A program is a spell cast over a computer, turning input into error messages.

—Anonymous

CHAPTER 8 System Software

8.1 INTRODUCTION

Over the course of your career, you may find yourself in a position where you are compelled to buy "suboptimal" computer hardware because a certain system is the only one that runs a particular software product needed by your employer. Although you may be tempted to see this situation as an insult to your better judgment, you have to recognize that a complete system requires software as well as hardware. Software is the window through which users see a system. If the software can't deliver services in accordance with users' expectations, they see the entire system as inadequate, regardless of the quality of its hardware.

In Chapter 1, we introduced a computer organization that consists of six machine levels, with each level above the gate level providing an abstraction for the layer below it. In Chapter 4, we discussed assemblers and the relationship of assembly language to the architecture. In this chapter, we study software found at the third level, and tie these ideas to software at the fourth and fifth levels. The collection of software at these three levels runs below application programs and just above the instruction set architecture level. These are the software components, the "machines," with which your application source code interacts. Programs at these levels work together to grant access to the hardware resources that carry out the commands contained in application programs. But to look at a computer system as if it were a single thread running from application source code down to the gate level is to limit our understanding of what a computer system is. We would be ignoring the rich set of services provided at each level.

Although our model of a computer system places only the operating system in the "system software" level, the study of system software often includes com-

pilers and other utilities, as well as a category of complex programs sometimes called *middleware*. Generally speaking, middleware is a broad classification for software that provides services above the operating system layer, but below the application program layer. You may recall that in Chapter 1 we discussed the semantic gap that exists between physical components and high-level languages and applications. We know this semantic gap must not be perceptible to the user, and middleware is the software that provides the necessary invisibility. Because the operating system is the foundation for all system software, virtually all system software interacts with the operating system to some extent. We start with a brief introduction to the inner workings of operating systems, and then we move on to the higher software layers.

8.2 OPERATING SYSTEMS

Originally, the main role of an operating system was to help various applications interact with the computer hardware. Operating systems provide a necessary set of functions allowing software packages to control the computer's hardware. Without an operating system, each program you run would need its own driver for the video card, the sound card, the hard drive, and so on.

Although modern operating systems still perform this function, users' expectations of operating systems have changed considerably. They assume that an operating system will make it easy for them to manage the system and its resources. This expectation has begotten "drag and drop" file management, as well as "plug and play" device management. From the programmer's perspective, the operating system obscures the details of the system's lower architectural levels, permitting greater focus on high-level problem solving. We have seen that it is difficult to program at the machine level or the assembly language level. The operating system works with numerous software components, creating a friendlier environment in which system resources are utilized effectively and efficiently and where programming in machine code is not required. The operating system not only provides this interface to the programmer, but it also acts as a layer between application software and the actual hardware of the machine. Whether looked at through the eyes of the user or the lines of code of an application, the operating system is, in essence, a virtual machine that provides an interface from hardware to software. It deals with real devices and real hardware so the application programs and users don't have to.

The operating system itself is a little more than an ordinary piece of software. It differs from most other software in that it is loaded by booting the computer and is then executed directly by the processor. The operating system must have control of the processor (as well as other resources), because one of its many tasks is scheduling the processes that use the CPU. It relinquishes control of the CPU to various application programs during the course of their execution. The operating system is dependent upon the processor to regain control when the application either no longer requires the CPU or gives up the CPU as it waits for other resources.

As we have mentioned, the operating system is an important interface to the underlying hardware, both for users and for application programs. In addition to

its role as an interface, it has three principle tasks. Process management is perhaps the most interesting of these three. The other two are system resource management and protection of those resources from errant processes. Before we discuss these duties, let's look at a short history of operating systems development to see how it parallels the evolution of computer hardware.

