CHAPTER 2

OPERATING SYSTEM OVERVIEW

We begin our study of operating systems (OSs) with a brief history. This history is itself interesting and also serves the purpose of providing an overview of OS principles. The first section examines the objectives and functions of operating systems. Then we look at how operating systems have evolved from primitive batch systems to sophisticated multitasking, multiuser systems. The remainder of the chapter looks at the history and general characteristics of the two operating systems that serve as examples throughout this book. All of the material in this chapter is covered in greater depth later in the book.

2.1 OPERATING SYSTEM OBJECTIVES AND FUNCTIONS

An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. It can be thought of as having three objectives:

- **Convenience:** An OS makes a computer more convenient to use.
- **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.
- **Ability to evolve:** An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

Let us examine these three aspects of an OS in turn.

The Operating System as a User/Computer Interface

The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion, as depicted in Figure 2.1. The user of those applications, the end user, generally is not concerned with the details of computer hardware. Thus, the end user views a computer system in terms of a set of applications. An application can be expressed in a programming language and is developed by an application programmer. If one were to develop an application program as a set of machine instructions that is completely responsible for controlling the computer hardware, one would be faced with an overwhelmingly complex undertaking. To ease this chore, a set of system programs is provided. Some of these programs are referred to as utilities. These implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices. A programmer will make use of these facilities in developing an application, and the application, while it is running, will invoke the utilities to perform certain functions. The most important collection of system programs comprises the OS. The OS masks the details of the hardware from the programmer and provides the programmer with a convenient interface for using the system. It acts as mediator, making it easier for the programmer and for application programs to access and use those facilities and services.

Briefly, the OS typically provides services in the following areas:

- **Program development:** The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs. Typically, these services are in the form of utility programs that, while not

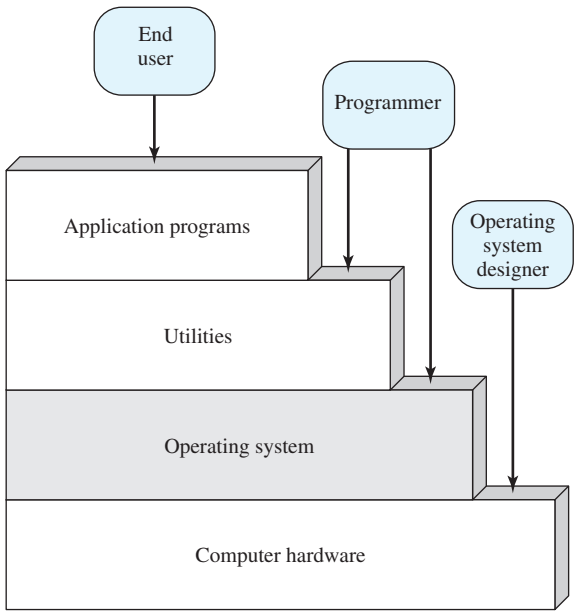

Figure 2.1 **Layers and Views of a Computer System**

strictly part of the core of the OS, are supplied with the OS and are referred to as application program development tools.

- **Program execution:** A number of steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The OS handles these scheduling duties for the user.
- **Access to I/O devices:** Each I/O device requires its own peculiar set of instructions or control signals for operation. The OS provides a uniform interface that hides these details so that programmers can access such devices using simple reads and writes.
- **Controlled access to files:** For file access, the OS must reflect a detailed understanding of not only the nature of the I/O device (disk drive, tape drive) but also the structure of the data contained in the files on the storage medium. In the case of a system with multiple users, the OS may provide protection mechanisms to control access to the files.
- **System access:** For shared or public systems, the OS controls access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.
- **Error detection and response:** A variety of errors can occur while a computer system is running. These include internal and external hardware errors, such as a memory error, or a device failure or malfunction; and various software errors, such as division by zero, attempt to access forbidden memory location,

and inability of the OS to grant the request of an application. In each case, the OS must provide a response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.

- **Accounting:** A good OS will collect usage statistics for various resources and monitor performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance. On a multiuser system, the information can be used for billing purposes.

The Operating System as Resource Manager

A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions. The OS is responsible for managing these resources.

Can we say that it is the OS that controls the movement, storage, and processing of data? From one point of view, the answer is yes: By managing the computer's resources, the OS is in control of the computer's basic functions. But this control is exercised in a curious way. Normally, we think of a control mechanism as something external to that which is controlled, or at least as something that is a distinct and separate part of that which is controlled. (For example, a residential heating system is controlled by a thermostat, which is separate from the heat-generation and heat-distribution apparatus.) This is not the case with the OS, which as a control mechanism is unusual in two respects:

- The OS functions in the same way as ordinary computer software; that is, it is a program or suite of programs executed by the processor.
- The OS frequently relinquishes control and must depend on the processor to allow it to regain control.

Like other computer programs, the OS provides instructions for the processor. The key difference is in the intent of the program. The OS directs the processor in the use of the other system resources and in the timing of its execution of other programs. But in order for the processor to do any of these things, it must cease executing the OS program and execute other programs. Thus, the OS relinquishes control for the processor to do some "useful" work and then resumes control long enough to prepare the processor to do the next piece of work. The mechanisms involved in all this should become clear as the chapter proceeds.

Figure 2.2 suggests the main resources that are managed by the OS. A portion of the OS is in main memory. This includes the **kernel**, or **nucleus**, which contains the most frequently used functions in the OS and, at a given time, other portions of the OS currently in use. The remainder of main memory contains user programs and data. The allocation of this resource (main memory) is controlled jointly by the OS and memory management hardware in the processor, as we shall see. The OS decides when an I/O device can be used by a program in execution and controls access to and use of files. The processor itself is a resource, and the OS must determine how much processor time is to be devoted to the execution of a particular user program. In the case of a multiple-processor system, this decision must span all of the processors.

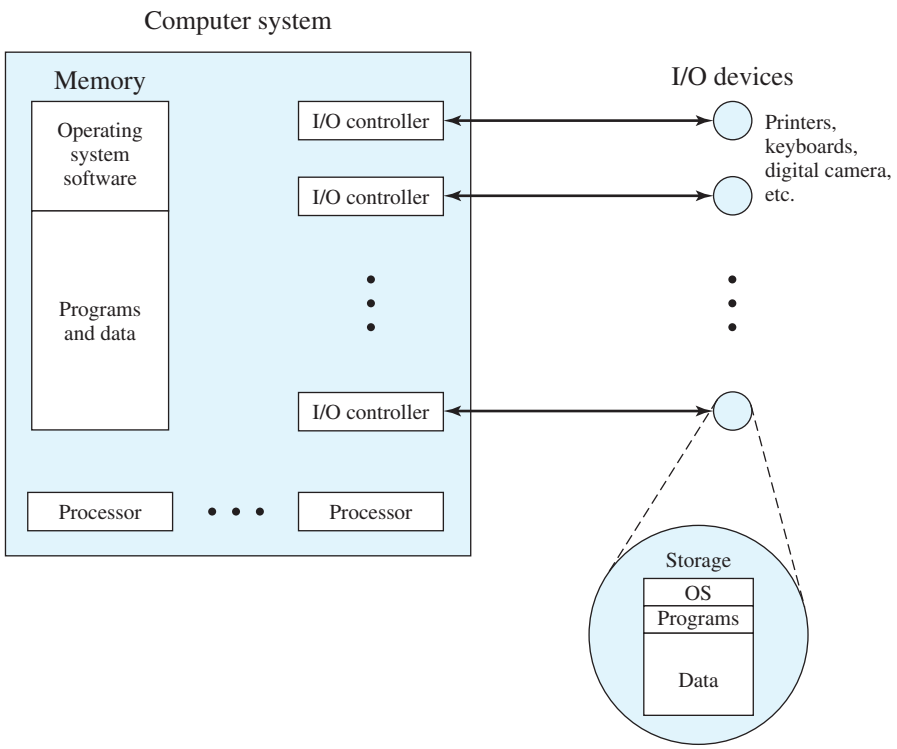

Figure 2.2 **The Operating System as Resource Manager**

Ease of Evolution of an Operating System

A major operating system will evolve over time for a number of reasons:

- **Hardware upgrades plus new types of hardware:** For example, early versions of UNIX and the Macintosh operating system did not employ a paging mechanism because they were run on processors without paging hardware.[1] Subsequent versions of these operating systems were modified to exploit paging capabilities. Also, the use of graphics terminals and page-mode terminals instead of line-at-a-time scroll mode terminals affects OS design. For example, a graphics terminal typically allows the user to view several applications at the same time through "windows" on the screen. This requires more sophisticated support in the OS.
- **New services:** In response to user demand or in response to the needs of system managers, the OS expands to offer new services. For example, if it is found to be difficult to maintain good performance for users with existing tools, new measurement and control tools may be added to the OS.
- **Fixes:** Any OS has faults. These are discovered over the course of time and fixes are made. Of course, the fix may introduce new faults.

[1]Paging is introduced briefly later in this chapter and is discussed in detail in Chapter 7.

