CHAPTER

3

Operating Systems

In this chapter we study operating systems, which are software packages that coordinate a computer's internal activities as well as oversee its communication with the outside world. It is a computer's operating system that transforms the computer hardware into a useful tool. Our goal is to understand what operating systems do and how they do it. Such a background is central to being an enlightened computer user.

**Asterisks indicate suggestions for optional sections.*

An **operating system** is the software that controls the overall operation of a computer. It provides the means by which a user can store and retrieve files, provides the interface by which a user can request the execution of programs, and provides the environment necessary to execute the programs requested.

Perhaps the best known example of an operating system is Windows, which is provided in numerous versions by Microsoft and widely used in the PC arena. Another well-established example is UNIX, which is a popular choice for larger computer systems as well as PCs. In fact, UNIX is the core of two other popular operating systems: Mac OS, which is the operating system provided by Apple for its range of Mac machines, and Solaris, which was developed by Sun Microsystems (now owned by Oracle). Still another example of an operating system found on both large and small machines is Linux, which was originally developed noncommercially by computer enthusiasts and is now available through many commercial sources, including IBM.

For casual computer users, the differences between operating systems are largely cosmetic. For computing professionals, different operating systems can represent major changes in the tools they work with or the philosophy they follow in disseminating and maintaining their work. Nevertheless, at their core all mainstream operating systems address the same kinds of problems that computing experts have faced for more than half a century.

3.1 The History of Operating Systems

Today's operating systems are large, complex software packages that have grown from humble beginnings. The computers of the 1940s and 1950s were not very flexible or efficient. Machines occupied entire rooms. Program execution required significant preparation of equipment in terms of mounting magnetic tapes, placing punched cards in card readers, setting switches, and so on. The execution of each program, called a **job,** was handled as an isolated activity—the machine was prepared for executing the program, the program was executed, and then all the tapes, punched cards, etc. had to be retrieved before the next program preparation could begin. When several users needed to share a machine, sign-up sheets were provided so that users could reserve the machine for blocks of time. During the time period allocated to a user, the machine was totally under that user's control. The session usually began with program setup, followed by short periods of program execution. It was often completed in a hurried effort to do just one more thing ("It will only take a minute") while the next user was impatiently starting to set up.

In such an environment, operating systems began as systems for simplifying program setup and for streamlining the transition between jobs. One early development was the separation of users and equipment, which eliminated the physical transition of people in and out of the computer room. For this purpose a computer operator was hired to operate the machine. Anyone wanting a program run was required to submit it, along with any required data and special directions about the program's requirements, to the operator and return later for the results. The operator, in turn, loaded these materials into the machine's mass storage where a program called the operating system could read and execute them one at a time. This was the beginning of **batch processing**—the execution of jobs by collecting them in a single batch, then executing them without further interaction with the user.

In batch processing systems, the jobs residing in mass storage wait for execution in a **job queue** (Figure 3.1). A **queue** is a storage organization in which objects (in this case, jobs) are ordered in **first-in, first-out** (abbreviated FIFO and pronounced "FI-foe") fashion. That is, the objects are removed from the queue in the order in which they arrived. In reality, most job queues do not rigorously follow the FIFO structure, since most operating systems provide for consideration of job priorities. As a result, a job waiting in the job queue can be bumped by a higher-priority job.

In early batch-processing systems, each job was accompanied by a set of instructions explaining the steps required to prepare the machine for that particular job. These instructions were encoded, using a system known as a job control language (JCL), and stored with the job in the job queue. When the job was selected for execution, the operating system printed these instructions at a printer where they could be read and followed by the computer operator. This communication between the operating system and the computer operator is still seen today, as witnessed by PC operating systems that report such errors as "network not available" and "printer not responding."

A major drawback to using a computer operator as an intermediary between a computer and its users is that the users have no interaction with their jobs once they are submitted to the operator. This approach is acceptable for some applications, such as payroll processing, in which the data and all processing decisions are established in advance. However, it is not acceptable when the user must interact with a program during its execution. Examples include reservation systems in which reservations and cancellations must be reported as they occur; word processing systems in which documents are developed in a dynamic write and rewrite manner; and computer games in which interaction with the machine is the central feature of the game.

To accommodate these needs, new operating systems were developed that allowed a program being executed to carry on a dialogue with the user through remote terminals—a feature known as **interactive processing** (Figure 3.2). (A terminal consisted of little more than an electronic typewriter by which the user could type input and read the computer's response that was printed on paper. Today terminals have evolved into more sophisticated devices called workstations and even into complete PCs that can function as stand-alone computers when desired.)

Figure 3.1 Batch processing

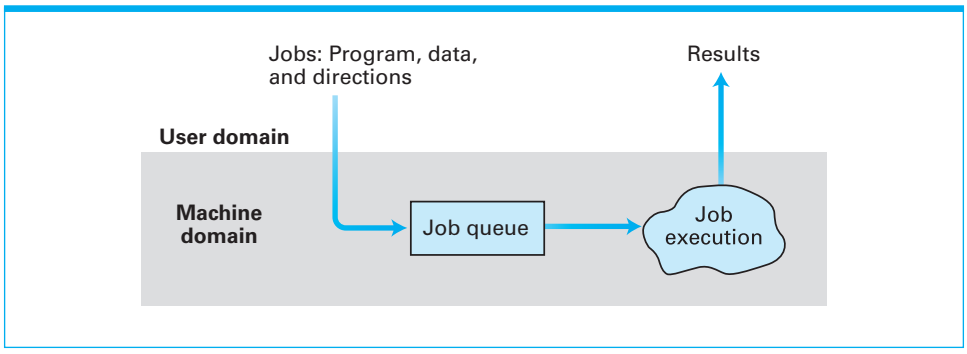
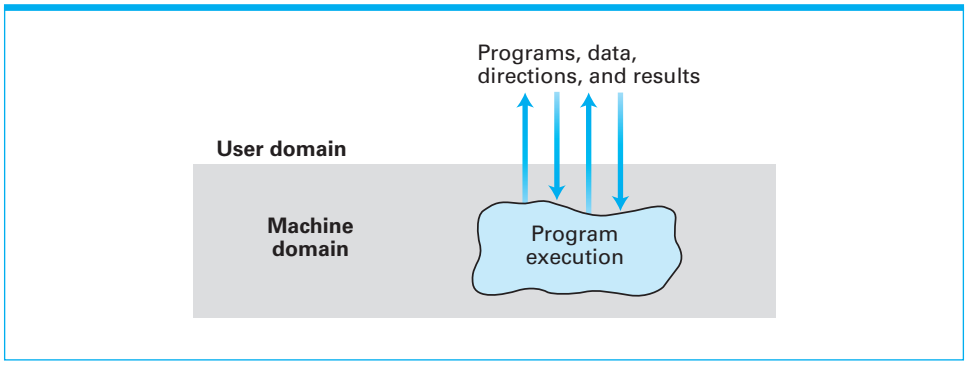


Figure 3.2 Interactive processing



Paramount to successful interactive processing is that the actions of the computer be sufficiently fast to coordinate with the needs of the user rather than forcing the user to conform to the machine's timetable. (The task of processing payroll can be scheduled to conform to the amount of time required by the computer, but using a word processor would be frustrating if the machine did not respond promptly as characters are typed.) In a sense, the computer is forced to execute tasks under a deadline, a process that became known as **real-time processing** in which the actions performed are said to occur in real-time. That is, to say that a computer performs a task in real time means that the computer performs the task in accordance with deadlines in its (external real-world) environment.

If interactive systems had been required to serve only one user at a time, real-time processing would have been no problem. But computers in the 1960s and 1970s were expensive, so each machine had to serve more than one user. In turn, it was common for several users, working at remote terminals, to seek interactive service from a machine at the same time, and real-time considerations presented obstacles. If the operating system insisted on executing only one job at a time, only one user would receive satisfactory real-time service.

The solution to this problem was to design operating systems that provided service to multiple users at the same time: a feature called **time-sharing.** One means of implementing time-sharing is to apply the technique called **multiprogramming** in which time is divided into intervals and then the execution of each job is restricted to only one interval at a time. At the end of each interval, the current job is temporarily set aside and another is allowed to execute during the next interval. By rapidly shuffling the jobs back and forth in this manner, the illusion of several jobs executing simultaneously is created. Depending on the types of jobs being executed, early time-sharing systems were able to provide acceptable real-time processing to as many as 30 users simultaneously. Today, multiprogramming techniques are used in single-user as well as multiuser systems, although in the former the result is usually called **multitasking.** That is, time-sharing refers to multiple users sharing access to a common computer, whereas multitasking refers to one user executing numerous tasks simultaneously.