8.2.1 Operating Systems History

Today's operating systems strive for optimum ease of use, providing an abundance of graphical tools to assist both novice and experienced users. But this wasn't always the case. A scant generation ago, computer resources were so precious that every machine cycle had to do useful work. Because of the enormously high cost of computer hardware, computer time was allotted with utmost care. In those days, if you wished to use a computer, your first step was to sign up for time on the machine. When your time arrived, you fed in a deck of punched cards yourself, running the machine in single-user, interactive mode. Before loading your program, however, you had to first load the compiler. The initial set of cards in the input deck included the bootstrap loader, which caused the rest of the cards to be loaded. At this point, you could compile your program. If there was an error in your code, you had to find it quickly, repunch the offending card (or cards) and feed the deck into the computer again in another attempt to compile your program. If you couldn't quickly locate the problem, you had to sign up for more time and try again later. If your program compiled, the next step was to link your object code with library code files to create the executable file that would actually be run. This was a terrible waste of expensive computer—and human—time. In an effort to make the hardware usable by more people, *batch processing* was introduced.

With batch processing, professional operators combined decks of cards into batches, or bundles, with the appropriate instructions allowing them to be processed with minimal intervention. These batches were usually programs of similar types. For example, there might be a batch of FORTRAN programs and then a batch of COBOL programs. This allowed the operator to set up the machine for FORTRAN programs, read and execute them all, and then switch to COBOL. A program called a *resident monitor* allowed programs to be processed without human interaction (other than placing the decks of cards into the card reader).

Monitors were the precursors of modern day operating systems. Their role was straightforward: The monitor started the job, gave control of the computer to the job, and when the job was done, the monitor resumed control of the machine. The work originally done by people was being done by the computer, thus increasing efficiency and utilization. As your authors remember, however, the turnaround time for batch jobs was quite large. (We recall the good old days of dropping off decks of assembly language cards for processing at the data center. We were thrilled at having to wait for anything less than 24 hours before getting results back!) Batch processing made debugging difficult, or more correctly, very time consuming. An infinite loop in a program could wreak havoc in a system.

Eventually, timers were added to monitors to prevent one process from monopolizing the system. However, monitors had a severe limitation in that they provided no additional protection. Without protection, a batch job could affect pending jobs. (For example, a "bad" job might read too many cards, thus rendering the next program incorrect.) Moreover, it was even possible for a batch job to affect the monitor code! To fix this problem, computer systems were provided with specialized hardware, allowing the computer to operate in either monitor mode or user mode. Programs were run in user mode, switching to monitor mode when certain system calls were necessary.

Increases in CPU performance made punched card batch processing increasingly less efficient. Card readers simply could not keep the CPU busy. Magnetic tape offered one way to process decks faster. Card readers and printers were connected to smaller computers, which were used to read decks of cards onto tape. One tape might contain several jobs. This allowed the mainframe CPU to continually switch among processes without reading cards. A similar procedure was followed for output. The output was written to tape, which was then removed and put on a smaller computer that performed the actual printing. It was necessary for the monitor to periodically check whether an I/O operation was needed. Timers were added to jobs to allow for brief interruptions so the monitor could send pending I/O to the tape units. This allowed I/O and CPU computations to occur in parallel. This process, prevalent in the late 60s to late 70s, was known as *Simultaneous Peripheral Operation Online*, or *SPOOLing*, and it is the simplest form of multiprogramming. The word has stuck in the computer lexicon, but its contemporary meaning refers to printed output that is written to disk prior to being sent to the printer.

Multiprogramming systems (established in the late 60s and continuing to the present day) extend the idea of spooling and batch processing to allow several executing programs to be in memory concurrently. This is achieved by cycling through processes, allowing each one to use the CPU for a specific slice of time. Monitors were able to handle multiprogramming to a certain extent. They could start jobs, spool operations, perform I/O, switch between user jobs, and give some protection between jobs. It should be clear, however, that the monitor's job was becoming more complex, necessitating software that was more elaborate. It was at this point that monitors evolved into the software we now know as *operating systems*.

