This page intentionally left blank

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.

Before we can explore the details of computer system operation, we need to know something about system structure. We begin by discussing the basic functions of system startup, I/O, and storage. We also describe the basic computer architecture that makes it possible to write a functional operating system.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. In this chapter, we provide a general overview of the major components of an operating system.

CHAPTER OBJECTIVES

- To provide a grand tour of the major components of operating systems.
- To describe the basic organization of computer systems.

1.1 What Operating Systems Do

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into
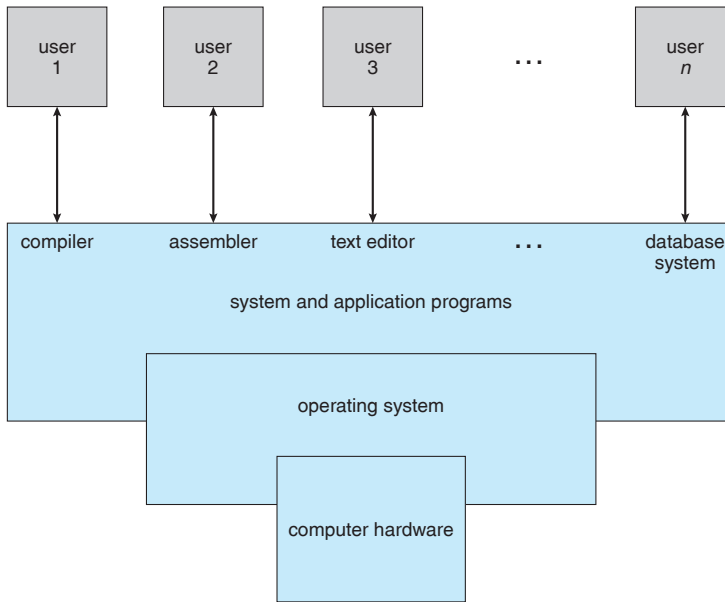
Figure 1.1 Abstract view of the components of a computer system.

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 the hardware and coordinates its use among the various application programs for the various users.

We can also view 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 system's role, we next explore operating systems from two viewpoints: that of the user and that of the 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 such systems

are optimized for the single-user experience rather than the requirements of multiple users.

In other cases, a user sits at a terminal connected to a **mainframe** or a **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. Most of these devices are standalone units for individual users. Some are connected to networks, either directly by wire or (more often) through wireless modems and networking. Because of power, speed, and interface limitations, they perform relatively few remote operations. Their operating systems are designed mostly for individual usability, but performance per unit 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 they and their operating systems are designed primarily 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 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

> **STORAGE DEFINITIONS AND NOTATION**
>
> A **bit** is the basic unit of computer storage. It can contain one of two values, zero and one. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native storage unit. A word is generally made up of one or more bytes. For example, a computer may have instructions to move 64-bit (8-byte) words.
>
> A kilobyte, or KB, is 1,024 bytes; a megabyte, or MB, is $1,024^2$ bytes; and a gigabyte, or GB, is $1,024^3$ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes.

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 more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer—usually called the **kernel**. (Along with the kernel, there are two other types of programs: **systems programs**, which are associated with the operating system but are not part of the kernel, and **application programs**, which include all programs not associated with the operation of the system.)

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 systems. As a result, Microsoft was found guilty of using its operating-system monopoly to limit competition.

1.2   Computer-System Organization

Before we can explore the details of how computer systems operate, we need general knowledge of the structure of a computer system. In this section, we look at several parts of this structure. The section is mostly concerned

THE STUDY OF OPERATING SYSTEMS

There has never been a more interesting time to study operating systems, and it has never been easier to do so. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. This list includes Linux, BSD UNIX, Solaris, and part of Mac OS X. The availability of source code allows us to study operating systems from the inside out. Questions that previously could be answered only by looking at documentation or the behavior of an operating system can now be answered by examining the code itself.

In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (`http://www.vmware.com`) provides a free "player" on which hundreds of free "virtual appliances" can run. Using this method, students can try out hundreds of operating systems within their existing operating systems at no cost.

Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but not complete list of open-source operating-system projects is available from `http://dmoz.org/Computers/Software/Operating_Systems/Open_Source/`. Simulators of specific hardware are also available in some cases, allowing the operating system to run on "native" hardware, all within the confines of a modern computer and modern operating system. For example, a DECSYSTEM-20 simulator running on Mac OS X can boot TOPS-20, load the source tapes, and modify and compile a new TOPS-20 kernel. An interested student can search the Internet to find the original papers that describe the operating system and the original manuals.

The advent of open-source operating systems also makes it easy to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution! Just a few years, ago it was difficult or impossible to get access to source code. Now that access is limited only by how much time and disk space a student has.

with computer-system organization, so you can skim or skip it if you already understand the concepts.

1.2.1 Computer-System Operation

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, 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.

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
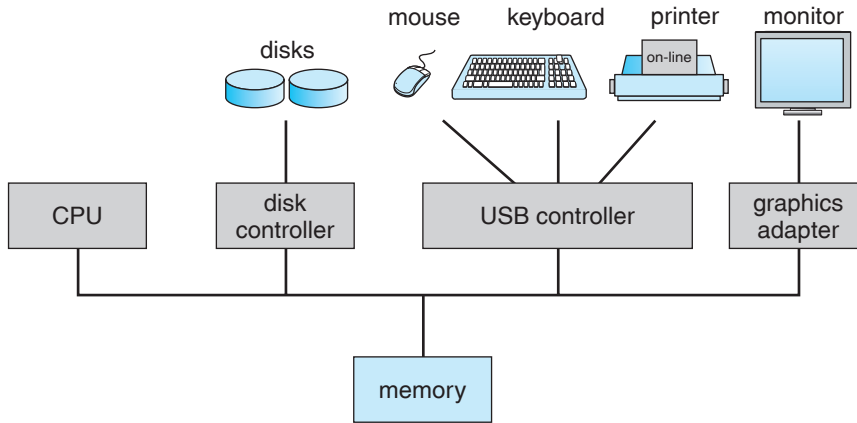
Figure 1.2   A modern computer system.

program, or **bootstrap program**, tends to be simple. Typically, it is stored in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware**, within the computer hardware. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate and load into memory the operating-system kernel. The operating system then starts executing the first process, such as "init," and waits for some event to occur.

