CHAPTER 2

Operating System Overview

Learning Objectives

After studying this chapter, you should be able to:

- Summarize, at a top level, the key functions of an operating system (OS).
- Discuss the evolution of operating systems for early simple batch systems to modern complex systems.
- Give a brief explanation of each of the major achievements in OS research, as defined in Section 2.3.
- Discuss the key design areas that have been instrumental in the development of modern operating systems.
- Define and discuss virtual machines and virtualization.
- Understand the OS design issues raised by the introduction of multiprocessor and multicore organization.
- Understand the basic structure of Windows.
- Describe the essential elements of a traditional UNIX system.
- Explain the new features found in modern UNIX systems.
- Discuss Linux and its relationship to UNIX.

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 will look at how operating systems have evolved from primitive batch systems to sophisticated multitasking, multiuser systems. The remainder of the chapter will look at the history and general characteristics of the two operating systems that serve as examples throughout this 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 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, or library programs. 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 a 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 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

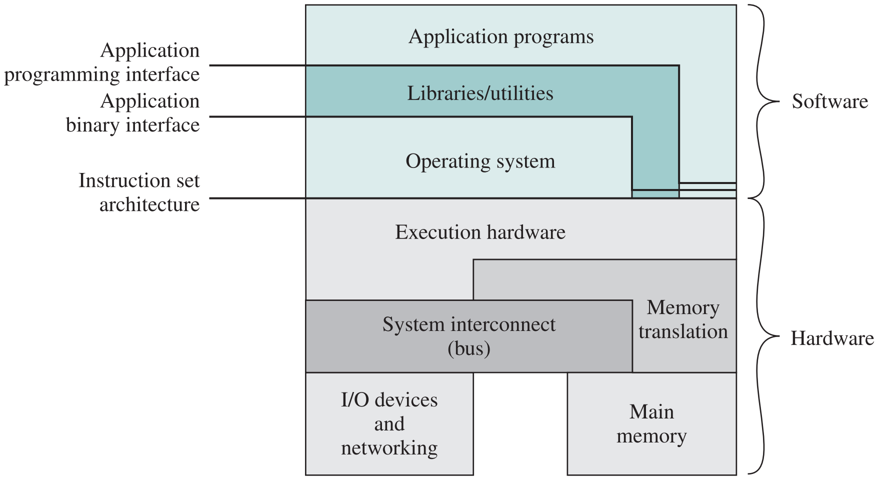

Application programming interface
Application binary interface
Instruction set architecture
Application programs
Libraries/utilities
Operating system
Execution hardware
Memory translation
System interconnect (bus)
I/O devices and networking
Main memory
Software
Hardware

Figure 2.1 Computer Hardware and Software Structure

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 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, or 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.

Figure 2.1 also indicates three key interfaces in a typical computer system:

- **Instruction set architecture (ISA):** The ISA defines the repertoire of machine language instructions that a computer can follow. This interface is the boundary between hardware and software. Note both application programs and utilities may access the ISA directly. For these programs, a subset of the instruction repertoire is available (user ISA). The OS has access to additional machine language instructions that deal with managing system resources (system ISA).
- **Application binary interface (ABI):** The ABI defines a standard for binary portability across programs. The ABI defines the system call interface to the operating system, and the hardware resources and services available in a system through the user ISA.
- **Application programming interface (API):** The API gives a program access to the hardware resources and services available in a system through the user ISA supplemented with high-level language (HLL) library calls. Any system calls are usually performed through libraries. Using an API enables application software to be ported easily, through recompilation, to other systems that support the same API.

The Operating System as Resource Manager

The OS is responsible for controlling the use of a computer's resources, such as I/O, main and secondary memory, and processor execution time. 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 consists of instructions executed by the processor. While executing, the OS decides how processor time is to be allocated and which computer resources are available for use. But in order for the processor to act on these decisions, it must cease executing the OS program and execute other programs. Thus, the OS relinquishes control for the processor to do some "useful" work, 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 and utility programs and data. The OS and the memory management hardware in the processor jointly control the allocation of main memory, 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.

Ease of Evolution of an Operating System

A major OS 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 OS 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.