The need to change an OS regularly places certain requirements on its design. An obvious statement is that the system should be modular in construction, with clearly defined interfaces between the modules, and that it should be well documented. For large programs, such as the typical contemporary OS, what might be referred to as straightforward modularization is inadequate [DENN80a]. That is, much more must be done than simply partitioning a program into modules. We return to this topic later in this chapter.

2.2 THE EVOLUTION OF OPERATING SYSTEMS

In attempting to understand the key requirements for an OS and the significance of the major features of a contemporary OS, it is useful to consider how operating systems have evolved over the years.

Serial Processing

With the earliest computers, from the late 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS. These computers were run from a console consisting of display lights, toggle switches, some form of input device, and a printer. Programs in machine code were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the lights. If the program proceeded to a normal completion, the output appeared on the printer.

These early systems presented two main problems:

- **Scheduling:** Most installations used a hardcopy sign-up sheet to reserve computer time. Typically, a user could sign up for a block of time in multiples of a half hour or so. A user might sign up for an hour and finish in 45 minutes; this would result in wasted computer processing time. On the other hand, the user might run into problems, not finish in the allotted time, and be forced to stop before resolving the problem.
- **Setup time:** A single program, called a **job**, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program) and then loading and linking together the object program and common functions. Each of these steps could involve mounting or dismounting tapes or setting up card decks. If an error occurred, the hapless user typically had to go back to the beginning of the setup sequence. Thus, a considerable amount of time was spent just in setting up the program to run.

This mode of operation could be termed *serial processing*, reflecting the fact that users have access to the computer in series. Over time, various system software tools were developed to attempt to make serial processing more efficient. These include libraries of common functions, linkers, loaders, debuggers, and I/O driver routines that were available as common software for all users.

Simple Batch Systems

Early computers were very expensive, and therefore it was important to maximize processor utilization. The wasted time due to scheduling and setup time was unacceptable.

To improve utilization, the concept of a batch operating system was developed. It appears that the first batch operating system (and the first OS of any kind) was developed in the mid-1950s by General Motors for use on an IBM 701 [WEIZ81]. The concept was subsequently refined and implemented on the IBM 704 by a number of IBM customers. By the early 1960s, a number of vendors had developed batch operating systems for their computer systems. IBSYS, the IBM operating system for the 7090/7094 computers, is particularly notable because of its widespread influence on other systems.

The central idea behind the simple batch-processing scheme is the use of a piece of software known as the **monitor**. With this type of OS, the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor. Each program is constructed to branch back to the monitor when it completes processing, at which point the monitor automatically begins loading the next program.

To understand how this scheme works, let us look at it from two points of view: that of the monitor and that of the processor.

- **Monitor point of view:** The monitor controls the sequence of events. For this to be so, much of the monitor must always be in main memory and available for execution (Figure 2.3). That portion is referred to as the **resident monitor**. The rest of the monitor consists of utilities and common functions that are loaded as subroutines to the user program at the beginning of any job that requires them. The monitor reads in jobs one at a time from the input device (typically a card reader or magnetic tape drive). As it is read in, the current job is placed in the user program area, and control is passed to this job. When the job is completed, it returns control to the monitor, which immediately reads in

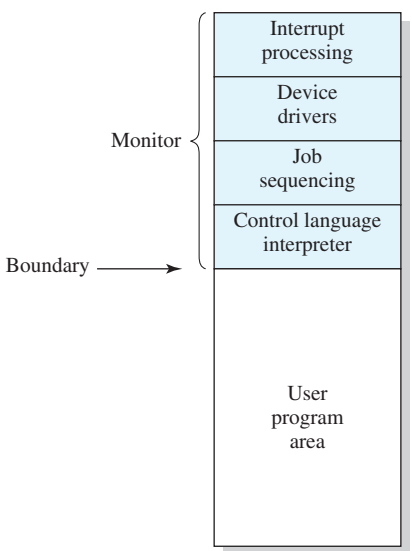

Figure 2.3 Memory Layout for a Resident Monitor

the next job. The results of each job are sent to an output device, such as a printer, for delivery to the user.

- **Processor point of view:** At a certain point, the processor is executing instructions from the portion of main memory containing the monitor. These instructions cause the next job to be read into another portion of main memory. Once a job has been read in, the processor will encounter a branch instruction in the monitor that instructs the processor to continue execution at the start of the user program. The processor will then execute the instructions in the user program until it encounters an ending or error condition. Either event causes the processor to fetch its next instruction from the monitor program. Thus the phrase "control is passed to a job" simply means that the processor is now fetching and executing instructions in a user program, and "control is returned to the monitor" means that the processor is now fetching and executing instructions from the monitor program.

The monitor performs a scheduling function: A batch of jobs is queued up, and jobs are executed as rapidly as possible, with no intervening idle time. The monitor improves job setup time as well. With each job, instructions are included in a primitive form of **job control language** (JCL). This is a special type of programming language used to provide instructions to the monitor. A simple example is that of a user submitting a program written in the programming language FORTRAN plus some data to be used by the program. All FORTRAN instructions and data are on a separate punched card or a separate record on tape. In addition to FORTRAN and data lines, the job includes job control instructions, which are denoted by the beginning $. The overall format of the job looks like this:

```
$JOB
$FTN
.
.          } FORTRAN instructions
.
$LOAD
$RUN
.
.          } Data
.
$END
```

To execute this job, the monitor reads the $FTN line and loads the appropriate language compiler from its mass storage (usually tape). The compiler translates the user's program into object code, which is stored in memory or mass storage. If it is stored in memory, the operation is referred to as "compile, load, and go." If it is stored on tape, then the $LOAD instruction is required. This instruction is read by the monitor, which regains control after the compile operation. The monitor invokes the loader, which loads the object program into memory (in place of the compiler)

and transfers control to it. In this manner, a large segment of main memory can be shared among different subsystems, although only one such subsystem could be executing at a time.

During the execution of the user program, any input instruction causes one line of data to be read. The input instruction in the user program causes an input routine that is part of the OS to be invoked. The input routine checks to make sure that the program does not accidentally read in a JCL line. If this happens, an error occurs and control transfers to the monitor. At the completion of the user job, the monitor will scan the input lines until it encounters the next JCL instruction. Thus, the system is protected against a program with too many or too few data lines.

The monitor, or batch operating system, is simply a computer program. It relies on the ability of the processor to fetch instructions from various portions of main memory to alternately seize and relinquish control. Certain other hardware features are also desirable:

- **Memory protection:** While the user program is executing, it must not alter the memory area containing the monitor. If such an attempt is made, the processor hardware should detect an error and transfer control to the monitor. The monitor would then abort the job, print out an error message, and load in the next job.
- **Timer:** A timer is used to prevent a single job from monopolizing the system. The timer is set at the beginning of each job. If the timer expires, the user program is stopped, and control returns to the monitor.
- **Privileged instructions:** Certain machine level instructions are designated privileged and can be executed only by the monitor. If the processor encounters such an instruction while executing a user program, an error occurs causing control to be transferred to the monitor. Among the privileged instructions are I/O instructions, so that the monitor retains control of all I/O devices. This prevents, for example, a user program from accidentally reading job control instructions from the next job. If a user program wishes to perform I/O, it must request that the monitor perform the operation for it.
- **Interrupts:** Early computer models did not have this capability. This feature gives the OS more flexibility in relinquishing control to and regaining control from user programs.

Considerations of memory protection and privileged instructions lead to the concept of modes of operation. A user program executes in a **user mode**, in which certain areas of memory are protected from the user's use and in which certain instructions may not be executed. The monitor executes in a system mode, or what has come to be called **kernel mode**, in which privileged instructions may be executed and in which protected areas of memory may be accessed.

Of course, an OS can be built without these features. But computer vendors quickly learned that the results were chaos, and so even relatively primitive batch operating systems were provided with these hardware features.

With a batch operating system, processor time alternates between execution of user programs and execution of the monitor. There have been two sacrifices: Some main memory is now given over to the monitor and some processor time is consumed by the monitor. Both of these are forms of overhead. Despite this overhead, the simple batch system improves utilization of the computer.