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

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

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information; the routine, in turn, would call the interrupt-specific handler. However, interrupts must be handled quickly. Since only a predefined number of interrupts is possible, a table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector**, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for
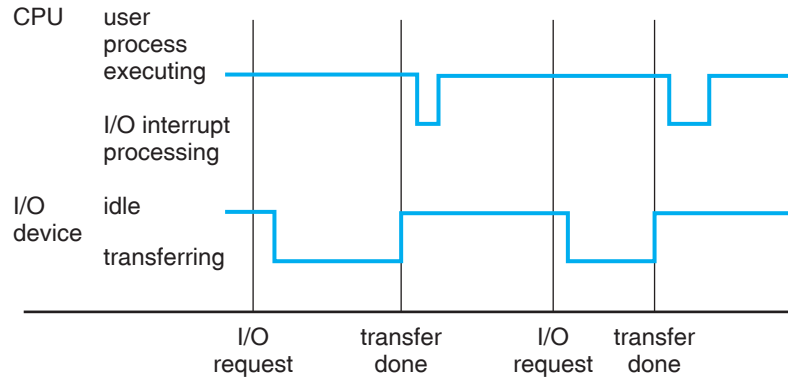
Figure 1.3   Interrupt time line for a single process doing output.

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

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

1.2.2   Storage Structure

The CPU can load instructions only from memory, so any programs to run must be stored there. General-purpose computers run most of their programs from rewriteable memory, called main memory (also called **random-access memory** or **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**. Computers use other forms of memory as well. Because the read-only memory (ROM) cannot be changed, only static programs are stored there. The immutability of ROM is of use in game cartridges. EEPROM cannot be changed frequently and so contains mostly static programs. For example, smartphones have EEPROM to store their factory-installed programs.

All forms of memory provide an array of 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.

A typical instruction–execution cycle, as executed on a system with a **von Neumann** architecture, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some

internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore *how* a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program.

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

1. Main memory is usually too small to store all needed programs and data permanently.

2. Main memory is a *volatile* storage device that loses its contents when power is turned off or otherwise lost.

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

The most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data. Most programs (system and application) are stored on a disk until they are loaded into memory. Many programs then use the disk as both the source and the destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system, as we discuss in Chapter 11.

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

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

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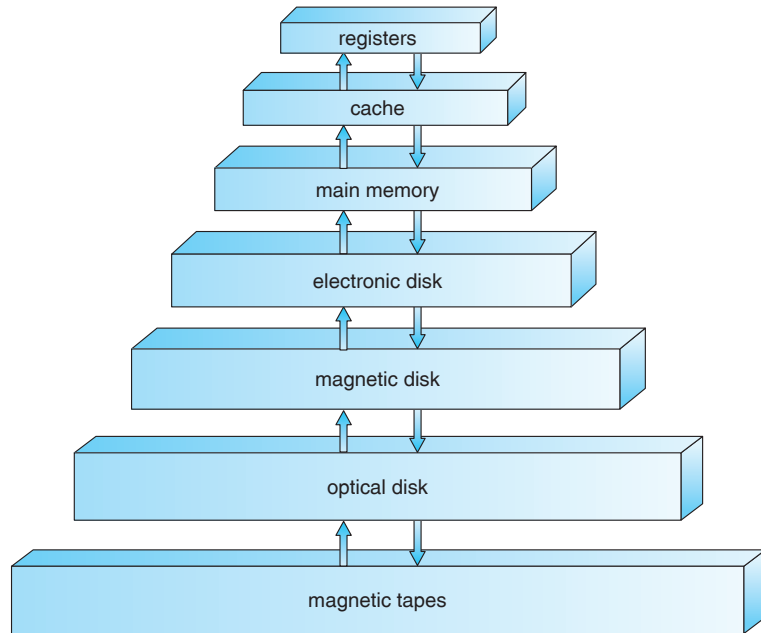
Figure 1.4   Storage-device hierarchy.

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 RAM. Another form of electronic disk is flash memory, which is popular in cameras and **personal digital assistants** (**PDAs**), in robots, and increasingly as removable storage on general-purpose computers. Flash memory is slower than DRAM but needs no power to retain its contents. Another form of nonvolatile storage is **NVRAM**, which is DRAM with battery backup power. This memory can be as fast as DRAM and (as long as the battery lasts) is nonvolatile.

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

1.2.3   I/O Structure

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

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Each device controller

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

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

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

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

1.3   Computer-System Architecture

In Section 1.2, we introduced the general structure of a typical computer system. A computer system may be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

1.3.1   Single-Processor Systems

Most systems use a single processor. The variety of single-processor systems may be surprising, however, since these systems range from PDAs through mainframes. On a single-processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes. Almost all systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk,
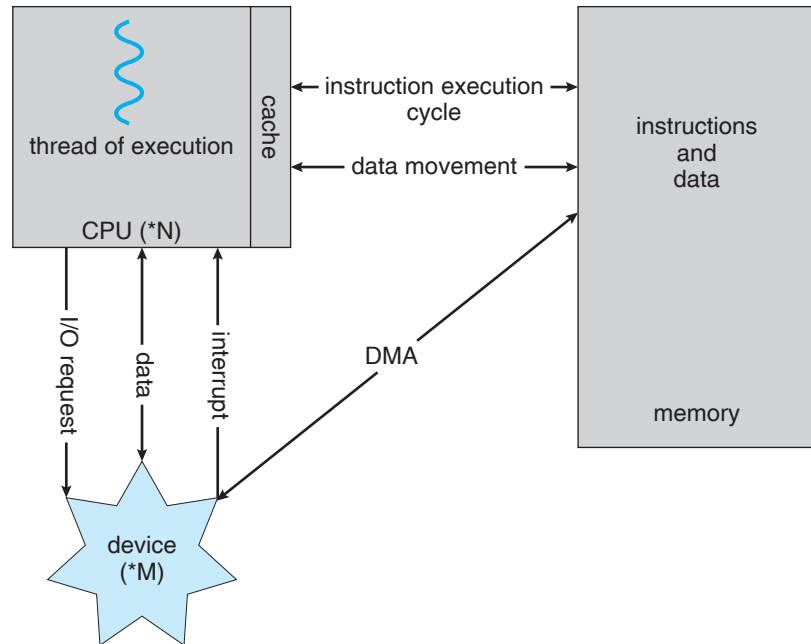
Figure 1.5 How a modern computer system works.

keyboard, and graphics controllers; or, on mainframes, they may take in the form of more general-purpose processors, such as I/O processors that move data rapidly among the components of the system.

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

1.3.2 Multiprocessor Systems

Although single-processor systems are most common, **multiprocessor systems** (also known as **parallel systems** or **tightly coupled systems**) are growing in importance. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

Multiprocessor systems have three main advantages:

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

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

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

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

The multiple-processor systems in use today are of two types. Some systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A 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.