Although operating systems relieved programmers (and operators) of a significant amount of work, users wanted closer interaction with computers. In particular, the concept of batch jobs was unappealing. Wouldn't it be nice if users could submit their own jobs, interactively, and get immediate feedback? *Timesharing systems* allowed exactly this. Terminals were connected to systems that allowed access by multiple concurrent users. Batch processing was soon outmoded, as interactive programming facilitated timesharing (also known as *timeslicing*). In a timesharing system, the CPU switches between user sessions very quickly, giving each user a small slice of processor time. This procedure of switching between processes is called *context switching*. The operating system

performs these context switches quickly, in essence, giving the user a personal virtual machine.

Timesharing permits many users to share the same CPU. By extending this idea, a system can allow many users to share a single application. Large interactive systems, such as airline reservation systems, service thousands of simultaneous users. As with timesharing systems, large interactive system users are unaware of the other users on the system.

The introduction of multiprogramming and timesharing required more complex operating system software. During a context switch, all pertinent information about the currently executing process must be saved, so that when the process is scheduled to use the CPU again, it can be restored to the exact state in which it was interrupted. This requires that the operating system know all the details of the hardware. Recall from Chapter 6 that virtual memory and paging are used in today's systems. Page tables and other information associated with virtual memory must be saved during a context switch. CPU registers must also be saved when a context switch occurs because they contain the current state of the executing process. These context switches are not cheap in terms of resources or time. To make them worthwhile, the operating system must deal with them quickly and efficiently.

It is interesting to note the close correlation between the advances in architecture and the evolution of operating systems. First-generation computers used vacuum tubes and relays and were quite slow. There was no need, really, for an operating system, because the machines could not handle multiple concurrent tasks. Human operators performed the required task management chores. Second-generation computers were built with transistors. This resulted in an increase in speed and CPU capacity. Although CPU capacity had increased, it was still costly and had to be utilized to the maximum possible extent. Batch processing was introduced as a means to keep the CPU busy. Monitors helped with the processing, providing minimal protection and handling interrupts. The third generation of computers was marked by the use of integrated circuits. This, again, resulted in an increase in speed. Spooling alone could not keep the CPU busy, so timesharing was introduced. Virtual memory and multiprogramming necessitated a more sophisticated monitor, which evolved into what we now call an operating system. Fourth-generation technology, VLSI, allowed for the personal computing market to flourish. Network operating systems and distributed systems are an outgrowth of this technology. Minimization of circuitry also saved on chip real estate, allowing more room for circuits that manage pipelining, array processing, and multiprocessing.

Early operating systems were divergent in design. Vendors frequently produced one or more operating systems specific to a given hardware platform. Operating systems from the same vendor designed for different platforms could vary radically both in their operation and in the services they provided. It wasn't uncommon for a vendor to introduce a new operating system when a new model of computer was introduced. IBM put an end to this practice in the mid-1960s when it introduced the 360 series of computers. Although each computer in the 360 family of machines differed greatly in performance and intended audience, all computers ran the same basic operating system, OS/360.

Unix is another operating system that exemplifies the idea of one operating system spanning many hardware platforms. Ken Thompson, of AT&T's Bell Laboratories, began working on Unix in 1969. Thompson originally wrote Unix in assembly language. Because assembly languages are hardware specific, any code written for one platform must be rewritten and assembled for a different platform. Thompson was discouraged by the thought of rewriting his Unix code for different machines. With the intention of sparing future labor, he created a new interpreted high-level language called *B*. It turned out that B was too slow to support operating system activities. Dennis Ritchie subsequently joined Thompson to develop the *C* programming language, releasing the first C compiler in 1973. Thompson and Ritchie rewrote the Unix operating system in C, forever dispelling the belief that operating systems must be written in assembly language. Because it was written in a high-level language and could be compiled for different platforms, Unix was highly portable. This major departure from tradition has allowed Unix to become extremely popular, and, although it found its way into the market slowly, it is currently the operating system of choice for millions of users. The hardware neutrality exhibited by Unix allows users to select the best hardware for their applications, instead of being limited to a specific platform. There are literally hundreds of different flavors of Unix available today, including Sun's Solaris, IBM's AIX, Hewlett-Packard's HP-UX, and Linux for PCs and servers.