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
Total	31 μs

$$\text{Percent CPU Utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

Figure 2.4 **System Utilization Example**

Multiprogrammed Batch Systems

Even with the automatic job sequencing provided by a simple batch operating system, the processor is often idle. The problem is that I/O devices are slow compared to the processor. Figure 2.4 details a representative calculation. The calculation concerns a program that processes a file of records and performs, on average, 100 machine instructions per record. In this example the computer spends over 96% of its time waiting for I/O devices to finish transferring data to and from the file. Figure 2.5a illustrates this situation, where we have a single program, referred to

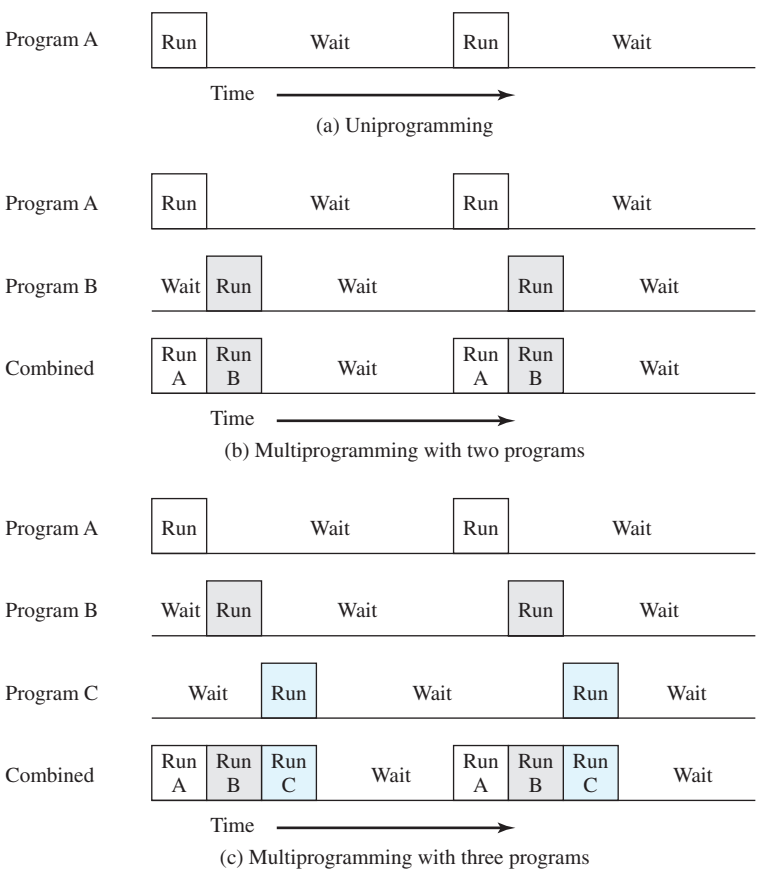

Figure 2.5 **Multiprogramming Example**

Table 2.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

as uniprogramming. The processor spends a certain amount of time executing, until it reaches an I/O instruction. It must then wait until that I/O instruction concludes before proceeding.

This inefficiency is not necessary. We know that there must be enough memory to hold the OS (resident monitor) and one user program. Suppose that there is room for the OS and two user programs. When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O (Figure 2.5b). Furthermore, we might expand memory to hold three, four, or more programs and switch among all of them (Figure 2.5c). The approach is known as **multiprogramming**, or **multitasking**. It is the central theme of modern operating systems.

To illustrate the benefit of multiprogramming, we give a simple example. Consider a computer with 250 Mbytes of available memory (not used by the OS), a disk, a terminal, and a printer. Three programs, JOB1, JOB2, and JOB3, are submitted for execution at the same time, with the attributes listed in Table 2.1. We assume minimal processor requirements for JOB2 and JOB3 and continuous disk and printer use by JOB3. For a simple batch environment, these jobs will be executed in sequence. Thus, JOB1 completes in 5 minutes. JOB2 must wait until the 5 minutes are over and then completes 15 minutes after that. JOB3 begins after 20 minutes and completes at 30 minutes from the time it was initially submitted. The average resource utilization, throughput, and response times are shown in the uniprogramming column of Table 2.2. Device-by-device utilization is illustrated in Figure 2.6a. It is evident that there is gross underutilization for all resources when averaged over the required 30-minute time period.

Table 2.2 Effects of Multiprogramming on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

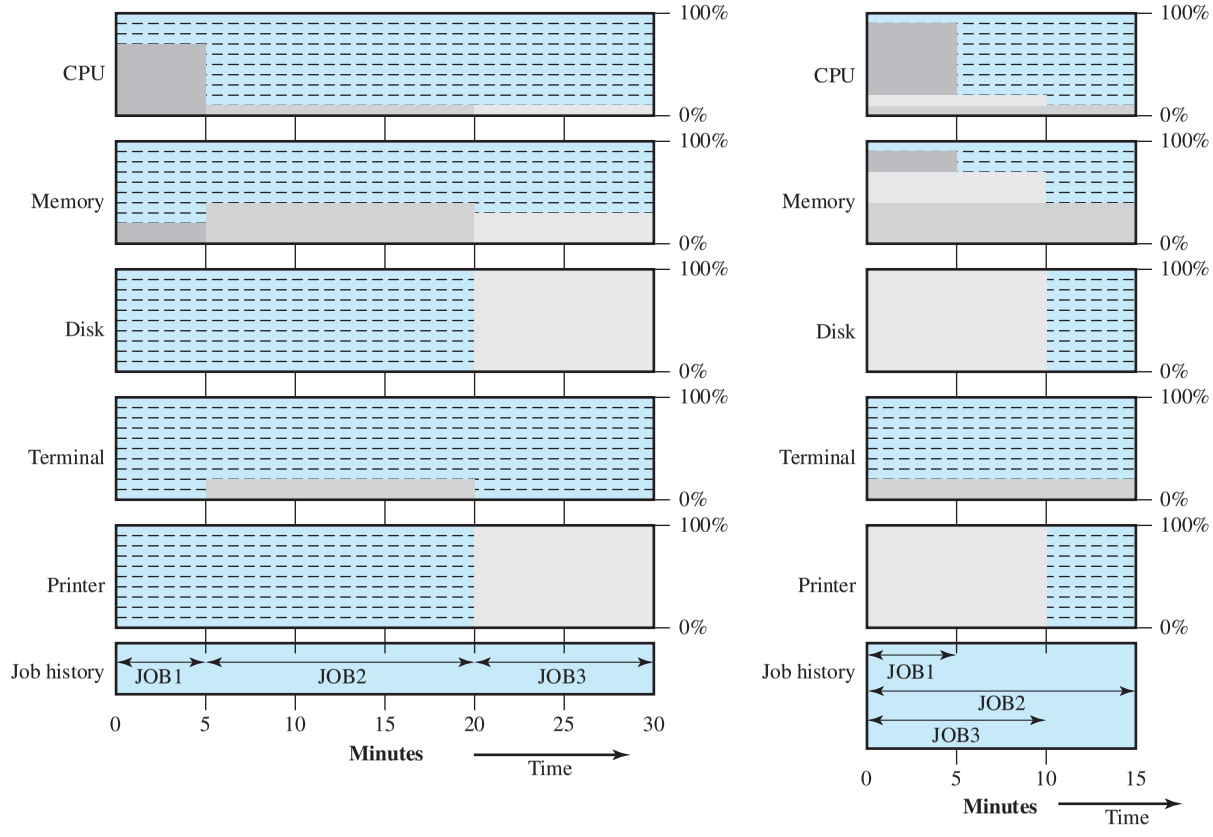

Figure 2.6 **Utilization Histograms**

Now suppose that the jobs are run concurrently under a multiprogramming operating system. Because there is little resource contention between the jobs, all three can run in nearly minimum time while coexisting with the others in the computer (assuming that JOB2 and JOB3 are allotted enough processor time to keep their input and output operations active). JOB1 will still require 5 minutes to complete, but at the end of that time, JOB2 will be one-third finished and JOB3 half finished. All three jobs will have finished within 15 minutes. The improvement is evident when examining the multiprogramming column of Table 2.2, obtained from the histogram shown in Figure 2.6b.

As with a simple batch system, a multiprogramming batch system must rely on certain computer hardware features. The most notable additional feature that is useful for multiprogramming is the hardware that supports I/O interrupts and DMA (direct memory access). With interrupt-driven I/O or DMA, the processor can issue an I/O command for one job and proceed with the execution of another job while the I/O is carried out by the device controller. When the I/O operation is complete, the processor is interrupted and control is passed to an interrupt-handling program in the OS. The OS will then pass control to another job.

Multiprogramming operating systems are fairly sophisticated compared to single-program, or **uniprogramming**, systems. To have several jobs ready to run, they must be kept in main memory, requiring some form of **memory management**. In addition, if several jobs are ready to run, the processor must decide which one to run, this decision requires an algorithm for scheduling. These concepts are discussed later in this chapter.

Time-Sharing Systems

With the use of multiprogramming, batch processing can be quite efficient. However, for many jobs, it is desirable to provide a mode in which the user interacts directly with the computer. Indeed, for some jobs, such as transaction processing, an interactive mode is essential.

Today, the requirement for an interactive computing facility can be, and often is, met by the use of a dedicated personal computer or workstation. That option was not available in the 1960s, when most computers were big and costly. Instead, time sharing was developed.

Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs. In this latter case, the technique is referred to as **time sharing**, because processor time is shared among multiple users. In a time-sharing system, multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation. Thus, if there are n users actively requesting service at one time, each user will only see on the average $1/n$ of the effective computer capacity, not counting OS overhead. However, given the relatively slow human reaction time, the response time on a properly designed system should be similar to that on a dedicated computer.

Both batch processing and time sharing use multiprogramming. The key differences are listed in Table 2.3.