The most common systems use **symmetric multiprocessing (SMP)**, in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no master–slave relationship exists between processors. Figure 1.6 illustrates a typical SMP architecture. Notice that each processor has its own set of registers, as well as a private—or local—cache; however, all processors share physical memory. An example of the SMP
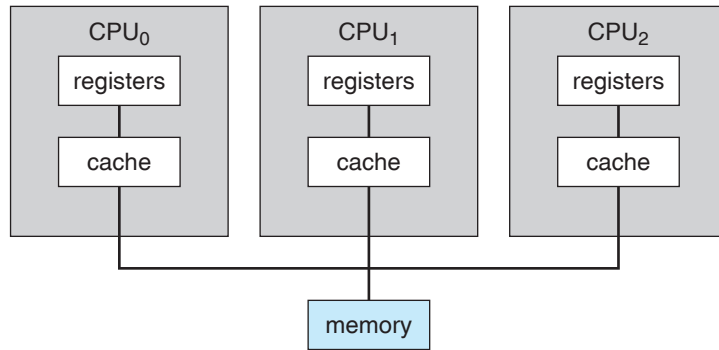
Figure 1.6 Symmetric multiprocessing architecture.

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 Solaris. 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. Such a system must be written carefully, as we shall see in Chapter 6. Virtually all modern operating systems—including Windows, Mac OS X, and Linux—now provide support for SMP.

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

Multiprocessing adds CPUs to increase computing power. If the CPU has an integrated memory controller, then adding CPUs can also increase the amount of memory addressable in the system. Either way, multiprocessing can cause a system to change its memory-access model from uniform memory access (**UMA**) to non-uniform memory access (**NUMA**). UMA is defined as the situation in which access to any RAM from any CPU takes the same amount of time. With NUMA, some parts of memory may take longer to access than other parts, creating a performance penalty. Operating systems can minimize the NUMA penalty through resource management, as discussed in Section 8.5.4.

A recent trend in CPU design is to include multiple computing **cores** on a single chip. In essence, these are multiprocessor chips. They can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips. As a result,
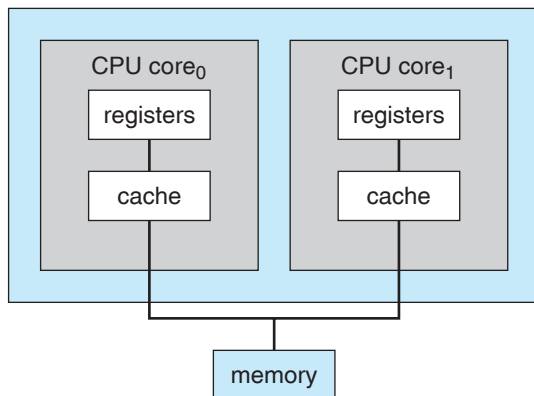
Figure 1.7 A dual-core design with two cores placed on the same chip.

multicore systems are especially well suited for server systems such as database and Web servers.

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

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

1.3.3 Clustered Systems

Another type of multiple-CPU system is the **clustered system**. Like multiprocessor systems, clustered systems gather together multiple CPUs to accomplish computational work. Clustered systems differ from multiprocessor systems, however, in that they are composed of two or more individual systems—or nodes—joined together. The definition of the term *clustered* is not concrete; many commercial packages wrestle with what a clustered system is and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via a **local-area network (LAN)** (as described in Section 1.10) or a faster interconnect, such as InfiniBand.

Clustering is usually used to provide **high-availability** service; that is, service will continue even if one or more systems in the cluster fail. High availability is generally obtained by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the

BEOWULF CLUSTERS

Beowulf clusters are designed for solving high-performance computing tasks. These clusters are built using commodity hardware—such as personal computers—that are connected via a simple local area network. Interestingly, a Beowulf cluster uses no one specific software package but rather consists of a set of open-source software libraries that allow the computing nodes in the cluster to communicate with one another. Thus, there are a variety of approaches for constructing a Beowulf cluster, although Beowulf computing nodes typically run the Linux operating system. Since Beowulf clusters require no special hardware and operate using open-source software that is freely available, they offer a low-cost strategy for building a high-performance computing cluster. In fact, some Beowulf clusters built from collections of discarded personal computers are using hundreds of computing nodes to solve computationally expensive problems in scientific computing.

applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

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

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

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN) (as described in Section 1.10). Parallel clusters allow multiple hosts to access the same data on the shared storage. Because most operating systems lack support for simultaneous data access by multiple hosts, parallel clusters are usually accomplished by use of special versions of software and special releases of applications. For example, Oracle Real Application Cluster is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access to data, the system must also supply access control and locking to ensure that no conflicting operations occur. This
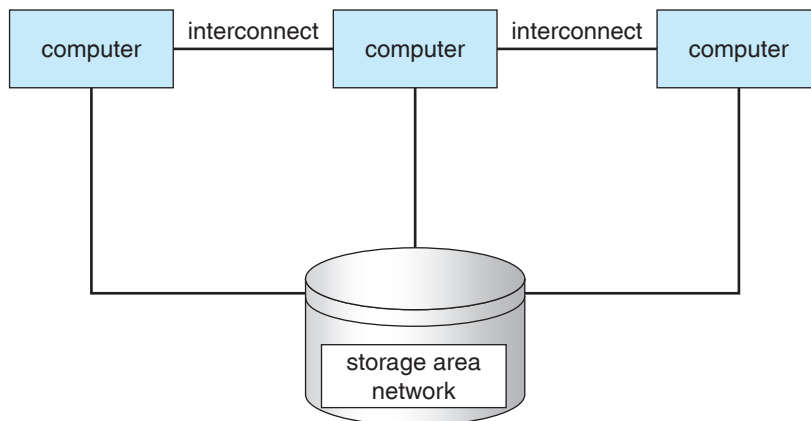
Figure 1.8   General structure of a clustered system.

function, commonly known as a **distributed lock manager (DLM)**, is included in some cluster technology.

Cluster technology is changing rapidly. Some cluster products support dozens of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks (SANs)**, as described in Section 11.3.3, which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability. Figure 1.8 depicts the general structure of a clustered system.

1.4   Operating-System Structure

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

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

The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.9). Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory.
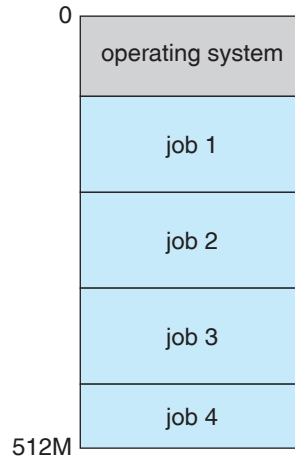
0

| operating system |
| job 1 |
| job 2 |
| job 3 |
| job 4 |

512M

Figure 1.9   Memory layout for a multiprogramming system.