Real-time, Multiprocessor, and Distributed/Networked Systems

Perhaps the biggest challenge to operating system designers in recent years has been the introduction of real-time, multiprocessor, and distributed/networked systems. *Real-time systems* are used for process control in manufacturing plants, assembly lines, robotics, and complex physical systems such as the space station, to name only a few. Real-time systems have severe timing constraints. If specific deadlines are not met, physical damage or other undesirable effects to persons or property can occur. Because these systems must respond to external events, correct process scheduling is critical. Imagine a system controlling a nuclear power plant that couldn't respond quickly enough to an alarm signaling critically high temperatures in the core! In *hard real-time systems* (with potentially fatal results if deadlines aren't met), there can be no errors. In *soft real-time systems*, meeting deadlines is desirable, but does not result in catastrophic results if deadlines are missed. QNX is an excellent example of a *real-time operating system* (*RTOS*) designed to meet strict scheduling requirements. QNX is also suitable for embedded systems because it is powerful yet has a small footprint (requires very little memory) and tends to be very secure and reliable.

Multiprocessor systems present their own set of challenges, because they have more than one processor that must be scheduled. The manner in which the operating system assigns processes to processors is a major design consideration. Typically, in a multiprocessing environment, the CPUs cooperate with each other to solve problems, working in parallel to achieve a common goal. Coordination of processor activities requires that they have some means of communicating with one

another. System synchronization requirements determine whether the processors are designed using tightly coupled or loosely coupled communication methods.

Tightly coupled multiprocessors share a single centralized memory, which requires that an operating system must synchronize processes very carefully to assure protection. This type of coupling is typically used for multiprocessor systems consisting of 16 or fewer processors. *Symmetric multiprocessors* (*SMPs*) are a popular form of tightly coupled architecture. These systems have multiple processors that share memory and I/O devices. All processors perform the same functions, with the processing load being distributed among all of them.

Loosely coupled multiprocessors have a physically distributed memory, and are also known as *distributed systems*. Distributed systems can be viewed in two different ways. A distributed collection of workstations on a LAN, each with its own operating system, is typically referred to as a *networked system*. These systems were motivated by a need for multiple computers to share resources. A network operating system includes the necessary provisions, such as remote command execution, remote file access, and remote login, to attach machines to the network. User processes also have the ability to communicate over the network with processes on other machines. Network file systems are one of the most important applications of networked systems. These allow multiple machines to share one logical file system, although the machines are located in different geographical locations and may have different architectures and unrelated operating systems. Synchronization among these systems is an important issue, but communication is even more important, because this communication may occur over large networked distances. Although networked systems may be distributed over geographical areas, they are not considered true distributed systems.

A truly distributed system differs from a network of workstations in one significant way: A distributed operating system runs concurrently on all of the machines, presenting to the user an image of one single machine. In contrast, in a networked system, the user is aware of the existence of different machines. Transparency, therefore, is an important issue in distributed systems. The user should not be required to use different names for files simply because they reside in different locations, provide different commands for different machines, or perform any other interaction dependent solely upon machine location.

For the most part, operating systems for multiprocessors need not differ significantly from those for uniprocessor systems. Scheduling is one of the main differences, however, because multiple CPUs must be kept busy. If scheduling is not done properly, the inherent advantages of the multiprocessor parallelism are not fully realized. In particular, if the operating system does not provide the proper tools to exploit parallelism, performance will suffer.

Real-time systems, as we have mentioned, require specially designed operating systems. Real-time as well as embedded systems require an operating system of minimal size and minimal resource utilization. Wireless networks, which combine the compactness of embedded systems with issues characteristic of networked systems, have also motivated innovations in operating systems design.

Operating Systems for Personal Computers

Operating systems for personal computers have a different goal than those for larger systems. Whereas larger systems want to provide for excellent performance and hardware utilization (while still making the system easy to use), operating systems for personal computers have one main objective: make the system user friendly.