Table 2.3 Batch Multiprogramming versus Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

One of the first time-sharing operating systems to be developed was the Compatible Time-Sharing System (CTSS) [CORB62], developed at MIT by a group known as Project MAC (Machine-Aided Cognition, or Multiple-Access Computers). The system was first developed for the IBM 709 in 1961 and later transferred to an IBM 7094.

Compared to later systems, CTSS is primitive. The system ran on a computer with 32,000 36-bit words of main memory, with the resident monitor consuming 5000 of that. When control was to be assigned to an interactive user, the user's program and data were loaded into the remaining 27,000 words of main memory. A program was always loaded to start at the location of the 5000th word; this simplified both the monitor and memory management. A system clock generated interrupts at a rate of approximately one every 0.2 seconds. At each clock interrupt, the OS regained control and could assign the processor to another user. This technique is known as **time slicing**. Thus, at regular time intervals, the current user would be preempted and another user loaded in. To preserve the old user program status for later resumption, the old user programs and data were written out to disk before the new user programs and data were read in. Subsequently, the old user program code and data were restored in main memory when that program was next given a turn.

To minimize disk traffic, user memory was only written out when the incoming program would overwrite it. This principle is illustrated in Figure 2.7. Assume that there are four interactive users with the following memory requirements, in words:

- JOB1: 15,000
- JOB2: 20,000
- JOB3: 5000
- JOB4: 10,000

Initially, the monitor loads JOB1 and transfers control to it (a). Later, the monitor decides to transfer control to JOB2. Because JOB2 requires more memory than JOB1, JOB1 must be written out first, and then JOB2 can be loaded (b). Next, JOB3 is loaded in to be run. However, because JOB3 is smaller than JOB2, a portion of JOB2 can remain in memory, reducing disk write time (c). Later, the monitor decides to transfer control back to JOB1. An additional portion of JOB2 must be written out when JOB1 is loaded back into memory (d). When JOB4 is loaded, part of JOB1 and the portion of JOB2 remaining in memory are retained (e). At this point, if either JOB1 or JOB2 is activated, only a partial load will be required. In this example, it is JOB2 that runs next. This requires that JOB4 and the remaining resident portion of JOB1 be written out and that the missing portion of JOB2 be read in (f).

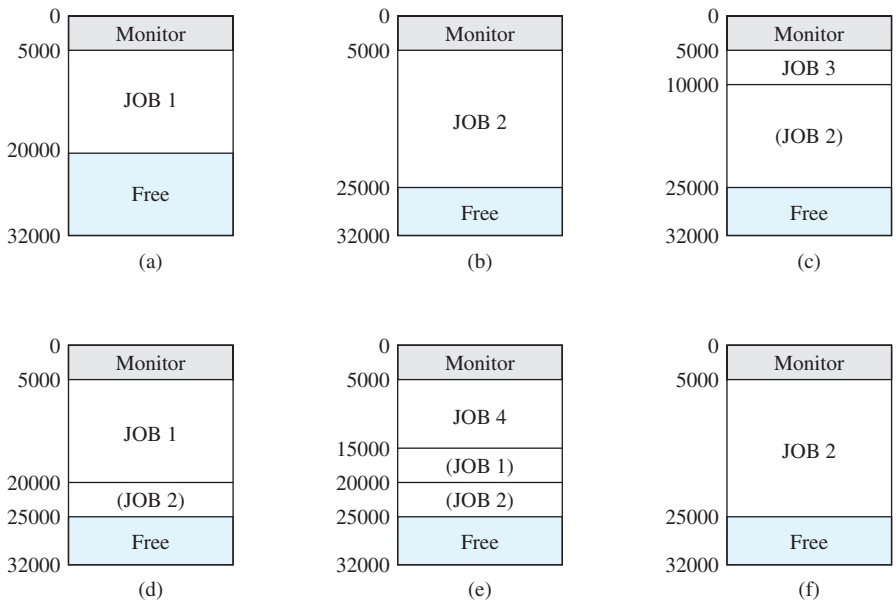

Figure 2.7 CTSS Operation

The CTSS approach is primitive compared to present-day time sharing, but it worked. It was extremely simple, which minimized the size of the monitor. Because a job was always loaded into the same locations in memory, there was no need for relocation techniques at load time (discussed subsequently). The technique of only writing out what was necessary minimized disk activity. Running on the 7094, CTSS supported a maximum of 32 users.

Time sharing and multiprogramming raise a host of new problems for the OS. If multiple jobs are in memory, then they must be protected from interfering with each other by, for example, modifying each other's data. With multiple interactive users, the file system must be protected so that only authorized users have access to a particular file. The contention for resources, such as printers and mass storage devices, must be handled. These and other problems, with possible solutions, will be encountered throughout this text.

2.3 MAJOR ACHIEVEMENTS

Operating systems are among the most complex pieces of software ever developed. This reflects the challenge of trying to meet the difficult and in some cases competing objectives of convenience, efficiency, and ability to evolve. [DENN80a] proposes that there have been five major theoretical advances in the development of operating systems:

- Processes
- Memory management

- Information protection and security
- Scheduling and resource management
- System structure

Each advance is characterized by principles, or abstractions, developed to meet difficult practical problems. Taken together, these five areas span many of the key design and implementation issues of modern operating systems. The brief review of these five areas in this section serves as an overview of much of the rest of the text.

The Process

The concept of process is fundamental to the structure of operating systems. This term was first used by the designers of Multics in the 1960s [DALE68]. It is a somewhat more general term than job. Many definitions have been given for the term *process*, including

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

This concept should become clearer as we proceed.

Three major lines of computer system development created problems in timing and synchronization that contributed to the development of the concept of the process: multiprogramming batch operation, time sharing, and real-time transaction systems. As we have seen, multiprogramming was designed to keep the processor and I/O devices, including storage devices, simultaneously busy to achieve maximum efficiency. The key mechanism is this: In response to signals indicating the completion of I/O transactions, the processor is switched among the various programs residing in main memory.

A second line of development was general-purpose time sharing. Here, the key design objective is to be responsive to the needs of the individual user and yet, for cost reasons, be able to support many users simultaneously. These goals are compatible because of the relatively slow reaction time of the user. For example, if a typical user needs an average of 2 seconds of processing time per minute, then close to 30 such users should be able to share the same system without noticeable interference. Of course, OS overhead must be factored into such calculations.

Another important line of development has been real-time transaction processing systems. In this case, a number of users are entering queries or updates against a database. An example is an airline reservation system. The key difference between the transaction processing system and the time-sharing system is that the former is limited to one or a few applications, whereas users of a time-sharing system can engage in program development, job execution, and the use of various applications. In both cases, system response time is paramount.

The principal tool available to system programmers in developing the early multiprogramming and multiuser interactive systems was the interrupt. The activity

of any job could be suspended by the occurrence of a defined event, such as an I/O completion. The processor would save some sort of context (e. g., program counter and other registers) and branch to an interrupt-handling routine, which would determine the nature of the interrupt, process the interrupt, and then resume user processing with the interrupted job or some other job.

The design of the system software to coordinate these various activities turned out to be remarkably difficult. With many jobs in progress at any one time, each of which involved numerous steps to be performed in sequence, it became impossible to analyze all of the possible combinations of sequences of events. In the absence of some systematic means of coordination and cooperation among activities, programmers resorted to ad hoc methods based on their understanding of the environment that the OS had to control. These efforts were vulnerable to subtle programming errors whose effects could be observed only when certain relatively rare sequences of actions occurred. These errors were difficult to diagnose because they needed to be distinguished from application software errors and hardware errors. Even when the error was detected, it was difficult to determine the cause, because the precise conditions under which the errors appeared were very hard to reproduce. In general terms, there are four main causes of such errors [DENN80a]:

- **Improper synchronization:** It is often the case that a routine must be suspended awaiting an event elsewhere in the system. For example, a program that initiates an I/O read must wait until the data are available in a buffer before proceeding. In such cases, a signal from some other routine is required. Improper design of the signaling mechanism can result in signals being lost or duplicate signals being received.
- **Failed mutual exclusion:** It is often the case that more than one user or program will attempt to make use of a shared resource at the same time. For example, two users may attempt to edit the same file at the same time. If these accesses are not controlled, an error can occur. There must be some sort of mutual exclusion mechanism that permits only one routine at a time to perform an update against the file. The implementation of such mutual exclusion is difficult to verify as being correct under all possible sequences of events.
- **Nondeterminate program operation:** The results of a particular program normally should depend only on the input to that program and not on the activities of other programs in a shared system. But when programs share memory, and their execution is interleaved by the processor, they may interfere with each other by overwriting common memory areas in unpredictable ways. Thus, the order in which various programs are scheduled may affect the outcome of any particular program.
- **Deadlocks:** It is possible for two or more programs to be hung up waiting for each other. For example, two programs may each require two I/O devices to perform some operation (e.g., disk to tape copy). One of the programs has seized control of one of the devices and the other program has control of the other device. Each is waiting for the other program to release the desired resource. Such a deadlock may depend on the chance timing of resource allocation and release.