The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job. When *that* job needs to wait, the CPU 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.)

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

Time sharing requires an **interactive** (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 input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.

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

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

Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision is **job scheduling**, which is discussed in Chapter 5. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management, which is covered in Chapters 7 and 8. In addition, if several jobs are ready to run at the same time, the system must choose among them. Making this decision is **CPU scheduling**, which is discussed in Chapter 5. Finally, running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. These considerations are discussed throughout the text.

In a time-sharing system, the operating system must ensure reasonable response time, which is sometimes accomplished through **swapping**, where processes are swapped in and out of main memory to the disk. A more common method for achieving this goal is **virtual memory**, a technique that allows the execution of a process that is not completely in memory (Chapter 8). 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.

Time-sharing systems must also provide a file system (Chapters 9 and 10). The file system resides on a collection of disks; hence, disk management must be provided (Chapter 11). Also, time-sharing systems provide a mechanism for protecting resources from inappropriate use (Chapter 13). To ensure orderly execution, the system must provide mechanisms for job synchronization and communication (Chapter 6), and it may ensure that jobs do not get stuck in a deadlock, forever waiting for one another (Section 6.9).

1.5   Operating-System Operations

As mentioned earlier, modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt

or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt.

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

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

1.5.1 Dual-Mode Operation

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

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

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

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that
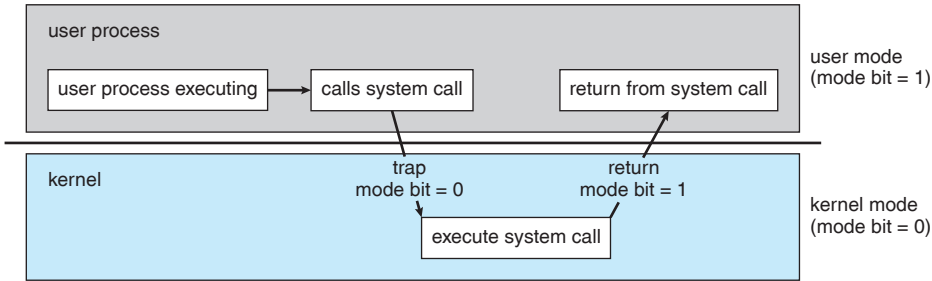
Figure 1.10   Transition from user to kernel mode.

may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

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

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

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic `trap` instruction, although some systems (such as the MIPS R2000 family) have a specific `syscall` instruction.

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

The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system. For instance, MS-DOS was written for the Intel 8088 architecture, which has no mode bit and therefore no dual mode. A user program running awry can wipe out the operating system by writing over it with data; also, multiple programs are able to write to a device at the same time,

with potentially disastrous results. Recent versions of the Intel CPU do provide dual-mode operation. Accordingly, most contemporary operating systems— such as Windows, as well as Unix, Linux, and Solaris—take advantage of this dual-mode feature and provide greater protection for the operating system.

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

1.5.2   Timer

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

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

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

1.6   Process Management

A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A time-shared user program such as a compiler is a process. A word-processing program being run by an individual user on a PC is a process. A system task, such as sending output

to a printer, can also be a process (or at least part of one). For now, you can consider a process to be a job or a time-shared program, but later you will learn that the concept is more general. As we shall see in Chapter 3, it is possible to provide system calls that allow processes to create subprocesses to execute concurrently.

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

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

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

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

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

We discuss process-management techniques in Chapters 3 through 6.

1.7   Memory Management

As we discussed in Section 1.2.2, the main memory is central to the operation of a modern computer system. Main memory is a large array of words or bytes, ranging in size from hundreds of thousands to billions. Each word or byte has

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

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

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

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

- Keeping track of which parts of memory are currently being used and by whom

- Deciding which processes (or parts thereof) and data to move into and out of memory

- Allocating and deallocating memory space as needed

Memory-management techniques are discussed in Chapters 7 and 8.

1.8   Storage Management

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

1.8.1   File-System Management

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic disk, optical disk, and magnetic tape are the most common. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, that

also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

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

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

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

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

File-management techniques are discussed in Chapters 9 and 10.

1.8.2 Mass-Storage Management

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

- Free-space management
- Storage allocation
- Disk scheduling

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

There are, however, many uses for storage that is slower and lower in cost (and sometimes of higher capacity) than secondary storage. Backups of disk data, seldom-used data, and long-term archival storage are some examples. Magnetic tape drives and their tapes and CD and DVD drives and platters are

typical **tertiary storage** devices. The media (tapes and optical platters) vary between **WORM** (write-once, read-many-times) and **RW** (read–write) formats.

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

Techniques for secondary and tertiary storage management are discussed in Chapter 11.

1.8.3 Caching

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

In addition, internal programmable registers, such as index registers, provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory. There are also caches that are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy. We are not concerned with these hardware-only caches in this text, since they are outside the control of the operating system.

Because caches have limited size, **cache management** is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance. Figure 1.11 compares storage performance in large workstations and small servers. Various replacement algorithms for software-controlled caches are discussed in Chapter 8.

Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use, and data must be in main memory before being moved to secondary storage for safekeeping. The file-system data, which resides permanently on secondary storage, may appear on several levels in the storage hierarchy. At the highest level, the operating system may maintain a cache of file-system data in main memory. In addition, electronic RAM disks (also known as **solid-state disks**) may be used for high-speed storage that is accessed through the file-system interface. The bulk of secondary storage is on magnetic disks. The magnetic-disk storage, in turn, is often backed up onto magnetic tapes or removable disks to protect against data loss in case of a hard-disk failure. Some systems automatically archive old file data from secondary storage to tertiary storage, such as tape jukeboxes, to lower the storage cost (see Chapter 11).

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 64 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

Figure 1.11   Performance of various levels of storage.

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

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

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

The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache (Figure 1.6). In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute
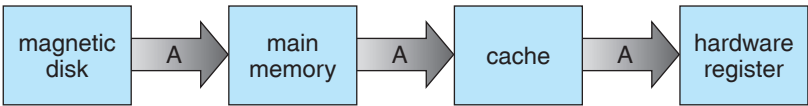

Figure 1.12   Migration of integer A from disk to register.

concurrently, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**, and it is usually a hardware problem (handled below the operating-system level).

1.8.4 I/O Systems

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

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

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

We discussed in Section 1.2.3 how interrupt handlers and device drivers are used in the construction of efficient I/O subsystems. In Chapter 12, we discuss how the I/O subsystem interfaces to the other system components, manages devices, transfers data, and detects I/O completion.