When Intel came out with the 8080 microprocessor in 1974, the company asked Gary Kildall to write an operating system. Kildall built a controller for a floppy disk, hooked the disk to the 8080, and wrote the software for the operating system to control the system. Kildall called this disk-based operating system *CP/M* (*Control Program for Microcomputers*). The *BIOS* (*basic input/output system*) allowed CP/M to be exported to different types of PCs easily because it provided the necessary interactions with input and output devices. Because the I/O devices are the most likely components to vary from system to system, by packaging the interfaces for these devices into one module, the actual operating systems could remain the same for various machines. Only the BIOS had to be altered.

Intel erroneously assumed disk-based machines had a bleak future. After deciding not to use this new operating system, Intel gave Kildall the rights to CP/M. In 1980, IBM needed an operating system for the IBM PC. Although IBM approached Kildall first, the deal ended up going to Microsoft, which had purchased a disk-based operating system named *QDOS* (*Quick and Dirty Operating System*) from the Seattle Computer Products Company for $15,000. The software was renamed *MS-DOS*, and the rest is history.

Operating systems for early personal computers operated on commands typed from the keyboard. Alan Key, inventor of the *GUI* (*graphical user interface*), and Doug Engelbart, inventor of the mouse, both of Xerox Palo Alto Research Center, changed the face of operating systems forever when their ideas were incorporated into operating systems. Through their efforts, command prompts were replaced by windows, icons, and drop-down menus. Microsoft popularized these ideas (but did not invent them) through its Windows series of operating systems: Windows 1.x, 2.x, 3.x, 95, 98, ME, NT2000, and XP. The Macintosh graphical operating system, *MacOS*, which preceded the Windows GUI by several years, has gone through numerous versions as well. Unix is gaining popularity in the personal computer world through Linux and OpenBSD. There are many other disk operating systems (such as DR DOS, PC DOS, and OS/2), but none are as popular as Windows and the numerous variants of Unix.

8.2.2 Operating System Design

Because the single most important piece of software used by a computer is its operating system, considerable care must be given to its design. The operating system controls the basic functions of the computer, including memory management and I/O, not to mention the "look and feel" of the interface. An operating system differs from most other software in that it is *event driven*, meaning it performs tasks in response to commands, application programs, I/O devices, and interrupts.

Four main factors drive operating system design: performance, power, cost, and compatibility. By now, you should have a feeling for what an operating system is, but there are many differing views regarding what an operating system should be, as evidenced by the various operating systems available today. Most operating systems have similar interfaces, but vary greatly in how tasks are carried out. Some operating systems are minimalistic in design, choosing to cover only the most basic functions, whereas others try to include every conceivable feature. Some have superior interfaces but lack in other areas, whereas others are superior in memory management and I/O, but fall short in the area of user-friendliness. No single operating system is superior in all respects.

Two components are crucial in operating system design: the kernel and the system programs. The *kernel* is the core of the operating system. It is used by the process manager, the scheduler, the resource manager, and the I/O manager. The kernel is responsible for scheduling, synchronization, protection/security, memory management, and dealing with interrupts. It has primary control of system hardware, including interrupts, control registers, status words, and timers. It loads all device drivers, provides common utilities, and coordinates all I/O activity. The kernel must know the specifics of the hardware to combine all of these pieces into a working system.

The two extremes of kernel design are *microkernel* architectures and *monolithic* kernels. Microkernels provide rudimentary operating system functionality, relying on other modules to perform specific tasks, thus moving many typical operating system services into user space. This permits many services to be restarted or reconfigured without restarting the entire operating system. Microkernels provide security, because services running at the user level have restricted access to system resources. Microkernels can be customized and ported to other hardware more easily than monolithic kernels. However, additional communication between the kernel and the other modules is necessary, often resulting in a slower and less efficient system. Key features of microkernel design are its smaller size, easy portability, and the array of services that run a layer above the kernel instead of in the kernel itself. Microkernel development has been significantly encouraged by the growth in SMP and other multiprocessor systems. Examples of microkernel operating systems include Windows 2000, Mach, and QNX.