What is needed to tackle these problems is a systematic way to monitor and control the various programs executing on the processor. The concept of the process provides the foundation. We can think of a process as consisting of three components:

- An executable program
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program

This last element is essential. The **execution context**, or **process state**, is the internal data by which the OS is able to supervise and control the process. This internal information is separated from the process, because the OS has information not permitted to the process. The context includes all of the information that the OS needs to manage the process and that the processor needs to execute the process properly. The context includes the contents of the various processor registers, such as the program counter and data registers. It also includes information of use to the OS, such as the priority of the process and whether the process is waiting for the completion of a particular I/O event.

Figure 2.8 indicates a way in which processes may be managed. Two processes, A and B, exist in portions of main memory. That is, a block of memory is allocated to each process that contains the program, data, and context information. Each process is recorded in a process list built and maintained by the OS. The process list contains one entry for each process, which includes a pointer to the location of the block of memory that contains the process. The entry may also include part or all of the execution context of the process. The remainder of the execution context is stored elsewhere, perhaps with the process itself (as indicated in Figure 2.8) or frequently in a separate region of memory. The process index register contains the index into the process list of the process currently controlling the processor. The program counter points to the next instruction in that process to be executed. The base and limit registers define the region in memory occupied by the process: The base register is the starting address of the region of memory and the limit is the size of the region (in bytes or words). The program counter and all data references are interpreted relative to the base register and must not exceed the value in the limit register. This prevents interprocess interference.

In Figure 2.8, the process index register indicates that process B is executing. Process A was previously executing but has been temporarily interrupted. The contents of all the registers at the moment of A's interruption were recorded in its execution context. Later, the OS can perform a process switch and resume execution of process A. The process switch consists of storing the context of B and restoring the context of A. When the program counter is loaded with a value pointing into A's program area, process A will automatically resume execution.

Thus, the process is realized as a data structure. A process can either be executing or awaiting execution. The entire **state** of the process at any instant is contained in its context. This structure allows the development of powerful techniques for ensuring coordination and cooperation among processes. New features can be designed and incorporated into the OS (e.g., priority) by expanding the context to include any new information needed to support the feature. Throughout this book,

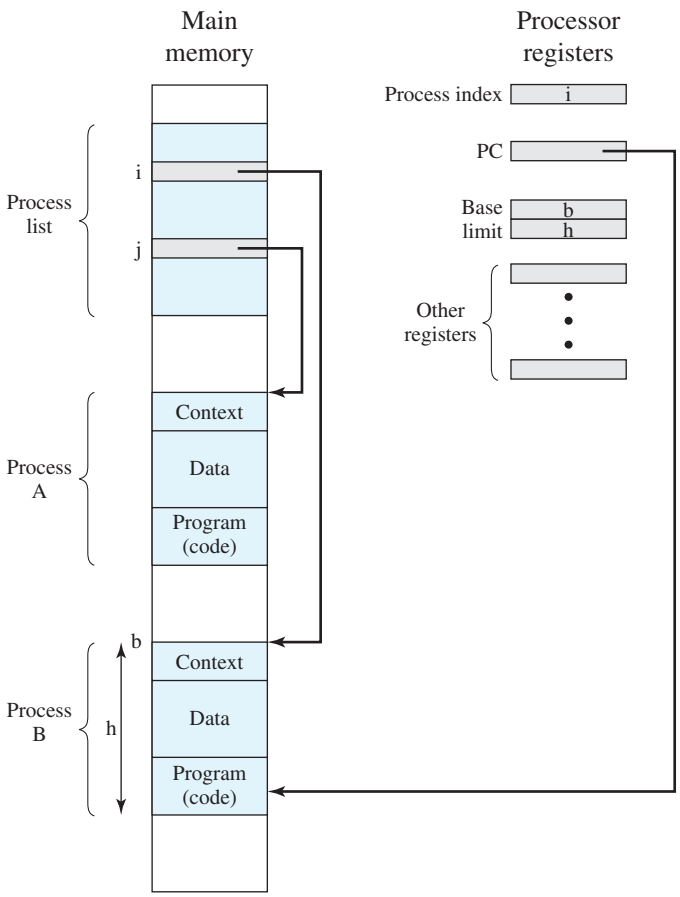

Figure 2.8 **Typical Process Implementation**

we will see a number of examples where this process structure is employed to solve the problems raised by multiprogramming and resource sharing.

Memory Management

The needs of users can be met best by a computing environment that supports modular programming and the flexible use of data. System managers need efficient and orderly control of storage allocation. The OS, to satisfy these requirements, has five principal storage management responsibilities:

- **Process isolation:** The OS must prevent independent processes from interfering with each other's memory, both data and instructions.
- **Automatic allocation and management:** Programs should be dynamically allocated across the memory hierarchy as required. Allocation should be transparent to the programmer. Thus, the programmer is relieved of concerns relating to memory limitations, and the OS can achieve efficiency by assigning memory to jobs only as needed.

- **Support of modular programming:** Programmers should be able to define program modules, and to create, destroy, and alter the size of modules dynamically.
- **Protection and access control:** Sharing of memory, at any level of the memory hierarchy, creates the potential for one program to address the memory space of another. This is desirable when sharing is needed by particular applications. At other times, it threatens the integrity of programs and even of the OS itself. The OS must allow portions of memory to be accessible in various ways by various users.
- **Long-term storage:** Many application programs require means for storing information for extended periods of time, after the computer has been powered down.

Typically, operating systems meet these requirements with virtual memory and file system facilities. The file system implements a long-term store, with information stored in named objects, called files. The file is a convenient concept for the programmer and is a useful unit of access control and protection for the OS.

Virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. Virtual memory was conceived to meet the requirement of having multiple user jobs reside in main memory concurrently, so that there would not be a hiatus between the execution of successive processes while one process was written out to secondary store and the successor process was read in. Because processes vary in size, if the processor switches among a number of processes, it is difficult to pack them compactly into main memory. Paging systems were introduced, which allow processes to be comprised of a number of fixed-size blocks, called pages. A program references a word by means of a **virtual address** consisting of a page number and an offset within the page. Each page of a process may be located anywhere in main memory. The paging system provides for a dynamic mapping between the virtual address used in the program and a **real address**, or physical address, in main memory.

With dynamic mapping hardware available, the next logical step was to eliminate the requirement that all pages of a process reside in main memory simultaneously. All the pages of a process are maintained on disk. When a process is executing, some of its pages are in main memory. If reference is made to a page that is not in main memory, the memory management hardware detects this and arranges for the missing page to be loaded. Such a scheme is referred to as **virtual memory** and is depicted in Figure 2.9.

The processor hardware, together with the OS, provides the user with a "virtual processor" that has access to a virtual memory. This memory may be a linear address space or a collection of segments, which are variable-length blocks of contiguous addresses. In either case, programming language instructions can reference program and data locations in the virtual memory area. Process isolation can be achieved by giving each process a unique, nonoverlapping virtual memory. Memory sharing can be achieved by overlapping portions of two virtual memory spaces. Files are maintained in a long-term store. Files and portions of files may be copied into the virtual memory for manipulation by programs.

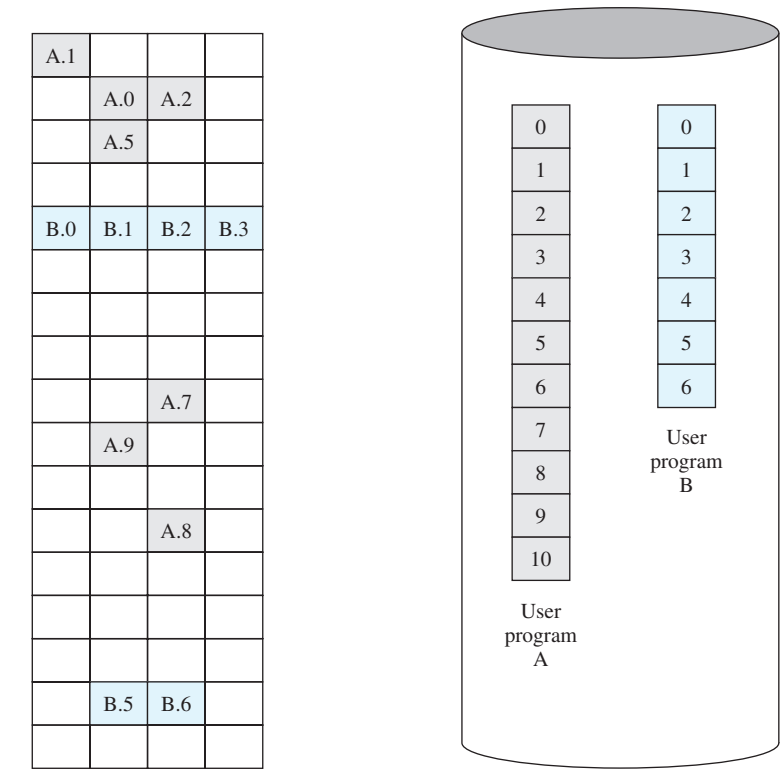

Figure 2.9 Virtual Memory Concepts

Figure 2.10 highlights the addressing concerns in a virtual memory scheme. Storage consists of directly addressable (by machine instructions) main memory and lower-speed auxiliary memory that is accessed indirectly by loading blocks into main memory. Address translation hardware (memory management unit) is interposed between the processor and memory. Programs reference locations using virtual addresses, which are mapped into real main memory addresses. If a reference is made to a virtual address not in real memory, then a portion of the contents of real memory is swapped out to auxiliary memory and the desired block of data is swapped in. During this activity, the process that generated the address reference must be suspended. The OS designer needs to develop an address translation mechanism that generates little overhead and a storage allocation policy that minimizes the traffic between memory levels.