1.9   Protection and Security

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

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

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage, as we discuss in Chapter 13.

A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information

(her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system). Prevention of some of these attacks is considered an operating-system function on some systems, while other systems leave the prevention to policy or additional software. Due to the alarming rise in security incidents, operating-system security features represent a fast-growing area of research and implementation. Security is discussed in Chapter 14.

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

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

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

1.10 Distributed Systems

A distributed system is a collection of physically separate, possibly heterogeneous, computer systems that are networked to provide the users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability. Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver. Others make users specifically invoke network functions. Generally, systems contain a mix of the two modes—for example FTP and NFS. The protocols that create a distributed system can greatly affect that system's utility and popularity.

A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary in the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol, although ATM and other protocols are in widespread use. Likewise, operating-system support of protocols varies. Most operating systems support TCP/IP, including the Windows and UNIX operating systems. Some systems support proprietary protocols to suit their needs. To an operating system, a network protocol simply needs an interface device—a network adapter, for example—with a device driver to manage it, as well as software to handle data. These concepts are discussed throughout this book.

Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a floor, or a building. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide. These networks may run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For example, a **metropolitan-area network (MAN)** can link buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **small-area network** such as might be found in a home.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very short-range infrared communication can be used for networking. At a rudimentary level, whenever computers communicate, they use or create a network. These networks also vary in their performance and reliability.

Some operating systems have taken the concept of networks and distributed systems further than merely providing network connectivity. A **network operating system** is an operating system that provides features such as file sharing across the network and includes a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operating system provides a less autonomous environment: The different operating systems communicate closely enough to provide the illusion that only a single operating system controls the network.

We do not cover computer networks and distributed systems further in this text. We encourage the interested reader to consult the bibliographical notes at the end of this chapter for sources of further information.

1.11   Special-Purpose Systems

The discussion thus far has focused on the general-purpose computer systems that we are all familiar with. There are, however, other classes of computer systems whose functions are more limited and whose objective is to deal with limited computation domains.

1.11.1 Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to DVDs and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems—such as UNIX—with special-purpose applications to implement the functionality. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired. Yet others are hardware devices with application-specific integrated circuits (**ASICs**) that perform their tasks without an operating system.

The use of embedded systems continues to expand. The power of these devices, both as standalone units and as elements of networks and the Web, is sure to increase as well. Even now, entire houses can be computerized, so that a central computer—either a general-purpose computer or an embedded system—can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home. Someday, the refrigerator may call the grocery store when it notices the milk is gone.

Embedded systems almost always run **real-time operating systems**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing *must* be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt *after* it had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this system with a time-sharing system, where it is desirable (but not mandatory) to respond quickly, or a batch system, which may have no time constraints at all.

We do not cover real-time systems further in this text. We encourage the interested reader to consult the bibliographical notes at the end of this chapter for sources of further information.

1.11.2 Multimedia Systems

Most operating systems are designed to handle conventional data such as text files, programs, word-processing documents, and spreadsheets. However, a recent trend in technology is the incorporation of **multimedia data** into computer systems. Multimedia data consist of audio and video files as well as conventional files. These data differ from conventional data in that multimedia

data—such as frames of video—must be delivered (streamed) according to certain time restrictions (for example, 30 frames per second).

Multimedia describes a wide range of applications in popular use today. These include audio files such as MP3, DVD movies, video conferencing, and short video clips of movie previews or news stories downloaded over the Internet. Multimedia applications may also include live webcasts (broadcasting over the World Wide Web) of speeches or sporting events and even live webcams that allow a viewer in Manhattan to observe customers at a Paris cafe. Multimedia applications need not be either audio or video; rather, a multimedia application often includes a combination of both. For example, a movie may consist of separate audio and video tracks. Nor must multimedia applications be delivered only to desktop personal computers. Increasingly, they are being directed toward smaller devices, including PDAs and cellular telephones. For example, a stock trader may have stock quotes delivered wirelessly and in real time to his PDA.

We do not cover multimedia systems in further this text. We encourage the interested reader to consult the bibliographical notes at the end of this chapter for sources of further information.

1.11.3 Handheld Systems

Handheld systems include personal digital assistants (PDAs), such as Palm and Pocket-PCs, and cellular telephones, many of which use special-purpose embedded operating systems. Developers of handheld systems and applications face many challenges, most of which are due to the limited size of such devices. For example, a PDA is typically about 5 inches in height and 3 inches in width, and it weighs less than one-half pound. Because of their size, most handheld devices have small amounts of memory, slow processors, and small display screens. We take a look now at each of these limitations.

The amount of physical memory in a handheld depends on the device, but typically it is somewhere between 1 MB and 1 GB. (Contrast this with a typical PC or workstation, which may have several gigabytes of memory.) As a result, the operating system and applications must manage memory efficiently. This includes returning all allocated memory to the memory manager when the memory is not being used. In Chapter 8, we explore virtual memory, which allows developers to write programs that behave as if the system has more memory than is physically available. Currently, not many handheld devices use virtual-memory techniques, so program developers must work within the confines of limited physical memory.

A second issue of concern to developers of handheld devices is the speed of the processor used in the devices. Processors for most handheld devices run at a fraction of the speed of a processor in a PC. Faster processors require more power. To include a faster processor in a handheld device would require a larger battery, which would take up more space and would have to be replaced (or recharged) more frequently. Most handheld devices use smaller, slower processors that consume less power. Therefore, the operating system and applications must be designed not to tax the processor.

The last issue confronting program designers for handheld devices is I/O. A lack of physical space limits input methods to small keyboards, handwriting recognition, or small screen-based keyboards. The small display screens limit

output options. Whereas a monitor for a home computer may measure up to 30 inches, the display for a handheld device is often no more than 3 inches square. Familiar tasks, such as reading e-mail and browsing Web pages, must be condensed into smaller displays. One approach for displaying the content in Web pages is **Web clipping**, where only a small subset of a Web page is delivered and displayed on the handheld device.

Some handheld devices use wireless technology, such as BlueTooth or 802.11, allowing remote access to e-mail and Web browsing. Cellular telephones with connectivity to the Internet fall into this category. However, for PDAs that do not provide wireless access, downloading data typically requires the user first to download the data to a PC or workstation and then download the data to the PDA. Some PDAs allow data to be directly copied from one device to another using an infrared link.

Generally, the limitations in the functionality of PDAs are balanced by their convenience and portability. Their use continues to expand as network connections become more available and other options, such as digital cameras and MP3 players, expand their utility.