Monolithic kernels provide all of their essential functionality through a single process. Consequently, they are significantly larger than microkernels. Typically targeted for specific hardware, monolithic kernels interact directly with the hardware, so they can be optimized more easily than can microkernel operating systems. It is for this reason that monolithic kernels are not easily portable. Examples of monolithic kernel operating systems include Linux, MacOS, and DOS.

Because an operating system consumes resources, in addition to managing them, designers must consider the overall size of the finished product. For example, Sun Microsystem's Solaris requires 8MB of disk space for a full installation; Windows 2000 requires about twice that amount. These statistics attest to the explosion of operating system functionality over the past two decades. MS-DOS 1.0 fit comfortably onto a single 100KB floppy diskette.

8.2.3 Operating System Services

Throughout the preceding discussion of operating system architecture, we mentioned some of the most important services that operating systems provide. The operating system oversees all critical system management tasks, including memory management, process management, protection, and interaction with I/O devices. In its role as an interface, the operating system determines how the user interacts with the computer, serving as a buffer between the user and the hardware. Each of these functions is an important factor in determining overall system performance and usability. In fact, sometimes we are willing to accept reduced performance if the system is easy to use. Nowhere is this tradeoff more apparent than in the area of graphical user interfaces.

The Human Interface

The operating system provides a layer of abstraction between the user and the hardware of the machine. Neither users nor applications see the hardware directly, as the operating system provides an interface to hide the details of the bare machine. Operating systems provide three basic interfaces, each providing a different view for a particular individual. Hardware developers are interested in the operating system as an interface to the hardware. Applications developers view the operating system as an interface to various application programs and services. Ordinary users are most interested in the graphical interface, which is the interface most commonly associated with the term *interface*.

Operating system user interfaces can be divided into two general categories: *command line interfaces* and *graphical user interfaces* (GUIs). Command line interfaces provide a prompt at which the user enters various commands, including those for copying files, deleting files, providing a directory listing, and manipulating the directory structure. Command line interfaces require the user to know the syntax of the system, which is often too complicated for the average user. However, for those who have mastered a particular command vocabulary, tasks are performed more efficiently with direct commands as opposed to using a graphical interface. GUIs, on the other hand, provide a more accessible interface for the casual user. Modern GUIs consist of windows placed on desktops. They include features such as icons and other graphical representations of files that are manipulated using a mouse. Examples of command line interfaces include Unix shells and DOS. Examples of GUIs include the various flavors of Microsoft Windows and MacOS. The decreasing cost of equipment, especially processors and memory, has made it practical to add GUIs to many other operating systems. Of particular interest is the generic X Window System provided with many Unix operating systems.

The user interface is a program, or small set of programs, that constitutes the *display manager*. This module is normally separate from the core operating system functions found in the kernel of the operating system. Most modern operating systems create an overall operating system package with modules for interfacing, handling files, and other applications that are tightly bound with the kernel. The

manner in which these modules are linked with one another is a defining characteristic of today's operating systems.

Process Management

Process management rests at the heart of operating system services. It includes everything from creating processes (setting up the appropriate structures to store information about each one), to scheduling processes' use of various resources, to deleting processes and cleaning up after their termination. The operating system keeps track of each process, its status (which includes the values of variables, the contents of CPU registers, and the actual state—running, ready, or waiting—of the process), the resources it is using, and those that it requires. The operating system maintains a watchful eye on the activities of each process to prevent *synchronization problems*, which arise when concurrent processes have access to shared resources. These activities must be monitored carefully to avoid inconsistencies in the data and accidental interference.

At any given time, the kernel is managing a collection of processes, consisting of user processes and system processes. Most processes are independent of each other. However, in the event that they need to interact to achieve a common goal, they rely on the operating system to facilitate their interprocess communication tasks.