With the development of multiuser, time-sharing operating systems, a typical computer installation was configured as a large central computer connected to numerous workstations. From these workstations, users could communicate directly with the computer from outside the computer room rather than submitting requests to a computer operator. Commonly used programs were stored in

the machine's mass storage devices, and operating systems were designed to execute these programs as requested from the workstations. In turn, the role of a computer operator as an intermediary between the users and the computer begins to fade.

Today, the existence of a computer operator has essentially disappeared, especially in the arena of personal computers where the computer user assumes all of the responsibilities of computer operation. Even most large computer installations run essentially unattended. Indeed, the job of computer operator has given way to that of a system administrator who manages the computer system—obtaining and overseeing the installation of new equipment and software, enforcing local regulations such as the issuing of new accounts and establishing mass storage space limits for the various users, and coordinating efforts to resolve problems that arise in the system—rather than operating the machines in a hands-on manner.

In short, operating systems have grown from simple programs that retrieved and executed programs one at a time into complex systems that coordinate time-sharing, maintain programs and data files in the machine's mass storage devices, and respond directly to requests from the computer's users.

But the evolution of operating systems continues. The development of multiprocessor machines has led to operating systems that provide time-sharing/multitasking capabilities by assigning different tasks to different processors as well as by sharing the time of each single processor. These operating systems must wrestle with such problems as **load balancing** (dynamically allocating tasks to the various processors so that all processors are used efficiently) as well as **scaling** (breaking tasks into a number of subtasks compatible with the number of processors available).

Moreover, the advent of computer networks in which numerous machines are connected over great distances has led to the creation of software systems to coordinate the network's activities. Thus the field of networking (which we will study in Chapter 4) is in many ways an extension of the subject of operating systems—the goal being to manage resources across many users on many machines rather than a single, isolated computer.

What's in a Smartphone?

As cell phones have become more powerful, it has become possible for them to offer services well beyond simply processing voice calls. A typical **smartphone** can now be used to text message, browse the Web, provide directions, view multimedia content—in short, it can be used to provide many of the same services as a traditional PC. As such, smartphones require full-fledged operating systems, not only to manage the limited resources of the smartphone hardware, but also to provide features that support the rapidly expanding collection of smartphone application software. The battle for dominance in the smartphone operating system market place promises to be fierce and will likely be settled on the basis of which system can provide the most imaginative features at the best price. Competitors in the smartphone operating system arena include Apple's iPhone OS, Research In Motion's BlackBerry OS, Microsoft's Windows Phone, Nokia's Symbian OS, and Google's Android.

Still another direction of research in operating systems focuses on devices that are dedicated to specific tasks such as medical devices, vehicle electronics, home appliances, cell phones, or other hand-held computers. The computer systems found in these devices are known as **embedded systems.** Embedded operating systems are often expected to conserve battery power, meet demanding real-time deadlines, or operate continuously with little or no human oversight. Successes in this endeavor are marked by systems such as VxWORKS, developed by Wind River Systems and used in the Mars Exploration Rovers named Spirit and Opportunity; Windows CE (also known as Pocket PC) developed by Microsoft; and Palm OS developed by PalmSource, Inc., especially for use in hand-held devices.

Questions & Exercises

1. Identify examples of queues. In each case, indicate any situations that violate the FIFO structure.
2. Which of the following activities require real-time processing?
   a. Printing mailing labels
   b. Playing a computer game
   c. Displaying numbers on a smartphone screen as they are dialed
   d. Executing a program that predicts the state of next year's economy
   e. Playing an MP3 recording
3. What is the difference between embedded systems and PCs?
4. What is the difference between time-sharing and multitasking?

3.2 Operating System Architecture

To understand the composition of a typical operating system, we first consider the complete spectrum of software found within a typical computer system. Then we will concentrate on the operating system itself.

A Software Survey

We approach our survey of the software found on a typical computer system by presenting a scheme for classifying software. Such classification schemes invariably place similar software units in different classes in the same manner as the assignment of time zones dictates that nearby communities must set their clocks an hour apart even though there is no significant difference between the occurrence of sunrise and sunset. Moreover, in the case of software classification, the dynamics of the subject and the lack of a definitive authority lead to contradictory terminology. For example, users of Microsoft's Windows operating systems will find groups of programs called "Accessories" and "Administrative Tools" that include software from what we will call the application and utility classes. The following taxonomy should therefore be viewed as a means of gaining a foothold in an extensive, dynamic subject rather than as a statement of universally accepted fact.

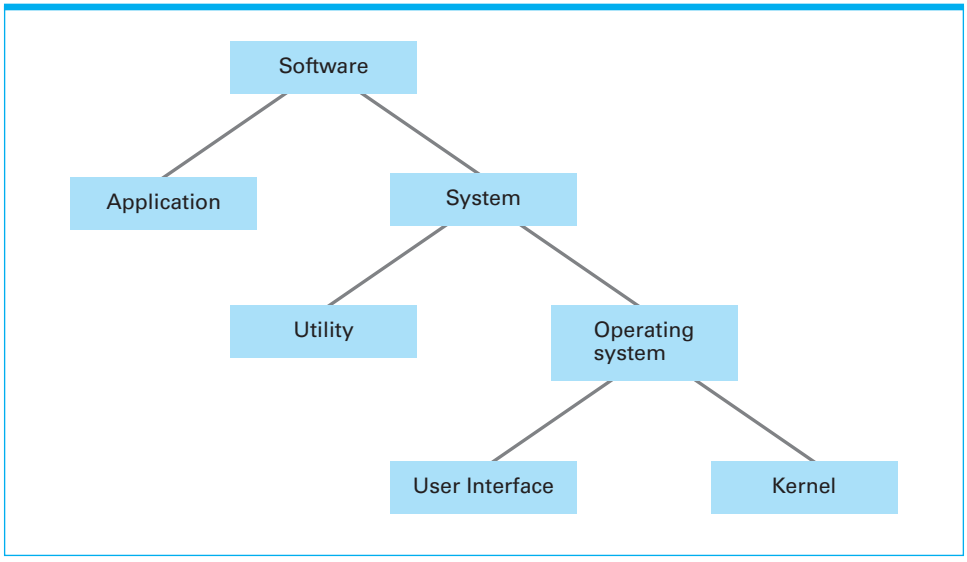
Let us begin by dividing a machine's software into two broad categories: **application software** and **system software** (Figure 3.3). Application software consists of the programs for performing tasks particular to the machine's utilization. A machine used to maintain the inventory for a manufacturing company will contain different application software from that found on a machine used by an electrical engineer. Examples of application software include spreadsheets, database systems, desktop publishing systems, accounting systems, program development software, and games.

In contrast to application software, system software performs those tasks that are common to computer systems in general. In a sense, the system software provides the infrastructure that the application software requires, in much the same manner as a nation's infrastructure (government, roads, utilities, financial institutions, etc.) provides the foundation on which its citizens rely for their individual lifestyles.

Within the class of system software are two categories: One is the operating system itself and the other consists of software units collectively known as **utility software.** The majority of an installation's utility software consists of programs for performing activities that are fundamental to computer installations but not included in the operating system. In a sense, utility software consists of software units that extend (or perhaps customize) the capabilities of the operating system. For example, the ability to format a magnetic disk or to copy a file from a magnetic disk to a CD is often not implemented within the operating system itself but instead is provided by means of a utility program. Other instances of utility software include software to compress and decompress data, software for playing multimedia presentations, and software for handling network communication.

Implementing certain activities as utility software allows system software to be customized to the needs of a particular installation more easily than if they were included in the operating system. Indeed, it is common to find companies or

Figure 3.3 Software classification



Linux

For the computer enthusiast who wants to experiment with the internal components of an operating system, there is Linux. Linux is an operating system originally designed by Linus Torvalds while a student at the University of Helsinki. It is a nonproprietary product and available, along with its source code (see Chapter 6) and documentation, without charge. Because it is freely available in source code form, it has become popular among computer hobbyists, students of operating systems, and programmers in general. Moreover, Linux is recognized as one of the more reliable operating systems available today. For this reason, several companies now package and market versions of Linux in an easily usable form, and these products are now challenging the long-established commercial operating systems on the market. You can learn more about Linux from the website at www.linux.org.

individuals who have modified, or added to, the utility software that was originally provided with their machine's operating system.

Unfortunately, the distinction between application software and utility software can be vague. From our point of view, the difference is whether the package is part of the computer's "software infrastructure." Thus a new application may evolve to the status of a utility if it becomes a fundamental tool. When still a research project, software for communicating over the Internet was considered application software; today such tools are fundamental to most PC usage and would therefore be classified as utility software.