1.12   Computing Environments

So far, we have provided an overview of computer-system organization and major operating-system components. We conclude with a brief overview of how these are used in a variety of computing environments.

1.12.1   Traditional Computing

As computing matures, the lines separating many of the traditional computing environments are blurring. Consider the "typical office environment." Just a few years ago, this environment consisted of PCs connected to a network, with servers providing file and print services. Remote access was awkward, and portability was achieved by use of laptop computers. Terminals attached to mainframes were prevalent at many companies as well, with even fewer remote access and portability options.

The current trend is toward providing more ways to access these computing environments. Web technologies are stretching the boundaries of traditional computing. Companies establish **portals**, which provide Web accessibility to their internal servers. **Network computers** are essentially terminals that understand Web-based computing. Handheld computers can synchronize with PCs to allow very portable use of company information. Handheld PDAs can also connect to **wireless networks** to use the company's Web portal (as well as the myriad other Web resources).

At home, most users had a single computer with a slow modem connection to the office, the Internet, or both. Today, network-connection speeds once available only at great cost are relatively inexpensive, giving home users more access to more data. These fast data connections are allowing home computers to serve up Web pages and to run networks that include printers, client PCs, and servers. Some homes even have **firewalls** to protect their networks from security breaches. Those firewalls cost thousands of dollars a few years ago and did not even exist a decade ago.

In the latter half of the previous century, computing resources were scarce. (Before that, they were nonexistent!) For a period of time, systems were either batch or interactive. Batch systems processed jobs in bulk, with predetermined input (from files or other sources of data). Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. Time-sharing systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources.

Today, traditional time-sharing systems are uncommon. The same scheduling technique is still in use on workstations and servers, but frequently the processes are all owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time. Consider the windows created while a user is working on a PC, for example, and the fact that they may be performing different tasks at the same time.

1.12.2   Client–Server Computing

As PCs have become faster, more powerful, and cheaper, designers have shifted away from centralized system architecture. Terminals connected to centralized systems are now being supplanted by PCs. Correspondingly, user-interface functionality once handled directly by centralized systems is increasingly being handled by PCs. As a result, many of today's systems act as **server systems** to satisfy requests generated by **client systems**. This form of specialized distributed system, called a **client–server** system, has the general structure depicted in Figure 1.13.

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data); in response, the server executes the action and sends back results to the client. A server running a database that responds to client requests for data is an example of such a system.

- The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a Web server that delivers files to clients running Web browsers.
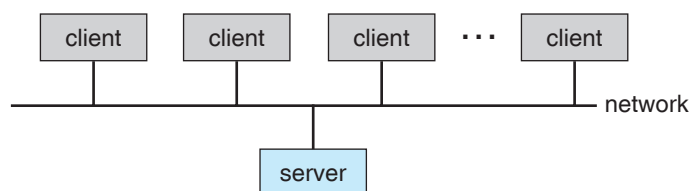

Figure 1.13   General structure of a client–server system.

1.12.3 Peer-to-Peer Computing

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another; instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client-server systems. In a client-server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.

- A peer acting as a client must first discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enable peers to exchange files with one another. The Napster system uses an approach similar to the first type described above: a centralized server maintains an index of all files stored on peer nodes in the Napster network, and the actual exchanging of files takes place between the peer nodes. The Gnutella system uses a technique similar to the second type: a client broadcasts file requests to other nodes in the system, and nodes that can service the request respond directly to the client. The future of exchanging files remains uncertain because many of the files are copyrighted (music, for example), and there are laws governing the distribution of copyrighted material. In any case, though, peer-to-peer technology undoubtedly will play a role in the future of many services, such as searching, file exchange, and e-mail.

1.12.4 Web-Based Computing

The Web has become ubiquitous, leading to more access by a wider variety of devices than was dreamt of a few years ago. PCs are still the most prevalent access devices, with workstations, handheld PDAs, and even cell phones also providing access.

Web computing has increased the emphasis on networking. Devices that were not previously networked now include wired or wireless access. Devices that were networked now have faster network connectivity, provided by either

improved networking technology, optimized network implementation code, or both.

The implementation of Web-based computing has given rise to new categories of devices, such as **load balancers**, which distribute network connections among a pool of similar servers. Operating systems like Windows 95, which acted as Web clients, have evolved into Linux and Windows XP and beyond, which can act as Web servers as well as clients. Generally, the Web has increased the complexity of devices because their users require them to be Web-enabled.

1.13   Open-Source Operating Systems

The study of operating systems, as noted earlier, is made easier by the availability of a vast number of open-source releases. **Open-source operating systems** are those made available in source-code format rather than as compiled binary code. Linux is the most famous open- source operating system, while Microsoft Windows is a well-known example of the opposite **closed-source** approach. Starting with the source code allows the programmer to produce binary code that can be executed on a system. Doing the opposite— **reverse engineering** the source code from the binaries—is quite a lot of work, and useful items such as comments are never recovered. Learning operating systems by examining the actual source code, rather than reading summaries of that code, can be extremely useful. With the source code in hand, a student can modify the operating system and then compile and run the code to try out those changes, which is another excellent learning tool. This text includes projects that involve modifying operating-system source code, while also describing algorithms at a high level to be sure all important operating system topics are covered. Throughout the text, we provide pointers to examples of open-source code for deeper study.

There are many benefits to open-source operating systems, including a community of interested (and usually unpaid) programmers who contribute to the code by helping to debug it, analyze it, provide support, and suggest changes. Arguably, open-source code is more secure than closed-source code because many more eyes are viewing the code. Certainly open-source code has bugs, but open-source advocates argue that bugs tend to be found and fixed faster owing to the number of people using and viewing the code. Companies that earn revenue from selling their programs tend to be hesitant to open-source their code, but Red Hat, SUSE, Sun, and a myriad of other companies are doing just that and showing that commercial companies benefit, rather than suffer, when they open-source their code. Revenue can be generated through support contracts and the sale of hardware on which the software runs, for example.

1.13.1   History

In the early days of modern computing (that is, the 1950s), a great deal of software was available in open-source format. The original hackers (computer enthusiasts) at MIT's Tech Model Railroad Club left their programs in drawers for others to work on. "Homebrew" user groups exchanged code during their meetings. Later, company-specific user groups, such as Digital Equipment

Corporation's DEC, accepted contributions of source-code programs, collected them onto tapes, and distributed the tapes to interested members.

