/O, or is offloaded to a DMA controller. The kernel module that controls a device is a device driver. The system-call interface provided to applications is designed to handle several basic categories of hardware, including block devices, character devices, memory-mapped files, network sockets, and programmed interval timers. The system calls usually block the process that issues them, but non-blocking and asynchronous calls are used by the kernel itself, and by applications that must not sleep while waiting for an I/O operation to complete.

The kernel's I/O subsystem provides numerous services. Among these are I/O scheduling, buffering, spooling, error handling, and device reservation. Another service is name translation, to make the connection between hardware devices and the symbolic file names used by applications. It involves several levels of mapping that translate from a character string name to a specific device driver and device address, and then to physical addresses of I/O ports or bus controllers. This mapping may occur within the file-system name space, as it does in UNIX, or in a separate device name space, as it does in MS-DOS.

STREAMS is an implementation and methodology for making drivers reusable and easy to use. Through them, drivers can be stacked, with data passed through them sequentially and bi-directionally for processing. I/O system calls are costly in terms of CPU consumption, because of the many layers of software between a physical device and the application. These layers imply the overheads of context switching to cross the kernel's protection boundary, of signal and interrupt handling to service the I/O devices, and of the load on the CPU and memory system to copy data between kernel buffers and application space.

EXERCISES

12.1 State three advantages of placing functionality in a device controller, rather than in the kernel. State three disadvantages.

12.2 Consider the following I/O scenarios on a single-user PC.

a. A mouse used with a graphical user interface

b. A tape drive on a multitasking operating system (assume no device pre-allocation is available)

c. A disk drive containing user files

d. A graphics card with direct bus connection, accessible through memory mapped I/O

For each of these I/O scenarios, would you design the operating system to use buffering, spooling, caching, or a combination? Would you use polled I/O or interrupt-driven I/O? Give reasons for your choices.

12.3 The example of handshaking in Section 13.2 used 2 bits: a busy bit and a command-ready bit. Is it possible to implement this handshaking with only 1 bit? If it is, describe the protocol. If not, explain why 1 bit is insufficient.

12.4 Describe three circumstances under which blocking I/O should be used. Describe three circumstances under which non-blocking I/O should be used. Why not just implement non-blocking I/O and have processes busy-wait until their device is ready?

12.5 Why might a system use interrupt-driven I/O to manage a single serial port, but polling I/O to manage a front-end processor, such as a terminal concentrator?

12.6 Polling for an I/O completion can waste a large number of CPU cycles if the processor iterates a busy-waiting loop many times before the I/O completes. But if the I/O device is ready for service, polling can be much more efficient than is catching and dispatching an interrupt. Describe a hybrid strategy that combines polling, sleeping, and interrupts for I/O device service. For each of these three strategies (pure polling, pure interrupts, hybrid), describe a computing environment in which that strategy is more efficient than is either of the others.

12.7 UNIX coordinates the activities of the kernel I/O components by manipulating shared in-kernel data structures, whereas Windows NT uses object-oriented message passing between kernel I/O components. Discuss three pros and three cons of each approach.

12.8 How does DMA increase system concurrency? How does it complicate the hardware design?

12.9 Write (in pseudocode) an implementation of virtual clocks, including the queueing and management of timer requests for the kernel and applications. Assume that the hardware provides three timer channels.

12.10 Why is it important to scale up system bus and device speeds as the CPU speed increases?

12.11 Distinguish between a STREAMS driver and a STREAMS module.

13. MASS-STORAGE STRUCTURE

13.1 Disk Structure

Disks provide the bulk of secondary storage for modern computer systems. Magnetic tape was used as an early secondary-storage medium, but the access time is much slower than for disks. Thus, tapes are currently used mainly for backup, for storage of infrequently used information, as a medium for transferring information from one system to another, and for storing quantities of data so large that they are impractical as disk systems. Modern disk drives are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be **low-level formatted** to choose a different logical block size, such as 1,024 bytes.

The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

By using this mapping, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform this translation, for two reasons. First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk. Second, the number of sectors per track is not a constant on some drives. On media that use **constant linear velocity (CLV)**, the density of bits per track is uniform.

The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives.

Alternatively, the disk rotation speed can stay constant, and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as **constant angular velocity (CAV)**. The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

13.2 Disk Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having a fast access time and disk bandwidth. The access time has two major components. The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector. The **rotational latency** is the additional time waiting for the disk to rotate the desired sector to the disk head. The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order.

As we discussed in Chapter 2, whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of information:

- Whether this operation is input or output
- What the disk address for the transfer is
- What the memory address for the transfer is
- What the number of bytes to be transferred is

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed on the queue of pending requests for that drive. For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next.

13.2.1 FCFS Scheduling

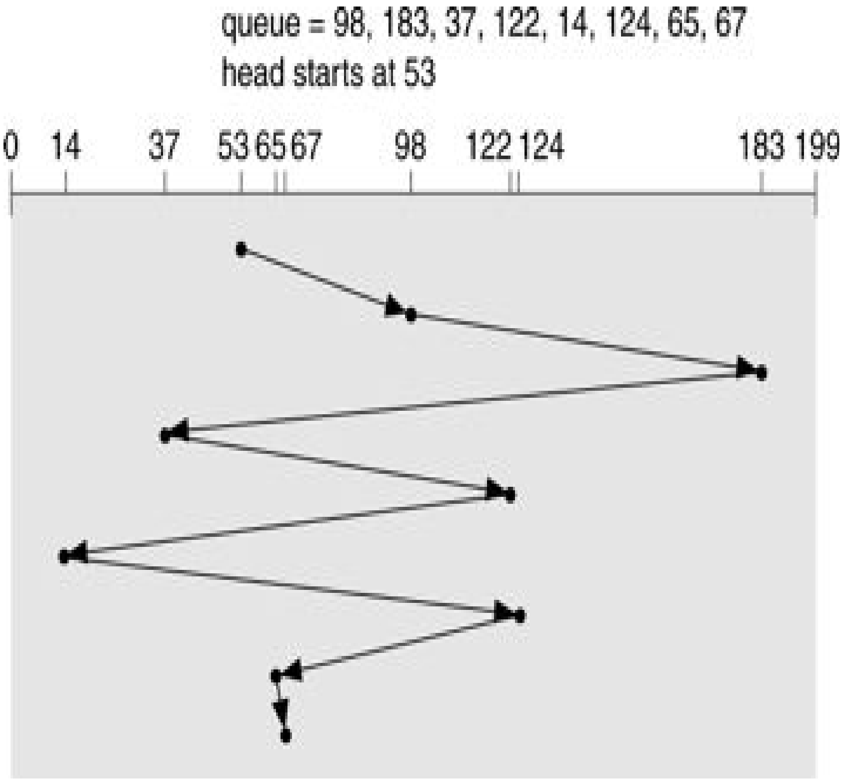
The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37, 122, 14, 124, 65, 67, in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 13.1.

The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests at 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

13.2.2 SSTF Scheduling

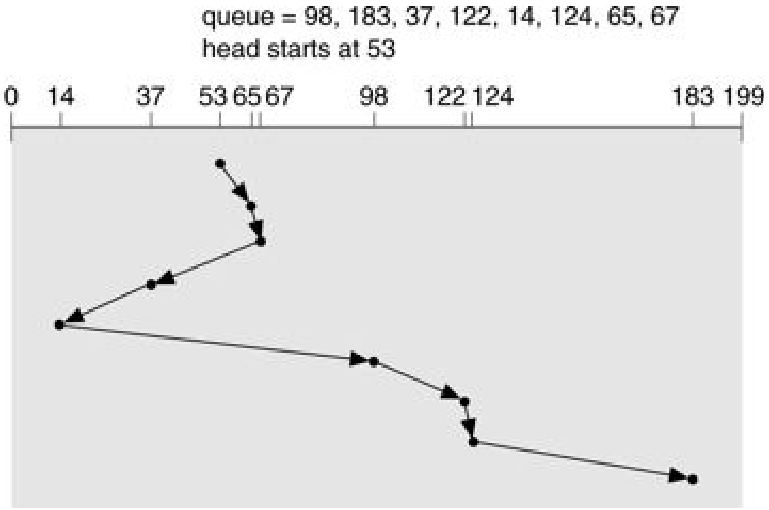
It seems reasonable to service all the requests close to the current head position, before moving the head far away to service other requests. This assumption is the basis for the **shortest-seek-time-first (SSTF) algorithm**. The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

Figure 14.1 FCFS disk scheduling.



For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183 (Figure 13.2). This scheduling method results in a total head movement of only 236 cylinders—little more than one-third of the distance needed for FCFS scheduling of this request queue. This algorithm gives a substantial improvement in performance. SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling, and, like SJF scheduling, it may cause starvation of some requests. Remember that requests may arrive at any time. Suppose that we have two requests in the queue, for cylinders 14 and 186, and while servicing the request from 14, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could arrive, causing the request for cylinder 186 to wait indefinitely. This scenario becomes increasingly likely if the pending-request queue grows long.

Figure 13.2 SSTF disk scheduling.



Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal. In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the total head movement to 208 cylinders.

13.2.3 SCAN Scheduling

In the **SCAN algorithm**, the disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. We again use our example. Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement, in addition to the head's current position (53). If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 13.3). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

Figure 13.3 SCAN disk scheduling.

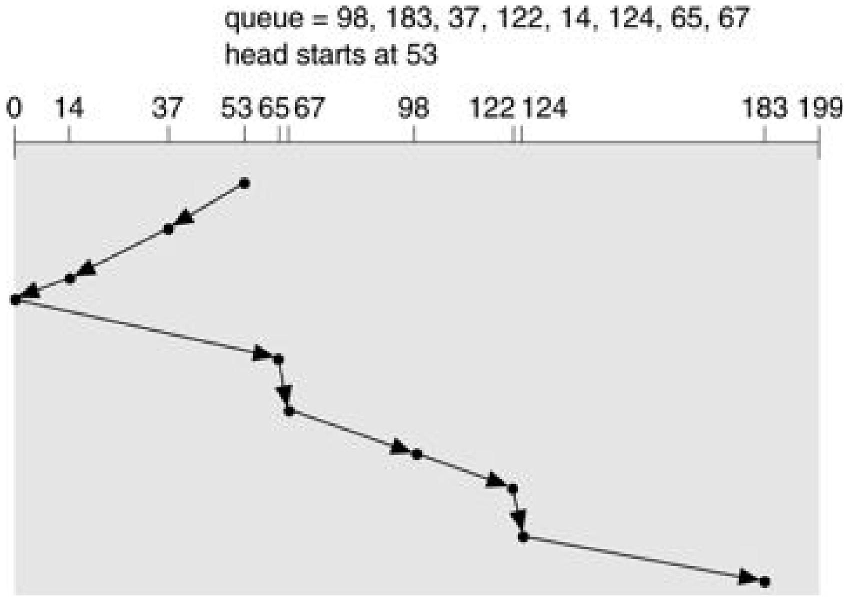
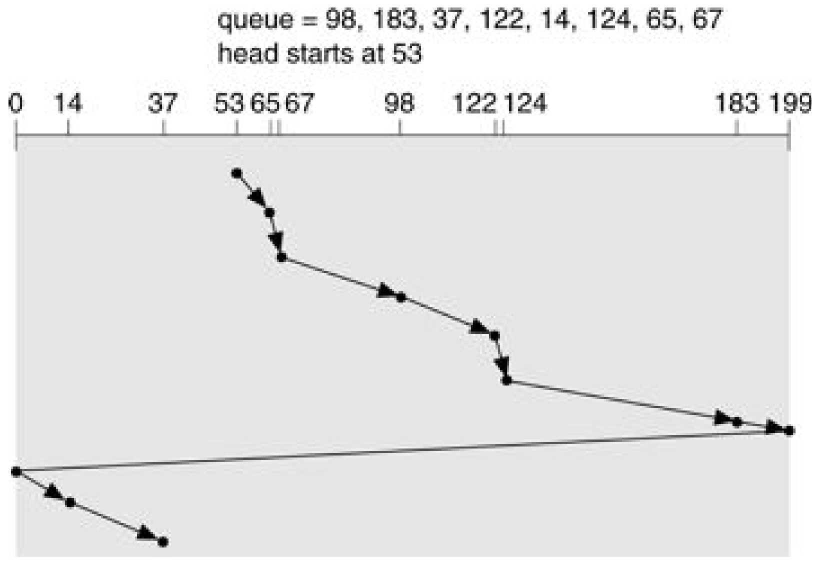


Figure 13.4 C-SCAN disk scheduling.



The SCAN algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up, and then reversing to service requests the other way. Assuming a uniform distribution of requests for cylinders, consider the density of requests when the head reaches one end and reverses direction. At this point, relatively few requests are immediately in front of the head, since these cylinders have recently been serviced. The heaviest density of requests is at the other end of the disk. These requests have also waited the longest, so why not go there first? That is the idea of the next algorithm.

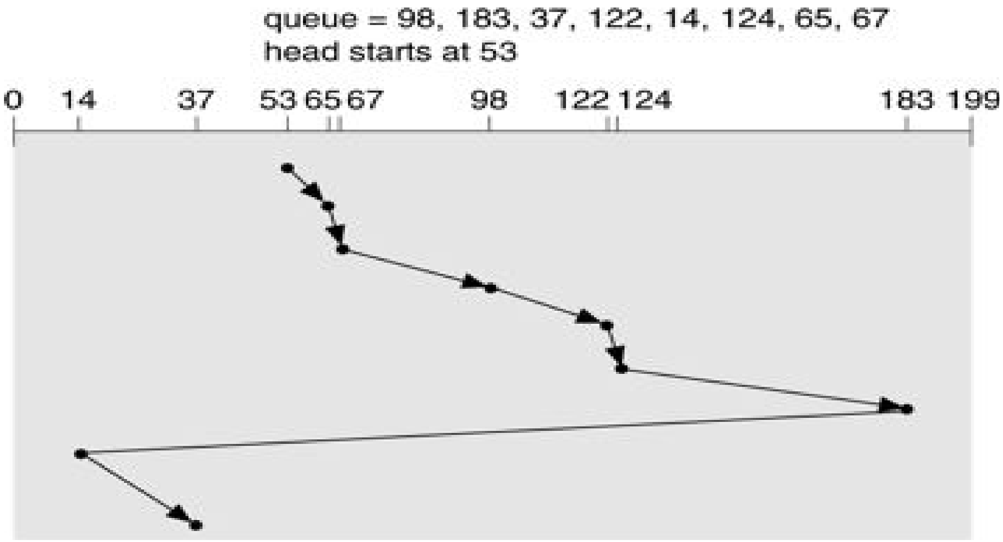
13.2.4 C-SCAN Scheduling

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip (Figure 13.4). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

13.2.5 LOOK Scheduling

As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is implemented this way. More commonly, the arm goes only as far as the final request in each direction.

Figure 13.5 C-LOOK disk scheduling.



Then, it reverses direction immediately, without going all the way to the end of the disk. These versions of SCAN and C-SCAN are called **LOOK** and **C-LOOK scheduling**, because they *look* for a request before continuing to move in a given direction (Figure 13.5).

13.2.6 Selection of a Disk-Scheduling Algorithm

Given so many disk-scheduling algorithms, how do we choose the best one? SSTF is common and has a natural appeal because it increases performance over FCFS. SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to have a starvation problem. For any particular list of requests, we can define an optimal order of retrieval, but the computation needed to find an optimal schedule may not justify the savings over SSTF or SCAN.

With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then, all scheduling algorithms are forced to behave the same, because they have only one choice for where to move the disk head: They all behave like FCFS scheduling.

The requests for disk service can be greatly influenced by the file-allocation method. A program reading a contiguously allocated file will generate several requests that are close together on the disk, resulting in limited head movement. A linked or indexed file, on the other hand, may include blocks that are widely scattered on the disk, resulting in greater head movement. The location of directories and index blocks is also important. Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently. Suppose that a directory entry is on the first cylinder and a file's data are on the final cylinder. In this case, the disk head has to move the entire width of the disk. If the directory entry were on the middle cylinder, the head would have to move, at most, one-half the width.

Caching the directories and index blocks in main memory can also help to reduce the disk-arm movement, particularly for read requests. Because of these complexities, the disk-scheduling algorithm should be written as a separate module of the operating system, so that it can be replaced with a different algorithm if necessary. Either SSTF or LOOK is a reasonable choice for the default algorithm. The scheduling algorithms described here consider only the seek distances.

For modern disks, the rotational latency can be nearly as large as the average seek time. But it is difficult for the operating system to schedule for improved rotational latency because modern disks do not disclose the physical location of logical blocks. Disk manufacturers have been alleviating this problem by implementing disk scheduling algorithms in the controller hardware built into the disk drive. If the operating system sends a batch of requests to the controller, the controller can queue them and then schedule them to improve both the seek time and the rotational latency. If I/O performance were the only consideration, the operating system would gladly turn over the responsibility of disk scheduling to the disk hardware.

In practice, however, the operating system may have other constraints on the service order for requests. For instance, demand paging may take priority over application I/O, and writes are more urgent than reads if the cache is running out of free pages. Also, it may be desirable to guarantee the order of a set of disk writes to make the file system robust in the face of system crashes. Consider what could happen if the operating system allocated a disk page to a file, and the application wrote data into that page before the operating system had a chance to flush the modified inode and free-space list back to disk. To accommodate such requirements, an operating system may choose to do its own disk scheduling and to spoon-feed the requests to the disk controller, one by one, for some types of I/O.

13.3 Disk Management

The operating system is responsible for several other aspects of disk management, too. Here we discuss disk initialization, booting from disk, and bad-block recovery.

13.3.1 Disk Formatting

A new magnetic disk is a blank slate: It is just platters of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting (or physical formatting). Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an error-correcting code (ECC). When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area.

When the sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad (Section 13.3.3). The ECC is an error-*correcting* code because it contains enough information that, if only a few bits of data have been corrupted, the controller can identify which bits have changed and can calculate what their correct values should be. The controller automatically does the ECC processing whenever a sector is read or written.

Most hard disks are low-level formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk. For many hard disks, when the disk controller is instructed to low-level format the disk, it can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is usually possible to choose among a few sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track, but that also means fewer headers and trailers are written on each track, and thus increases the space available for user data. Some operating systems can handle only a sector size of 512 bytes.

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps. The first step is to partition the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files. After partitioning, the second step is logical formatting (or creation of a file system).

In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or inodes) and an initial empty directory. Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the raw disk, and I/O to this array is termed raw I/O. For example, some database systems prefer raw I/O because it enables them to control the exact disk location where each database record is stored. Raw I/O bypasses all the file-system services, such as the buffer cache, file locking, prefetching, space allocation, file names, and directories.

We can make certain applications more efficient by implementing their own special-purpose storage services on a raw partition, but most applications perform better when they use the regular file-system services.

13.3.2 Boot Block

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 bootstrap program tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system. To do its job, the bootstrap program finds the operating-system kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution.

For most computers, the bootstrap is stored in **read-only memory (ROM)**. This location is convenient, because ROM needs no initialization and is at a fixed location that the processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM hardware chips. For this reason, most systems store a tiny bootstrap loader program in the boot ROM , whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in a partition called the boot blocks, at a fixed location on the disk.