The distinction between utility software and the operating system is equally vague. In particular, antitrust lawsuits in the United States and Europe have been founded on questions regarding whether units such as browsers and media players are components of Microsoft's operating systems or utilities that Microsoft has included merely to squash competition.

Components of an Operating System

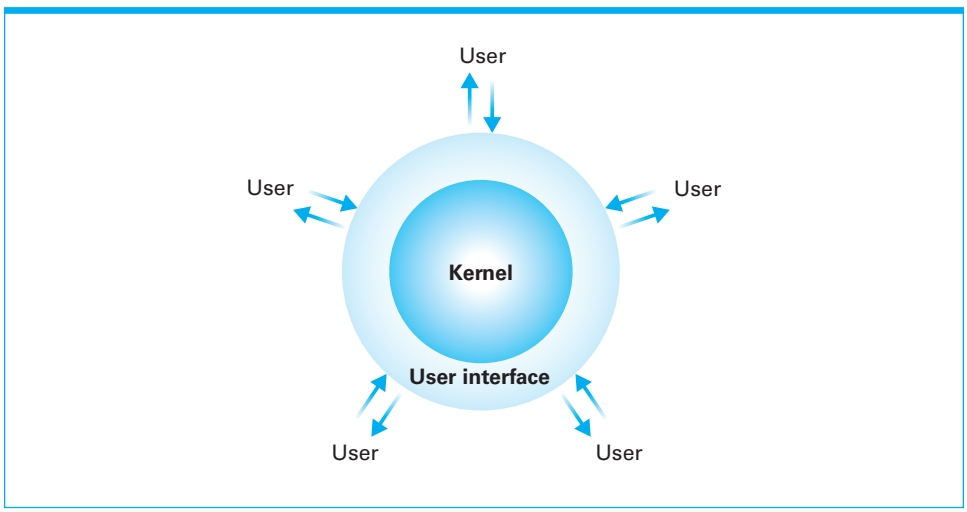
Let us focus now on components that are within the domain of an operating system. In order to perform the actions requested by the computer's users, an operating system must be able to communicate with those users. The portion of an operating system that handles this communication is often called the **user interface.** Older user interfaces, called **shells,** communicated with users through textual messages using a keyboard and monitor screen. More modern systems perform this task by means of a **graphical user interface (GUI**—pronounced "GOO–ee") in which objects to be manipulated, such as files and programs, are represented pictorially on the display as icons. These systems allow users to issue commands by using one of several common input devices. For example, a computer mouse, with one or more buttons, can be used to click or drag icons on the screen. In place of a mouse, special-purpose pointing devices or styluses are often used by graphic artists or on several types of handheld devices. More recently, advances in fine-grained touch screens allow users to manipulate icons directly with their fingers. Whereas today's GUIs use two-dimensional image projection systems, three-dimensional interfaces that allow human users to communicate with computers by means of 3D projection systems, tactile sensory devices, and surround sound audio reproduction systems are subjects of current research.

Although an operating system's user interface plays an important role in establishing a machine's functionality, this framework merely acts as an intermediary between the computer's user and the real heart of the operating system (Figure 3.4). This distinction between the user interface and the internal parts of the operating system is emphasized by the fact that some operating systems allow a user to select among different interfaces to obtain the most comfortable interaction for that particular user. Users of the UNIX operating system, for example, can select among a variety of shells including the Bourne shell, the C shell, and the Korn shell, as well as a GUI called X11. The earliest versions of Microsoft Windows were a GUI application program that could be loaded from the MS-DOS operating system's command shell. The DOS cmd.exe shell can still be found as a utility program in the latest versions of Windows, although this interface is almost never required by casual users. Similarly, Apple's OS X retains a Terminal utility shell that hearkens back to that system's UNIX ancestors.

An important component within today's GUI shells is the **window manager,** which allocates blocks of space on the screen, called windows, and keeps track of which application is associated with each window. When an application wants to display something on the screen, it notifies the window manager, and the window manager places the desired image in the window assigned to the application. In turn, when a mouse button is clicked, it is the window manager that computes the mouse's location on the screen and notifies the appropriate application of the mouse action. Window managers are responsible for what is generally called the "style" of a GUI, and most managers offer a range of configurable choices. Linux users even have a range of choices for a window manager, with popular choices including KDE and Gnome.

In contrast to an operating system's user interface, the internal part of an operating system is called the **kernel.** An operating system's kernel contains those software components that perform the very basic functions required by the computer installation. One such unit is the **file manager,** whose job is to coordinate the use of the machine's mass storage facilities. More precisely, the file manager maintains records of all the files stored in mass storage, including where

Figure 3.4 The user interface acts as an intermediary between users and the operating system's kernel



each file is located, which users are allowed to access the various files, and which portions of mass storage are available for new files or extensions to existing files. These records are kept on the individual storage medium containing the related files so that each time the medium is placed online, the file manager can retrieve them and thus know what is stored on that particular medium.

For the convenience of the machine's users, most file managers allow files to be grouped into a bundle called a **directory** or **folder.** This approach allows a user to organize his or her files according to their purposes by placing related files in the same directory. Moreover, by allowing directories to contain other directories, called subdirectories, a hierarchical organization can be constructed. For example, a user may create a directory called `MyRecords` that contains subdirectories called `FinancialRecords`, `MedicalRecords`, and `HouseHoldRecords`. Within each of these subdirectories could be files that fall within that particular category. (Users of a Windows operating system can ask the file manager to display the current collection of folders by executing the utility program Windows Explorer.)

A chain of directories within directories is called a **directory path.** Paths are often expressed by listing the directories along the path separated by slashes. For instance, `animals/prehistoric/dinosaurs` would represent the path starting at the directory named `animals`, passing through its subdirectory named `prehistoric`, and terminating in the sub-subdirectory `dinosaurs`. (For Windows users the slashes in such a path expression are reversed as in `animals\prehistoric\dinosaurs`.)

Any access to a file by other software units is obtained at the discretion of the file manager. The procedure begins by requesting that the file manager grant access to the file through a procedure known as opening the file. If the file manager approves the requested access, it provides the information needed to find and to manipulate the file.

Another component of the kernel consists of a collection of **device drivers,** which are the software units that communicate with the controllers (or at times, directly with peripheral devices) to carry out operations on the peripheral devices attached to the machine. Each device driver is uniquely designed for its particular type of device (such as a printer, disk drive, or monitor) and translates generic requests into the more technical steps required by the device assigned to that driver. For example, a device driver for a printer contains the software for reading and decoding that particular printer's status word as well as all the other handshaking details. Thus, other software components do not have to deal with those technicalities in order to print a file. Instead, the other components can merely rely on the device driver software to print the file, and let the device driver take care of the details. In this manner, the design of the other software units can be independent of the unique characteristics of particular devices. The result is a generic operating system that can be customized for particular peripheral devices by merely installing the appropriate device drivers.

Still another component of an operating system's kernel is the **memory manager,** which is charged with the task of coordinating the machine's use of main memory. Such duties are minimal in an environment in which a computer is asked to perform only one task at a time. In these cases, the program for performing the current task is placed at a predetermined location in main memory, executed, and then replaced by the program for performing the next task. However, in multiuser or multitasking environments in which the computer is asked to address

many needs at the same time, the duties of the memory manager are extensive. In these cases, many programs and blocks of data must reside in main memory concurrently. Thus, the memory manager must find and assign memory space for these needs and ensure that the actions of each program are restricted to the program's allotted space. Moreover, as the needs of different activities come and go, the memory manager must keep track of those memory areas no longer occupied.

The task of the memory manager is complicated further when the total main memory space required exceeds the space actually available in the computer. In this case the memory manager may create the illusion of additional memory space by rotating programs and data back and forth between main memory and mass storage (a technique called **paging**). Suppose, for example, that a main memory of 8GB is required but the computer only has 4GB. To create the illusion of the larger memory space, the memory manager reserves 4GB of storage space on a magnetic disk. There it records the bit patterns that would be stored in main memory if main memory had an actual capacity of 8GB. This data is divided into uniform sized units called **pages,** which are typically a few KB in size. Then the memory manager shuffles these pages back and forth between main memory and mass storage so that the pages that are needed at any given time are actually present in the 4GB of main memory. The result is that the computer is able to function as though it actually had 8GB of main memory. This large "fictional" memory space created by paging is called **virtual memory.**

Two additional components within the kernel of an operating system are the **scheduler** and **dispatcher,** which we will study in the next section. For now we merely note that in a multiprogramming system the scheduler determines which activities are to be considered for execution, and the dispatcher controls the allocation of time to these activities.