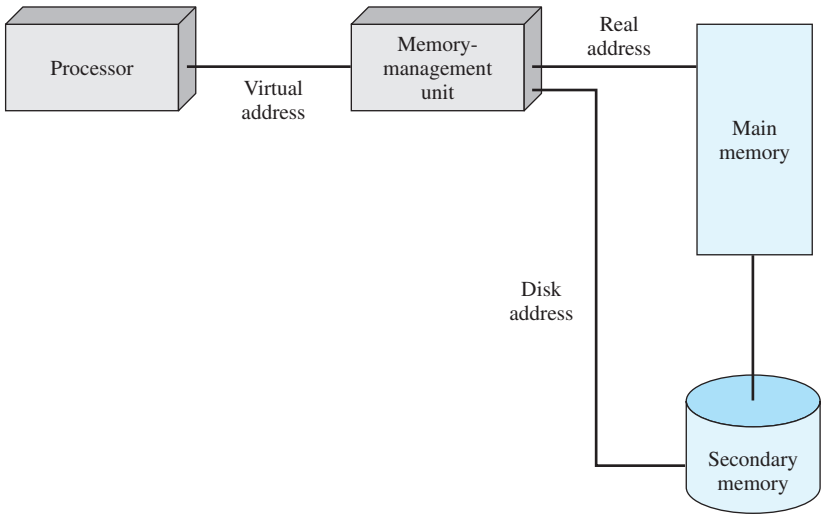

Figure 2.10 **Virtual Memory Addressing**

Information Protection and Security

The growth in the use of time-sharing systems and, more recently, computer networks has brought with it a growth in concern for the protection of information. The nature of the threat that concerns an organization will vary greatly depending on the circumstances. However, there are some general-purpose tools that can be built into computers and operating systems that support a variety of protection and security mechanisms. In general, we are concerned with the problem of controlling access to computer systems and the information stored in them.

Much of the work in security and protection as it relates to operating systems can be roughly grouped into four categories:

- **Availability:** Concerned with protecting the system against interruption
- **Confidentiality:** Assures that users cannot read data for which access is unauthorized
- **Data integrity:** Protection of data from unauthorized modification
- **Authenticity:** Concerned with the proper verification of the identity of users and the validity of messages or data

Scheduling and Resource Management

A key responsibility of the OS is to manage the various resources available to it (main memory space, I/O devices, processors) and to schedule their use by the various active processes. Any resource allocation and scheduling policy must consider three factors:

- **Fairness:** Typically, we would like all processes that are competing for the use of a particular resource to be given approximately equal and fair access to that

resource. This is especially so for jobs of the same class, that is, jobs of similar demands.

- **Differential responsiveness:** On the other hand, the OS may need to discriminate among different classes of jobs with different service requirements. The OS should attempt to make allocation and scheduling decisions to meet the total set of requirements. The OS should also make these decisions dynamically. For example, if a process is waiting for the use of an I/O device, the OS may wish to schedule that process for execution as soon as possible to free up the device for later demands from other processes.
- **Efficiency:** The OS should attempt to maximize throughput, minimize response time, and, in the case of time sharing, accommodate as many users as possible. These criteria conflict; finding the right balance for a particular situation is an ongoing problem for operating system research.

Scheduling and resource management are essentially operations-research problems and the mathematical results of that discipline can be applied. In addition, measurement of system activity is important to be able to monitor performance and make adjustments.

Figure 2.11 suggests the major elements of the OS involved in the scheduling of processes and the allocation of resources in a multiprogramming environment. The OS maintains a number of queues, each of which is simply a list of processes waiting for some resource. The short-term queue consists of processes that are in main memory (or at least an essential minimum portion of each is in main memory) and are ready to run as soon as the processor is made available. Any one of these processes

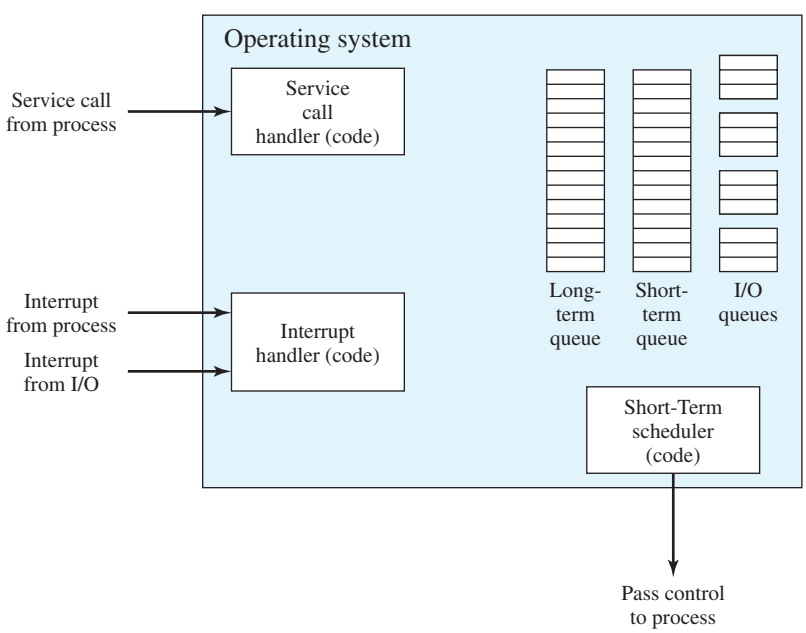

Figure 2.11 **Key Elements of an Operating System for Multiprogramming**

could use the processor next. It is up to the short-term scheduler, or dispatcher, to pick one. A common strategy is to give each process in the queue some time in turn; this is referred to as a **round-robin** technique. In effect, the round-robin technique employs a circular queue. Another strategy is to assign priority levels to the various processes, with the scheduler selecting processes in priority order.

The long-term queue is a list of new jobs waiting to use the processor. The OS adds jobs to the system by transferring a process from the long-term queue to the short-term queue. At that time, a portion of main memory must be allocated to the incoming process. Thus, the OS must be sure that it does not overcommit memory or processing time by admitting too many processes to the system. There is an I/O queue for each I/O device. More than one process may request the use of the same I/O device. All processes waiting to use each device are lined up in that device's queue. Again, the OS must determine which process to assign to an available I/O device.

The OS receives control of the processor at the interrupt handler if an interrupt occurs. A process may specifically invoke some operating system service, such as an I/O device handler by means of a service call. In this case, a service call handler is the entry point into the OS. In any case, once the interrupt or service call is handled, the short-term scheduler is invoked to pick a process for execution.

The foregoing is a functional description; details and modular design of this portion of the OS will differ in various systems. Much of the research and development effort in operating systems has been directed at picking algorithms and data structures for this function that provide fairness, differential responsiveness, and efficiency.

System Structure

As more and more features have been added to operating systems, and as the underlying hardware has become more capable and versatile, the size and complexity of operating systems has grown. CTSS, put into operation at MIT in 1963, consisted of approximately 32,000 36-bit words of storage. OS/360, introduced a year later by IBM, had more than a million machine instructions. By 1975, the Multics system, developed by MIT and Bell Laboratories, had grown to more than 20 million instructions. It is true that more recently, some simpler operating systems have been introduced for smaller systems, but these have inevitably grown more complex as the underlying hardware and user requirements have grown. Thus, the UNIX of today is far more complex than the almost toy system put together by a few talented programmers in the early 1970s, and the simple MS-DOS has given way to the rich and complex power of OS/2 and Windows. For example, Windows NT 4.0 contains 16 million lines of code, and Windows 2000 has well over twice that number.

The size of a full-featured OS, and the difficulty of the problem it addresses, has led to four unfortunate but all-too-common problems. First, operating systems are chronically late in being delivered. This goes for new operating systems and upgrades to older systems. Second, the systems have latent bugs that show up in the field and must be fixed and reworked. Third, performance is often not what was expected. Fourth, it has proved impossible to deploy a complex OS that is not vulnerable to a variety of security attacks, including viruses, worms, and unauthorized access.