A disk that has a boot partition is called a **boot disk** or **system disk**. The code in the boot ROM instructs the disk controller to read the boot blocks into memory (no device drivers are loaded at this point), and then starts executing that code. The full bootstrap program is more sophisticated than the bootstrap loader in the boot ROM; it is able to load the entire operating system from a nonfixed location on disk, and to start the operating system running. Even so, the full bootstrap code may be small. For example, MS-DOS uses one 512-byte block for its boot program (Figure 13.6).

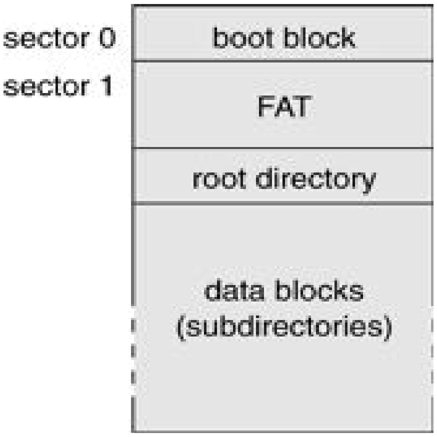
13.3.3 Bad Blocks

Because disks have moving parts and small tolerances (recall that the disk head flies just above the disk surface), they are prone to failure. Sometimes the failure is complete, and the disk needs to be replaced, and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with **bad blocks**. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

On simple disks, such as some disks with IDE controllers, bad blocks are handled manually. For instance, the MS-DOS format command does a logical format and, as a part of the process, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block. If blocks go bad during normal operation, a special program (such as chkdsk) must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

More sophisticated disks, such as the SCSI disks used in high-end PCs and most workstations and servers, are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level format at the factory, and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**.

Figure 13.6 MS-DOS disk layout.



A typical bad-sector transaction might be as follows:

• The operating system tries to read logical block 87.

• The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.

• The next time that the system is rebooted, a special command is run to tell the SCSI controller to replace the bad sector with a spare.

• After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Such a redirection by the controller could invalidate any optimization by the operating system's disk-scheduling algorithm! For this reason, most disks are formatted to provide a few spare sectors in each cylinder, and a spare cylinder as well. When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible. As an alternative to sector sparing, some controllers can be instructed to replace a bad block by **sector slipping**. Here is an example: Suppose that logical block 17 becomes defective, and the first available spare follows sector 202. Then, sector slipping would remap all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 would be copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping

the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

The replacement of a bad block generally is not a totally automatic process because the data in the bad block are usually lost. Thus, whatever file was using that block must be repaired (for instance, by restoration from a backup tape), and that requires manual intervention.

13.4 Swap-Space Management

Swap-space management is another low-level task of the operating system. Virtual memory uses disk space as an extension of main memory. Since disk access is much slower than memory access, using swap space significantly decreases system performance. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual-memory system. In this section, we discuss how swap space is used, where swap space is located on disk, and how swap space is managed.

13.4.1 Swap-Space Use

Swap space is used in various ways by different operating systems, depending on the implemented memory-management algorithms. For instance, systems that implement swapping may use swap space to hold the entire process image, including the code and data segments. Paging systems may simply store pages that have been pushed out of main memory. The amount of swap space needed on a system can therefore vary depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used. It can range from a few megabytes of disk space to gigabytes.

Some operating systems, such as UNIX , allow the use of multiple swap spaces. These swap spaces are usually put on separate disks, so the load placed on the I/O system by paging and swapping can be spread over the system's I/O devices. Note that it is safer to overestimate than to underestimate swap space, because if a system runs out of swap space it may be forced to abort processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but does no other harm.

13.4.2 Swap-Space Location

A swap space can reside in two places: Swap space can be carved out of the normal file system, or it can be in a separate disk partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space. This approach, though easy to implement, is also inefficient. Navigating the directory structure and the disk-allocation data structures takes time and (potentially) extra disk accesses. External fragmentation can greatly increase swapping times by forcing multiple seeks during reading or writing of a process image. We can improve performance by caching the block location information in physical memory, and by using special tools to allocate physically contiguous blocks for the swap file, but the cost of traversing the file-system data structures still remains.

Alternatively, swap space can be created in a separate disk partition. No file system or directory structure is placed on this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks. This manager uses algorithms optimized for speed, rather than for storage efficiency. Internal fragmentation may increase, but this tradeoff is acceptable because data in the swap space generally live for much shorter amounts of time than do files in the file system, and the swap area may be accessed much more frequently. This approach creates a fixed amount of swap space during disk partitioning. Adding more swap space can be done only via repartitioning of the disk (which involves moving or destroying and restoring the other file-system partitions from backup), or via adding another swap space elsewhere.

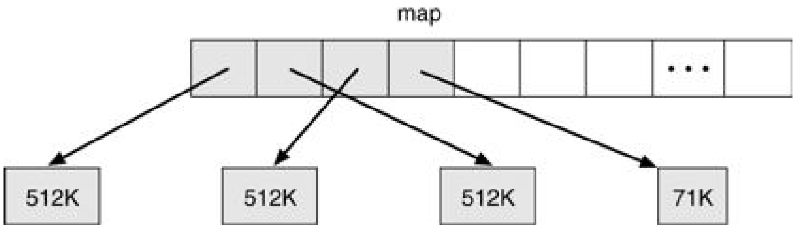
Some operating systems are flexible and can swap both in raw partitions and in filesystem space. Solaris 2 is an example. The policy and implementation are separate, allowing the machine's administrator to decide which type to use. The tradeoff is between the convenience of allocation and management in the file system, and the performance of swapping in raw partitions.

13.4.3 Swap-Space Management: An Example

To illustrate the methods used to manage swap space, we now follow the evolution of swapping and paging in UNIX. As discussed fully in Appendix A, UNIX started with an implementation of swapping that copied entire processes between contiguous disk regions and memory. UNIX evolved to a combination of swapping and paging, as paging hardware became available. In 4.3 BSD, swap space is allocated to a process when the process is started. Enough space is set aside to hold the program, known as the **text pages** or the **text segment**, and the **data segment** of the process.

Preallocating all the needed space in this way generally prevents a process from running out of swap space while it executes. When a process starts, its text is paged in from the file system. These pages are written out to swap when necessary, and are read back in from there, so the file system is consulted only once for each text page. Pages from the data segment are read in from the file system, or are created (if they are uninitialized), and are written to swap space and paged back in as needed. One optimization (for instance, when two users run the same editor) is that processes with identical text pages share these pages, both in physical memory and in swap space. Two per-process **swap maps** are used by the kernel to track swap-space use. The text segment is a fixed size, so its swap space is allocated in 512 KB chunks, except for the final chunk, which holds the remainder of the pages, in 1 KB increments (Figure 13.7).

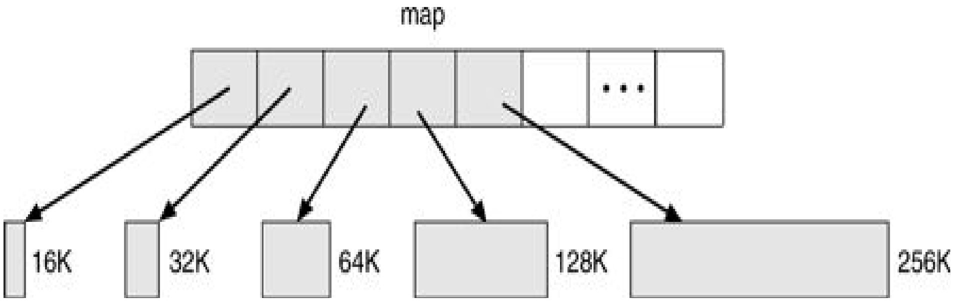
Figure 13.7 4.3 BSD text-segment swap map.



The data-segment swap map is more complicated, because the data segment can grow over time. The map is of fixed size, but contains swap addresses for blocks of varying size. Given index *i*, a block pointed to by swap-map entry *i* is of size 2*i* × 16 KB, to a maximum of 2 MB. This data structure is shown in Figure 13.8. (The block size minimum and maximum are variable, and can be changed at system reboot.) When a process tries to grow its data segment beyond the final allocated block in its swap area, the operating system allocates another block, twice as large as the previous one. This scheme results in small processes using only small blocks. It also minimizes fragmentation. The blocks of large processes can be found quickly, and the swap map remains small.

In Solaris 1 (sunOS 4), the designers made changes to standard UNIX methods to improve efficiency and reflect technological changes. When a process executes, text-segment pages are brought in from the file system, accessed in main memory, and thrown away if selected for pageout. It is more efficient to reread a page from the file system than to write it to swap space and then to reread it from there. More changes were made in Solaris 2. The biggest change is that Solaris 2 allocates swap space only when a page is forced out of physical memory, rather than when the virtual-memory page is first created. This change gives better performance on modern computers, which have more physical memory than older systems and tend to page less.

Figure 13.8 4.3 BSD data-segment swap map.



13.5 RAID Structure

Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach a large number of disks to a computer system. Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data. A variety of disk-organization techniques, collectively called redundant arrays of inexpensive disks (RAID), are commonly used to address the performance and reliability issues.

In the past, RAIDs composed of small cheap disks were viewed as a cost-effective alternative to large, expensive disks; today, RAIDs are used for their higher reliability and higher data-transfer rate, rather than for economic reasons. Hence, the *I* in *RAID* stands for "independent", instead of "inexpensive."

13.5.1 Improvement of Reliability via Redundancy

Let us first consider reliability. The chance that some disk out of a set of *N* disks will fail is much higher than the chance that a specific single disk will fail. Suppose that the **mean time to failure** of a single disk is 100,000 hours. Then, the mean time to failure of some disk in an array of 100 disks will be 100,000/100 = 1,000 hours, or 41.66 days, which is not long at all! If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data—such a high rate of data loss is unacceptable.

The solution to the problem of reliability is to introduce redundancy; we store extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information. Thus, even if a disk fails, data are not lost. The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called mirroring (or shadowing). A logical disk then consists of two physical disks, and every write is carried out on both disks. If one of the disks fails, the data can be read from the other. Data will be lost only if the second disk fails before the first failed disk is replaced. The mean time to failure—where *failure* is the loss of data—of a mirrored disk depends on two factors: the mean time to failure of the individual disks, as well as on the mean time to repair, which is the time it takes (on average) to replace a failed disk and to restore the data on it. Suppose that the failures of the two disks are independent; that is, the failure of one disk is not connected to the failure of the other. Then, if the mean time to failure of a single disk is 100,000 hours and the mean time to repair is 10 hours, then the mean time to data loss of a mirrored disk system is 100,0002/(2 * 10) = 500 * 106 hours, or 57,000 years! You should be aware that the assumption of independence of disk failures is not valid.

Power failures and natural disasters, such as earthquakes, fires, and floods, may result in damage to both disks at the same time. Also, manufacturing defects in a batch of disks can cause correlated failures. As disks age, the probability of failure increases, increasing the chance that a second disk will fail while the first is being repaired. In spite of all these considerations, however, mirrored-disk systems offer much higher reliability than do single-disk systems. Power failures are a particular source of concern, since they occur far more frequently than do natural disasters. However, even with mirroring of disks, if writes are in progress to the same block in both disks, and power fails before both blocks are fully written, the two blocks can be in an inconsistent state.

The solution to this problem is to write one copy first, then the next, so that one of the two copies is always consistent. Some extra actions are required when we restart after a power failure, to recover from incomplete writes.

13.5.2 Improvement in Performance via Parallelism

Now let us consider the benefit of parallel access to multiple disks. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single disk system, but the number of reads per unit time has doubled. With multiple disks, we can improve the transfer rate as well (or instead) by striping data across multiple disks. In its simplest form, **data striping** consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**.

For example, if we have an array of eight disks, we write bit *i* of each byte to disk *i*. The array of eight disks can be treated as a single disk with sectors that are eight times the normal size, and, more important, that have eight times the access rate. In such an organization, every disk participates in every access (read or write), so the number of accesses that can be processed per second is about the same as on a

single disk, but each access can read eight times as many data in the same time as on a single disk.

Bit-level striping can be generalized to a number of disks that either is a multiple of 8 or divides 8. For example, if we use an array of four disks, bits *i* and 4 + *i* of each byte go to disk *i*. Further, striping does not need to be at the level of bits of a byte: For example, in block-level striping, blocks of a file are striped across multiple disks; with *n* disks, block *i* of a file goes to disk (*i* mod *n*) + 1. Other levels of striping, such as bytes of a sector or sectors of a block, also are possible.

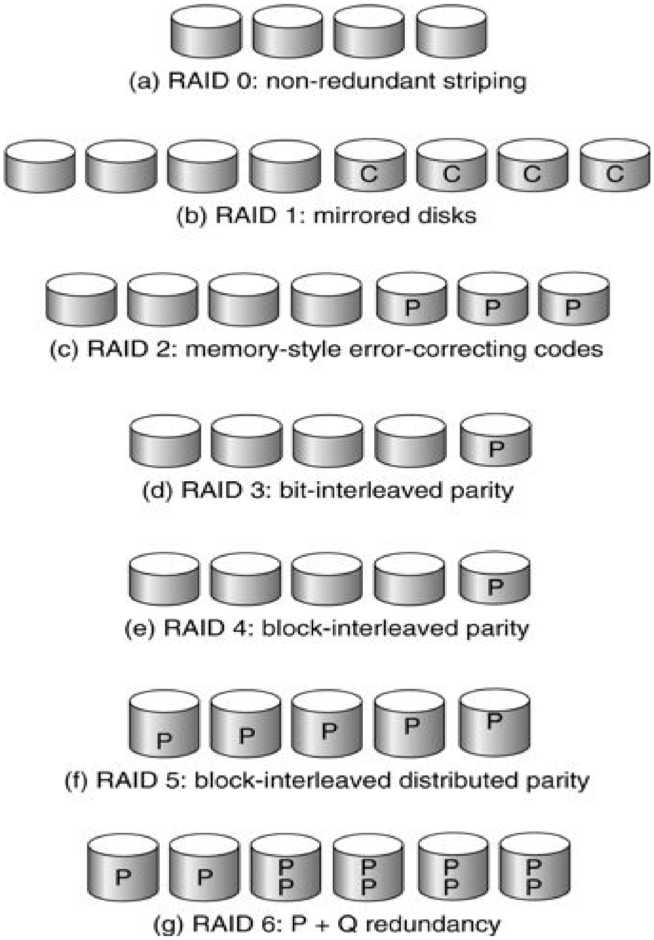
In summary, there are two main goals of parallelism in a disk system:

1. Increase the throughput of multiple small accesses (that is, page accesses) by load balancing.
2. Reduce the response time of large accesses.

13.5.3 RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost by using the idea of disk striping combined with "parity" bits (which we describe next) have been proposed. These schemes have different cost-performance tradeoffs and are classified into levels called **RAID levels**. We describe the various levels here; Figure 13.9 shows them pictorially (in the figure, *P* indicates error-correcting bits and *C* indicates a second copy of the data). In all cases depicted in the figure, four disks' worth of data is stored, and the extra disks are used to store redundant information for failure recovery.

Figure 14.9 RAID levels.



(a) RAID 0: non-redundant striping

(b) RAID 1: mirrored disks

(c) RAID 2: memory-style error-correcting codes

(d) RAID 3: bit-interleaved parity

(e) RAID 4: block-interleaved parity

(f) RAID 5: block-interleaved distributed parity

(g) RAID 6: P + Q redundancy

• **RAID Level 0:** RAID level 0 refers to disk arrays with striping at the level of blocks, but without any redundancy (such as mirroring or parity bits). Figure 13.9a shows an array of size 4.

• **RAID Level 1:** RAID level 1 refers to disk mirroring. Figure 13.9b shows a mirrored organization that holds four disks' worth of data.

• **RAID Level 2:** RAID level 2 is also known as **memory-style error correcting-code (ECC) organization**. Memory systems have long implemented error detection using parity bits. Each byte in a memory system may have a parity bit associated with it that records whether the numbers of bits in the byte set to 1 is even (parity=0) or odd (parity=1). If one of the bits in the byte gets damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity. Similarly, if the stored parity bit gets damaged, it will not match the computed parity.

Thus, all single-bit errors are detected by the memory system. Error-correcting schemes store two or more extra bits, and can reconstruct the data if a single bit gets damaged. The idea of ECC can be used directly in disk arrays via striping of bytes across disks. For example, the first bit of each byte could be stored in disk 1, the second bit in disk 2, and so on until the eighth bit is stored in disk 8, and the error-correction bits are stored in further disks. This scheme is shown pictorially in Figure 13.9, where the disks labeled *P* store the error-correction bits. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks and be used to reconstruct the damaged data. Figure 13.9c shows an array of size 4; note RAID level 2 requires only three disks' overhead for four disks of data, unlike RAID level 1, which required four disks' overhead.

• **RAID level 3:** RAID level 3, or **bit-interleaved parity organization**, improves on level 2 by noting that, unlike memory systems, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection. The idea is as follows. If one of the sectors gets damaged, we know exactly which sector it is, and, for each bit in the sector, we can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1. RAID level 3 is as good as level 2 but is less expensive in the number of extra disks (it has only a one-disk overhead), so level 2 is not used in practice. This scheme is shown pictorially in Figure 13.9d. RAID level 3 has two benefits over level 1. Only one parity disk is needed for several regular disks, unlike one mirror disk for every disk in level 1, thus reducing the storage overhead. Since reads and writes of a byte are spread out over multiple disks, with *N*-way striping of data, the transfer rate for reading or writing a single block is *N* times as fast as with a RAID-level-1 organization using *N*-way striping. On the other hand, RAID level 3 supports a lower number of I/Os per second, since every disk has to participate in every I/O request.

A further performance problem with RAID 3 (as with all parity-based RAID levels) is the expense of computing and writing the parity. This overhead results in significantly slower writes, as compared to non-parity RAID arrays. To moderate this performance penalty, many RAID storage arrays include a hardware controller with dedicated parity hardware. This offloads the parity computation from the CPU to the array. The array has a **nonvolatile RAM (NVRAM)** cache as well, to store the blocks while the parity is computed and to buffer the writes from the controller to the spindles. This combination can make parity RAID almost as fast as non-parity. In fact, a caching array doing parity RAID can outperform a non-caching non-parity RAID.

• **RAID Level 4:** RAID level 4, or **block-interleaved parity organization**, uses block-level striping, as in RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from *N* other disks. This scheme is shown pictorially in Figure 13.9e. If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.

A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. The transfer rates for large reads is high, since all the disks can be read in parallel; large writes also have high transfer rates, since the data and parity can be written in parallel.

Small independent writes, on the other hand, cannot be performed in parallel. A write of a block has to access the disk on which the block is stored, as well as the parity disk, since the parity block has to be updated. Moreover, both the old value of the parity block and the old value of the block being written have to be read for the new parity to be computed. This is known as the **read-modify-write**. Thus, a single write requires four disk accesses: two to read the two old blocks, and two to write the two new blocks.

• **RAID level 5:** RAID level 5, or **block-interleaved distributed parity**, differs from level 4 by spreading data and parity among all *N* + 1 disks, rather than storing data in *N* disks and parity in one disk. For each block, one of the disks stores the parity, and the others store data. For example, with an array of five disks, the parity for the *n*th block is stored in disk (*n* mod 5) + 1; the *n*th blocks of the other four disks store actual data for that block. This setup is denoted pictorially in Figure 13.9f, where the *P*s are distributed across all the disks. A parity block cannot store parity for blocks in the same disk, because a disk failure would result in loss of data as well as of parity, and hence would not be recoverable. By spreading the parity across all the disks in the set, RAID 5 avoids the potential overuse of a single parity disk that can occur with RAID 4.