Getting It Started

We have seen that an operating system provides the software infrastructure required by other software units, but we have not considered how the operating system gets started. This is accomplished through a procedure known as

Firmware

In addition to the boot loader, a PC's ROM contains a collection of software routines for performing fundamental input/output activities such as receiving information from the keyboard, displaying messages on the computer screen, and reading data from mass storage. Being stored in nonvolatile memory such as FlashROM, this software is not immutably etched into the silicon of the machine—the hardware—but is also not as readily changeable as the rest of the programs in mass storage—the software. The term **firmware** was coined to describe this middle ground. Firmware routines can be used by the boot loader to perform I/O activities before the operating system becomes functional. For example, they are used to communicate with the computer user before the boot process actually begins and to report errors during booting. Widely used firmware systems include the BIOS (Basic Input/Output System) long used in PCs, the newer EFI (Extensible Firmware Interface), Sun's Open Firmware (now a product of Oracle), and the CFE (Common Firmware Environment) used in many embedded devices.

boot strapping (often shortened to **booting**) that is performed by a computer each time it is turned on. It is this procedure that transfers the operating system from mass storage (where it is permanently stored) into main memory (which is essentially empty when the machine is first turned on). To understand the boot strap process and the reason it is necessary, we begin by considering the machine's CPU.

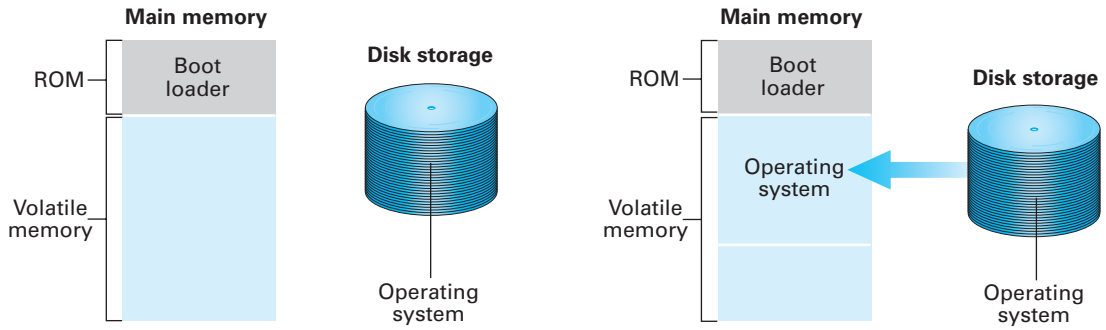
A CPU is designed so that its program counter starts with a particular predetermined address each time the CPU is turned on. It is at this location that the CPU expects to find the beginning of the program to be executed. Conceptually, then, all that is needed is to store the operating system at this location. However, for technical reasons, a computer's main memory is typically constructed from volatile technologies—meaning that the memory loses the data stored in it when the computer is turned off. Thus, the contents of main memory must be replenished each time the computer is restarted.

In short, we need a program (preferably the operating system) to be present in main memory when the computer is first turned on, but the computer's volatile memory is erased each time the machine is turned off. To resolve this dilemma, a small portion of a computer's main memory where the CPU expects to find its initial program is constructed from special nonvolatile memory cells. Such memory is known as **read-only memory (ROM)** because its contents can be read but not altered. As an analogy, you can think of storing bit patterns in ROM as blowing tiny fuses (some blown open—ones—and some blown closed—zeros), although the technology used is more advanced. More precisely, most ROM in today's PCs is constructed with flash memory technology (which means that it is not strictly ROM because it can be altered under special circumstances).

In a general-purpose computer, a program called the **boot loader** is permanently stored in the machine's ROM. This, then, is the program that is initially executed when the machine is turned on. The instructions in the boot loader direct the CPU to transfer the operating system from a predetermined location into the volatile area of main memory (Figure 3.5). Modern boot loaders can copy an operating system into main memory from a variety of locations. For example, in embedded systems, such as smartphones, the operating system is copied from special flash (nonvolatile) memory; in the case of small workstations at large companies or universities, the operating system may be copied from a distant machine over a network. Once the operating system has been placed in main memory, the boot loader directs the CPU to execute a jump instruction to that area of memory. At this point, the operating system takes over and begins controlling the machine's activities. The overall process of executing the boot loader and thus starting the operating system is called **booting** the computer.

You may ask why desktop computers are not provided with enough ROM to hold the entire operating system so that booting from mass storage would not be necessary. While this is feasible for embedded systems with small operating systems, devoting large blocks of main memory in general-purpose computers to nonvolatile storage is not efficient with today's technology. Moreover, computer operating systems undergo frequent updates in order to maintain security and keep abreast of new and improved device drivers for the latest hardware. While it is possible to update operating systems and boot loaders stored in ROM (often

Figure 3.5 The booting process



Step 1: Machine starts by executing the boot loader program already in memory. Operating system is stored in mass storage.

Step 2: Boot loader program directs the transfer of the operating system into main memory and then transfers control to it.

called a **firmware update**), the technological limits make mass storage the most common choice for more traditional computer systems.

In closing we should point out that understanding the boot process as well as the distinctions between an operating system, utility software, and application software allows us to comprehend the overall methodology under which most general-purpose computer systems operate. When such a machine is first turned on, the boot loader loads and activates the operating system. The user then makes requests to the operating system regarding the utility or application programs to be executed. As each utility or application is terminated, the user is put back in touch with the operating system, at which time the user can make additional requests. Learning to use such a system is therefore a two-layered process. In addition to learning the details of the specific utility or application desired, one must learn enough about the machine's operating system to navigate among the applications.

Questions & Exercises

1. List the components of a typical operating system and summarize the role of each in a single phrase.
2. What is the difference between application software and utility software?
3. What is virtual memory?
4. Summarize the booting procedure.

3.3 Coordinating the Machine's Activities

In this section we consider how an operating system coordinates the execution of application software, utility software, and units within the operating system itself. We begin with the concept of a process.

The Concept of a Process

One of the most fundamental concepts of modern operating systems is the distinction between a program and the activity of executing a program. The former is a static set of directions, whereas the latter is a dynamic activity whose properties change as time progresses. (This distinction is analogous to a piece of sheet music, sitting inert in a book on the shelf, versus a musician performing that piece by taking actions that the sheet music describes.) The activity of executing a program under the control of the operating system is known as a **process.** Associated with a process is the current status of the activity, called the **process state.** This state includes the current position in the program being executed (the value of the program counter) as well as the values in the other CPU registers and the associated memory cells. Roughly speaking, the process state is a snapshot of the machine at a particular time. At different times during the execution of a program (at different times in a process) different snapshots (different process states) will be observed.

Unlike a musician, who normally tries to play only one musical piece at a time, typical time-sharing/multitasking computers are running many processes, all competing for the computer's resources. It is the task of the operating system to manage these processes so that each process has the resources (peripheral devices, space in main memory, access to files, and access to a CPU) that it needs, that independent processes do not interfere with one another, and that processes that need to exchange information are able to do so.

Process Administration

The tasks associated with coordinating the execution of processes are handled by the scheduler and dispatcher within the operating system's kernel. The scheduler maintains a record of the processes present in the computer system, introduces new processes to this pool, and removes completed processes from the pool. Thus when a user requests the execution of an application, it is the scheduler that adds the execution of that application to the pool of current processes.

To keep track of all the processes, the scheduler maintains a block of information in main memory called the **process table.** Each time the execution of a program is requested, the scheduler creates a new entry for that process in the process table. This entry contains such information as the memory area assigned to the process (obtained from the memory manager), the priority of the process, and whether the process is ready or waiting. A process is **ready** if it is in a state in which its progress can continue; it is **waiting** if its progress is currently delayed until some external event occurs, such as the completion of a mass storage operation, the pressing of a key at the keyboard, or the arrival of a message from another process.

The dispatcher is the component of the kernel that oversees the execution of the scheduled processes. In a time-sharing/multitasking system this task is

accomplished by **multiprogramming**; that is, dividing time into short segments, each called a **time slice** (typically measured in milliseconds or microseconds), and then switching the CPU's attention among the processes as each is allowed to execute for one time slice (Figure 3.6). The procedure of changing from one process to another is called a **process switch** (or a **context switch**).