Process scheduling is a large part of the operating system's normal routine. First, the operating system must determine which processes to admit to the system (often called *long-term scheduling*). Then it must determine which process will be granted the CPU at any given instant (*short-term scheduling*). To perform short-term scheduling, the operating system maintains a list of ready processes, so it can differentiate between processes that are waiting on resources and those that are ready to be scheduled and run. If a running process needs I/O or other resources, it voluntarily relinquishes the CPU and places itself in a waiting list, and another process is scheduled for execution. This sequence of events constitutes a *context switch*.

During a context switch, all pertinent information about the currently executing process is saved, so that when that process resumes execution, it can be restored to the exact state in which it was interrupted. Information saved during a context switch includes the contents of all CPU registers, page tables, and other information associated with virtual memory. Once this information is safely tucked away, a previously interrupted process (the one preparing to use the CPU) is restored to its exact state prior to its interruption. (New processes, of course, have no previous state to restore.)

A process can give up the CPU in two ways. In *nonpreemptive scheduling,* a process relinquishes the CPU voluntarily (possibly because it needs another unscheduled resource). However, if the system is set up with time slicing, the process might be taken from a running state and placed into a waiting state by the operating system. This is called *preemptive scheduling* because the process is preempted and the CPU is taken away. Preemption also occurs when processes are

scheduled and interrupted according to priority. For example, if a low priority job is running and a high priority job needs the CPU, the low priority job is placed in the ready queue (a context switch is performed), allowing the high priority job to run immediately.

The operating system's main task in process scheduling is to determine which process should be next in line for the CPU. Factors affecting scheduling decisions include CPU utilization, throughput, turnaround time, waiting time, and response time. Short-term scheduling can be done in a number of ways. The approaches include first-come, first-served (FCFS), shortest job first (SJF), round robin, and priority scheduling. In *first-come*, *first-served scheduling*, processes are allocated processor resources in the order in which they are requested. Control of the CPU is relinquished when the executing process terminates. FCFS scheduling is a nonpreemptive algorithm that has the advantage of being easy to implement. However, it is unsuitable for systems that support multiple users because there is a high variance in the average time a process must wait to use the CPU. In addition, a process could monopolize the CPU, causing inordinate delays in the execution of other pending processes.

In *shortest job first scheduling*, the process with the shortest execution time takes priority over all others in the system. SJF is a provably optimal scheduling algorithm. The main trouble with it is that there is no way of knowing in advance exactly how long a job is going to run. Systems that employ shortest job first apply some heuristics in making "guesstimates" of job run time, but these heuristics are far from perfect. Shortest job first can be nonpreemptive or preemptive. (The preemptive version is often called *shortest remaining time first*.)

Round robin scheduling is an equitable and simple preemptive scheduling scheme. Each process is allocated a certain slice of CPU time. If the process is still running when its timeslice expires, it is swapped out through a context switch. The next process waiting in line is then awarded its own slice of CPU time. Round robin scheduling is used extensively in timesharing systems. When the scheduler employs sufficiently small timeslices, users are unaware that they are sharing the resources of the system. However, the timeslices should not be so small that the context switch time is large by comparison.

Priority scheduling associates a priority with each process. When the short-term scheduler selects a process from the ready queue, the process with the highest priority is chosen. FCFS gives equal priority to all processes. SJF gives priority to the shortest job. The foremost problem with priority scheduling is the potential for starvation, or indefinite blocking. Can you imagine how frustrating it would be to try to run a large job on a busy system when users continually submit shorter jobs that run before yours? Folklore has it that when a mainframe in a large university was halted, a job was found in the ready queue that had been trying to run for several years!

Some operating systems offer a combination of scheduling approaches. For example, a system might use a preemptive, priority-based, first-come, first-served algorithm. Highly complex operating systems that support enterprise class systems allow some degree of user control over timeslice duration, the number of allowable concurrent tasks, and assignment of priorities to different job classes.