• **RAID Level 6:** RAID level 6, also called the **P+Q redundancy scheme**, is much like RAID level 5, but stores extra redundant information to guard against multiple disk failures. Instead of using parity, error-correcting codes such as the **Reed-Solomon codes** are used. In the scheme shown in Figure 13.9g, 2 bits of redundant data are stored for every 4 bits of data—unlike 1 parity bit in level 5— and the system can tolerate two disk failures.

• **RAID level 0 + 1:** RAID level 0 + 1 refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability. Generally, it provides better performance than RAID 5. It is common in environments where both performance and reliability are important. Unfortunately, it doubles the number of disks needed for storage, as does RAID 1, so it is also more expensive. In RAID 0 + 1, a set of disks are striped, and then the stripe is mirrored to another, equivalent stripe. Another RAID option that is becoming available commercially is RAID 1 + 0, in which disks are mirrored in pairs, and then the resulting mirror pairs are striped. This RAID has some theoretical advantages over RAID 0 + 1. For example, if a single disk fails in RAID 0 + 1, the entire stripe is inaccessible, leaving only the other stripe available. With a failure in RAID 1 + 0, the single disk is unavailable, but its mirrored pair is still available as are all the rest of the disks (Figure 13.10).

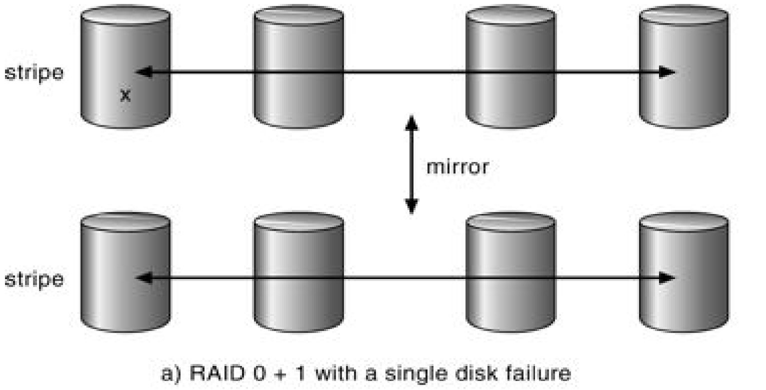
Finally, we note that numerous variations have been proposed to the basic RAID schemes described here. As a result, some confusion may exist about the exact definitions of the different RAID levels.

13.5.4 Selecting a RAID Level

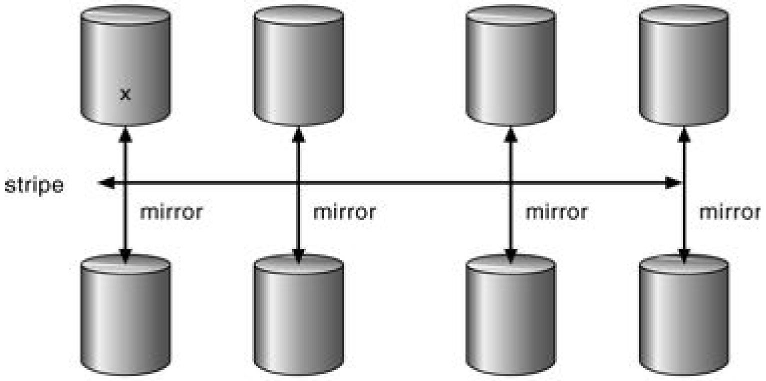
If a disk fails, the time to rebuild its data can be significant and will vary with the RAID level used. Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data in a failed disk. The rebuild performance of a RAID system may be an important factor if continuous supply of data is required, as it is in high performance or interactive database systems. Furthermore, rebuild performance influences the mean time to failure. RAID level 0 is used in high-performance applications where data loss is not critical. RAID level 1 is popular for applications that require high reliability with fast recovery. RAID 0 + 1 and 1 + 0 are used where performance and reliability are important, for example for small databases. Due to RAID 1's high space overhead, RAID level 5 is often preferred for storing large volumes of data. Level 6 is not supported currently by many RAID implementations, but it should offer better reliability than level 5.

RAID system designers have to make several other decisions as well. For example, how many disks should be in an array? How many bits should be protected by each parity bit? If more disks are in an array, data-transfer rates are higher, but the system is more expensive. If more bits are protected by a parity bit, the space overhead due to parity bits is lower, but the chance that a second disk will fail before the first failed disk is repaired is greater, and that will result in data loss. One other aspect of most RAID implementations is a hot spare disk or disks. A **hot spare** is not used for data, but is configured to be used as a replacement should any other disk fail. For instance, a hot spare can be used to rebuild a mirror pair should one of the disks in the pair fail. In this way, the RAID level can be reestablished automatically, without waiting for the failed disk to be replaced. Allocating more than one hot spare allows more than one failure to be repaired without human intervention.

Figure 13.10 RAID 0 + 1 and 1 + 0.



a) RAID 0 + 1 with a single disk failure



b) RAID 1 + 0 with a single disk failure

13.5.5 Extensions

The concepts of RAID have been generalized to other storage devices, including arrays of tapes, and even to the broadcast of data over wireless systems. When applied to arrays of tapes, the RAID structures are able to recover data even if one of the tapes in an array of tapes is damaged. When applied to broadcast of data, a block of data is split into short units and is broadcast along with a parity unit; if one of the units is not received for any reason, it can be reconstructed from the other units. Commonly, tape-drive robots containing multiple tape drives will stripe data across all the drives to increase throughput and decrease backup time

.

13.6 Disk Attachment

Computers access disk storage in two ways. One way is via I/O ports (or **host attached storage**); this is common on small systems. The other way is via a remote host via a distributed file system; this is referred to as **network-attached storage**.

13.6.1 Host-Attached Storage

Host-attached storage is storage accessed via local I/O ports. These ports are available in several technologies. The typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus. High-end workstations and servers generally use more sophisticated I/O architectures such as SCSI and fibre channel (FC).SCSI is a bus architecture. Its physical medium is usually a ribbon cable having a large number of conductors (typically 50 or 68). The SCSI protocol supports a maximum of 16 devices on the bus. Typically this consists of one controller card in the host (the **SCSI initiator**), and up to 15 storage devices (the **SCSI targets**).

A SCSI disk is a typical SCSI target, but the protocol provides the ability to address up to 8 **logical units** in each SCSI target. A typical use of logical unit addressing is to direct commands to components of a RAID array, or components of a removable media library (such as a CD jukebox sending commands to the media changer mechanism or to one of the drives). FC is a high-speed serial architecture. This architecture can operate over optical fiber or over a 4-conductor copper cable. It has two variants. One is a large switched fabric having a 24-bit address space. This method is expected to dominate in the future, and is the basis of **storage-area networks (SANs)**. Because of the large address space and the switched nature of the communication, multiple hosts and storage devices can attach to the fabric, allowing great flexibility in I/O communication. The other is an **arbitrated loop (FC-AL)** that can address 126 devices (drives and controllers).

A wide variety of storage devices are suitable for use as host-attached storage. Among these are hard disk drives, RAID arrays, and CD, DVD, and tape drives. The I/O commands that initiate data transfers to a host-attached storage device are reads and writes of logical data blocks, directed to specifically identified storage units (such as bus ID, SCSI ID, and target logical unit, for example).

13.6.2 Network-Attached Storage

A network-attached storage device is a special-purpose storage system that is accessed remotely over a data network (Figure 13.11). Clients access network attached storage (NAS) via a remote-procedure-call interface such as NFS for UNIX systems, or CIFS for Windows machines. The remote procedure calls (RPCs) are carried via TCP or

UDP over an IP network—usually the same local-area network (LAN) that carries all data traffic to the clients. The network-attached storage unit is

usually implemented as a RAID array with software that implements the remote procedure call interface. It is easiest to think of NAS as simply another storage access protocol. For example, rather than using a SCSI device driver and SCSI protocols to access storage, a system using NAS would use RPC over TCP/IP. Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage, with the same ease of naming and access enjoyed with local host-attached storage. However, it tends to be less efficient and have lower performance than some direct-attached storage options.

13.6.3 Storage-Area Network

One drawback of network-attached storage systems is that the storage I/O operations consume bandwidth on the data network, thereby increasing the latency of network communication. This problem can be particularly acute in large client-server installations—the communication between servers and clients competes for bandwidth with the communication among servers and storage devices.

Figure 13.11 Network-attached storage.

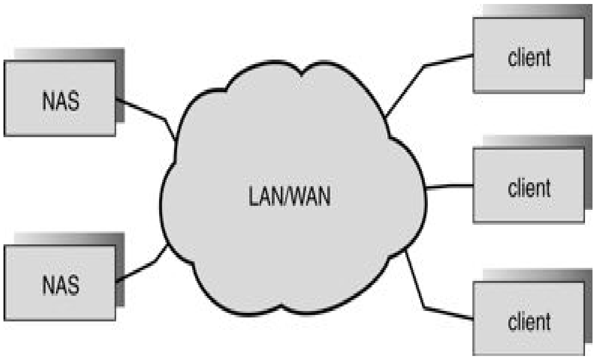
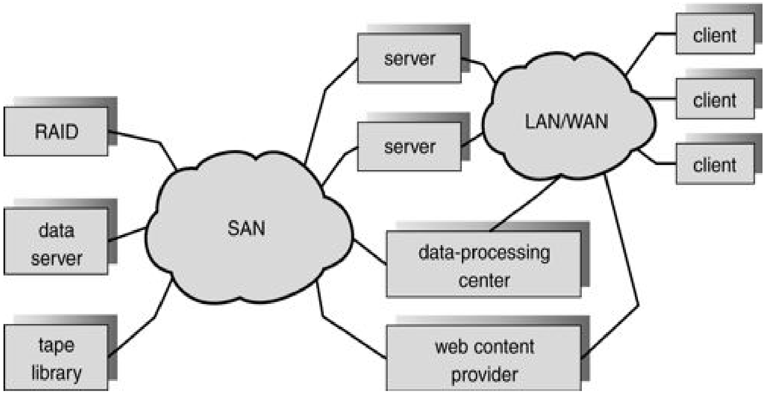


Figure 13.12 Storage-area network.



A storage-area network (SAN) is a private network (using storage protocols rather than networking protocols) among the servers and storage units, separate from the LAN or WAN that connects the servers to the clients (Figure 13.12). The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN , and storage can be dynamically allocated to hosts. As one example, if a host is running low on disk space, the SAN can be configured to allocate more storage to that host. In 2003, many proprietary single-vendor SAN systems are available, but SAN components are not well standardized or interoperable.

Most SAN systems in 2003 are based on fibre-channel loops or fibre-channel switched networks. One emerging alternative to a fibre-channel interconnect for the SAN is storage over IP network infrastructure such as Gigabit Ethernet. Another potential alternative is a special-purpose SAN architecture named Infiniband, which provides hardware and software support for high-speed interconnection networks for servers and storage units.

13.7 Stable-Storage Implementation

In the previous Chapters, we introduced the write-ahead log, which required the availability of stable storage. By definition, information residing in stable storage is *never* lost. To implement such storage, we need to replicate the needed information on multiple storage devices (usually disks) with independent failure modes. We need to coordinate the writing of updates in a way that guarantees that a failure during an update will not leave all the copies in a damaged state, and that, when we are recovering from a failure, we can force all copies to a consistent and correct value, even if another failure occurs during the recovery. In the remainder of this section, we discuss how to meet our needs.

A disk write results in one of three outcomes:

1. **Successful completion:** The data were written correctly on disk.

2. **Partial failure:** A failure occurred in the midst of transfer, so only some of the sectors were written with the new data, and the sector being written during the failure may have been corrupted.

3. **Total failure:** The failure occurred before the disk write started, so the

previous data values on the disk remain intact.

We require that, whenever a failure occurs during writing of a block, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do that, the system must maintain two physical blocks for each logical block. An output operation is executed as follows:

1. Write the information onto the first physical block.

2. When the first write completes successfully, write the same information onto the second physical block.

3. Declare the operation complete only after the second write completes successfully.

During recovery from a failure, each pair of physical blocks is examined. If both are the same and no detectable error exists, then no further action is necessary. If one block contains a detectable error, then we replace its contents with the value of the other block. If both blocks contain no detectable error, but they differ in content, then we replace the content of the first block with the value of the second. This recovery procedure ensures that a write to stable storage either succeeds completely or results in no change.

We can extend this procedure easily to allow the use of an arbitrarily large number of copies of each block of stable storage. Although a large number of copies further reduces the probability of a failure, it is usually reasonable to simulate stable storage with only two copies. The data in stable storage are guaranteed to be safe unless a failure destroys all the copies.

Because waiting for disk writes to complete (synchronous I/O) is time consuming, many storage arrays add NVRAM as a cache. Because the memory is nonvolatile (usually it has battery power as a backup to the unit's power), it can be trusted to store the data on its way to the disks. It is thus considered part of the stable storage. Writes to it are much faster than to disk, so performance is greatly improved.

13.8 Tertiary-Storage Structure

Would you buy a VCR that had inside it only one tape that you could not take out or replace? Or an audio cassette player or CD player that had one album sealed inside? Of course not. You expect to use a VCR or CD player with many relatively inexpensive tapes or disks. On a computer as well, using many inexpensive cartridges with one drive lowers the overall cost.

13.8.1 Tertiary-Storage Devices

Low cost is the defining characteristic of tertiary storage. So, in practice, tertiary storage is built with **removable media**. The most common examples of removable media are floppy disks, CD-ROMs, and tapes; many other kinds of tertiary-storage devices are available as well.

13.8.1.1 Removable Disks

Removable disks are one kind of tertiary storage. Floppy disks are an example of removable magnetic disks. They are made from a thin flexible disk coated with magnetic material, enclosed in a protective plastic case. Although common floppy disks can hold only about 1 MB, similar technology is used for removable magnetic disks that hold more than 1 GB. Removable magnetic disks can be nearly as fast as hard disks, although the recording surface is at greater risk of damage from scratches.

A **magneto-optic disk** is another kind of removable disk. It records data on a rigid platter coated with magnetic material, but the recording technology is quite different from that for a magnetic disk. The magneto-optic head flies much farther from the disk surface than a magnetic disk head does, and the magnetic material is covered with a thick protective layer of plastic or glass. This arrangement makes the disk much more resistant to head crashes.

The drive has a coil that produces a magnetic field; at room temperature, the field is too large and too weak to magnetize a bit on the disk. To write a bit, the disk head flashes a laser beam at the disk surface. The laser is aimed at a tiny spot where a bit is to be written. The laser heats this spot, which makes the spot susceptible to the magnetic field. So the large, weak magnetic field can record a tiny bit.

The magneto-optic head is too far from the disk surface to read the data by detecting the tiny magnetic fields in the way that the head of a hard disk does. Instead, the drive reads a bit using a property of laser light called the **Kerr effect**. When a laser beam is bounced off of a magnetic spot, the polarization of the laser beam is rotated clockwise or counter-clockwise, depending on the orientation of the magnetic field. This rotation is what the head detects to read a bit.

Another category of removable disk is the **optical disk**. These disks do not use magnetism at all. They use special materials that can be altered by laser light to have relatively dark or bright spots. One example of optical-disk technology is the phase-change disk. The **phase-change disk** is coated with a material that can freeze into either a crystalline or an amorphous state. The crystalline state is more transparent, and hence a laser beam is brighter when it passes through the phase-change material and bounces off the reflective layer.

The phase-change drive uses laser light at three different powers: low power to read data, medium power to erase the disk by melting and refreezing the recording medium into the crystalline state, and a high power to melt the medium into the amorphous state to write to the disk. The most common examples of this technology are the re-recordable CD-RW and DVDRW. The kinds of disks described here can be used over and over. They are called **read-write disks**. In contrast, **write-once, read-many-times (WORM) disks** form another category. An old way to make a WORM disk is to manufacture a thin aluminum film sandwiched between two glass or plastic platters.

To write a bit, the drive uses a laser light to burn a small hole through the aluminum. Because this burning cannot be reversed, any sector on the disk can be written only once. Although it is possible to destroy the information on a WORM disk by burning holes everywhere, it is virtually impossible to alter data on the disk, because holes can only be added, and the ECC code associated with each sector is likely to detect such additions. WORM disks are considered to be durable and reliable because the metal layer is safely encapsulated between the protective glass or plastic platters, and magnetic fields cannot damage the recording. A newer write-once technology records on an organic polymer dye instead of an aluminum layer: the dye absorbs laser light to form marks.

This technology is used in the recordable CD-R and DVD-R. **Read-only disks**, such as CD-ROM and DVD, come from the factory with the data pre-recorded. They use technology similar to that of WORM disks (although the pits are pressed, not burnt), and they are very durable. Most removable disks are slower than their non-removable counterparts. The writing process is slower, as are rotation and sometimes seek time.

13.8.1.2 Tapes

Magnetic tape is another type of removable medium. As a general rule, a tape holds more data than an optical or magnetic disk cartridge. Tape drives and disk drives have similar transfer rates. But random access to tape is much slower than a disk seek, because it requires a fast-forward or rewind operation that takes tens of seconds, or even minutes. Although a typical tape drive is more expensive than a typical disk drive, the price of a tape cartridge is lower than the price of the equivalent capacity of magnetic disks. So tape is an economical medium for purposes that do not require fast random access. Tapes are commonly used to hold backup copies of disk data. They are also used in large supercomputer centers to hold the enormous volumes of data used in scientific research and by large commercial enterprises.

Some tapes can hold much more data than can a disk drive; the surface area of a tape is much larger than the surface area of a disk. The storage capacity of tapes could improve even further, because at present the **areal density** (or bits per square inch) of tape technology is much less than that for magnetic disks. Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library. These mechanisms give the computer automated access to a large number of tape cartridges. A robotic tape library can lower the overall cost of data storage. A disk-resident file that will not be needed for a while can be **archived** to tape, where the cost per gigabyte can be lower; if the file is needed in the future, the computer can **stage** it

back into disk storage for active use. A robotic tape library is sometimes called **near-line** storage, since it is between the high performance of **on-line** magnetic disks and the low cost of **off-line** tapes sitting on shelves in a storage room.

13.8.1.3 Future Technology

In the future, other storage technologies may become important. One promising storage technology, **holographic storage**, uses laser light to record holographic photographs on special media. We can think of a black-and-white photograph as a two-dimensional array of pixels. Each pixel represents one bit: 0 for black, or 1 for white. A sharp photograph can hold millions of bits of data. And all the pixels in a hologram are transferred in one flash of laser light, so the data rate is extremely high. With continued development, holographic storage may become commercially viable.

Another storage technology under active research is based on **microelectronic mechanical systems (MEMS)**. The idea is to apply the fabrication technologies that produce electronic chips in order to manufacture small data storage machines. One proposal calls for the fabrication of an array of 10,000 tiny disk heads, with a square centimeter of magnetic storage material suspended above the array. When the storage material is moved lengthwise over the heads, each head accesses its own linear track of data on the material. The storage material can be shifted sideways slightly to enable all the heads to access their next track. Although it remains to be seen whether this technology can be successful, it may provide a nonvolatile data storage technology that is faster than magnetic disk and cheaper than semiconductor DRAM.

Whether the storage medium is a removable magnetic disk, a DVD, or a magnetic tape, the operating system needs to provide several capabilities to use removable media for data storage. These capabilities are discussed in Section 13.8.2.