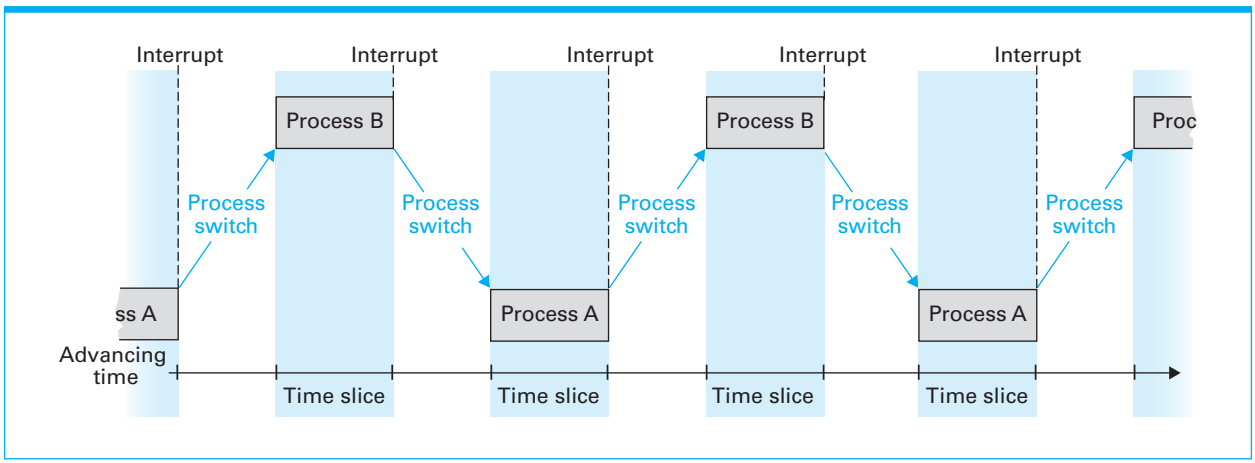
Each time the dispatcher awards a time slice to a process, it initiates a timer circuit that will indicate the end of the slice by generating a signal called an **interrupt.** The CPU reacts to this interrupt signal in much the same way that you react when interrupted from a task. You stop what you are doing, record where you are in the task (so that you will be able to return at a later time), and take care of the interrupting entity. When the CPU receives an interrupt signal, it completes its current machine cycle, saves its position in the current process, and begins executing a program, called an **interrupt handler,** which is stored at a predetermined location in main memory. This interrupt handler is a part of the dispatcher, and it describes how the dispatcher should respond to the interrupt signal.

Thus, the effect of the interrupt signal is to preempt the current process and transfer control back to the dispatcher. At this point, the dispatcher selects the process from the process table that has the highest priority among the ready processes (as determined by the scheduler), restarts the timer circuit, and allows the selected process to begin its time slice.

Paramount to the success of a multiprogramming system is the ability to stop, and later restart, a process. If you are interrupted while reading a book, your ability to continue reading at a later time depends on your ability to remember your location in the book as well as the information that you had accumulated to that point. In short, you must be able to re-create the environment that was present immediately prior to the interruption.

In the case of a process, the environment that must be re-created is the process's state, which as already mentioned, includes the value of the program counter as well as the contents of the registers and pertinent memory cells. CPUs designed for multiprogramming systems incorporate the task of saving this information as part of the CPU's reaction to the interrupt signal. These CPUs

Figure 3.6 Multiprogramming between process A and process B



Interrupts

The use of interrupts for terminating time slices, as described in the text, is only one of many applications of a computer's interrupt system. There are many situations in which an interrupt signal is generated, each with its own interrupt routine. Indeed, interrupts provide an important tool for coordinating a computer's actions with its environment. For example, both clicking a mouse and pressing a key on the keyboard generate interrupt signals that cause the CPU to set aside its current activity and address the cause of the interrupt.

To manage the task of recognizing and responding to incoming interrupts, the various interrupt signals are assigned priorities so that the more important tasks can be taken care of first. The highest priority interrupt is usually associated with a power failure. Such an interrupt signal is generated if the computer's power is unexpectedly disrupted. The associated interrupt routine directs the CPU through a series of "housekeeping" chores during the milliseconds before the voltage level drops below an operational level.

also tend to have machine-language instructions for reloading a previously saved state. Such features simplify the task of the dispatcher when performing a process switch and exemplify how the design of modern CPUs is influenced by the needs of today's operating systems.

In closing, we should note that the use of multiprogramming has been found to increase the overall efficiency of a machine. This is somewhat counterintuitive since the shuffling of processes required by multiprogramming introduces an overhead. However, without multiprogramming each process runs to completion before the next process begins, meaning that the time that a process is waiting for peripheral devices to complete tasks or for a user to make the next request is wasted. Multiprogramming allows this lost time to be given to another process. For example, if a process executes an I/O request, such as a request to retrieve data from a magnetic disk, the scheduler will update the process table to reflect that the process is waiting for an external event. In turn, the dispatcher will cease to award time slices to that process. Later (perhaps several hundred milliseconds), when the I/O request has been completed, the scheduler will update the process table to show that the process is ready, and thus that process will again compete for time slices. In short, progress on other tasks will be made while the I/O request is being performed, and thus the entire collection of tasks will be completed in less time than if executed in a sequential manner.

Questions & Exercises

1. Summarize the difference between a program and a process.
2. Summarize the steps performed by the CPU when an interrupt occurs.
3. In a multiprogramming system, how can high-priority processes be allowed to run faster than others?

4. If each time slice in a multiprogramming system is 50 milliseconds and each context switch requires at most a microsecond, how many processes can the machine service in a single second?
5. If each process uses its complete time slice in the machine in question 4, what fraction of the machine's time is spent actually performing processes? What would this fraction be if each process executed an I/O request after only a microsecond of its time slice?

3.4 Handling Competition Among Processes

An important task of an operating system is the allocation of the machine's resources to the processes in the system. Here we are using the term *resource* in a broad sense, including the machine's peripheral devices as well as features within the machine itself. The file manager allocates access to files as well and allocates mass storage space for the construction of new files; the memory manager allocates memory space; the scheduler allocates space in the process table; and the dispatcher allocates time slices. As with many problems in computer systems, this allocation task may appear simple at first glance. Below the surface, however, lie several subtleties that can lead to malfunctions in a poorly designed system. Remember, a machine does not think for itself; it merely follows directions. Thus, to construct reliable operating systems, we must develop algorithms that cover every possible contingency, regardless of how minuscule it may appear.

Semaphores

Let us consider a time-sharing/multitasking operating system controlling the activities of a computer with a single printer. If a process needs to print its results, it must request that the operating system give it access to the printer's device driver. At this point, the operating system must decide whether to grant this request, depending on whether the printer is already being used by another process. If it is not, the operating system should grant the request and allow the process to continue; otherwise, the operating system should deny the request and

Microsoft's Task Manager

You can gain insight to some of the internal activity of a Microsoft Windows operating system by executing the utility program called Task Manager. (Press the Ctrl, Alt, and Delete keys simultaneously.) In particular, by selecting the Processes tab in the Task Manager window, you can view the process table. Here is an experiment you can perform: Look at the process table before you activate any application program. (You may be surprised that so many processes are already in the table. These are necessary for the system's basic operation.) Now activate an application and confirm that an additional process has entered the table. You will also be able to see how much memory space was allocated to the process.

perhaps classify the process as a waiting process until the printer becomes available. After all, if two processes were given simultaneous access to the computer's printer, the results would be worthless to both.

To control access to the printer, the operating system must keep track of whether the printer has been allocated. One approach to this task would be to use a flag, which in this context refers to a bit in memory whose states are often referred to as *set* and *clear*, rather than 1 and 0. A clear flag (value 0) indicates that the printer is available and a set flag (value 1) indicates that the printer is currently allocated. On the surface, this approach seems well-founded. The operating system merely checks the flag each time a request for printer access is made. If it is clear, the request is granted and the operating system sets the flag. If the flag is set, the operating system makes the requesting process wait. Each time a process finishes with the printer, the operating system either allocates the printer to a waiting process or, if no process is waiting, merely clears the flag.

However, this simple flag system has a problem. The task of testing and possibly setting the flag may require several machine instructions. (The value of the flag must be retrieved from main memory, manipulated within the CPU, and finally stored back in memory.) It is therefore possible for a task to be interrupted after a clear flag has been detected but before the flag has been set. In particular, suppose the printer is currently available, and a process requests use of it. The flag is retrieved from main memory and found to be clear, indicating that the printer is available. However, at this point, the process is interrupted and another process begins its time slice. It too requests the use of the printer. Again, the flag is retrieved from main memory and found still clear because the previous process was interrupted before the operating system had time to set the flag in main memory. Consequently, the operating system allows the second process to begin using the printer. Later, the original process resumes execution where it left off, which is immediately after the operating system found the flag to be clear. Thus the operating system continues by setting the flag in main memory and granting the original process access to the printer. Two processes are now using the same printer.

The solution to this problem is to insist that the task of testing and possibly setting the flag be completed without interruption. One approach is to use the interrupt disable and interrupt enable instructions provided in most machine languages. When executed, an interrupt disable instruction causes future interrupts to be blocked, whereas an interrupt enable instruction causes the CPU to resume responding to interrupt signals. Thus, if the operating system starts the flag-testing routine with a disable interrupt instruction and ends it with an enable interrupt instruction, no other activity can interrupt the routine once it starts.