Computer and software companies eventually sought to limit the use of their software to authorized computers and paying customers. Releasing only the binary files compiled from the source code, rather than the source code itself, helped them to achieve this goal, as well as protecting their code and their ideas from their competitors. Another issue involved copyrighted material. Operating systems and other programs can limit the ability to play back movies and music or display electronic books to authorized computers. Such **copy protection** or **digital rights management (DRM)** would not be effective if the source code that implemented these limits were published. Laws in many countries, including the U.S. Digital Millennium Copyright Act (DMCA), make it illegal to reverse-engineer DRM code or otherwise try to circumvent copy protection.

To counter the move to limit software use and redistribution, Richard Stallman in 1983 started the GNU project to create a free, open-source UNIX-compatible operating system. In 1985, he published the GNU Manifesto, which argues that all software should be free and open-sourced. He also formed the **Free Software Foundation (FSF)** with the goal of encouraging the free exchange of software source code and the free use of that software. Rather than copyright its software, the FSF "copylefts" the software to encourage sharing and improvement. The **GNU General Public License (GPL)** codifies copylefting and is a common license under which free software is released. Fundamentally, GPL requires that the source code be distributed with any binaries and that any changes made to the source code be released under the same GPL license.

1.13.2   Linux

As an example of an open-source operating system, consider **GNU/Linux**. The GNU project produced many UNIX-compatible tools, including compilers, editors, and utilities, but never released a kernel. In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide. The advent of the Internet meant that anyone interested could download the source code, modify it, and submit changes to Torvalds. Releasing updates once a week allowed this so-called Linux operating system to grow rapidly, enhanced by several thousand programmers.

The resulting GNU/Linux operating system has spawned hundreds of unique **distributions**, or custom builds, of the system. Major distributions include RedHat, SUSE, Fedora, Debian, Slackware, and Ubuntu. Distributions vary in function, utility, installed applications, hardware support, user interface, and purpose. For example, RedHat Enterprise Linux is geared to large commercial use. PCLinuxOS is a **LiveCD**—an operating system that can be booted and run from a CD-ROM without being installed on a system's hard disk. One variant of PCLinuxOS, "PCLinuxOS Supergamer DVD," is a **LiveDVD** that includes graphics drivers and games. A gamer can run it on any compatible system simply by booting from the DVD. When the gamer is finished, a reboot of the system resets it to its installed operating system.

Access to the Linux source code varies by release. Here, we consider Ubuntu Linux. Ubuntu is a popular Linux distribution that comes in a variety

of types, including those tuned for desktops, servers, and students. Its founder pays for the printing and mailing of DVDs containing the binary and source code (which helps to make it popular). The following steps outline a way to explore the Ubuntu kernel source code on systems that support the free "VMware Player" tool:

- Download the player from `http://www.vmware.com/download/player/` and install it on your system.

- Download a virtual machine containing Ubuntu. Hundreds of "appliances", or virtual machine images, pre-installed with operating systems and applications, are available from VMware at `http://www.vmware.com/appliances/`.

- Boot the virtual machine within VMware Player.

- Get the source code of the kernel release of interest, such as 2.6, by executing `wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.18.1.tar.bz2` within the Ubuntu virtual machine.

- Uncompress and untar the downloaded file via `tar xjf linux-2.6.18.1.tar.bz2`.

- Explore the source code of the Ubuntu kernel, which is now in `./linux-2.6.18.1`.

For more about Linux, see Chapter 15. For more about virtual machines, see Section 2.8.

1.13.3 BSD UNIX

BSD UNIX has a longer and more complicated history than Linux. It started in 1978 as a derivative of AT&T's UNIX. Releases from the University of California at Berkeley came in source and binary form, but they were not open-source because a license from AT&T was required. BSD UNIX's development was slowed by a lawsuit by AT&T, but eventually a fully functional open-source version, 4.4BSD-lite, was released in 1994.

Just as with Linux, there are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD. To explore the source code of FreeBSD, simply download the virtual machine image of the version of interest and boot it within VMware, as described above for Ubuntu Linux. The source code comes with the distribution and is stored in `/usr/src/`. The kernel source code is in `/usr/src/sys`. For example, to examine the virtual-memory implementation code in the FreeBSD kernel, see the files in `/usr/src/sys/vm`.

Darwin, the core kernel component of Mac OS X, is based on BSD UNIX and is open-sourced as well. That source code is available from `http://www.opensource.apple.com/darwinsource/`. Every Mac OS X release has its open-source components posted at that site. The name of the package that contains the kernel is "xnu." The source code for Mac OS X kernel revision 1228 (the source code to Mac OS X Leopard) can be found at www.opensource.apple.com/darwinsource/tarballs/apsl/xnu-1228.tar.gz. Apple also provides extensive developer tools, documentation, and support at `http://connect.apple.com`. For more information, see Appendix A.

1.13.4 Solaris

Solaris is the commercial UNIX-based operating system of Sun Microsystems. Originally, Sun's **SunOS** operating system was based on BSD UNIX. Sun moved to AT&T's System V UNIX as its base in 1991. In 2005, Sun open-sourced some of the Solaris code, and over time, the company has added more and more to that open-source code base. Unfortunately, not all of Solaris is open-sourced, because some of the code is still owned by AT&T and other companies. However, Solaris can be compiled from the open source and linked with binaries of the close-sourced components, so it can still be explored, modified, compiled, and tested.

The source code is available from `http://opensolaris.org/os/downloads/`. Also available there are pre-compiled distributions based on the source code, documentation, and discussion groups. It is not necessary to download the entire source-code bundle from the site, because Sun allows visitors to explore the source code on-line via a source code browser.

1.13.5 Utility

The free software movement is driving legions of programmers to create thousands of open-source projects, including operating systems. Sites like `http://freshmeat.net/` and `http://distrowatch.com/` provide portals to many of these projects. Open-source projects enable students to use source code as a learning tool. They can modify programs and test them, help find and fix bugs, and otherwise explore mature, full-featured operating systems, compilers, tools, user interfaces, and other types of programs. The availability of source code for historic projects, such as Multics, can help students to understand those projects and to build knowledge that will help in the implementation of new projects.

GNU/Linux, BSD UNIX, and Solaris are all open-source operating systems, but each has its own goals, utility, licensing, and purpose. Sometimes licenses are not mutually exclusive and cross-pollination occurs, allowing rapid improvements in operating-system projects. For example, several major components of Solaris have been ported to BSD UNIX. The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

1.14 Summary

An operating system is software that manages the computer hardware, as well as providing an environment for application programs to run. Perhaps the most visible aspect of an operating system is the interface to the computer system it provides to the human user.