13.8.2 Operating-System Jobs

Two major jobs of an operating system are to manage physical devices and to present a virtual-machine abstraction to applications. In this chapter, we saw that, for hard disks, the operating system provides two abstractions. One is the raw device, which is just an array of data blocks. The other is a file system. For a file system on a magnetic disk, the operating system queues and schedules the interleaved requests from several applications. Now, we shall see how the operating system does its job when the storage media are removable.

13.8.2.1 Application Interface

Most operating systems can handle removable disks almost exactly as they do fixed disks. When a blank cartridge is inserted into the drive (or mounted), the cartridge must be formatted, and then an empty file system is generated on the disk. This file system is used just like a file system on a hard disk. Tapes are often handled differently. The operating system usually presents a tape as a raw storage medium. An application does not open a file on the tape; it opens the whole tape drive as a raw device. Usually, the tape drive then is reserved for the exclusive use of that application until the application exits or closes the tape device.

This exclusivity makes sense, because random access on a tape can take tens of seconds, or even a few minutes, so interleaving random accesses to tapes from more than one application would be likely to cause thrashing. When the tape drive is presented as a raw device, the operating system does not provide file-system services. The application must decide how to use the array of blocks. For instance, a program that backs up a hard disk to tape might store a list of file names and sizes at the beginning of the tape, and then copy the data of the files to the tape in that order.

It is easy to see the problems that can arise from this way of using tape. Since every application makes up its own rules for how to organize a tape, a tape full of data can generally be used by only the program that created it. For instance, even if we know that a backup tape contains a list of file names and file sizes followed by the file data in that order, we still would find it difficult to use the tape. How exactly are the file names stored? Are the file sizes in binary or in ASCII? Are the files written one per block, or are they all concatenated together in one tremendously long string of bytes? We do not even know the block size on the tape, because this variable is generally one that can be chosen separately for each block written.

For a disk drive, the basic operations are read, write, and seek. Tape drives, on the other hand, have a different set of basic operations. Instead of seek, a tape drive uses the locate operation. The tape locate operation is more precise than the disk seek operation, because it positions the tape to a specific logical block, rather than an entire track. Locating to block 0 is the same as rewinding the tape. For most kinds of tape drives, it is possible to locate to any block that has been written on a tape. In a partly filled tape, however, it is not possible to locate into the empty space beyond the written area, because most tape drives manage their physical space differently from disk drives. For a disk drive, the sectors have a fixed size, and the formatting process must be used to place empty sectors in their final positions before any data can be written. Most tape drives have a variable block size, and the size of each block is determined on the fly, when that block is written. If an area of defective tape is encountered during writing, the bad area is skipped and the block is written again.

This operation explains why it is not possible to locate into the empty space beyond the written area—the positions and numbers of the logical blocks have not yet been determined. Most tape drives have a read position operation that returns the logical block number where the tape head is. Many tape drives also support a space operation for relative motion. So, for example, the operation space -2 would locate backward over two logical blocks. For most kinds of tape drives, writing a block has the side effect of logically erasing everything beyond the position of the write. In practice, this side effect means that most tape drives are append-only devices, because updating a block in the middle of the tape also effectively erases everything beyond that block. The tape drive implements this appending by placing an end-of-tape (EOT) mark after a block that is written. The drive refuses to locate past the EOT mark, but it is possible to locate to the EOT and then to start writing. Doing so overwrites the old EOT mark, and places a new one at the end of the new blocks just written. In principle, a file system can be implemented on a tape. But many of the file system data structures and algorithms would be different from those used for disks, because of the append-only property of tape.

13.8.2.2 File Naming

Another question that the operating system needs to handle is how to name files on removable media. For a fixed disk, naming is not difficult. On a PC, the file name consists of a drive letter followed by a path name. In UNIX, the file name does not contain a drive letter, but the mount table enables the operating system to discover on what drive the file is located. But if the disk is removable, knowing a drive that contained the cartridge at some time in the past does not mean knowing how to find the file. If every removable cartridge in the world had a different serial number, the name of a file on a removable device could be prefixed with the serial number, but to ensure that no two serial numbers are the same would require each one to be about 12 digits in length. Who could remember the names of her files if she had to memorize a 12-digit serial number for each one? The problem becomes even more difficult when we want to write data on a removable cartridge on one computer, and then use the cartridge in another computer. If both machines are of the same type and have the same kind of removable drive, the only difficulty is knowing the contents and data layout on the cartridge. But if the machines or drives are different, many additional problems can arise. Even if the drives are compatible, different computers may store bytes in different orders, and may use different encodings for binary numbers and even for letters (such as ASCII on PCs versus EBCDIC on mainframes).

Today's operating systems generally leave the name-space problem unsolved for removable media, and depend on applications and users to figure out how to access and interpret the data. Fortunately, a few kinds of removable media are so well standardized that all computers use them the same way. One example is the CD. Music CDs use a universal format that is understood by any CD drive. Data CDs are available in only a few different formats, so it is usual for a CD drive and the operating-system device driver to be programmed to handle all the common formats. DVD formats are also well standardized.

13.8.2.3 Hierarchical Storage Management

A **robotic jukebox** enables the computer to change the removable cartridge in a tape or disk drive without human assistance. Two major uses of this technology are for backups and hierarchical storage systems. The use of a jukebox for backups is simple: when one cartridge becomes full, the computer instructs the jukebox to switch to the next cartridge. Some jukeboxes hold tens of drives and thousands of cartridges, with robotic arms managing the movement of tapes to the drives.

A hierarchical storage system extends the storage hierarchy beyond primary memory and secondary storage (that is, magnetic disk) to incorporate tertiary storage. Tertiary storage is usually implemented as a jukebox of tapes or removable disks. This level of the storage hierarchy is larger, cheaper, and probably slower. Although the virtual-memory system can be extended in a straightforward manner to tertiary storage, this extension is rarely carried out in practice. The reason is that a retrieval from a jukebox can take tens of seconds or even minutes, and such a long delay is intolerable for demand paging and for other forms of virtual memory use.

The usual way to incorporate tertiary storage is to extend the file system. Small and frequently used files remain on magnetic disk, while large and old files that are not actively used are archived to the jukebox. In some file-archiving systems, the directory entry for the file continues to exist, but the contents of the file no longer occupy space in secondary storage. If an application tries to open the file, the open system call is suspended until the file contents can be staged in from tertiary storage. When the contents are again available from magnetic disk, the open operation returns control to the application, which proceeds to use the disk resident copy of the data. **Hierarchical storage management (HSM)** has been implemented in ordinary time-sharing systems such as TOPS-20, which ran on minicomputers from Digital Equipment Corporation in the late 1970s. Today, HSM is usually found in supercomputing centers and other large installations that have enormous volumes of data.

13.8.3 Performance Issues

As with any component of the operating system, the three most important aspects of tertiary-storage performance are speed, reliability, and cost.

13.8.3.1 Speed

The speed of tertiary storage has two aspects: bandwidth and latency. We measure the bandwidth in bytes per second. The **sustained bandwidth** is the average data rate during a large transfer, that is, the number of bytes divided by the transfer time. The **effective bandwidth** calculates the average over the entire I/O time, including the time for seek or locate and any cartridge-switching time in a jukebox. In essence, the sustained bandwidth is the data rate when the data stream is actually flowing, and the effective bandwidth is the overall data rate provided by the drive. The *bandwidth of a drive* is generally understood to mean the sustained bandwidth.

For removable disks, the bandwidth ranges from less than 0.25 MB per second for the slowest, to several megabytes per second for the fastest. Tapes have an even wider range of bandwidths, from less than 0.25 MB per second to over 30 MB per second. The fastest tape drives have significantly higher bandwidth than do removable disk drives. The second aspect of speed is the **access latency**. By this performance measure, disks are much faster than tapes: Disk storage is essentially two-dimensional—all the bits are out in the open. A disk access simply moves the arm to the selected cylinder and waits for the rotational latency, which may take less than 5 milliseconds. By contrast, tape storage is three-dimensional. At any time, a small portion of the tape is accessible to the head, whereas most of the bits are buried below hundreds or thousands of layers of tape wound on the reel. A random access on tape requires winding the tape reels until the selected block reaches the tape head, which can take tens or hundreds of seconds. So we can generally say that random access within a tape cartridge is more than a thousand times slower than random access on disk.

If a jukebox is involved, the access latency can be significantly higher. For a removable disk to be changed, the drive must stop spinning, then the robotic arm must switch the disk cartridges, and the drive must spin up the new cartridge. This operation takes several seconds—about a hundred times longer than the random access time within one disk. So switching disks in a jukebox incurs a relatively high performance penalty. For tapes, the robotic-arm time is about the same as for disk. But for tapes to be switched, the old tape generally must rewind before it can be ejected, and that operation can take as long as 4 minutes. And, after a new tape is loaded into the drive, many seconds can be required for the drive to calibrate itself to the tape and to prepare for I/O. Although a slow tape jukebox can have a tape switch time of 1 or 2 minutes, this time is not enormously larger than the random-access time within one tape.

So, to generalize, we say that random access in a disk jukebox has a latency of tens of seconds, whereas random access in a tape jukebox has a latency of hundreds of seconds; switching tapes is expensive, but switching disks is not. Be careful not to over generalize: Some expensive tape jukeboxes can rewind, eject, load a new tape, and fast forward to a random item of data all in less than 30 seconds. If we pay attention to only the performance of the drives in a jukebox, the bandwidth and latency seem reasonable. But if we focus our attention on the cartridges instead, there is a terrible bottleneck. Consider first the bandwidth. By comparison with a fixed disk, the bandwidth-to-storage-capacity ratio of a robotic library is much less favorable. To read all the data stored on a large hard disk could take about an hour. To read all the data stored in a large tape library could take years. The situation with respect to access latency is nearly as bad. To illustrate this, if 100 requests are queued for a disk drive, the average waiting time will be about 1 second. If 100 requests are queued for a tape library, the average waiting time could be over 1 hour. The low cost of tertiary storage results from having many cheap cartridges share a few expensive drives. But a removable library is best devoted to the storage of infrequently used data, because the library can satisfy only a relatively small number of I/O requests per hour.

13.8.3.2 Reliability

Although we often think *good performance* means *high speed*, another important aspect of performance is *reliability*. If we try to read some data and are unable to do so because of a drive or media failure, for all practical purposes the access time is infinitely long and the bandwidth is infinitely small. So it is important to understand the reliability of removable media. Removable magnetic disks are somewhat less reliable than are fixed hard disks because the cartridge is more likely to be exposed to harmful environmental conditions such as dust, large changes in temperature and humidity, and mechanical forces such as shock and bending. Optical disks are considered very reliable, because the layer that stores the bits is protected by a transparent plastic or glass layer. The reliability of magnetic tape varies widely, depending on the kind of drive. Some inexpensive drives wear out tapes after a few dozen uses; other kinds are gentle enough to allow millions of reuses. By comparison with a magnetic disk, the head in a magnetic-tape drive is a weak spot. A disk head flies above the media, but a tape head is in close contact with the tape. The scrubbing action of the tape can wear out the head after a few thousands or tens of thousands of hours.

In summary, we say that a fixed disk drive is likely to be more reliable than a removable disk or tape drive, and an optical disk is likely to be more reliable than a magnetic disk or tape. But a fixed magnetic disk has one weakness. A head crash in a hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.

13.8.3.3 Cost

Storage cost is another important factor. Here is a concrete example of how removable media may lower the overall storage cost. Suppose that a hard disk that holds *X* GB has a price of \$200; of this amount, \$190 is for the housing, motor, and controller, and \$10 is for the magnetic platters. Then, the storage cost for this disk is \$200/*X* per gigabyte. Now, suppose that we can manufacture the platters in a removable cartridge. For one drive and 10 cartridges, the total price is \$190 +\$100 and the capacity is 10*X* GB, so the storage cost is \$29/*X* per gigabyte. Even if it is a little more expensive to make a removable cartridge, the cost per gigabyte of removable storage may well be lower than the cost per gigabyte of a hard disk, because the expense of one drive is averaged with the low price of many 3removable cartridges. Figures 1.13, 13.14, and 13.15 show the cost trends per megabyte for DRAM memory, magnetic hard disks, and tape drives. The prices in the graphs are the lowest price found in advertisements in *BYTE* magazine and *PC Magazine* at the end of each year. These prices reflect the small-computer marketplace of the readership of these magazines, where prices are low by comparison with the mainframe and minicomputer markets. In the case of tape, the price is for a drive with one tape. The overall cost of tape storage becomes much lower as more tapes are purchased for use with the drive, because the price of a tape is a small fraction of the price of the drive. However, in a huge tape library containing thousands of cartridges, the storage cost is dominated by the cost of the tape cartridges. As of this writing in 2003, the cost per GB of tape cartridges can be approximated as \$2.

Figure 13.13 Price per megabyte of DRAM, from 1981to 2000.

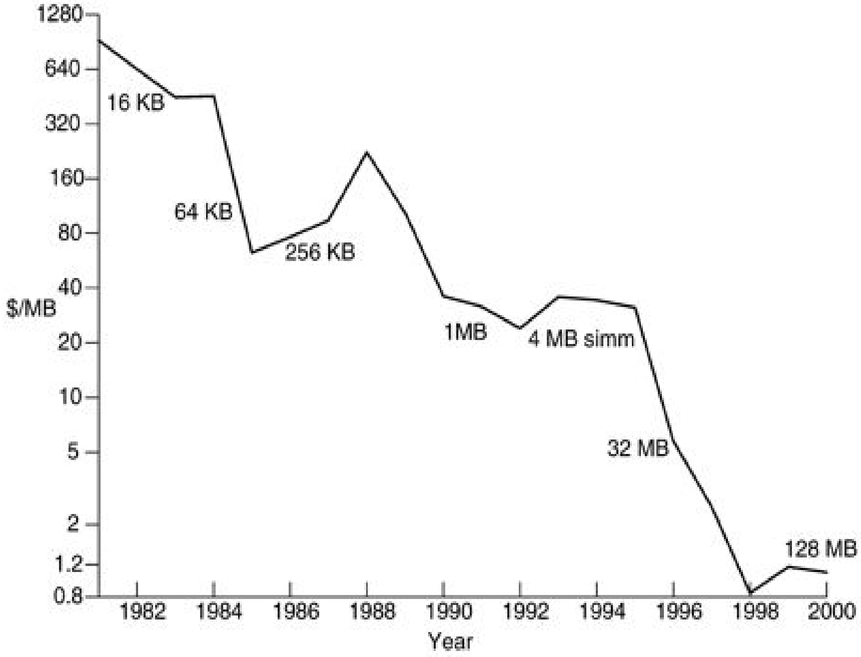
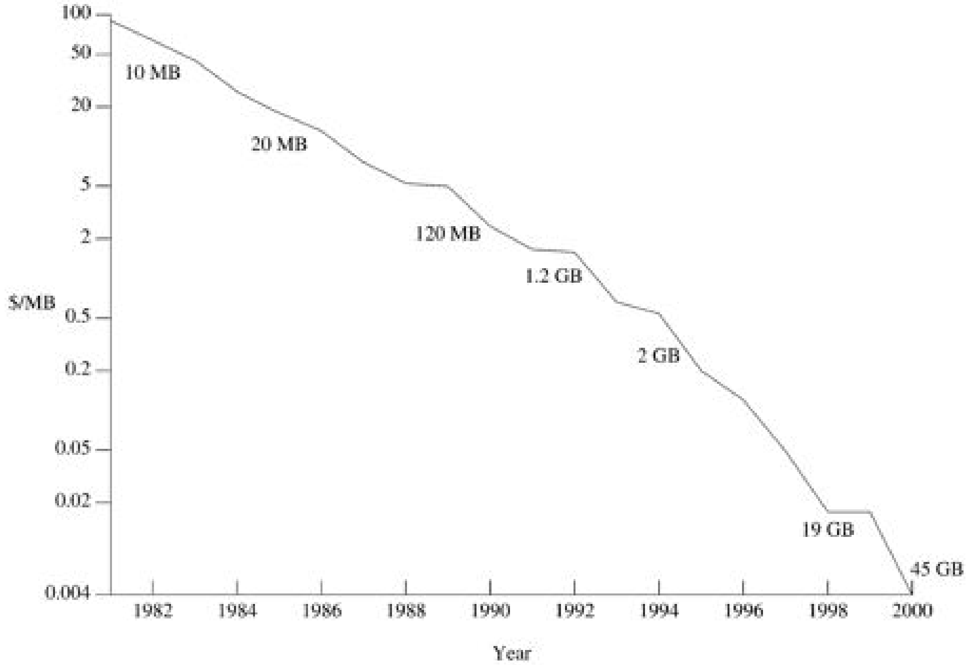


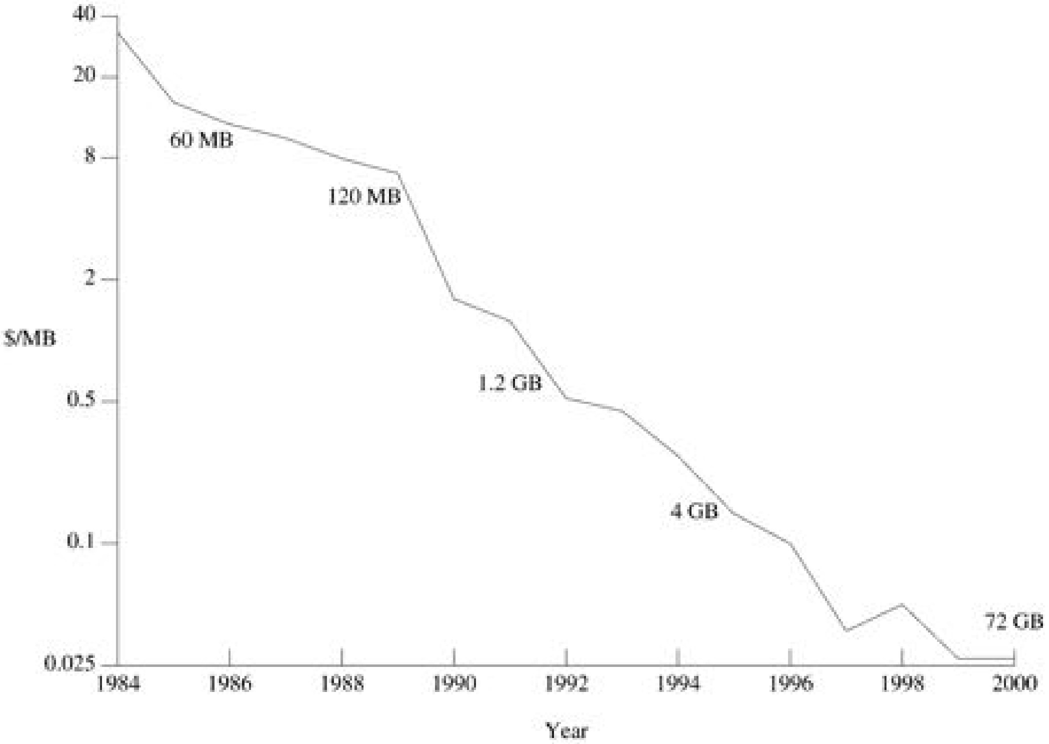
Figure 13.14 Price per megabyte of magnetic hard disk, from 1981 to 2000.