Another approach is to use the **test-and-set** instruction that is available in many machine languages. This instruction directs the CPU to retrieve the value of a flag, note the value received, and then set the flag—all within a single machine instruction. The advantage here is that because the CPU always completes an instruction before recognizing an interrupt, the task of testing and setting the flag cannot be split when it is implemented as a single instruction.

A properly implemented flag, as just described, is called a **semaphore,** in reference to the railroad signals used to control access to sections of track. In fact, semaphores are used in software systems in much the same way as they are in railway systems. Corresponding to the section of track that can contain only

one train at a time is a sequence of instructions that should be executed by only one process at a time. Such a sequence of instructions is called a **critical region.** The requirement that only one process at a time be allowed to execute a critical region is known as **mutual exclusion.** In summary, a common way of obtaining mutual exclusion to a critical region is to guard the critical region with a semaphore. To enter the critical region, a process must find the semaphore clear and then set the semaphore before entering the critical region; then upon exiting the critical region, the process must clear the semaphore. If the semaphore is found in its set state, the process trying to enter the critical region must wait until the semaphore has been cleared.

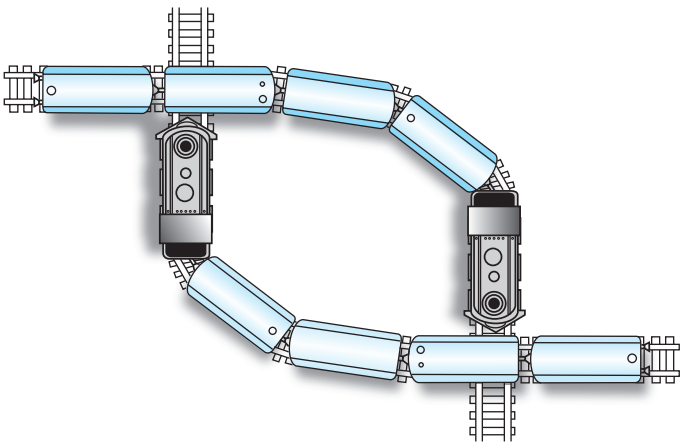
Deadlock

Another problem that can arise during resource allocation is **deadlock,** the condition in which two or more processes are blocked from progressing because each is waiting for a resource that is allocated to another. For example, one process may have access to the computer's printer but be waiting for access to the computer's CD player, while another process has access to the CD player but is waiting for the printer. Another example occurs in systems in which processes are allowed to create new processes (an action called **forking** in the UNIX vernacular) to perform subtasks. If the scheduler has no space left in the process table and each process in the system must create an additional process before it can complete its task, then no process can continue. Such conditions, as in other settings (Figure 3.7), can severely degrade a system's performance.

Analysis of deadlock has revealed that it cannot occur unless all three of the following conditions are satisfied:

1. There is competition for nonsharable resources.
2. The resources are requested on a partial basis; that is, having received some resources, a process will return later to request more.
3. Once a resource has been allocated, it cannot be forcibly retrieved.

Figure 3.7 A deadlock resulting from competition for nonsharable railroad intersections



Python and Operating Systems

When a Python script is executed by a user, the operating system launches a new process to run the script. Such scripts are often applications, although they can also be considered utility software if they extend or customize the capabilities of the system. Python scripts interact with components of the operating system to accomplish their work, such as the file manager for reading and writing files, or the GUI or shell for providing user interactions. The Python module "os" provides a variety of predefined, system-agnostic functions for accessing common operating system features, such as forking new Python processes, or executing other utility or application programs.

The point of isolating these conditions is that the deadlock problem can be removed by attacking any one of the three. Techniques that attack the third condition fall into the category known as deadlock detection and correction schemes. In these cases, the occurrence of deadlock is considered so remote that no effort is made to avoid the problem. Instead, the approach is to detect it should it occur and then correct it by forcibly retrieving some of the allocated resources. Our example of a full process table might fall in this class. If deadlock should occur due to a full table, routines within the operating system (or perhaps a human administrator using his or her powers as "super user") can remove (the technical term is **kill**) some of the processes. This releases space in the process table, breaking the deadlock and allowing the remaining processes to continue their tasks.

Techniques that attack the first two conditions are known as deadlock avoidance schemes. One, for example, attacks the second condition by requiring each process to request all its resources at one time. Another scheme attacks the first condition, not by removing the competition directly but by converting nonsharable resources into sharable ones. For example, suppose the resource in question is a printer and a variety of processes require its use. Each time a process requests the printer, the operating system could grant the request. However, instead of connecting the process to the printer's device driver, the operating system would connect it to a device driver that stores the information to be printed in mass storage rather than sending it to the printer. Thus each process, thinking it has access to the printer, could execute in its normal way. Later, when the printer is available, the operating system could transfer the data from mass storage to the printer. In this manner, the operating system would make the nonsharable resource appear sharable by creating the illusion of more than one printer. This technique of holding data for output at a later but more convenient time is called **spooling.**

We have introduced spooling as a technique for granting several processes access to a common resource—a theme that has many variations. For example, a file manager could grant several processes access to the same file if the processes are merely reading data from the file, but conflicts can occur if more than one process tries to alter a file at the same time. Thus, a file manager may allocate file access according to the needs of the processes, allowing several processes to have read access but allowing only one to have write access. Other systems may divide the file into pieces so that different processes can alter different parts of the file concurrently. Each of these techniques, however, has subtleties that must be resolved to obtain a reliable system. How, for example, should those processes with only read access to a file be notified when a process with write access alters the file?

Multi-Core Operating Systems

Traditional time-sharing/multitasking systems give the illusion of executing many processes at once by switching rapidly between time slices faster than a human can perceive. Modern systems continue to multitask in this way, but in addition, the latest multi-core CPUs are genuinely capable of running two, four, or many more processes simultaneously. Unlike a group of single-core computers working together, a multi-core machine contains multiple independent processors (in this case called cores) that share the computer's peripherals, memory, and other resources. For a multi-core operating system, this means that the dispatcher and scheduler must consider which processes to execute on each core. With different processes running on different cores, handling competition among processes becomes more challenging because disabling interrupts on all cores whenever one needs to enter a critical region would be highly inefficient. Computer science has many active research areas related to building operating system mechanisms better suited to the new multi-core world.

Questions & Exercises

1. Suppose process A and process B are sharing time on the same machine, and each needs the same nonsharable resource for short periods of time. (For example, each process may be printing a series of independent, short reports.) Each process may then repeatedly acquire the resource, release it, and later request it again. What is a drawback to controlling access to the resource in the following manner:

   > Begin by assigning a flag the value 0. If process A requests the resource and the flag is 0, grant the request. Otherwise, make process A wait. If process B requests the resource and the flag is 1, grant the request. Otherwise, make process B wait. Each time process A finishes with the resource, change the flag to 1. Each time process B finishes with the resource, change the flag to 0.

2. Suppose a two-lane road converges to one lane to pass through a tunnel. To coordinate the use of the tunnel, the following signal system has been installed:

   > A car entering either end of the tunnel causes red lights above the tunnel entrances to be turned on. As the car exits the tunnel, the lights are turned off. If an approaching car finds a red light on, it waits until the light is turned off before entering the tunnel.

   What is the flaw in this system?

3. Suppose the following solutions have been proposed for removing the deadlock that occurs on a single-lane bridge when two cars meet. Identify which condition for deadlock given in the text is removed by each solution.

   a. Do not let a car onto the bridge until the bridge is empty.
   b. If cars meet, make one of them back up.
   c. Add a second lane to the bridge.

4. Suppose we represent each process in a multiprogramming system with a dot and draw an arrow from one dot to another if the process represented by the first dot is waiting for a (nonsharable) resource being used by the second. Mathematicians call the resulting picture a **directed graph.** What property of the directed graph is equivalent to deadlock in the system?

3.5 Security

Since the operating system oversees the activities in a computer, it is natural for it to play a vital role in maintaining security as well. In the broad sense, this responsibility manifests itself in multiple forms, one of which is reliability. If a flaw in the file manager causes the loss of part of a file, then the file was not secure. If a defect in the dispatcher leads to a system failure (often called a system crash) causing the loss of an hour's worth of typing, we would argue that our work was not secure. Thus the security of a computer system requires a well-designed, dependable operating system.

The development of reliable software is not a subject that is restricted to operating systems. It permeates the entire software development spectrum and constitutes the field of computer science known as software engineering, which we will study in Chapter 7. In this section, then, we focus on security problems that are more closely related to the specifics of operating systems.