Multitasking (allowing multiple processes to run concurrently) and *multithreading* (allowing a process to be subdivided into different threads of control) provide interesting challenges for CPU scheduling. A *thread* is the smallest schedulable unit in a system. Threads share the same execution environment as their parent process, including its CPU registers and page table. Because of this, context switching among threads generates less overhead so they can occur much faster than a context switch involving the entire process. Depending on the degree of concurrency required, it is possible to have one process with one thread, one process with multiple threads, multiple single-threaded processes, or multiple multithreaded processes. An operating system that supports multithreading must be able to handle all combinations.

Resource Management

In addition to process management, the operating system manages system resources. Because these resources are relatively expensive, it is preferable to allow them to be shared. For example, multiple processes can share one processor, multiple programs can share physical memory, and multiple users and files can share one disk. There are three resources that are of major concern to the operating system: the CPU, memory, and I/O. Access to the CPU is controlled by the scheduler. Memory and I/O access requires a different set of controls and functions.

Recall from Chapter 6 that most modern systems have some type of virtual memory that extends RAM. This implies that parts of several programs may coexist in memory, and each process must have a page table. Originally, before operating systems were designed to deal with virtual memory, the programmer implemented virtual memory using the *overlay* technique. If a program was too large to fit into memory, the programmer divided it into pieces, loading only the data and instructions necessary to run at a given moment. If new data or instructions were needed, it was up to the programmer (with some help from the compiler) to make sure the correct pieces were in memory. The programmer was responsible for managing memory. Now, operating systems have taken over that chore. The operating system translates virtual addresses to physical addresses, transfers pages to and from disk, and maintains memory page tables. The operating system also determines main memory allocation and tracks free frames. As it deallocates memory space, the operating system performs "garbage collection," which is the process of coalescing small portions of free memory into larger, more usable chunks.

In addition to processes sharing a single finite memory, they also share I/O devices. Most input and output is done at the request of an application. The operating system provides the necessary services to allow input and output to occur. It's possible for applications to handle their own I/O without using the operating system, but, in addition to duplicating effort, this presents protection and access issues. If several different processes try to use the same I/O device simultaneously, the requests must be mediated. It falls upon the operating system to perform this task. The operating system provides a generic interface to I/O through

various system calls. These calls allow an application to request an I/O service through the operating system. The operating system then calls upon device drivers that contain software implementing a standard set of functions relevant to particular I/O devices.

The operating system also manages disk files. The operating system takes care of file creation, file deletion, directory creation, and directory deletion, and also provides support for primitives that manipulate files and directories and their mapping onto secondary storage devices. Although I/O device drivers take care of many of the particular details, the operating system coordinates device driver activities that support I/O system functions.

Security and Protection

In its role as a resource and process manager, the operating system has to make sure that everything works correctly, fairly, and efficiently. Resource sharing, however, creates a multitude of exposures, such as the potential for unauthorized access or modification of data. Therefore, the operating system also serves as a resource protector, making sure "bad guys" and buggy software don't ruin things for everyone else. Concurrent processes must be protected from each other, and operating system processes must be protected from all user processes. Without this protection, a user program could potentially wipe out the operating system code for dealing with, say, interrupts. Multiuser systems require additional security services to protect both shared resources (such as memory and I/O devices) and nonshared resources (such as personal files). Memory protection safeguards against a bug in one user's program affecting other programs or a malicious program taking control of the entire system. CPU protection makes sure user programs don't get stuck in infinite loops, consuming CPU cycles needed by other jobs.

The operating system provides security services in a variety of ways. First, active processes are limited to execution within their own memory space. All requests for I/O or other resources from the process pass through the operating system, which then processes the request. The operating system executes most commands in user mode and others in kernel mode. In this way, the resources are protected against unauthorized use. The operating system also provides facilities to control user access, typically through login names and passwords. Stronger protection can be effected by restricting processes to a single subsystem or partition.

8.3 PROTECTED ENVIRONMENTS

To provide protection, multiuser operating systems guard against processes running amok in the system. Process execution must be isolated from both the operating system and other processes. Access to shared resources must be controlled and mediated to avoid conflicts. There are a number of ways in which protective