To manage the complexity of operating systems and to overcome these problems, there has been much focus over the years on the software structure of the OS. Certain points seem obvious. The software must be modular. This will help organize the software development process and limit the effort of diagnosing and fixing errors. The modules must have well-defined interfaces to each other, and the interfaces must be as simple as possible. Again, this eases the programming burden. It also facilitates system evolution. With clean, minimal interfaces between modules, one module can be changed with minimal impact on other modules.

For large operating systems, which run from millions to tens of millions of lines of code, modular programming alone has not been found to be sufficient. Instead there has been increasing use of the concepts of hierarchical layers and information abstraction. The hierarchical structure of a modern OS separates its functions according to their characteristic time scale and their level of abstraction. We can view the system as a series of levels. Each level performs a related subset of the functions required of the OS. It relies on the next lower level to perform more primitive functions and to conceal the details of those functions. It provides services to the next higher layer. Ideally, the levels should be defined so that changes in one level do not require changes in other levels. Thus, we have decomposed one problem into a number of more manageable subproblems.

In general, lower layers deal with a far shorter time scale. Some parts of the OS must interact directly with the computer hardware, where events can have a time scale as brief as a few billionths of a second. At the other end of the spectrum, parts of the OS communicate with the user, who issues commands at a much more leisurely pace, perhaps one every few seconds. The use of a set of levels conforms nicely to this environment.

The way in which these principles are applied varies greatly among contemporary operating systems. However, it is useful at this point, for the purpose of gaining an overview of operating systems, to present a model of a hierarchical OS. Let us consider the model proposed in [BROW84] and [DENN84]. Although it does not correspond to any particular OS, this model provides a useful high-level view of OS structure. The model is defined in Table 2.4 and consists of the following levels:

- **Level 1:** Consists of electronic circuits, where the objects that are dealt with are registers, memory cells, and logic gates. The operations defined on these objects are actions, such as clearing a register or reading a memory location.
- **Level 2:** The processor's instruction set. The operations at this level are those allowed in the machine language instruction set, such as add, subtract, load, and store.
- **Level 3:** Adds the concept of a procedure or subroutine, plus the call/return operations.
- **Level 4:** Introduces interrupts, which cause the processor to save the current context and invoke an interrupt-handling routine.

These first four levels are not part of the OS but constitute the processor hardware. However, some elements of the OS begin to appear at these levels, such as the interrupt-handling routines. It is at level 5 that we begin to reach the OS proper and that the concepts associated with multiprogramming begin to appear.

Table 2.4 Operating System Design Hierarchy

Level	Name	Objects	Example Operations
13	Shell	User programming environment	Statements in shell language
12	User processes	User processes	Quit, kill, suspend, resume
11	Directories	Directories	Create, destroy, attach, detach, search, list
10	Devices	External devices, such as printers, displays, and keyboards	Open, close, read, write
9	File system	Files	Create, destroy, open, close, read, write
8	Communications	Pipes	Create, destroy, open, close, read, write
7	Virtual memory	Segments, pages	Read, write, fetch
6	Local secondary store	Blocks of data, device channels	Read, write, allocate, free
5	Primitive processes	Primitive processes, semaphores, ready list	Suspend, resume, wait, signal
4	Interrupts	Interrupt-handling programs	Invoke, mask, unmask, retry
3	Procedures	Procedures, call stack, display	Mark stack, call, return
2	Instruction set	Evaluation stack, microprogram interpreter, scalar and array data	Load, store, add, subtract, branch
1	Electronic circuits	Registers, gates, buses, etc.	Clear, transfer, activate, complement

Gray shaded area represents hardware.

- **Level 5:** The notion of a process as a program in execution is introduced at this level. The fundamental requirements on the OS to support multiple processes include the ability to suspend and resume processes. This requires saving hardware registers so that execution can be switched from one process to another. In addition, if processes need to cooperate, then some method of synchronization is needed. One of the simplest techniques, and an important concept in OS design, is the semaphore, a simple signaling technique that is explored in Chapter 5.
- **Level 6:** Deals with the secondary storage devices of the computer. At this level, the functions of positioning the read/write heads and the actual transfer of blocks of data occur. Level 6 relies on level 5 to schedule the operation and to notify the requesting process of completion of an operation. Higher levels are concerned with the address of the needed data on the disk and provide a request for the appropriate block to a device driver at level 5.
- **Level 7:** Creates a logical address space for processes. This level organizes the virtual address space into blocks that can be moved between main memory and secondary memory. Three schemes are in common use: those using fixed-size pages, those using variable-length segments, and those using both. When a needed block is not in main memory, logic at this level requests a transfer from level 6.

Up to this point, the OS deals with the resources of a single processor. Beginning with level 8, the OS deals with external objects such as peripheral devices and possibly networks and computers attached to the network. The objects at these upper levels are logical, named objects that can be shared among processes on the same computer or on multiple computers.

- **Level 8:** Deals with the communication of information and messages between processes. Whereas level 5 provided a primitive signal mechanism that allowed for the synchronization of processes, this level deals with a richer sharing of information. One of the most powerful tools for this purpose is the pipe, which is a logical channel for the flow of data between processes. A pipe is defined with its output from one process and its input into another process. It can also be used to link external devices or files to processes. The concept is discussed in Chapter 6.
- **Level 9:** Supports the long-term storage of named files. At this level, the data on secondary storage are viewed in terms of abstract, variable-length entities. This is in contrast to the hardware-oriented view of secondary storage in terms of tracks, sectors, and fixed-size blocks at level 6.
- **Level 10:** Provides access to external devices using standardized interfaces.
- **Level 11:** Is responsible for maintaining the association between the external and internal identifiers of the system's resources and objects. The external identifier is a name that can be employed by an application or user. The internal identifier is an address or other indicator that can be used by lower levels of the OS to locate and control an object. These associations are maintained in a directory. Entries include not only external/internal mapping, but also characteristics such as access rights.
- **Level 12:** Provides a full-featured facility for the support of processes. This goes far beyond what is provided at level 5. At level 5, only the processor register contents associated with a process are maintained, plus the logic for dispatching processes. At level 12, all of the information needed for the orderly management of processes is supported. This includes the virtual address space of the process, a list of objects and processes with which it may interact and the constraints of that interaction, parameters passed to the process upon creation, and any other characteristics of the process that might be used by the OS to control the process.
- **Level 13:** Provides an interface to the OS for the user. It is referred to as the **shell** because it separates the user from OS details and presents the OS simply as a collection of services. The shell accepts user commands or job control statements, interprets these, and creates and controls processes as needed. For example, the interface at this level could be implemented in a graphical manner, providing the user with commands through a list presented as a menu and displaying results using graphical output to a specific device such as a screen.

This hypothetical model of an OS provides a useful descriptive structure and serves as an implementation guideline. The reader may refer back to this structure during the course of the book to observe the context of any particular design issue under discussion.

2.4 DEVELOPMENTS LEADING TO MODERN OPERATING SYSTEMS

Over the years, there has been a gradual evolution of OS structure and capabilities. However, in recent years a number of new design elements have been introduced into both new operating systems and new releases of existing operating systems that create a major change in the nature of operating systems. These modern operating systems respond to new developments in hardware, new applications, and new security threats. Among the key hardware drivers are multiprocessor systems, greatly increased processor speed, high-speed network attachments, and increasing size and variety of memory storage devices. In the application arena, multimedia applications, Internet and Web access, and client/server computing have influenced OS design. With respect to security, Internet access to computers has greatly increased the potential threat and increasingly sophisticated attacks, such as viruses, worms, and hacking techniques, have had a profound impact on OS design.

The rate of change in the demands on operating systems requires not just modifications and enhancements to existing architectures but new ways of organizing the OS. A wide range of different approaches and design elements has been tried in both experimental and commercial operating systems, but much of the work fits into the following categories:

- Microkernel architecture
- Multithreading
- Symmetric multiprocessing
- Distributed operating systems
- Object-oriented design

Most operating systems, until recently, featured a large **monolithic kernel**. Most of what is thought of as OS functionality is provided in these large kernels, including scheduling, file system, networking, device drivers, memory management, and more. Typically, a monolithic kernel is implemented as a single process, with all elements sharing the same address space. A **microkernel architecture** assigns only a few essential functions to the kernel, including address spaces, interprocess communication (IPC), and basic scheduling. Other OS services are provided by processes, sometimes called servers, that run in user mode and are treated like any other application by the microkernel. This approach decouples kernel and server development. Servers may be customized to specific application or environment requirements. The microkernel approach simplifies implementation, provides flexibility, and is well suited to a distributed environment. In essence, a microkernel interacts with local and remote server processes in the same way, facilitating construction of distributed systems.

Multithreading is a technique in which a process, executing an application, is divided into threads that can run concurrently. We can make the following distinction:

- **Thread:** A dispatchable unit of work. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a

stack (to enable subroutine branching). A thread executes sequentially and is interruptable so that the processor can turn to another thread.

- **Process:** A collection of one or more threads and associated system resources (such as memory containing both code and data, open files, and devices). This corresponds closely to the concept of a program in execution. By breaking a single application into multiple threads, the programmer has great control over the modularity of the application and the timing of application-related events.