Attacks from the Outside

An important task performed by operating systems is to protect the computer's resources from access by unauthorized personnel. In the case of computers used by multiple people, this is usually approached by means of establishing "accounts" for the various authorized users—an account being essentially a record within the operating system containing such entries as the user's name, password, and privileges to be granted to that user. The operating system can then use this information during each **login** procedure (a sequence of transactions in which the user establishes initial contact with a computer's operating system) to control access to the system.

Accounts are established by a person known as the **super user** or the **administrator.** This person gains highly privileged access to the operating system by identifying him- or herself as the administrator (usually by name and password) during the login procedure. Once this contact is established, the administrator can alter settings within the operating system, modify critical software packages, adjust the privileges granted to other users, and perform a variety of other maintenance activities that are denied normal users.

From this "lofty perch," the administrator is also able to monitor activity within the computer system in an effort to detect destructive behavior, whether malicious or accidental. To assist in this regard, numerous software utilities, called **auditing software,** have been developed that record and then analyze the activities taking place within the computer system. In particular, auditing software may expose a flood of attempts to login using incorrect passwords, indicating that an unauthorized user may be trying to gain access to the computer.

Auditing software may also identify activities within a user's account that do not conform to that user's past behavior, which may indicate that an unauthorized user has gained access to that account. (It is unlikely that a user who traditionally uses only word processing and spreadsheet software will suddenly begin to access highly technical software applications or try to execute utility packages that lie outside that user's privileges.)

Another culprit that auditing systems are designed to detect is the presence of **sniffing software**, which is software that, when left running on a computer, records activities and later reports them to a would-be intruder. An old, well-known example is a program that simulates the operating system's login procedure. Such a program can be used to trick authorized users into thinking they are communicating with the operating system, whereas they are actually supplying their names and passwords to an impostor.

With all the technical complexities associated with computer security, it is surprising to many that one of the major obstacles to the security of computer systems is the carelessness of the users themselves. They select passwords that are relatively easy to guess (such as names and dates), they share their passwords with friends, they fail to change their passwords on a timely basis, they subject offline mass storage devices to potential degradation by transferring them back and forth between machines, and they import unapproved software into the system that might subvert the system's security. For problems like these, most institutions with large computer installations adopt and enforce policies that catalog the requirements and responsibilities of the users.

Attacks from Within

Once an intruder (or perhaps an authorized user with malicious intent) gains access to a computer system, the next step is usually to explore, looking for information of interest or for places to insert destructive software. This is a straightforward process if the prowler has gained access to the administrator's account, which is why the administrator's password is closely guarded. If, however, access is through a general user's account, it becomes necessary to trick the operating system into allowing the intruder to reach beyond the privileges granted to that user. For example, the intruder may try to trick the memory manager into allowing a process to access main memory cells outside its allotted area, or the prowler may try to trick the file manager into retrieving files whose access should be denied.

Today's CPUs are enhanced with features that are designed to foil such attempts. As an example, consider the need to restrict a process to the area of main memory assigned to it by the memory manager. Without such restrictions, a process could erase the operating system from main memory and take control of the computer itself. To counter such attempts, CPUs designed for multiprogramming systems typically contain special-purpose registers in which the operating system can store the upper and lower limits of a process's allotted memory area. Then, while performing the process, the CPU compares each memory reference to these registers to ensure that the reference is within the designated limits. If the reference is found to be outside the process's designated area, the CPU automatically transfers control back to the operating system (by performing an interrupt sequence) so that the operating system can take appropriate action.

Embedded in this illustration is a subtle but significant problem. Without further security features, a process could still gain access to memory cells outside of

its designated area merely by changing the special-purpose registers that contain its memory limits. That is, a process that wanted access to additional memory could merely increase the value in the register containing the upper memory limit and then proceed to use the additional memory space without approval from the operating system.

To protect against such actions, CPUs for multiprogramming systems are designed to operate in one of two **privilege levels**; we will call one "privileged mode," the other we will call "nonprivileged mode." When in privileged mode, the CPU is able to execute all the instructions in its machine language. However, when in nonprivileged mode, the list of acceptable instructions is limited. The instructions that are available only in privileged mode are called **privileged instructions.** (Typical examples of privileged instructions include instructions that change the contents of memory limit registers and instructions that change the current privilege mode of the CPU.) An attempt to execute a privileged instruction when the CPU is in nonprivileged mode causes an interrupt. This interrupt converts the CPU to privileged mode and transfers control to an interrupt handler within the operating system.

When first turned on, the CPU is in privileged mode. Thus, when the operating system starts at the end of the boot process, all instructions are executable. However, each time the operating system allows a process to start a time slice, it switches the CPU to nonprivileged mode by executing a "change privilege mode" instruction. In turn, the operating system will be notified if the process attempts to execute a privileged instruction, and thus the operating system will be in position to maintain the integrity of the computer system.

Privileged instructions and the control of privilege levels is the major tool available to operating systems for maintaining security. However, the use of these tools is a complex component of an operating system's design, and errors continue to be found in current systems. A single flaw in privilege level control can open the door to disaster from malicious programmers or from inadvertent programming errors. If a process is allowed to alter the timer that controls the system's multiprogramming system, that process can extend its time slice and dominate the machine. If a process is allowed to access peripheral devices directly, then it can read files without supervision by the system's file manager. If a process is allowed to access memory cells outside its allotted area, it can read and even alter data being used by other processes. Thus, maintaining security continues to be an important task of an administrator as well as a goal in operating system design.

Questions & Exercises

1. Give some examples of poor choices for passwords and explain why they would be poor choices.
2. Processors in Intel's Pentium series provide for four privilege levels. Why would the designers of CPUs decide to provide four levels rather than three or five?
3. If a process in a multiprogramming system could access memory cells outside its allotted area, how could it gain control of the machine?

Chapter Review Problems

(Asterisked problems are associated with optional sections.)

1. List four activities of a typical operating system.
2. What led to the development of the interactive operating system?
3. Suppose three items R, S, and T are placed in a queue in that order. Then one item is removed from the queue before a fourth item, X, is placed in the queue. Then one item is removed from the queue, the items Y and Z are placed in the queue, and then the queue is emptied by removing one item at a time. List all the items in the order in which they were removed.
4. What is the significance of the window manager in current GUI shells?
5. What is a real-time operating system?
6. If you have a PC, identify some situations in which you can take advantage of its multitasking capabilities.
7. On the basis of a computer system with which you are familiar, identify two units of application software and two units of utility software. Then explain why you classified them as you did.
8. a. What is the role of the user interface of an operating system?
   b. What is the role of the kernel of an operating system?
9. What is the use of the process table in program execution?
10. Give some examples of functions that are provided by the Python module "os".
11. What is the Linux or UNIX command for creating new processes?
12. What is the difference between a process that is ready and a process that is waiting?
13. What is the difference between virtual memory and main memory?
14. Suppose a computer contained 512MB (MiB) of main memory, and an operating system needed to create a virtual memory of twice that size using pages of 2KB (KiB). How many pages would be required?
15. What complications could arise in a time-sharing/multitasking system if two processes require access to the same file at the same time? Are there cases in which the file manager should grant such requests? Are there cases in which the file manager should deny such requests?
16. How is firmware different from hardware and software? What do you mean by a firmware update?
17. Define load balancing and scaling in the context of multiprocessor architectures.
18. What is a context switch?
19. What are the flaws of privilege level control?
20. If you have a PC, record the sequence activities that you can observe when you turn it on. Then determine what messages appear on the computer screen before the booting process actually begins. What software writes these messages?
21. Suppose a multiprogramming operating system allocated time slices of 10 milliseconds and the machine executed an average of five instructions per nanosecond. How many instructions could be executed in a single time slice?
22. If a typist types 60 words per minute (where a word is considered five characters), how much time would pass between typing each character? If a multiprogramming operating system allocated time slices in 10 millisecond units and we ignore the time required for process switches, how many time slices could be allocated between characters being typed?
23. Suppose a multiprogramming operating system is allotting time slices of 50 milliseconds. If it normally takes 8 milliseconds to position a disk's read/write head over the desired track and another 17 milliseconds for the desired data to rotate around to the read/write head, how much of a program's time slice can be spent waiting for a read operation from a disk to take place? If the machine is capable of executing 10 instructions each nanosecond,

how many instructions can be executed during this waiting period? (This is why when a process performs an operation with a peripheral device, a multiprogramming system terminates that process's time slice and allows another process to run while the first process is waiting for the services of the peripheral device.)

24. What is the drawback of using the set and clear flag system while allocating devices?

25. A process is said to be I/O-bound if it requires a lot of I/O operations, whereas a process that consists of mostly computations within the CPU/memory system is said to be compute-bound. If both a compute-bound process and an I/O-bound process are waiting for a time slice, which should be given priority? Why?