The cost of DRAM fluctuates widely. In the period from 1981 to 2000, we can see three price crashes (around 1981, 1989, and 1996), as excess roduction caused a glut in the marketplace. We can also see two periods (around 1987 and 1993) where shortages in the marketplace caused significant price increases. In the case of hard disks, the price declines have been much steadier, although the price decline appears to have accelerated since 1992. Tape-drive prices also fell steadily up to 1997. Since 1997 the price per gigabyte of inexpensive tape drives has ceased its dramatic fall, although mid-range tape technology (such as DAT/DDS) has continued to fall, and is now approaching that of the inexpensive drives. Tape-drive prices are not shown prior to 1984, because *BYTE* magazine is targeted to the small-computer marketplace, and tape drives were not widely used with small computers prior to 1984.

By comparing these graphs we see that the price of disk storage has plummeted relative to the price of DRAM and tape.

Figure 13.15 Price per megabyte of a tape drive, from 1984 to 2000.



The price per megabyte of magnetic disk has improved by more than four orders of magnitude during the past two decades, whereas the corresponding improvement for main memory has only been three orders of magnitude. Main memory today is more expensive than disk storage by a factor of 100. The price per megabyte has dropped much more rapidly for disk drives than for tape drives. In fact, the price per megabyte of magnetic disk drives is approaching that of a tape cartridge without the tape drive. Consequently, small- and medium size tape libraries have a higher storage cost than disk systems with equivalent capacity. The dramatic fall in disk prices has largely rendered tertiary storage obsolete: We no longer have any tertiary storage technology that is orders of magnitude less expensive than magnetic disk. It appears that the revival of tertiary storage must await a revolutionary technology breakthrough. Meanwhile, tape storage will find its use mostly limited to purposes such as backups of disk drives and archival storage in enormous tape libraries that greatly exceed the practical storage capacity of large disk farms.

13.9 SUMMARY

Disk drives are the major secondary-storage I/O device on most computers. Requests for disk I/O are generated by the file system and by the virtual-memory system. Each request specifies the address on the disk to be referenced, in the form of a logical block number. Disk-scheduling algorithms can improve the effective bandwidth, the average response time, and the variance in response time. Algorithms such as SSTF, SCAN, CSCAN, LOOK, and C-LOOK are designed to make such improvements by strategies for disk-queue ordering. Performance can be harmed by external fragmentation. Some systems have utilities that scan the file system to identify fragmented files; they then move blocks around to decrease the fragmentation. Defragmenting a badly fragmented file system can significantly improve the performance, but the system may have reduced performance while the defragmentation is in progress. Sophisticated file systems, such as the UNIX Fast File System, incorporate many strategies to control fragmentation during space allocation, so that disk reorganization is not needed.

The operating system manages the disk blocks. First, a disk must be low-level formatted to create the sectors on the raw hardware—new disks usually come preformatted. Then, the disk is partitioned and file systems created, and boot blocks are allocated to store the system's bootstrap program. Finally, when a block is corrupted, the system must have a way to lock out that block, or to replace it logically with a spare. Because an efficient swap space is a key to good performance, systems usually bypass the file system and use raw disk access for paging I/O. Some systems dedicate a raw disk partition to swap space, and others use a file within the file system instead.

Other systems allow the user or system administrator to make the decision by providing both options. The write-ahead log scheme requires the availability of stable storage. To implement such storage, we need to replicate the needed information on multiple nonvolatile storage devices (usually disks) with independent failure modes. We also need to update the information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery. Because of the amount of storage required on large systems, disks are frequently made redundant via RAID algorithms. These algorithms allow more than one disk to be used for a given operation, and allow continued operation and even automatic recovery in the face of a disk failure. RAID algorithms are organized into different levels where each level provides some combination of reliability and high transfer rates.

Disks may be attached to a computer system one of two ways: (1) using the local I/O ports on the host computer or (2) using a network connection such as storage area networks. Tertiary storage is built from disk and tape drives that use removable media. Many different technologies are available, including magnetic tape, removable magnetic

and magneto-optic disks, and optical disks. For removable disks, the operating system generally provides the full services of a file-system interface, including space management and request-queue scheduling. For many operating systems, the name of a file on a removable cartridge is a combination of a drive name and a file name within that drive. This convention is simpler but potentially more confusing than is using a name that identifies a specific cartridge. For tapes, the operating system generally just provides a raw interface. Many operating systems have no built-in support for jukeboxes. Jukebox support can be provided by a device driver or by a privileged application designed for backups or for HSM.

Three important aspects of performance are bandwidth, latency, and reliability. A wide variety of bandwidths is available for both disks and tapes, but the random access latency for a tape is generally much slower than that for a disk. Switching cartridges in a jukebox is also relatively slow. Because a jukebox has a low ratio of drives to cartridges, reading a large fraction of the data in a jukebox can take a long time. Optical media, which protect the sensitive layer by a transparent coating, are generally more robust than magnetic media, which expose the magnetic material to a greater possibility of physical damage.

EXERCISES

13.1 None of the disk-scheduling disciplines, except FCFS, are truly *fair*

(starvation may occur).

a. Explain why this assertion is true.

b. Describe a way to modify algorithms such as SCAN to ensure fairness.

c. Explain why fairness is an important goal in a time-sharing system.

d. Give three or more examples of circumstances in which it is important that the operating system be *un*fair in serving I/O requests.

13.2 Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130.

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

a. FCFS

b. SSTF

c. SCAN

d. LOOK

e. C-SCAN

f. C-LOOK

13.3 Elementary physics states that when an object is subjected to a constant acceleration a, the relationship between distance d and time t is given by $d = 1/2at2$. Suppose that, during a seek, the disk in Exercise13.2 accelerates the disk arm at a constant rate for the first half of the seek, then decelerates the disk arm at the same rate for the second half of the seek. Assume that the disk can perform a seek to an adjacent cylinder in 1 millisecond, and a full-stroke seek over all 5,000 cylinders in 18 milliseconds.

a. The distance of a seek is the number of cylinders that the head moves. Explain why the seek time is proportional to the square root of the seek distance.

b. Write an equation for the seek time as a function of the seek distance. This equation should be of the form $t = x + y\sqrt{L}$, where t is the time in milliseconds and L is the seek distance in cylinders.

c. Calculate the total seek time for each of the schedules in Exercise 13.2. Determine which schedule is the fastest (has the smallest total seek time).

e. The *percentage speedup* is the time saved divided by the original time. What is the percentage speedup of the fastest schedule over FCFS?

13.4 Suppose that the disk in Exercise 13.3 rotates at 7,200 RPM.

a. What is the average rotational latency of this disk drive?

b. What seek distance can be covered in the time that you found for part a?

13.5 The accelerating seek described in Exercise 13.3 is typical of hard-disk drives. By contrast, floppy disks (and many hard disks manufactured before the mid-1980s) typically seek at a fixed rate. Suppose that the disk in Exercise 13.3 has a constant-rate seek, rather than a constant-acceleration seek, so the seek time is of the form $t = x + yL$, where t is the time in milliseconds and L is the seek distance. Suppose that the time to seek to an adjacent cylinder is 1 millisecond, as before, and is 0.5 milliseconds for each additional cylinder.

a. Write an equation for this seek time as a function of the seek distance.

b. Using the seek-time function from part a, calculate the total seek time for each of the schedules in Exercise 13.2. Is your answer the same as it was for Exercise 13.3c? Explain why it is the same or why it is different.

c. What is the percentage speedup of the fastest schedule over FCFS in this case?

13.6 Write a Java program for disk scheduling using the SCAN and C-SCAN disk scheduling algorithms.

13.7 Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request's service), the variation in response time, and the effective bandwidth. How does performance depend on the relative sizes of seek time and rotational latency?

13.8 Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.

13.9 Explain why SSTF scheduling tends to favor middle cylinders over the innermost and outermost cylinders.

13.10 Requests are not usually uniformly distributed. For example, a cylinder containing the file system FAT or inodes can be expected to be accessed more frequently than a cylinder that contains only files. Suppose that you know that 50 percent of the requests are for a small, fixed number of cylinders.

a. Would any of the scheduling algorithms discussed in this chapter be particularly good for this case? Explain your answer.

b. Propose a disk-scheduling algorithm that gives even better performance by taking advantage of this "hot spot" on the disk.

c. File systems typically find data blocks via an indirection table, such as a FAT in DOS or inodes in UNIX. Describe one or more ways to take advantage of this indirection to improve the disk performance.

13.11 Why is rotational latency usually not considered in disk scheduling? How would you modify SSTF, SCAN, and C-SCAN to include latency optimization?

13.12 How would the use of a RAM disk affect your selection of a disk scheduling algorithm? What factors would you need to consider? Do the same considerations apply to hard-disk scheduling, given that the file system stores recently used blocks in a buffer cache in main memory?

13.13 Why is it important to balance file system-I/O among the disks and controllers on a system in a multitasking environment?

13.14 What are the tradeoffs involved in rereading code pages from the file system, versus using swap space to store them?

13.15 Is there any way to implement truly stable storage? Explain your answer.

13.16 The reliability of a hard-disk drive is typically described in terms of a quantity called *mean time between failures* (*MTBF*). Although this quantity is called a "time," the MTBF actually is measured in drive-hours per failure.

a. If a system contains 1,000 disk drives, each of which has a 750,000 hour MTBF, which of the following best describes how often a drive failure will occur in that disk farm: once per thousand years, once per century, once per decade, once per year, once per month, once per week, once per day, once per hour, once per minute, or once per second?

b. Mortality statistics indicate that, on the average, a U.S. resident has about 1:1,000 chance of dying between ages 20 and 21 years. Deduce the MTBF hours for 20 year olds. Convert this figure from hours to years. What does this MTBF tell you about the expected lifetime of a 20 year old?

c. The manufacturer guarantees a 1-million hour MTBF for a certain model of disk drive. What can you conclude about the number of years for which one of these drives is under warranty?

13.17 The term *fast wide SCSI-II* denotes a SCSI bus that operates at a data rate of 20 MB per second when it moves a packet of bytes between the host and a device. Suppose that a fast wide SCSI-II disk drive spins at 7,200 RPM , has a sector size of 512 bytes, and holds 160 sectors per track.

a. Estimate the sustained transfer rate of this drive in megabytes per second.

b. Suppose that the drive has 7,000 cylinders, 20 tracks per cylinder, a heads witch time (from one platter to another) of 0.5 milliseconds, and an adjacent cylinder seek time of 2 milliseconds. Use this additional information to give an accurate estimate of the sustained transfer rate for a huge transfer.

c. Suppose that the average seek time for the drive is 8 milliseconds. Estimate the I/Os per second and the effective transfer rate for a random access workload that reads individual sectors scattered across the disk.

d. Calculate the random-access I/Os per second and transfer rate for I/O sizes of 4 KB, 8 KB, and 64 KB.

e. If multiple requests are in the queue, a scheduling algorithm such as SCAN should be able to reduce the average seek distance. Suppose that a random access workload is reading 8 KB pages, the average queue length is 10, and the scheduling algorithm reduces the average seek time to 3 milliseconds. Calculate the I/Os per second and the effective transfer rate of the drive.

13.18 More than one disk drive can be attached to a SCSI bus. In particular, a fast wide SCSI-II bus (Exercise 13.17) can be connected to at most 15 disk drives. Recall that this bus has a bandwidth of 20 MB per second. At any time, only one packet can be transferred on the bus between some disk's internal cache and the host. However, a disk can be moving its disk arm while some other disk is transferring a packet on the bus. Also, a disk can be transferring data between its magnetic platters and its internal cache while some other disk is transferring a packet on the bus. Considering the transfer rates that you calculated for the various workloads in Exercise 13.17, discuss how many disks can be used effectively by one fast wide SCSI-II bus.

13.19 Remapping of bad blocks by sector sparing or sector slipping could influence performance. Suppose that the drive in Exercise 14.17 has a total of 100 bad sectors at random locations, and that each bad sector is mapped to a spare that is located on a different track, but within the same cylinder. Estimate the number of I/Os per second and the effective transfer rate for a random-access workload consisting of 8 KB reads, with a queue length of 1 (that is, the choice of scheduling algorithm is not a factor). What is the effect of a bad sector on performance?

13.20 Discuss the relative advantages and disadvantages of sector sparing and sector slipping.

13.21 The operating system generally treats removable disks as shared file systems, but assigns a tape drive to only one application at a time. Give three reasons that could explain this difference in treatment of disks and tapes. Describe the additional features that an operating system would need to support shared file-system access to a tape jukebox. Would the applications sharing the tape jukebox need any special properties, or could they use the files as though the files were disk-resident? Explain your answer.

13.22 In a disk jukebox, what would be the effect if the number of open files was greater than the number of drives in the jukebox?

13.23 What would be the effects on cost and performance if tape storage had the same areal density as disk storage?

13.24 If magnetic hard disks eventually have the same cost per gigabyte as do tapes, will tapes become obsolete, or will they still be needed? Explain your answer.

13.25 You can use simple estimates to compare the cost and performance of a terabyte storage system made entirely from disks with one that incorporates tertiary storage. Suppose that magnetic disks each hold 10 GB, cost $1,000, 5 MB per second, and have an average access latency of 15 milliseconds. Suppose that a tape library costs $10 per gigabyte, transfers 10 MB per second, and has an average access latency of 20 seconds. Compute the total cost, the maximum total data rate, and the average waiting time for a pure disk system. If you make any assumptions about the workload, describe and justify them. Now, suppose that 5 percent of the data are frequently used, so they must reside on disk, but the other 95 percent are archived in the tape library. Further suppose that the disk system handles 95 percent of the requests, and the library handles the other 5 percent. What are the total cost, the maximum total data rate, and the average waiting time for this hierarchical storage system?

13.26 It is sometimes said that tape is a sequential-access medium, whereas magnetic disk is a random-access medium. In fact, the suitability of a storage device for random access depends on the transfer size. The term *streaming transfer rate* denotes the data rate for a transfer underway, excluding the effect of access latency. By contrast, the *effective transfer rate* is the ratio of total bytes per total seconds, including overhead time such as the access latency. Suppose that, in a computer, the level-2 cache has an access latency of 8 nanoseconds and a streaming transfer rate of 800 MB per second, the main memory has an access latency of 60 nanoseconds and a streaming transfer rate of 80 MB per second, the magnetic disk has an access latency of 15 millisecond and a streaming transfer rate of 5 MB per second, and a tape drive has an access latency of 60 seconds and a streaming transfer rate of 2 MB per second.

a. Random access causes the effective transfer rate of a device to decrease, because no data are transferred during the access time. For the disk described, what is the effective transfer rate if a streaming transfer of 512 bytes, 8 KB, 1MB, and 16 MB follows an average access?

b. The utilization of a device is the ratio of effective transfer rate to streaming transfer rate. Calculate the utilization of the disk drive for random access that performs transfers in each of the four sizes given in part a.

c. Suppose that a utilization of 25 percent (or higher) is considered acceptable. Using the performance figures given, compute the smallest transfer size for disk that gives acceptable utilization.

d. Complete the following sentence: A disk is a random-access device for transfers larger than _____ bytes, and is a sequential-access device for smaller transfers.

e. Compute the minimum transfer sizes that give acceptable utilization for cache, memory, and tape.

f. When is a tape a random-access device, and when is it a sequential-access device?

13.27 Imagine that a holographic storage drive has been invented. Suppose that the holographic drive costs $10,000 and has an average access time of 40 milliseconds. Suppose that it uses a $100 cartridge the size of a CD. This cartridge holds 40,000 images, and each image is a square black-and-white picture with resolution 6,000 ×

6,000 pixels (each pixel stores 1 bit). Suppose that the drive can read or write one picture in 1 millisecond. Answer the following questions.

a. What would be some good uses for this device?

b. How would this device affect the I/O performance of a computing system?

c. Which other kinds of storage devices, if any, would become obsolete as a result of this device being invented?

13.28 Suppose that a one-sided 5.25-inch optical-disk cartridge has an areal density of 1 gigabit per square inch. Suppose that a magnetic tape has an areal density of 20 megabits per square inch, and is 1/2 inch wide and 1,800 feet long. Calculate an estimate of the storage capacities of these two kinds of storage cartridges. Suppose that an optical tape exists that has the same physical size as the tape, but the same storage density as the optical disk. What volume of data could the optical tape hold? What would be a marketable price for the optical tape if the magnetic tape cost $25?

13.29 Suppose that we agree that 1 KB is 1,024 bytes, 1 MB is 1,0242 bytes, and 1 GB is 1,0243 bytes. This progression continues through terabytes, petabytes, and exabytes (1,0246). Several newly proposed scientific projects plan to be able to record and store a few exabytes of data during the next decade. To answer the following questions, you will need to make a few reasonable assumptions; state the assumptions that you make.

a. How many disk drives would be required to hold 4 exabytes of data?

b. How many magnetic tapes would be required to hold 4 exabytes of data?

c. How many optical tapes would be required to hold 4 exabytes of data (Exercise 13.28)?

d. How many holographic storage cartridges would be required to hold 4 exabytes of data (Exercise 13.27)?

e. How many cubic feet of storage space would each option require?

13.30 Discuss how an operating system could maintain a free-space list for a tape-resident file system. Assume that the tape technology is appendonly, and that it uses the EOT mark and locate, space, and read position commands as described in Section 13.8.2.1.

UNIT - V

14. SECURITY

14.1 The Security Problem

In the previous Chapter, we discussed mechanisms that the operating system can provide (with appropriate aid from the hardware) that allow users to protect their resources (usually 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. Among the forms of malicious access are the following:

- Unauthorized reading of data (or theft of information)
- Unauthorized modification of data
- Unauthorized destruction of data
- Preventing legitimate use of the system (or denial of service)

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, if not all, unauthorized attempts to access the information residing in the system.

To protect the 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.

2. **Human:** Users must be screened carefully to reduce the chance of authorizing a user who then gives access to an intruder (in exchange for a bribe, for example).

3. **Network:** Much computer data in modern systems travels over private leased lines, shared lines like the Internet, 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.

4. **Operating system:** The system must protect itself from accidental or purposeful security breaches. 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.

Furthermore, the system hardware must provide protection (Chapter 18) to allow the implementation of security features. Most contemporary operating systems are now designed to provide security features. 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 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 far beyond the scope of this text. Security within the operating system and between operating systems is implemented in several ways, ranging from passwords for access to the system to the isolation of concurrent processes running within the system. The file system also provides some degree of protection.

14.2 User Authentication

A major security problem for operating systems is **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. A user normally identifies himself. How do we determine whether a user's identity is authentic? Generally, authentication is based on one or more of three items: user possession (a key or card), user knowledge (a user identifier and password), and/or a user attribute (fingerprint, retina pattern, or signature).

14.2.1 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 user is legitimate. 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 could 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.

14.2.2 Password Vulnerabilities

Passwords are extremely common because they are easy to understand and use. Unfortunately, passwords can often be guessed, accidentally exposed, sniffed, 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 letters, numbers, and punctuation, until the password is found. Short passwords are especially vulnerable to this method. For example, a four-decimal 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 1 millisecond would take only about 5 seconds to guess a four-digit password. Enumeration is not as successful at finding passwords in systems that allow longer passwords, that differentiate between uppercase and lowercase letters, and that allow use of numbers and all punctuation characters in passwords. 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 could seamlessly add a network monitor, allowing her to watch all data being transferred on the network (*sniffing*), including user IDs and passwords. Encrypting the data stream containing the password solves this problem. Exposure is a particularly severe problem if the password is written down where it can be read or lost. As we shall see, some systems force users to select hard to remember or long passwords, which may cause a user to record the password. As a result, such systems provide much less security than systems that allow easy passwords!