For a computer to do its job of executing programs, the programs must be in main memory. Main memory is the only large storage area that the processor can access directly. It is an array of words or bytes, ranging in size from millions to billions. Each word in memory has its own address. The main memory is usually a volatile storage device that loses its contents when power is turned

off or lost. Most computer systems provide secondary storage as an extension of main memory. Secondary storage provides a form of nonvolatile storage that is capable of holding large quantities of data permanently. The most common secondary-storage device is a magnetic disk, which provides storage of both programs and data.

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.

There are several different strategies for designing a computer system. Uniprocessor systems have only a single processor, while multiprocessor systems contain two or more processors that share physical memory and peripheral devices. The most common multiprocessor design is symmetric multiprocessing (or SMP), where all processors are considered peers and run independently of one another. Clustered systems are a specialized form of multiprocessor systems and consist of multiple computer systems connected by a local area network.

To best utilize the CPU, modern operating systems employ multiprogramming, which allows several jobs to be in memory at the same time, thus ensuring that the CPU always has a job to execute. Time-sharing systems are an extension of multiprogramming wherein CPU scheduling algorithms rapidly switch between jobs, thus providing the illusion that each job is running concurrently.

The operating system must ensure correct operation of the computer system. To prevent user programs from interfering with the proper operation of the system, the hardware has two modes: user mode and kernel mode. Various instructions (such as I/O instructions and halt instructions) are privileged and can be executed only in kernel mode. The memory in which the operating system resides must also be protected from modification by the user. A timer prevents infinite loops. These facilities (dual mode, privileged instructions, memory protection, and timer interrupt) are basic building blocks used by operating systems to achieve correct operation.

A process (or job) is the fundamental unit of work in an operating system. Process management includes creating and deleting processes and providing mechanisms for processes to communicate and synchronize with each other. An operating system manages memory by keeping track of what parts of memory are being used and by whom. The operating system is also responsible for dynamically allocating and freeing memory space. Storage space is also managed by the operating system; this includes providing file systems for representing files and directories and managing space on mass-storage devices.

Operating systems must also be concerned with protecting and securing the operating system and users. Protection measures are mechanisms that control the access of processes or users to the resources made available by the computer system. Security measures are responsible for defending a computer system from external or internal attacks.

Distributed systems allow users to share resources on geographically dispersed hosts connected via a computer network. Services may be provided through either the client–server model or the peer-to-peer model. In a clustered system, multiple machines can perform computations on data residing on shared storage, and computing can continue even when some subset of cluster members fails.

LANs and WANs are the two basic types of networks. LANs enable processors distributed over a small geographical area to communicate, whereas WANs allow processors distributed over a larger area to communicate. LANs typically are faster than WANs.

There are several computer systems that serve specific purposes. These include real-time operating systems designed for embedded environments such as consumer devices, automobiles, and robotics. Real-time operating systems have well-defined fixed-time constraints. Processing *must* be done within the defined constraints, or the system will fail. Multimedia systems involve the delivery of multimedia data and often have special requirements of displaying or playing audio, video, or synchronized audio and video streams.

Recently, the influence of the Internet and the World Wide Web has encouraged the development of operating systems that include Web browsers and networking and communication software as integral features.

The free software movement has created thousands of open-source projects, including operating systems. Because of these projects, students are able to use source code as a learning tool. They can modify programs and test them, help find and fix bugs, and otherwise explore mature, full-featured operating systems, compilers, tools, user interfaces, and other types of programs.

GNU/Linux, BSD UNIX, and Solaris are all open-source operating systems. The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

Practice Exercises

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

1.2   What are the main differences between operating systems for mainframe computers and personal computers?

1.3   List the four steps that are necessary to run a program on a completely dedicated machine—a computer that is running only that program.

1.4   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 "waste" resources? Why is such a system not really wasteful?

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

1.6   Consider the various definitions of *operating system*. Consider whether the operating system should include applications such as Web browsers and mail programs. Argue both that it should and that it should not, and support your answers.

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

1.8   Which of the following instructions should be privileged?

   a.   Set value of timer.

b. Read the clock.

c. Clear memory.

d. Issue a trap instruction.

e. Turn off interrupts.

f. Modify entries in device-status table.

g. Switch from user to kernel mode.

h. Access I/O device.

1.9 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.

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

1.11 Timers can be used to compute the current time. Provide a short description of how this could be accomplished.

1.12 Is the Internet a LAN or a WAN?

Exercises

1.13 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 in a dedicated machine? Explain your answer.

1.14 The issue of resource utilization shows up in different forms in different types of operating systems. List what resources must be managed carefully in the following settings:

a. Mainframe or minicomputer systems

b. Workstations connected to servers

c. Handheld computers

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

1.16 Identify which of the functionalities listed below need to be supported by the operating system for (a) handheld devices and (b) real-time systems.

a. Batch programming

b. Virtual memory

c. Time sharing

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

1.18 How do clustered systems differ from multiprocessor systems? What is required for two machines belonging to a cluster to cooperate to provide a highly available service?

1.19 Distinguish between the client–server and peer-to-peer models of distributed systems.

1.20 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 disadvantages of each.

1.21 How are network computers different from traditional personal computers? Describe some usage scenarios in which it is advantageous to use network computers.

1.22 What is the purpose of interrupts? What are the differences between a trap and an interrupt? Can traps be generated intentionally by a user program? If so, for what purpose?

1.23 Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.

   a. How does the CPU interface with the device to coordinate the transfer?

   b. How does the CPU know when the memory operations are complete?

   c. The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

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

1.25 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?

1.26 Consider an SMP system similar to that shown in Figure 1.6. Illustrate with an example how data residing in memory could in fact have two different values in each of the local caches.

1.27 Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:

   a. Single-processor systems

    b.   Multiprocessor systems

    c.   Distributed systems

1.28 Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

1.29 What network configuration would best suit the following environments?

    a.   A dormitory floor

    b.   A university campus

    c.   A state

    d.   A nation

1.30 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

1.31 What are the tradeoffs inherent in handheld computers?

1.32 Identify several advantages and several disadvantages of open-source operating systems. Include the types of users who would find each aspect to be an advantage or a disadvantage.

Bibliographical Notes

Brookshear [2003] provides an overview of computer science in general.

An overview of the Linux operating system is presented in Bovet and Cesati [2006]. Solomon and Russinovich [2000] give an overview of Microsoft Windows and considerable technical detail about the system internals and components. Russinovich and Solomon [2009] update this information to WinSEVEN. McDougall and Mauro [2007] cover the internals of the Solaris operating system. Mac OS X is presented at `http://www.apple.com/macosx`. Mac OS X internals are discussed in Singh [2007].

Coverage of peer-to-peer systems includes Parameswaran et al. [2001], Gong [2002], Ripeanu et al. [2002], Agre [2003], Balakrishnan et al. [2003], and