26. Would greater throughput be achieved by a system running two processes in a multiprogramming environment if both processes were I/O-bound (refer to problem 25) or if one were I/O-bound and the other were compute-bound? Why?

27. Write a set of directions that tells an operating system's dispatcher what to do when a process's time slice is over.

28. What are the various functions of the memory manager in an operating system?

29. Identify a situation in a multiprogramming system in which a process does not consume the entire time slice allocated to it.

30. What is meant by an interrupt handler in multiprogramming systems and what is its significance?

31. Answer each of the following in terms of an operating system that you use:
a. How do you ask the operating system to copy a file from one location to another?
b. How do you ask the operating system to show you the directory on a disk?
c. How do you ask the operating system to execute a program?

32. Answer each of the following in terms of an operating system that you use:
a. How does the operating system restrict access to only those who are approved users?
b. How do you ask the operating system to show you what processes are currently in the process table?
c. How do you tell the operating system that you do not want other users of the machine to have access to your files?

***33.** Explain an important use for the test-and-set instruction found in many machine languages. Why is it important for the entire test-and-set process to be implemented as a single instruction?

***34.** A banker with only $100,000 loans $50,000 to each of two customers. Later, both customers return with the story that before they can repay their loans they must each borrow another $10,000 to complete the business deals in which their previous loans are involved. The banker resolves this deadlock by borrowing the additional funds from another source and passing on this loan (with an increase in the interest rate) to the two customers. Which of the three conditions for deadlock has the banker removed?

***35.** Students who want to enroll in Model Railroading II at the local university are required to obtain permission from the instructor and pay a laboratory fee. The two requirements are fulfilled independently in either order and at different locations on campus. Enrollment is limited to 20 students; this limit is maintained by both the instructor, who will grant permission to only 20 students, and the financial office, which will allow only 20 students to pay the laboratory fee. Suppose that this registration system has resulted in 19 students having successfully registered for the course, but with the final space being claimed by two students—one who has only obtained permission from the instructor and another who has only paid the fee. Which requirement for deadlock is removed by each of the following solutions to the problem?
a. Both students are allowed in the course.
b. The class size is reduced to 19, so neither of the two students is allowed to register for the course.
c. The competing students are both denied entry to the class and a third student is given the twentieth space.
d. It is decided that the only requirement for entry into the course is the payment of the fee. Thus the student who has paid the fee gets into the course, and entry is denied to the other student.

*36. Since each area on a computer's display can be used by only one process at a time (otherwise the image on the screen would be unreadable), these areas are nonsharable resources that are allocated by the window manager. Which of the three conditions necessary for deadlock does the window manager remove in order to avoid deadlock?

*37. Suppose each nonsharable resource in a computer system is classified as a level 1, level 2, or level 3 resource. Moreover, suppose each process in the system is required to request the resources it needs according to this classification. That is, it must request all the required level 1 resources at once before requesting any level 2 resources. Once it receives the level 1 resources, it can request all the required level 2 resources, and so on. Can deadlock occur in such a system? Why or why not?

*38. Each of two robot arms is programmed to lift assemblies from a conveyor belt, test them for tolerances, and place them in one of two bins depending on the results of the test. The assemblies arrive one at a time with a sufficient interval between them. To keep both arms from trying to grab the same assembly, the computers controlling the arms share a common memory cell. If an arm is available as an assembly approaches, its controlling computer reads the value of the common cell. If the value is nonzero, the arm lets the assembly pass. Otherwise, the controlling computer places a nonzero value in the memory cell, directs the arm to pick up the assembly, and places the value 0 back into the memory cell after the action is complete. What sequence of events could lead to a tug-of-war between the two arms?

*39. Why is disabling the interrupts in a multicore operating system not considered to be an efficient approach?

*40. A process that is waiting for a time slice is said to suffer **starvation** if it is never given a time slice.

a. The pavement in the middle of an intersection can be considered as a nonsharable resource for which cars approaching the intersection compete. A traffic light rather than an operating system is used to control the allocation of the resource. If the light is able to sense the amount of traffic arriving from each direction and is programmed to give the green light to the heavier traffic, the lighter traffic might suffer from starvation. How is starvation avoided?

b. In what sense can a process starve if the dispatcher always assigns time slices according to a priority system in which the priority of each process remains fixed? (Hint: What is the priority of the process that just completed its time slice in comparison to the processes that are waiting, and consequently which routine gets the next time slice?) How, would you guess, do many operating systems avoid this problem?

*41. Why can't special-purpose registers restrict a process in its allotted memory area? Explain your answer.

*42. The following is the "dining philosophers" problem that was originally proposed by E. W. Dijkstra and is now a part of computer science folklore. Five philosophers are sitting at a round table. In front of each is a plate of spaghetti. There are five forks on the table, one between each plate. Each philosopher wants to alternate between thinking and eating. To eat, a philosopher requires possession of both the forks that are adjacent to the philosopher's plate. Identify the possibilities of deadlock and starvation (see problem 40) that are present in the dining philosophers' problem.

*43. What problem arises as the lengths of the time slices in a multiprogramming system are made shorter and shorter? What about as they become longer and longer?

*44. When is it preferable to use the deadlock prevention scheme, the deadlock avoidance scheme, and the deadlock detection and recovery scheme? Can you provide scenarios where more than one of these schemes can be used?

45. Identity two activities that can be performed by an operating system's administrator but not by a typical user.

46. How is the read action different from the write action when multiple processes access the same file?

47. Suppose a password consisted of a string of nine characters from the English alphabet

(26 characters). If each possible password could be tested in a millisecond, how long would it take to test all possible passwords?

48. Why are CPUs that are designed for multitasking operating systems capable of operating at different privilege levels?
49. How are privileged instructions handled in nonprivileged mode?
50. Identify three ways in which a process could challenge the security of a computer system if not prevented from doing so by the operating system.
51. What are the conditions that lead to a deadlock?
52. What are the policies that a user should follow to manage login passwords?
53. Give appropriate examples of auditing software and sniffing software.
54. How is security relevant in current operating systems?
55. How is the booting process different in embedded systems from traditional systems?

Social Issues

The following questions are intended as a guide to the ethical/social/legal issues associated with the field of computing. The goal is not merely to answer these questions. You should also consider why you answered as you did and whether your justifications are consistent from one question to the next.

1. Suppose you are using a multiuser operating system that allows you to view the names of the files belonging to other users as well as to view the contents of those files that are not otherwise protected. Would viewing such information without permission be similar to wandering through someone's unlocked home without permission, or would it be more like reading materials placed in a common lounge such as a physician's waiting room?
2. When you have access to a multiuser computer system, what responsibilities do you have when selecting your password?
3. If a flaw in an operating system's security allows a malicious programmer to gain unauthorized access to sensitive data, to what extent should the developer of the operating system be held responsible?
4. Is it your responsibility to lock your house in such a way that intruders cannot get in, or is it the public's responsibility to stay out of your house unless invited? Is it the responsibility of an operating system to guard access to a computer and its contents, or is it the responsibility of hackers to leave the machine alone?
5. In *Walden,* Henry David Thoreau argues that we have become tools of our tools; that is, instead of benefiting from the tools that we have, we spend our time obtaining and maintaining our tools. To what extent is this true with regard to computing? For example, if you own a personal computer, how much time do you spend earning the money to pay for it, learning how to use its operating system, learning how to use its utility and application software, maintaining it, and downloading upgrades to its software in comparison to the amount of time you spend benefiting from it? When you use it, is your time well spent? Are you more socially active with or without a personal computer?

Additional Reading

Bishop, M. *Introduction to Computer Security.* Boston, MA: Addison-Wesley, 2005.

Craig, B. *Cyberlaw: The Law of the Internet and Information Technology.* Upper Saddle River, NJ: Prentice-Hall, 2012.

Davis, W. S., and T. M. Rajkumar. *Operating Systems: A Systematic View,* 6th ed. Boston, MA: Addison-Wesley, 2005.

Deitel, H. M., P. J. Deitel, and D. R. Choffnes. *Operating Systems,* 3rd ed. Upper Saddle River, NJ: Prentice-Hall, 2005.

Silberschatz, A., P. B. Galvin, and G. Gagne. *Operating System Concepts,* 9th ed., New York: Wiley, 2012.

Stallings, W. *Operating Systems,* 8th ed. Upper Saddle River, NJ: Prentice-Hall, 2014.

Tanenbaum, A. S. *Modern Operating Systems,* 3rd ed. Upper Saddle River, NJ: Prentice-Hall, 2008.