The final method 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 that user 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 her account. 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. User-selected passwords, however, are often easy to guess (the user's name or favorite car, for example). At some sites, administrators occasionally check user passwords and notify a user if her password is too short or otherwise easy to guess. 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 frequently. In 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 misused, it can be used 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.

14.2.3 Encrypted 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 **encryption** to avoid the necessity of keeping its password list secret. 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 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 encoded 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 function *f*(*x*) is typically an **encryption algorithm** that has been designed and tested rigorously, as discussed in Section 14.7.2.

The flaw in this method is that the system no longer has control over the passwords. Although the passwords are encrypted, anyone with a copy of the password file can run fast encryption routines against it—encrypting 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 encryption algorithm, a hacker might keep a cache of passwords that have been cracked previously. For this reason, new versions of UNIX store the encrypted password entries in a file readable only by the **superuser**. The programs that compare a presented password to the stored password 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. To avoid the dictionary encryption method, some systems disallow 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 thrown in for good measure. For example, the phrase "My mother's name is Katherine" might yield the password "MmnisK.!". The password is hard to crack but easy for the user to remember.

14.2.4 One-Time Passwords

To avoid the problems of password sniffing and shoulder surfing, a system could 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. The algorithm might be an integer function, for example. The system selects a random integer and presents it to the user. The user applies the function and replies with the correct result. The system also applies the function. If the two results match, access is allowed.

Such algorithmic passwords are not susceptible to exposure; that is, a user can type in a password, and no entity intercepting that password will be able to reuse it. In this variation, the system and the user share a secret. The secret is never transmitted over a medium that allows exposure. Rather, the secret is used as input to the function, along with a shared seed. A seed is a random number or alphanumeric sequence. The seed is the authentication challenge from the computer. The secret and the seed are used as input to the function *f*(*secret, seed*). The result of this function is transmitted as the password to the computer. Because the computer also knows the secret and the seed, it can perform the same computation. If the results match, the user is authenticated. The next time that the user needs to be authenticated, another seed is generated, and the same steps ensue. This time, the password is different.

In this **one-time password** system, the password is different in each instance. Anyone capturing the password from one session and trying to reuse it in another session will fail. One-time passwords are among the only ways to prevent improper authentication due to password exposure. Commercial implementations of one-time password systems such as securID, for example, use hardware calculators. Most of these calculators are in the shape of a credit card but have a keypad and display. Some use the current time as the random seed. The user uses the keypad to enter the shared secret, also known as a personal identification number (PIN). The display shows the one-time password. The use of both a one-time password generator and a PIN is one form of two-factor authentication. Two different types of components are needed in this case. Two-factor authentication offers far better authentication protection than single-factor authentication.

Another variation on one-time passwords is the use of a code book, or one-time pad, which is a list of single-use passwords. In this method, each password on the list is used, in order, 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.

14.2.5 Biometrics

There are many other variations on the use of passwords for authentication. Palmor 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 your finger's 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 the finger on the pad is the same as the stored one. 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.

14.3 Program Threats

When a program written by one user may be used by another, misuse and unexpected behavior may result. In this section, we describe common methods by which users gain access to the programs of others: Trojan horses, trap doors, and stack and buffer overflow.

14.3.1 Trojan Horse

Many systems have mechanisms for allowing programs written by users to be executed by other users. If these programs are executed in a domain that provides the access rights of the executing user, the other users may misuse these rights. A text-editor program, for example, may include code to search the file to be edited for certain keywords. If any are found, the entire file may be copied to a special area accessible to the creator of the text editor. A code segment that misuses its environment is called a Trojan horse. Long search paths, such as are common on UNIX systems, exacerbate the Trojan-horse problem. The search path lists the set of directories to search when an ambiguous program name is given. The path is searched for a file of that name, and the file is executed. All the directories in the search path must be secure, or a Trojan horse could be slipped into the user's path and executed accidentally.

For instance, consider the use of the "." character in a search path. The "." tells the shell to include the current directory in the search. Thus, if a user has "." in her search path, has set her current directory to a friend's directory, and enters the name of a normal system command, the command may be executed from the friend's directory instead. The program would run within the user's domain, allowing the program to do anything that the user is allowed to do, including deleting the user's files, for instance. A variation of the Trojan horse is a program that emulates a login program. An unsuspecting user starts to log in at a terminal and notices that he has apparently mistyped his password. He tries again and is successful. What has happened is that his authentication key and password have been stolen by the login emulator, which was left running on the terminal by the thief. The emulator stored away the password, printed out a login error message, and exited; the user was then provided with a genuine login prompt. This type of attack can be defeated by having the operating system print a usage message at the end of an interactive session or by a non trappable key sequence, such as the control-alt-delete combination used by all Windows operating systems.

14.3.2 Trap Door

The designer of a program or system might leave a hole in the software that only she is capable of using. This type of security breach (or **trap door**) was shown in the movie *War Games*. For instance, the code might check for a specific user ID or password, and it might circumvent normal security procedures. Programmers have been arrested for embezzling from banks by including rounding errors in their code and having the occasional half-cent credited to their accounts. This account crediting can add up to a large amount of money, considering the number of transactions that a large bank executes.

A clever trap door could be included in a compiler. The compiler could generate standard object code as well as a trap door, regardless of the source code being compiled. This activity is particularly nefarious, since a search of the source code of the program will not reveal any problems. Only the source code of the compiler would contain the information. Trap doors pose a difficult problem because, to detect them, we have to analyze all the source code for all components of a system. Given that software systems may consist of millions of lines of code, this analysis is not done frequently, and frequently it is not done at all!

14.3.3 Stack and Buffer Overflow

The stack- or buffer-overflow attack is the most common way for an attacker outside of the system, on a network or dial-up connection, to gain unauthorized access to the target system. An authorized user of the system may also use this exploit for **privilege escalation**, to gain privileges beyond those allowed for that user. Essentially, the attack exploits a bug in a program. The bug can be a simple case of poor programming, in which the programmer neglected to code bounds checking on an input field. In this case, the attacker sends more data than the program expecting. Using trial and error, or by examination of the source code of the attacked program if it is available, the attacker determines the vulnerability and writes a program to do the following:

1. Overflow an input field, command-line argument, or input buffer—for example, on a network daemon—until it writes into the stack
2. Overwrite the current return address on the stack with the address of the exploit code loaded in step 3
3. Write a simple set of code for the next space in the stack that includes the commands that the attacker wishes to execute—for instance, spawn a shell The result of this attack program's execution will be a root shell or other privileged command execution.

For instance, if a web page form expects a user name to be entered into a field, the attacker could send the user name, plus extra characters to overflow the buffer and reach the stack, plus a new return address to load onto the stack, plus the code the attacker wants to run. When the buffer-reading subroutine returns from execution, the return address is the exploit code, and the code is run. The buffer-overflow attack is especially pernicious, as it can be run within a system and can travel over allowed communications channels. Such attacks can occur within protocols that are expected to be used to communicate with the machine, and they can therefore be hard to detect and prevent. They can even bypass the security added by firewalls (Section 14.5).

One solution to this problem is for the CPU to have a feature that disallows execution of code in a stack section of memory. Recent versions of Sun's SPARC chip include this setting, and recent versions of Solaris enable it. The return address of the overflowed routine can still be modified; but when the return address is within the stack and the code there attempts to execute, an exception is generated, and the program is halted with an error.

14.4 System Threats

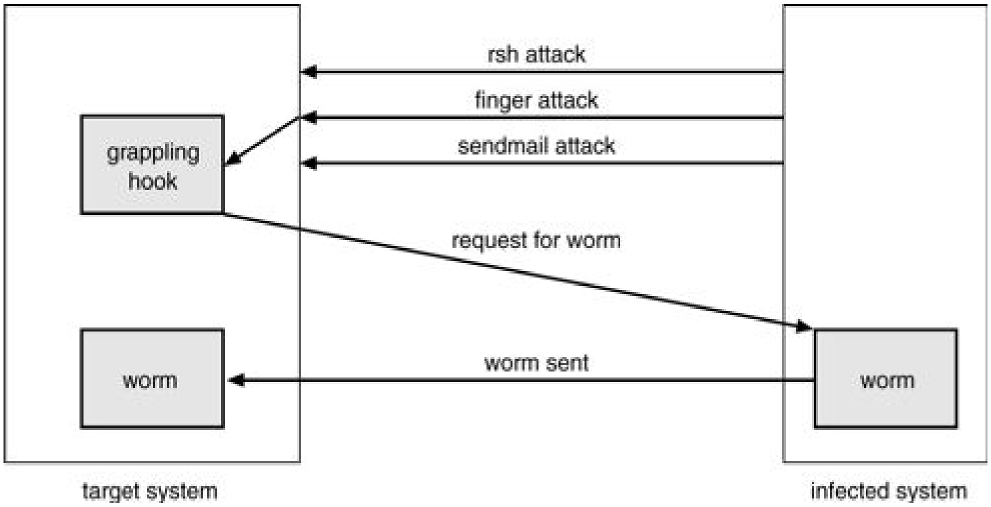
Most operating systems provide a means by which processes can spawn other processes. In such an environment, it is possible to create a situation where operating system resources and user files are misused. The two most common methods for was achieving this misuse are worms and viruses. We discuss each below, along with a somewhat different form of system threat: denial of service.

14.4.1 Worms

A **worm** is a process that uses the **spawn** mechanism to ravage system performance. The worm spawns copies of itself, using up system resources and perhaps locking out all other processes. On computer networks, worms are particularly potent, since they may reproduce themselves among systems and thus shut down the entire network. Such an event occurred in 1988 to UNIX systems on the Internet, causing millions of dollars of lost system and system administrator time. At the close of the workday on November 2, 1988, Robert Tappan Morris, Jr., a first-year Cornell graduate student, unleashed a worm program on one or more hosts connected to the Internet. Targeting Sun Microsystems' Sun 3 workstations and VAX computers running variants of Version 4 BSD UNIX, the worm quickly spread over great distances; within a few hours of its release, it had consumed system resources to the point of bringing down the infected machines.

Although Robert Morris designed the self-replicating program for rapid reproduction and distribution, some of the features of the UNIX networking environment provided the means to propagate the worm throughout the system. It is likely that Morris chose for initial infection an Internet host left open for and accessible to outside users. From there, the worm program exploited flaws in the UNIX operating system's security routines and took advantage of UNIX utilities that simplify resource sharing in local-area networks to gain unauthorized access to thousands of other connected sites. Morris's methods of attack are outlined next. The worm was made up of two programs, a **grappling hook** (also called a **bootstrap** or **vector**) program and the main program. Named *l1.c*, the grappling hook consisted of 99 lines of C code compiled and run on each machine it accessed. Once established on the computer system under attack, the grappling hook connected to the machine where it originated and uploaded a copy of the main worm onto the *hooked* system (Figure 14.1). The main program proceeded to search for other machines to which the newly infected system could connect easily. In these actions, Morris exploited the UNIX networking utility, rsh, for easy remote task execution. By setting up special files that list the host-login name pairs, users can omit entering a password each time they access a remote account on the paired list. The worm searched these special files for site names that would allow remote execution without a password. Where remote shells were established, the worm program was uploaded and began executing anew.

Figure 14.1 The Morris Internet worm.



The attack via remote access was one of three infection methods built into the worm. The other two methods involved operating-system bugs in the UNIX *finger* and *sendmail* programs. The *finger* utility functions as an electronic telephone directory; the command

finger user-name@hostname

returns a person's real and login names along with other information that the user may have provided, such as office and home address and telephone number, research plan, or clever quotation. *Finger* runs as a background process (or daemon) at each BSD site and responds to queries throughout the Internet. The point vulnerable to malicious entry involved reading input without checking bounds for overflow. The code executed a buffer-overflow attack. Morris's program queried *finger* with a 536-byte string crafted to exceed the buffer allocated for input and to overwrite the stack frame. Instead of returning to the *main* routine it was in before Morris's call, the *finger* daemon was routed to a procedure within the invading 536-byte string now residing on the stack. The new procedure executed */bin/sh*, which, if successful, gave the worm a remote shell on the machine under attack.

The bug exploited in *sendmail* also involved using a daemon process for malicious entry. *Sendmail* routes electronic mail in a network environment. Debugging code in the utility permits testers to verify and display the state of the mail system. The debugging option is useful to system administrators and is often left on as a background process. Morris included in his attack arsenal a call to debug that, instead of specifying a user address, as would be normal in testing, issued a set of commands that mailed and executed a copy of the grappling-hook program.

Once in place, the main worm undertook systematic attempts to discover user passwords. It began by trying simple cases of no password or of passwords constructed of account-user-name combinations, then used comparisons with an internal dictionary of 432 favorite password choices, and then went to the final stage of trying each word in the standard UNIX on-line dictionary as a possible password. This elaborate and efficient three-stage password-cracking algorithm enabled the worm to gain access to other user accounts on the infected system. The worm then searched for *rsh* data files in these newly broken accounts and used them as described previously to gain access to user accounts on remote systems.

With each new access, the worm program searched for already active copies of itself. If it found one, the new copy exited, except in every seventh instance. Had the worm exited on all duplicate sightings, it might have remained undetected. Allowing every seventh duplicate to proceed (possibly to confound efforts to stop its spread by baiting with *fake* worms) created a wholesale infestation of Sun and VAX systems on the Internet. The very features of the UNIX network environment that assisted the worm's propagation also helped to stop its advance. Ease of electronic communication, mechanisms to copy source and binary files to remote machines, and access to both source code and human expertise allowed cooperative efforts to develop solutions quickly. By the evening of the next day, November 3, methods of halting the invading program were circulated to system administrators via the Internet. Within days, specific software patches for the exploited security flaws were available.

Why did Morris unleash the worm? The action has been characterized as both a harmless prank gone awry and a serious criminal offense. Based on the complexity of starting the attack, it is unlikely that the worm's release or the scope of its spread was unintentional. The worm program took elaborate steps to cover its tracks and to repel efforts to stop its spread. Yet the program contained no code aimed at damaging or destroying the systems on which it ran. The author clearly had the expertise to include such commands; in fact, data structures were present in the bootstrap code that could have been used to transfer Trojan-horse or virus programs (Section 14.4.2). The behavior of the program may lead to interesting observations, but it does not provide a sound basis for inferring motive. What is not open to speculation, however, is the legal outcome: A federal court convicted Morris and handed down a sentence of three years' probation, 400 hours of community service, and a $10,000 fine. Morris' legal costs were probably in excess of $100,000.

A more recent event shows that worms are still a fact of life on the Internet. It also shows that as the Internet grows, the damage that even "harmless" worms can do also grows, and can be significant. This example occurred during August, 2003. The fifth version of the "Sobig" worm, more properly known as "W32.Sobig.F@mm", was released by persons at this time unknown. It is the fastest-spreading worm released to date, at its peak infecting hundreds of thousands of computers and one in seventeen e-mail messages on the Internet. It clogged e-mail inboxes, slowed networks, and took a huge number of man-hours to clean up. Sobig.F was launched by being uploaded to a pornography newsgroup, via an account created using a stolen credit card. It was disguised as a photo.

The virus targeted Microsoft Windows systems, and used its own SMTP engine to e-mail itself to all of the addresses found on an infected system. It used a variety of subject lines to help avoid detection, including "Thank You!, Your details," and "Re: Approved". It also used a random address on the host as the "From:" address, making it difficult to determine via the message which machine was the infected source. Sobig.F included an attachment for the target e-mail reader to click on, again with a variety of names. If the payload was executed, it stored a program called

WINPPR32.EXE into the default Windows directory, along with a text file. It also modified the Windows registry.

It was also programmed to periodically attempt to connect to one of twenty servers, and download and execute a program from them. Fortunately, the servers were disabled before the code could be downloaded. The contents of the program from these servers has not yet been determined. Should the code have been malevelent, untold damage to a vast number of machines could have resulted. Security experts are still evaluating methods to decrease or eliminate worms.

14.4.2 Viruses

Another form of computer attack is a **virus**. Like worms, viruses are designed to spread into other programs and can wreak havoc in a system by modifying or destroying files and causing system crashes and program malfunctions. Whereas a worm is structured as a complete, standalone program, a virus is a fragment of code embedded in a legitimate program. Viruses are a major problem for computer users, especially users of microcomputer systems. Multiuser computers generally are not susceptible to viruses because the executable programs are protected from writing by the operating system. Even if a virus does infect a program, its powers are limited because other aspects of the system are protected. Single-user systems have no such protections and, as a result, a virus has free run.

Viruses are usually spread when users download viral programs from public bulletin boards or exchange disks containing an infection. In February 1992, for example, two Cornell University students developed three Macintosh game programs with an embedded virus and distributed them to worldwide software archives via the Internet. The virus was discovered when amathematics professor in Wales downloaded the games and antivirus programs on his system alerted him to an infection. Some 200 other users had already downloaded the games. Although the virus was not designed to destroy data, it could spread to application files and cause such problems as long delays and program malfunctions. The authors were easy to trace, since the games had been mailed electronically from a Cornell account. New York State authorities arrested the students on misdemeanor charges of computer tampering.

In recent years, a common form of virus transmission has been via the exchange of Microsoft Office files, such as Microsoft Word documents. These documents can contain so-called *macros* (or Visual Basic programs) that programs in the Office suite (Word, PowerPoint, or Excel) will execute automatically. Because these programs run under the user's own account, the macros can run largely unconstrained (for example, deleting user files at will). On occasion, upcoming viral infections are announced in high-profile media events. Such was the case with the *Michelangelo* virus, which was scheduled to erase infected hard disk files on March 6, 1992, the Renaissance artist's 517th birthday. Because of the extensive publicity surrounding the virus, most sites had located and destroyed the virus before it was activated, so it caused little or no damage.

Such cases both alert the general public to and alarm it about the virus problem. Antivirus programs are currently excellent sellers. Most commercial antivirus packages are effective against only particular known viruses. They work by searching all the programs on a system for the specific pattern of instructions known to make up the virus. When they find a known pattern, they remove the instructions, *disinfecting* the program. These commercial packages have catalogs of thousands of viruses for which they search. Viruses and the antivirus software continue to become more sophisticated. Some viruses modify themselves as they infect other software to avoid the basic pattern-match approach of antivirus software. The antivirus software in turn now looks for families of patterns rather than a single pattern to identify a virus.

The best protection against computer viruses is prevention, or the practice of **safe computing**. Purchasing unopened software from vendors and avoiding free or pirated copies from public sources or disk exchange are the safest route to preventing infection. However, even new copies of legitimate software applications are not immune to virus infection: There have been cases where disgruntled employees of a software company have infected the master copies of software programs to do economic harm to the company selling the software. For macro viruses, one defense is to exchange Word documents in an alternative file format called **rich text format (RTF)**. Unlike the native Word format, RTF does not include the capability to attach macros.

Another defense is to avoid opening any e-mail attachments from unknown users. Unfortunately, history has shown that e-mail vulnerabilities appear as fast as they are fixed. For example, in 2000 the *love bug* virus became very widespread by appearing to be a love note sent by a friend of the receiver. Once the attached Visual Basic script was opened, the virus propagated by sending itself to the first users in the user's e-mail contact list. Fortunately, except for clogging e-mail systems and users' inboxes, it was relatively harmless. It did, however, effectively negate the defensive strategy of opening attachments only from people known to the receiver. A more effective defense method is to avoid opening any e-mail attachment that contains executable code. Some companies now enforce this as policy by removing all incoming attachments to e-mail messages.