Multithreading is useful for applications that perform a number of essentially independent tasks that do not need to be serialized. An example is a database server that listens for and processes numerous client requests. With multiple threads running within the same process, switching back and forth among threads involves less processor overhead than a major process switch between different processes. Threads are also useful for structuring processes that are part of the OS kernel as described in subsequent chapters.

Until recently, virtually all single-user personal computers and workstations contained a single general-purpose microprocessor. As demands for performance increase and as the cost of microprocessors continues to drop, vendors have introduced computers with multiple microprocessors. To achieve greater efficiency and reliability, one technique is to employ **symmetric multiprocessing (SMP)**, a term that refers to a computer hardware architecture and also to the OS behavior that exploits that architecture. A symmetric multiprocessor can be defined as a standalone computer system with the following characteristics:

1. There are multiple processors.
2. These processors share the same main memory and I/O facilities, interconnected by a communications bus or other internal connection scheme.
3. All processors can perform the same functions (hence the term *symmetric*).

In recent years, systems with multiple processors on a single chip have become widely used, referred to as chip multiprocessor systems. Many of the design issues are the same, whether dealing with a chip multiprocessor or a multiple-chip SMP.

The OS of an SMP schedules processes or threads across all of the processors. SMP has a number of potential advantages over uniprocessor architecture, including the following:

- **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type. This is illustrated in Figure 2.12. With multiprogramming, only one process can execute at a time; meanwhile all other processes are waiting for the processor. With multiprocessing, more than one process can be running simultaneously, each on a different processor.
- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the system. Instead, the system can continue to function at reduced performance.
- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.

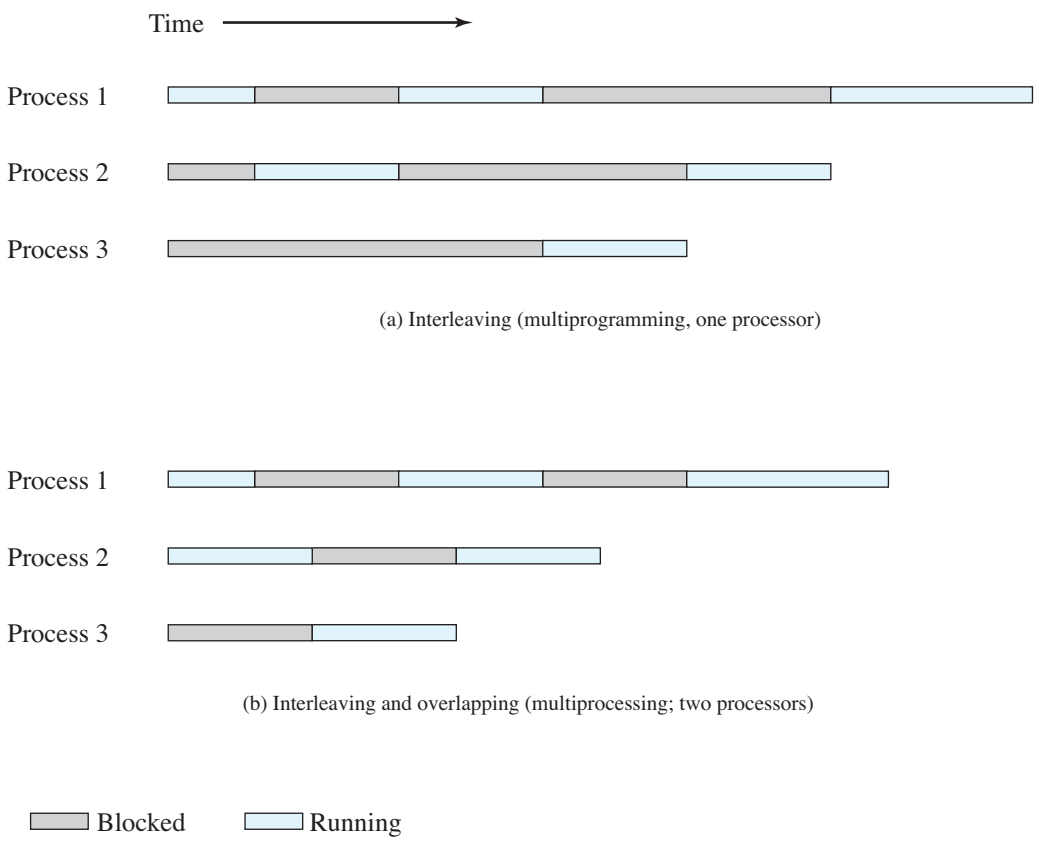

Figure 2.12 **Multiprogramming and Multiprocessing**

- **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

It is important to note that these are potential, rather than guaranteed, benefits. The OS must provide tools and functions to exploit the parallelism in an SMP system.

Multithreading and SMP are often discussed together, but the two are independent facilities. Even on a uniprocessor system, multithreading is useful for structuring applications and kernel processes. An SMP system is useful even for nonthreaded processes, because several processes can run in parallel. However, the two facilities complement each other and can be used effectively together.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The OS takes care of scheduling of threads or processes on individual processors and of synchronization among processors. This book discusses the scheduling and synchronization mechanisms used to provide the single-system appearance to the user. A different problem is to provide the appearance of a single system for a cluster of separate computers—a multicomputer system. In this case, we are dealing with a collection of entities (computers), each with its own main memory,

- **UNIX Guru Universe:** Excellent source of UNIX information.
- **Linux Documentation Project:** The name describes the site.
- **IBM's Linux Web site:** Provides a wide range of technical and user information on Linux. Much of it is devoted to IBM products, but there is a lot of useful general technical information.
- **Windows Development:** Good source of information on Windows internals.

2.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

batch processing	multiprogramming	serial processing
batch system	multlitasking	symmetric multiprocessing
execution context	multithreading	task
interrupt	nucleus	thread
job	operating system (OS)	time sharing
job control language	physical address	time-sharing system
kernel	privileged instruction	uniprogramming
memory management	process	virtual address
microkernel	process state	
monitor	real address	
monolithic kernel	resident monitor	
multiprogrammed batch system	round robin	
	scheduling	

Review Questions

2.1 What are three objectives of an OS design?
2.2 What is the kernel of an OS?
2.3 What is multiprogramming?
2.4 What is a process?
2.5 How is the execution context of a process used by the OS?
2.6 List and briefly explain five storage management responsibilities of a typical OS.
2.7 Explain the distinction between a real address and a virtual address.
2.8 Describe the round-robin scheduling technique.
2.9 Explain the difference between a monolithic kernel and a microkernel.
2.10 What is multithreading?

Problems

2.1 Suppose that we have a multiprogrammed computer in which each job has identical characteristics. In one computation period, T, for a job, half the time is spent in I/O and the other half in processor activity. Each job runs for a total of N periods. Assume that a simple round-robin scheduling is used, and that I/O operations can overlap with processor operation. Define the following quantities:
- Turnaround time = actual time to complete a job
- Throughput = average number of jobs completed per time period T
- Processor utilization = percentage of time that the processor is active (not waiting)

Compute these quantities for one, two, and four simultaneous jobs, assuming that the period T is distributed in each of the following ways:

a. I/O first half, processor second half

b. I/O first and fourth quarters, processor second and third quarter

2.2 An I/O-bound program is one that, if run alone, would spend more time waiting for I/O than using the processor. A processor-bound program is the opposite. Suppose a short-term scheduling algorithm favors those programs that have used little processor time in the recent past. Explain why this algorithm favors I/O-bound programs and yet does not permanently deny processor time to processor-bound programs.

2.3 Contrast the scheduling policies you might use when trying to optimize a time-sharing system with those you would use to optimize a multiprogrammed batch system.

2.4 What is the purpose of system calls, and how do system calls relate to the OS and to the concept of dual-mode (kernel mode and user mode) operation?

2.5 In IBM's mainframe operating system, OS/390, one of the major modules in the kernel is the System Resource Manager (SRM). This module is responsible for the allocation of resources among address spaces (processes). The SRM gives OS/390 a degree of sophistication unique among operating systems. No other mainframe OS, and certainly no other type of OS, can match the functions performed by SRM. The concept of resource includes processor, real memory, and I/O channels. SRM accumulates statistics pertaining to utilization of processor, channel, and various key data structures. Its purpose is to provide optimum performance based on performance monitoring and analysis. The installation sets forth various performance objectives, and these serve as guidance to the SRM, which dynamically modifies installation and job performance characteristics based on system utilization. In turn, the SRM provides reports that enable the trained operator to refine the configuration and parameter settings to improve user service.

This problem concerns one example of SRM activity. Real memory is divided into equal-sized blocks called frames, of which there may be many thousands. Each frame can hold a block of virtual memory referred to as a page. SRM receives control approximately 20 times per second and inspects each and every page frame. If the page has not been referenced or changed, a counter is incremented by 1. Over time, SRM averages these numbers to determine the average number of seconds that a page frame in the system goes untouched. What might be the purpose of this and what action might SRM take?