Another safeguard, although it does not prevent infection, does permit early detection. A user must begin by completely reformatting the hard disk, especially the boot sector, which is often targeted for viral attack. Only secure software is uploaded, and a checksum for each file is calculated. The checksum list must then be kept free from unauthorized access. Periodically, a program can re-compute the checksums and compare them to the original list; any differences serve as a warning of possible infection. Because they usually work among systems and not programs, processes, or users, both worms and viruses generally pose security, rather than protection, problems.

14.4.3 Denial of Service

The last attack category, **denial of service**, is aimed not at gaining information or stealing resources but rather at disrupting legitimate use of a system or facility. An intruder could delete all the files on a system, for example. Most denial-of-service attacks involve systems that the attacker has not penetrated. Indeed, 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. The first case is an attack that uses so many facility resources that, in essence, no useful work can be done. For example, a web-site click could download a Java applet that proceeds to use all available CPU time. The second case involves disrupting the network of the facility. There have been several successful denial-of-service attacks against major web sites. They 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 several partially started TCP sessions. These sessions 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.

These 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. Frequently, it is difficult to determine if a system slowdown is just a surge in use or an attack!

14.5 Securing Systems and Facilities

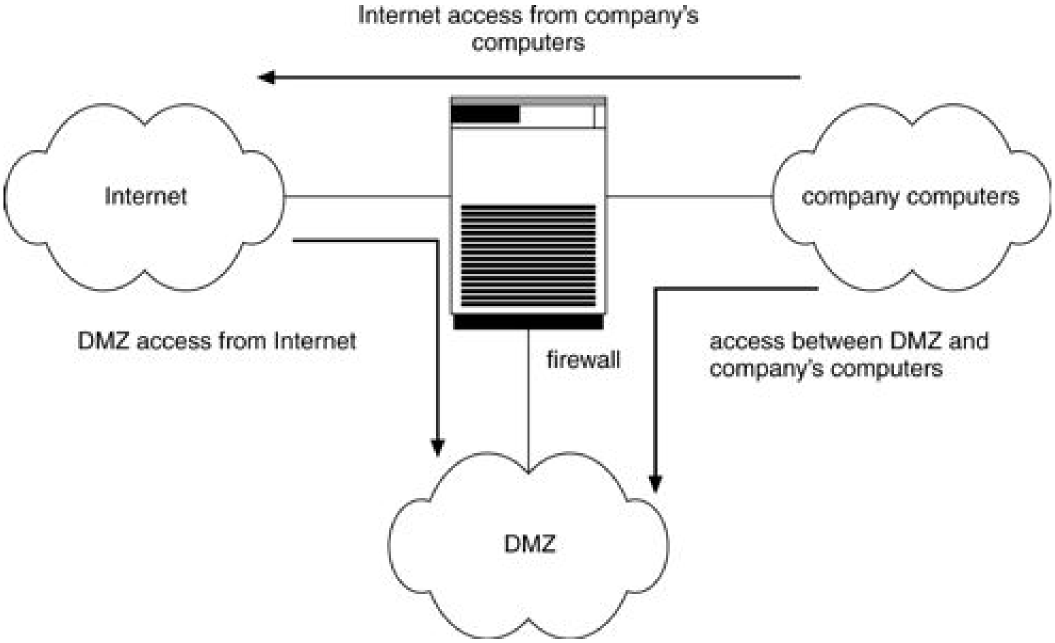
Securing systems and facilities is intimately linked to intrusion detection (Section 14.6). Both techniques need to work together to assure that a system is secure and that, if a security breach happens, it is detected. One method of improving system security is periodically to scan the system for security holes. These scans can be done at times when computer use is relatively low, so they will have less effect than logging. A scan can check a variety of aspects of the system:

- Short or easy-to-guess passwords
- Unauthorized privileged programs, such as *setuid* programs
- Unauthorized programs in system directories
- Unexpected long-running processes
- Improper directory protections on both user and system directories
- Improper protections on system data files, such as the password file, device drivers, or even the operating-system kernel itself
- Dangerous entries in the program search path (for example, the Trojan horse discussed in Section 14.3.1)
- Changes to system programs detected with checksum values
- Unexpected or hidden network daemons

Any problems found by a security scan can either be fixed automatically or reported to the managers of the system.

Networked computers are much more susceptible to security attacks than are standalone systems. Rather than attacks from a known set of access points, such as directly connected terminals, we face attacks from an unknown and large set of access points—a potentially severe security problem. To a lesser extent, systems connected to telephone lines via modems are also more exposed. In fact, the U.S. government considers a system to be only as secure as its most farreaching connection. For instance, a top-secret system may be accessed only from within a building also considered top-secret. The system loses its top-secret rating if any form of communication can occur outside that environment. Some government facilities take extreme security precautions. The connectors that plug a terminal into the secure computer are locked in a safe in the office when the terminal is not in use. A person must know a physical lock combination, as well as authentication information for the computer itself, to gain access to the computer.

Figure 14.2 Network security through domain separation via firewall.


Internet access from company's computers
Internet
company computers
DMZ access from Internet
firewall
access between DMZ and company's computers
DMZ

Unfortunately for systems administrators and computer-security professionals, it is frequently impossible to lock a machine in a room and disallow all remote access. For instance, the Internet network currently connects millions of computers. It is becoming a mission-critical, indispensable resource for many companies and individuals. If you consider the Internet a *club*, then, as in any club with millions of *members*, there are many good members and some bad members. The bad members have many tools they can use to attempt to gain access to the interconnected computers, just as Morris did with his worm.

How can trusted computers be connected safely to an untrustworthy network? One solution is the use of a firewall to separate trusted and untrusted systems. A **firewall** is a computer or router that sits between the trusted and the untrusted. It limits network access between the two **security domains** and monitors and logs all connections. It can also limit connections based on source or destination address, source or destination port, or direction of the connection. For instance, web servers use HTTP to communicate with web browsers. A firewall therefore may allow only HTTP to pass from all hosts outside the firewall only to the web server within the firewall. The Morris Internet worm used the finger protocol to break into computers, so *finger* would not be allowed to pass.

In fact, a firewall can separate a network into multiple domains. A common implementation has the Internet as the untrusted domain; a semitrusted and semisecure network, called the **demilitarized zone (DMZ)**, as another domain; and a company's computers as a third domain (Figure 14.2). Connections are allowed from the Internet to the DMZ computers and from the company computers to the Internet but are not allowed from the Internet or DMZ computers to the company computers. Optionally, controlled communications may be allowed between the DMZ and one or more company computers. For instance, a web server on the DMZ may need to query a database server on the corporate network. With a firewall, all access is contained, and any DMZ systems that are broken into based on the protocols allowed through the firewall still are unable to access the company computers.

Of course, a firewall itself must be secure and attack-proof; otherwise, its ability to secure connections can be compromised. Firewalls do not prevent attacks that **tunnel**, or travel within protocols or connections that the firewall allows. A buffer-overflow attack to a web server will not be stopped by the firewall, for example, because the HTTP connection is allowed; it is the contents of the HTTP connection that house the attack. Likewise, denial-of-service attacks can affect firewalls as much as any other machines. Another vulnerability of firewalls is **spoofing**, in which an unauthorized host pretends to be an authorized host by meeting some authorization criterion. For example, if a firewall rule allows a connection from a host and identifies that host by its IP address, then another host could send packets using that same address and be allowed through the firewall.

14.6 Intrusion Detection

Intrusion detection, as its name suggests, strives to detect attempted or successful intrusions into computer systems and to initiate appropriate responses to the intrusions. Intrusion detection encompasses a wide array of techniques that vary on a number of *axes*. These axes include:

- The time that detection occurs. Detection can occur in real time (while the intrusion is occurring) or after the fact.
- The types of inputs examined to detect intrusive activity. These may include user shell commands, process system calls, and network packet headers or contents. Some forms of intrusions might be detected only by correlating information from several such sources.
- The range of response capabilities. Simple forms of response include alerting an administrator to the potential intrusion or somehow halting the potentially intrusive activity—for example, killing a process engaged in apparently intrusive activity. In a sophisticated form of response, a system might transparently divert an intruder's activity to a **honeypot**—a false resource exposed to the attacker. The resource appears

real to the attacker and enables the system to monitor and gain information about the attack. These degrees of freedom in the design space for detecting intrusions have yielded a wide range of solutions, known as **intrusion-detection systems (IDS)**.

14.6.1 What Constitutes an Intrusion?

Defining a suitable specification of intrusion turns out to be quite difficult, and thus automatic IDSs today typically settle for one of two less ambitious approaches. In the first, called **signature-based detection**, system input or network traffic is examined for specific behavior patterns (or **signatures**) known to indicate attacks. A simple example of signature-based detection is scanning network packets for the string */etc/passwd/* targeted for a UNIX system. Another example is virus-detection software, which scans binaries or network packets for known viruses. The second approach, typically called **anomaly detection**, attempts through various techniques to detect anomalous behavior within computer systems.

Of course, not all anomalous system activity indicates an intrusion, but the presumption is that intrusions often induce anomalous behavior. An example of anomaly detection is monitoring system calls of a daemon process to detect whether the system-call behavior deviates from normal patterns, possibly indicating that a buffer overflow has been exploited in the daemon to corrupt its behavior. Another example is monitoring shell commands to detect anomalous commands for a given user or detecting an anomalous login time for a user, either of which may indicate that an attacker has succeeded in gaining access to that user's account.

Signature-based detection and anomaly detection can be viewed as two sides of the same coin: Signature-based detection attempts to characterize dangerous behaviors and detects when one of these behaviors occurs, whereas anomaly detection attempts to characterize normal (or non-dangerous) behaviors and detects when something other than these behaviors occurs. These different approaches yield IDSs with very different properties, however. In particular, anomaly detection can detect previously unknown methods of intrusion. Signature-based detection will identify only known attacks that can be codified in a recognizable pattern. Thus, new attacks that were not contemplated when the signatures were generated will evade signature-based detection. This problem is well known to vendors of virus-detection software, who must release new signatures with great frequency as new viruses are generated and detected manually.

Anomaly detection is not necessarily superior to signature-based detection, however. Indeed, a significant challenge for systems that attempt anomaly detection is to benchmark "normal" system behavior accurately. If the system is already penetrated when it is benchmarked, then the intrusive activity may be included in the "normal" benchmark. Even if the system is benchmarked cleanly, without influence from intrusive behavior, the benchmark must give a fairly complete picture of normal behavior. Otherwise, the number of false alarms will be excessive.

To illustrate the impact of even a marginally high rate of false alarms, consider an installation consisting of a few tens of UNIX workstations from which records of security-relevant events are recorded for purposes of intrusion detection. A small installation such as this could easily generate a million audit records per day. Only one or two might be worthy of an administrator's investigation.

If we suppose, optimistically, that each such attack is reflected in ten audit records, we can then roughly compute the rate of occurrence of audit records reflecting truly intrusive activity as 2intrusions/day · 10records/intrusion/106records/day = 0.00002 Interpreting this as a "probability of occurrence of intrusive records," we denote it as $P(I)$; that is, event I is the occurrence of a record reflecting truly intrusive behavior. Since $P(I) = 0.00002$, we also know that $P(\neg I) = 1 - P(I) = 0.99998$. Now let A denote the event of the IDS raising an alarm. An accurate IDS should maximize both $P(I|A)$ and $P(\neg I|\neg A)$—that is, the probabilities that an alarm indicates an intrusion and that no alarm indicates no intrusion. Focusing on $P(I|A)$ for the moment, we can compute it using **Bayes' theorem**:

$$P(I|A) = P(I) \cdot P(A|I)/P(I) \cdot P(A|I) + P(\neg I) \cdot P(A|\neg I) = 0.00002 \cdot P(A|I)/0.00002 \cdot P(A|I) + 0.99998 \cdot P(A|\neg I)$$

Now consider the impact of the false-alarm rate $P(A|\neg I)$ on $P(I|A)$. Even with a very good, true alarm-rate $P(A|I) = 0.8$, a seemingly good false-alarm rate $P(A|\neg I) = 0.0001$ yields $P(I|A) \approx 0.14$. That is, fewer than one in every seven alarms indicates a real intrusion! In systems where a security administrator investigates each alarm, this rate of false alarms is exceedingly wasteful and would quickly teach the administrator to ignore alarms. This example illustrates a general principle for IDSs: For usability, an IDS must offer an extremely low false-alarm rate. Achieving a sufficiently low false-alarm rate is a serious challenge for anomaly-detection systems, because of the difficulties of adequately benchmarking normal system behavior. However, research continues to improve anomaly-detection techniques.

14.6.2 Auditing and Logging

A common method of intrusion detection is **audit-trail processing**, in which security-relevant events are logged to an audit trail and then matched against attack signatures (in signature-based detection) or analyzed for anomalous behavior (in anomaly detection). On UNIX systems, a simple pair of programs that can be used to create and analyze audit trails and initiate responses are syslog and swatch. The program syslog creates audit trails and provides a dispatching facility for security-relevant messages. The program swatchapplies simple signature-based detection to syslog audit trails as they are created and initiates responses as indications of intrusions are found.

When syslog is installed on a UNIX system, a daemon process called syslogd is created at system startup. Process syslogd waits for messages from various possible sources and dispatches them as instructed in the syslog.conf configuration file (usually in the */etc/* directory). The process syslogd can be set up to accept messages from a range of different sources, by default including the streams log driver */dev/log*; the process can also accept messages that come over a network from other machines. The file syslog.conf consists of pairs, each containing a selector field and an action field. The selector field identifies the messages to which the corresponding action should be applied. The types of actions that can be applied to messages include sending the message to a file, to a list of users, to the syslogd daemons on other machines, or to a program via a UNIX pipe. In this way, syslogd provides a centralized facility through which various security-relevant messages can be dispatched.

The swatch program is available from http://swatch.sourceforge.net/. It processes messages dispatched to it by the syslogd daemon (or other logging programs) and initiates activities when certain patterns are detected. These messages need not be dispatched to swatch directly but may be written to a file that swatch reads and analyzes. The swatch program matches each audit message against regular expressions listed in a configuration file. With each regular expression, actions to be taken if a match occurs are also listed. Allowable actions include echoing a line to swatch's controlling terminal, ringing a bell at swatch's controlling terminal, sending a message to a user list via e-mail or the write command, or executing a program with arguments drawn from the audit message.

Each regular expression listed in the swatch configuration file can be viewed as a primitive signature for use in signature-based intrusion detection. For example, a useful signature might match any string containing "permission denied" and instruct that a bell signal be directed to swatch's controlling terminal. However, the signatures supported by swatch are fairly rudimentary, since they are applied to only a single audit message at a time. Attack signatures made up of multiple actions that would typically be reflected across multiple audit messages will not be detected by this approach. Commercial audit-trail-processing tools tend to offer richer processing capabilities.

14.6.3 Tripwire

An example of a simple anomaly-detection tool is the **Tripwire file system** integrity-checking tool for UNIX, developed at Purdue University. Tripwire operates on the premise that many intrusions result in anomalous modification of system directories and files. For example, an attacker might modify the */etc/passwd* file of a system to enable the intruder to gain access easily in the future. An intruder might modify system programs, perhaps inserting copies with Trojan horses, or insert new programs into directories commonly found in user shell search paths. Or an intruder might remove system log files to cover his tracks. Tripwire is a tool to monitor file systems for added, deleted, or changed files and to alert system administrators to these modifications.

The operation of Tripwire is controlled by a configuration file tw.config that enumerates the directories and files to be monitored for changes, deletions, or additions. Each entry in this configuration file includes a selection mask to specify the file attributes (inode attributes) to monitor for changes. For example, the selection mask might specify that a file's permissions be monitored but its access time be ignored. In addition, the selection mask can instruct that the file contents— or, more specifically, the results of applying a predefined hash function to the contents of the file—be monitored for changes. Tripwire offers several collision resistant hash functions for this purpose, where a hash function f is collision resistant if, given the result $f(x)$ when f is applied to an input file x, it is computationally infeasible to compute a different file x' such that $f(x') = f(x)$. Thus, monitoring the hash of a file for changes is as good as monitoring the file itself, but storing hashes of files requires far less room than copying the files themselves.

When run initially, Tripwire takes as input the tw.config file and computes a signature for each file or directory consisting of its monitored attributes (inode attributes and hash values). These signatures are stored in a database. When run subsequently, Tripwire inputs both tw.config and the previously stored database, recomputes the signature for each file or directory named in tw.config, and compares this signature with the signature (if any) in the previously computed database. Events reported to an administrator include any monitored file or directory whose signature differs from that in the database (a changed file), any file or directory in a monitored directory for which a signature does not exist in the database (an added file), and any signature in the database for which the corresponding file or directory no longer exists (a deleted file). Although effective for a wide class of attacks, Tripwire does have limitations. Perhaps the most obvious is the need to protect the Tripwire program and its associated files, especially the database file, from unauthorized modification.

For this reason, Tripwire and its associated files should be stored on some tamper-proof medium, such as a write-protected disk or a secure server where logins can be tightly controlled. Unfortunately, this makes it less convenient to update the database after authorized updates to monitored directories and files. A second limitation is that some security-relevant files—for example, system log files—are *supposed* to change over time, and Tripwire does not provide a way to distinguish between an authorized and unauthorized change. So, for example, an attack that modifies (without deleting) a system log that would normally change anyway would escape Tripwire's detection capabilities. The best Tripwire can do in this case is to detect certain obvious inconsistencies (for example, if the log file shrinks). Free and commercial versions of Tripwire are available from http://tripwire.com and httpd://tripwire.org.

14.6.4 System-Call Monitoring

A more recent and speculative form of anomaly detection is system-call monitoring. This approach monitors process system calls to detect in real time when a process is deviating from its expected system-call behavior. It leverages the fact that a program implicitly defines the sequences of system calls that it can make, as determined by the set of possible execution paths through the program. For large and complex programs, the set of possible system-call sequences will be enormous and may not be easily determined, much less stored explicitly. Nevertheless, this approach attempts to characterize "usual" system-call behavior in an abbreviated form so that actual system-call traces can be compared against this characterization to detect anomalous behavior. A type of intrusion that might be detected this way is an attack that *takes over* the process—for example, by exploiting a buffer-overflow vulnerability in the process's program—and then causes the process to execute the attacker's code rather than the original program code. If the attacker's code induces an anomalous system-call sequence, then the intrusion detection system should detect this and take action (for example, kill the process).

One example of this approach, for UNIX processes, was initiated at the University of New Mexico. In this approach, the "usual" system-call sequence for a program is generated by running the program on various inputs, both actual user inputs and inputs designed to elicit a range of behaviors from the program. The system-call sequences thus generated (ignoring parameters) are recorded. For example, one such sequence might be:

open, read, mmap, mmap, open, getrlimit, mmap, close

This sequence is then used to populate a table indicating, for each system call, which system calls can follow it at a distance of one, two, and so forth, up to some distance *k*. For example, suppose we choose *k* = 3. Then, examining the first open, we see that read follows at distance one and mmap follows at distances two and three. Similarly, mmap follows the read at distances one and two, and open follows at distance three. Performing this exercise for each system call, we obtain the table depicted in Figure 14.3.

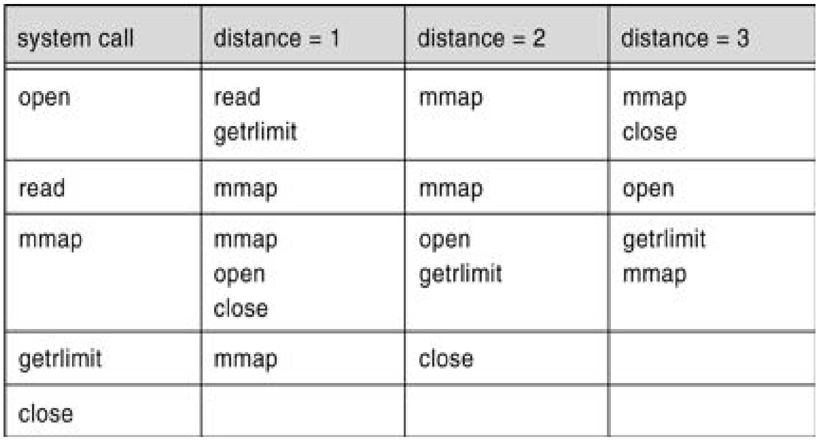
This table is then stored. During subsequent executions of the program, the systemcall sequence is compared with this table to detect discrepancies. For example, if the sequence

open, read, mmap, open, open, getrlimit, mmap, close

is observed (that is, one mmap is replaced with an open), then the following discrepancies are noted: open follows open at a distance of three; open follows read at a distance of two; open follows open at a distance of one; and getrlimit follows open at a distance of two. Whether the IDS determines that this behavior indicates an intrusion might depend on the number of discrepancies observed—perhaps as a fraction of the total possible number of discrepancies—or on the particular system calls involved.

This approach and others like it have been applied primarily to root processes, such as sendmail, in an attempt to detect intrusions. Root processes are attractive targets for this technique because they typically have a limited set of behaviors and do not change often. In addition, they are common targets for attackers because of their privileged status and because, in the case of network services that accept communication over the network, they can easily be probed by attackers from a distance.

Figure 19.3 Data structure derived from system-call sequence.



system call	distance = 1	distance = 2	distance = 3
open	read getrlimit	mmap	mmap close
read	mmap	mmap	open
mmap	mmap open close	open getrlimit	getrlimit mmap
getrlimit	mmap	close	
close			

Although system-call monitoring techniques can be useful, an attacker may craft an attack that does not perturb the sequence of system calls enough to alert the IDS. For example, the attacker may have a copy of the software that he plans to attack and may try various intrusions until one is found that leaves the system-call sequence largely intact. Additional security can be achieved by extending these detection techniques to take into account the parameters of the system calls.

14.7 Cryptography

In an isolated computer, the operating system can reliably determine the sender and recipient of all inter process 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.

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 to write a file when it cannot trust the named source of the request? And how is it supposed to provide read protection for a file when it cannot determine who will receive the file 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 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—the key is the *source* of the message.

Similarly, a sender can encode its message so that only a computer with a certain key can decode the message, so that the key becomes the *destination*. Unlike network addresses, however, keys are designed so that it is computationally infeasible 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.

14.7.1 Authentication

Constraining the set of potential senders of a message is called *authentication.* An authentication algorithm enables the recipient of a message to verify that a message was created by some computer possessing a certain key. More precisely, an authentication algorithm 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 $k \varepsilon K$, $S(k)$ is a function for generating authenticators from messages. Both S and $S(k)$ for any k should be efficiently computable functions.
- A function $V : K \rightarrow (M \times A \rightarrow \{\text{true, false}\})$. That is, for each $k \varepsilon K$, $V(k)$ 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 $a \varepsilon A$ such that $V(k)(m, a) = \text{true}$ only if it possesses $S(k)$. Thus, a computer holding $S(k)$ can generate authenticators on messages so that any other computer possessing $V(k)$ can verify them. However, a computer not holding $S(k)$ is unable to generate authenticators on messages that can be verified using $V(k)$. Since authenticators are generally exposed (for example, they are sent on the network with the messages themselves), it must not be feasible to derive $S(k)$ from the authenticators.

Authentication algorithms come in two main varieties. In a **message authentication code (MAC)**, knowledge of $V(k)$ and knowledge of $S(k)$ are equivalent: one can be derived from the other. For this reason, it is as important to protect $V(k)$ as it is to protect $S(k)$. A simple example of a MAC defines $S(k)(m) = f(k, m)$, where f is a function that is one-way on its first argument (that is, its first argument cannot be derived from its output) and collision-resistant on its second (that is, it is infeasible to find an $m' \neq m$ such that $f(k, m) = f(k, m')$). A suitable verification algorithm is then $V(k)(m, a) \equiv (f(k, m) = a)$. Note that k is needed to compute both $S(k)$ and $V(k)$; that is, 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*. In a digital signature algorithm, it is computationally infeasible to derive $S(k)$ from $V(k)$; in particular, V is a one-way function. Thus, $V(k)$ need not be kept secret and can be widely distributed. For this reason, $V(k)$ is called the **public key**, and conversely $S(k)$ (or just k) is the **private key**. Here we describe one digital signature algorithm, known as *RSA* after the names of its inventors. In the RSA scheme, the key k is a pair $\langle d, N\rangle$ where N is the product of two large, randomly chosen prime numbers p and q (for example, p and q are 512 bits each). The signature algorithm is $S(\langle d, N\rangle)(m) = f(m)d \bmod N$, where f is a collision-resistant function. The verification algorithm is then $V(\langle d, N\rangle)(m, a) \equiv (ae \bmod N = f(m))$, where e satisfies $ed \bmod (p - 1)(q - 1) = 1$. It is widely believed to be computationally infeasible for a computer possessing $V(\langle d, N\rangle)$ (that is, possessing $\langle e, N\rangle$ but not e) to generate $S(\langle d, N\rangle)$ (that is, d).

14.7.2 Encryption

Encryption is a means for constraining the possible receivers of a message. Encryption is thus complementary to authentication, and to emphasize this we provide a parallel treatment of it. An encryption algorithm enables the sender of a message to ensure that only a computer possessing a certain key can read the message. More precisely, an encryption algorithm consists of the following components:

- A set K of keys.
- A set M of messages.
- A set C of cipher texts.
- A function $E : K \rightarrow (M \rightarrow C)$. That is, for each k ε K, $E(k)$ is a function for generating cipher texts from messages. Both E and $E(k)$ for any k should be efficiently computable functions.
- A function $D : K \rightarrow (C \rightarrow M)$. That is, for each k ε K, $D(k)$ is a function for generating messages from cipher texts. Both D and $D(k)$ for any k should be efficiently computable functions.

The essential property that an encryption algorithm must provide is that, given a cipher text c ε C, a computer can compute m such that $E(k)(m) = c$ only if it possesses $D(k)$. Thus, a computer holding $D(k)$ can decrypt cipher texts to the plaintexts used to produce them. However, a computer not holding $D(k)$ is unable to decrypt cipher texts. Since cipher texts are generally exposed (for example, they are sent on the network), it is important that it be infeasible to derive $D(k)$ from the Cipher texts.

Just as there are two types of authentication algorithms, there are two main types of encryption algorithms. In the first type, called a **symmetric encryption algorithm**, $E(k)$ can be derived from $D(k)$ and vice versa. Therefore, the secrecy of $E(k)$ must be protected to the same extent as that of $D(k)$. For the past 20 years or so, the most commonly used symmetric encryption algorithm in the United States for civilian applications has been the **data-encryption standard (DES)** adopted by the National Institute of Standards and Technology (NIST). However, DES is now considered insecure for many applications because its keys, of 56 bits in length, can be exhaustively searched with moderate computing resources. NIST has since adopted a new encryption algorithm, called the **advanced encryption standard (AES)**, to replace the DES.

In an **asymmetric encryption algorithm**, it is computationally infeasible to derive $D(k)$ from $E(k)$, and so $E(k)$ need not be kept secret and can be widely disseminated; $E(k)$ is the public key, and $D(k)$ (or just k) is the private key. Interestingly, the mechanism underlying the RSA signature algorithm can also be used to yield an asymmetric encryption algorithm. Again, the key k is a pair $\langle d, N \rangle$, where N is the product of two large, randomly chosen prime numbers p and q. The encryption algorithm is $E(\langle d, N\rangle)(m) = me \bmod N$, where e, d and N are as before. The decryption algorithm is then $D(\langle d, N\rangle)(c) = cd \bmod N$.

14.7.3 Use of Cryptography

Network protocols are typically organized in **layers**, each layer acting as a client to 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 TCP peer at the other end of the TCP 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 IP peer on the destination computer. This IP peer then delivers the TCP packet up 873 to the TCP peer on that machine. All in all, the OSI Reference Model, which has been almost universally adopted for data networking, defines seven such protocol layers.

Network-layer security that has been standardized (or **IPSec**) defines IP packet formats that allow the insertion of authenticators and the encryption of packet contents. IPSec is becoming widely used as the basis for **virtual private networks**. Numerous protocols also have been developed for applications to use. 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 runs over 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—for example, the user may be required to type a password.

14.8 Computer-Security Classifications

The U.S. Department of Defense Trusted Computer System Evaluation Criteria specify four divisions of security in systems: A, B, C, and D. The lowest-level classification is division D, or minimal protection. Division D comprises only one class and is used for systems that have failed to meet the requirements of any of the other security classes. For instance, MS-DOS and Windows 3.1 are in division D. Division C, the next level of security, provides discretionary protection and accountability of users and their actions through the use of audit capabilities. Division C has two levels: C1 and C2. A C1-class system incorporates some form of controls that allow users to protect private information and to keep other users from accidentally reading or destroying their data. A C1 environment is one in which cooperating users access data at the same levels of sensitivity. Most versions of UNIX are C1 class.

The sum total of all protection systems within a computer system (hardware, software, firmware) that correctly enforce a security policy is known as a **Trusted Computer Base (TCB)**. The TCB of a C1 system controls access between users and files by allowing the user to specify and control sharing of objects by named individuals or defined groups. In addition, the TCB requires that the users identify themselves before they start any activities that the TCB is expected to mediate. This identification is accomplished via a protected mechanism or password; the TCB protects the authentication data so that they are inaccessible to unauthorized users. A C2-class system adds an individual-level access control to the requirements of a C1 system. For example, access rights of a file can be specified to the level of a single individual. In addition, the system administrator can selectively audit the actions of any one or more users based on individual identity. The TCB also protects itself from modification of its code or data structures. In addition, no information produced by a prior user is available to another user who accesses a storage object that has been released back to the system. Some special, secure versions of UNIX have been certified at the C2 level.

Division-B mandatory-protection systems have all the properties of a class-C2 system; in addition, they attach a sensitivity label to each object. The B1-class TCB maintains the security label of each object in the system; the label is used for decisions pertaining to mandatory access control. For example, a user at the confidential level could not access a file at the more sensitive secret level. The TCB also denotes the sensitivity level at the top and bottom of each page of any human-readable output. In addition to the normal user-name-password authentication information, the TCB also maintains the clearance and authorizations of individual users and will support at least two levels of security. These levels are hierarchical, so that a user may access any objects that carry sensitivity labels equal to or lower than his security clearance. For example, a secret-level user could access a file at the confidential level in the absence of other access controls. Processes are also isolated through the use of distinct address spaces.

A B2-class system extends the sensitivity labels to each system resource, such as storage objects. Physical devices are assigned minimum- and maximum-security levels that the system uses to enforce constraints imposed by the physical environments in which the devices are located. In addition, a B2 system supports covert channels and the auditing of events that may lead to the exploitation of a covert channel. A B3-class system allows the creation of access-control lists that denote users or groups *not* granted access to a given named object. The TCB also contains a mechanism to monitor events that may indicate a violation of security policy. The mechanism notifies the security administrator and, if necessary, terminates the event in the least disruptive manner. The highest-level classification is division A. A class-A1 system is functionally equivalent to a B3 system architecturally, but it uses formal design specifications and verification techniques, granting a high degree of assurance that the TCB has been implemented correctly. A system beyond class A1 might be designed and developed in a trusted facility by trusted personnel.

The use of a TCB merely ensures that the system can enforce aspects of a security policy; the TCB does not specify what the policy should be. Typically, a given computing environment develops a security policy for **certification** and has the plan **accredited** by a security agency, such as the National Computer Security Center. Certain computing environments may require other certification, such as by

TEMPEST, which guards against electronic eavesdropping. For example, a TEMPEST-certified system has terminals that are shielded to prevent electromagnetic fields from escaping. This shielding ensures that equipment outside the room or building where the terminal is housed cannot detect what information is being displayed by the terminal.

14.9 An Example: Windows NT

Microsoft Windows NT is a general-purpose operating system designed to support a variety of security features and methods. In this section, we examine features that Windows NT uses to perform security functions. The NT security model is based on the notion of **user accounts**. NT allows the creation of any number of user accounts, which can be grouped in any manner. Access to system objects can then be permitted or denied as desired. Users are identified to the system by a *unique* Security ID. When a user logs on, NT creates a **security access token** that includes the Security ID for the user, Security IDs for any groups of which the user is a member, and a list of any special privileges that the user has.

Examples of special privileges include backing up files and directories, shutting down the computer, logging on interactively, and changing the system clock. Every process that NT runs on behalf of a user will receive a copy of the access token. The system uses the Security IDs in the access token to permit or deny access to system objects whenever the user, or a process on behalf of the user, attempts to access the object. Authentication of a user account is typically accomplished via a user name and password, although the modular design of NT allows the development of custom authentication packages. For example, a retinal (or eye) scanner might be used to verify that the user is who she says she is. NT uses the idea of a subject to ensure that programs run by a user do not get greater access to the system than the user is authorized to have. A **subject** is used to track and manage permissions for each program that a user runs; it is composed of the user's access token and the program acting on behalf of the user. Since NT operates with a client-server model, two classes of subjects are used to control access.

An example of a **simple subject** is the typical application program that a user executes after she logs on. The simple subject is assigned a **security context** based on the security access token of the user. A **server subject**is a process implemented as a protected server that uses the security context of the client when acting on the client's behalf. The technique whereby one process takes on the security attributes of another is called **impersonation**.

As mentioned in Section 14.5, auditing is a useful security technique. NT has built-in auditing that allows many common security threats to be monitored. Examples include failure auditing for logon and logoff events to detect random password breakins, success auditing for logon and logoff events to detect logon activity at strange hours, success and failure write-access auditing for executable files to track a virus outbreak, and success and failure auditing for file access to detect access to sensitive files. Security attributes of an object in NT are described by a **security descriptor**. The security descriptor contains the Security ID of the owner of the object (who can change the access permissions), a group Security ID used only by the POSIX subsystem, a discretionary access-control list that identifies which users or groups are allowed (and which are not allowed) access, and a system access-control list that controls which auditing messages the system will generate. For example, the security descriptor of the file *foo.bar* might have owner avi and this discretionary accesscontrol list:

- avi—all access
- group cs—read-write access
- user cliff—no access.

In addition, it might have a system access-control list of audit writes by everyone. An access-control list is composed of access-control entries that contain the Security ID of the individual and an access mask that defines all possible actions on the object, with a value of AccessAllowed or AccessDenied for each action. Files in NT may have the following access types: ReadData, WriteData, AppendData, Execute, ReadExtendedAttribute, WriteExtendedAttribute, ReadAttributes, and WriteAttributes. We can see how this allows a fine degree of control over access to objects.

NT classifies objects as either container objects or noncontainer objects. **Container objects**, such as directories, can logically contain other objects. By default, when an object is created within a container object, the new object inherits permissions from the parent object. It does so as well if the user copies a file from one directory to a new directory—the file will inherit the permissions of the destination directory. **Noncontainer objects** inherit no other permissions.

If a permission is changed on a directory, however, the new permissions do not automatically apply to existing files and subdirectories; the user may explicitly apply them if she so desires. Also, if the user moves a file into a new directory, the current permissions of the file move with it. The system administrator can prohibit printing to a printer on the system for all or part of a day and can use the NT Performance Monitor to help her spot approaching problems. In general, NT does a good job of providing features to help ensure a secure computing environment. Many of these features are not enabled by default, however. For a real multiuser environment, the system administrator should formulate a security plan and implement it, using the features that NT provides.

14.10 SUMMARY

Protection is an internal problem. Security, in contrast, must consider both the computer system and the environment—people, buildings, businesses, valuable objects, and threats—within which the system is used. The data stored in the computer system must be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. It is easier to protect against accidental loss of data consistency than to protect against malicious access to the data. Absolute protection of the information stored in a computer system from malicious abuse is not possible; but the cost to the perpetrator can be made sufficiently high to deter most, if not all, attempts to access that information without proper authority. Authentication methods are used to identify legitimate users of a system.

In addition to standard user-name and password protection, several authentication methods are used. One-time passwords, for example, change from session to session to avoid replay attacks. Two-factor authentication requires two forms of authentication, such as a hardware calculator with an activation PIN. Along with biometric devices, these methods greatly decrease the chance of authentication forgery. Several types of attacks can be launched against individual computers or the masses. Stack- and buffer-overflow techniques allow successful attackers to change their level of system access. Viruses and worms are self-perpetuating, sometimes infecting thousands of computers. Denial-of-service attacks prevent legitimate use of target systems. Methods of preventing or detecting security incidents include intrusion-detection systems, auditing and logging of system events, monitoring of system software changes, and system-call monitoring. Cryptography can be used both within a system and between systems to prevent interception of information by hackers.

EXERCISES

14.1 A password may become known to other users in a variety of ways. Is there a simple method for detecting that such an event has occurred? Explain your answer.

14.2 The list of all passwords is kept within the operating system. Thus, if a user manages to read this list, password protection is no longer provided. Suggest a scheme that will avoid this problem. (Hint: Use different internal and external representations.)

14.3 An experimental addition to UNIX allows a user to connect a *watchdog* program to a file. The watchdog is invoked whenever a program requests access to the file. The watchdog then either grants or denies access to the file. Discuss two pros and two cons of using watchdogs for security.

14.4 The UNIX program COPS scans a given system for possible security holes and alerts the user to possible problems. What are two potential hazards of using such a system for security? How can these problems be limited or eliminated?

14.5 Discuss a means by which managers of systems connected to the Internet could have designed their systems to limit or eliminate the damage done by a worm. What are the drawbacks of making the change that you suggest?

14.6 Argue for or against the judicial sentence handed down against Robert Morris, Jr., for his creation and execution of the Internet worm.

14.7 Make a list of six security concerns for a bank's computer system. For each item on your list, state whether this concern relates to physical, human, or operating-system security.

14.8 What are two advantages of encrypting data stored in the computer system?

REFERENCES

1. ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE "OPERATING SYSTEM CONCEPTS *with JAVA* " JOHN WILEY & SONS, INC., SIXTH EDITION.
2. HARRIS JA " OPERATING SYSTEMS" SCHANM'S OUTLINE SERIES"
3. BHATT PCP "AN INTRODUCTION TO OPERATING SYSTMES", PHI PUBLICATIONS.
4. D M DHAMDHERE, "OPERATING SYSTEMS A CNCEPT BAED APPROACH" The McGraw-Hill Companies., second Edition.
6. William Stallings " Operating Systems Internals and Design Principles", Sixth Edition.