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

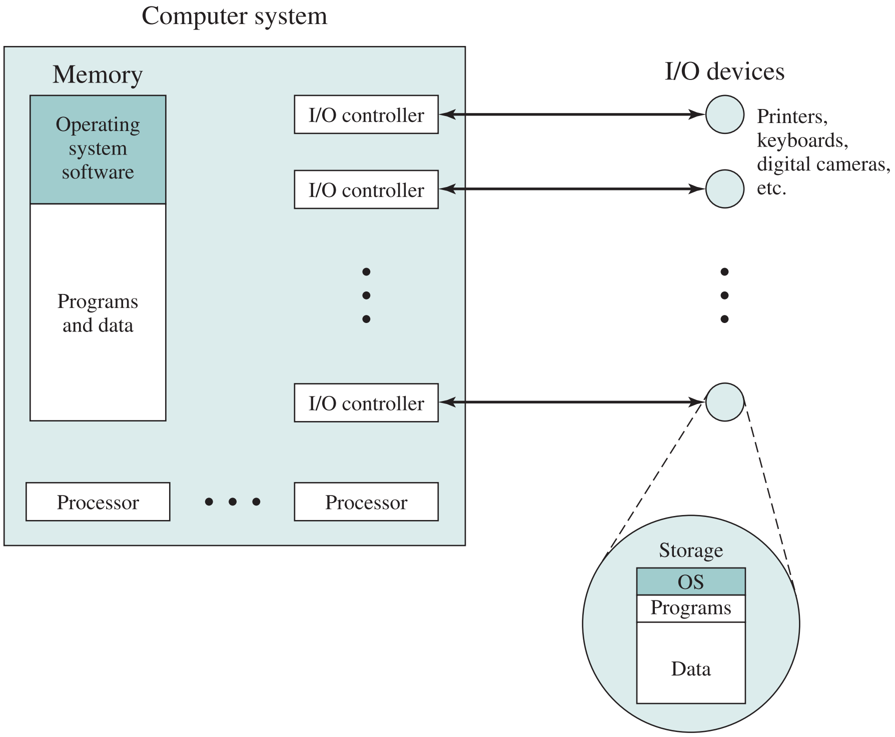


Figure 2.2 **The Operating System as Resource Manager**

- **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.

The need to regularly update an OS 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 will 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), 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 OS was developed. It appears that the first batch OS (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 OS 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

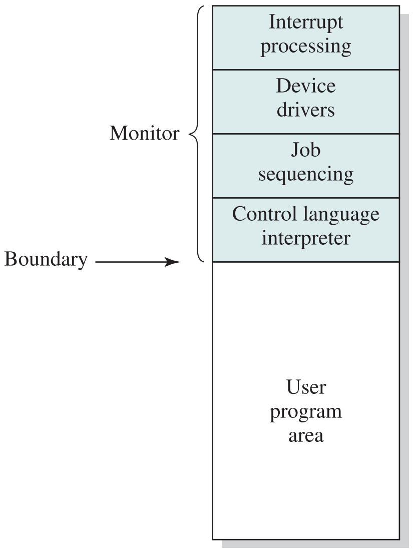


Figure 2.3 Memory Layout for a Resident Monitor

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 (see 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 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 the processor is now fetching and executing instructions in a user program, and "control is returned to the monitor" means 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 OS, 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 as 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 OS, 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 monitors 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.

Multiprogrammed Batch Systems

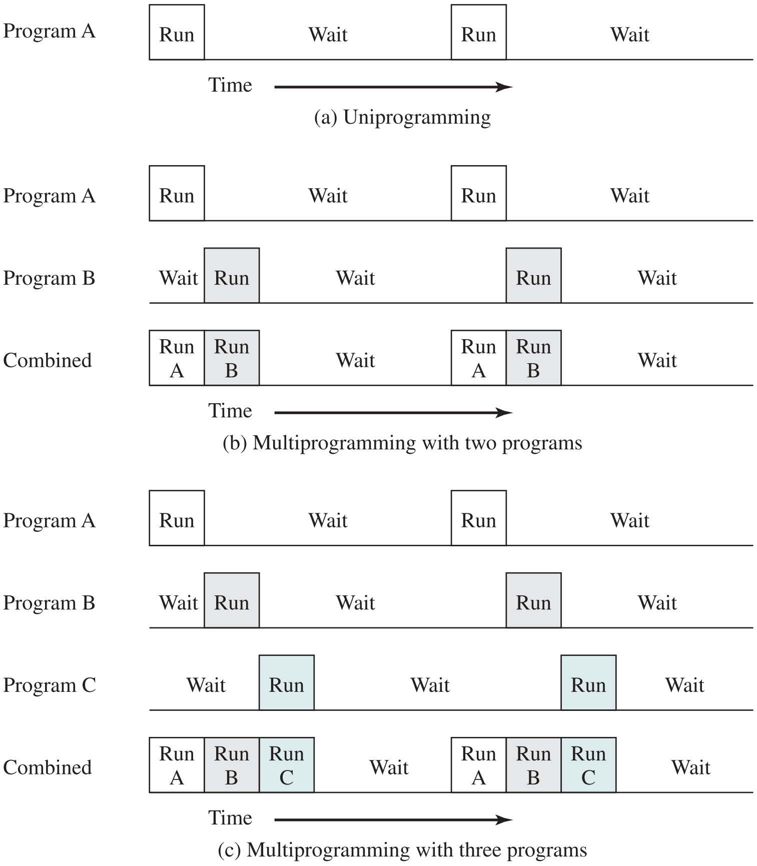
Even with the automatic job sequencing provided by a simple batch OS, the processor is often idle. The problem is I/O devices are slow compared to the processor.

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

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 as uniprogramming. The processor



(a) Uniprogramming

(b) Multiprogramming with two programs

(c) Multiprogramming with three programs

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

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.

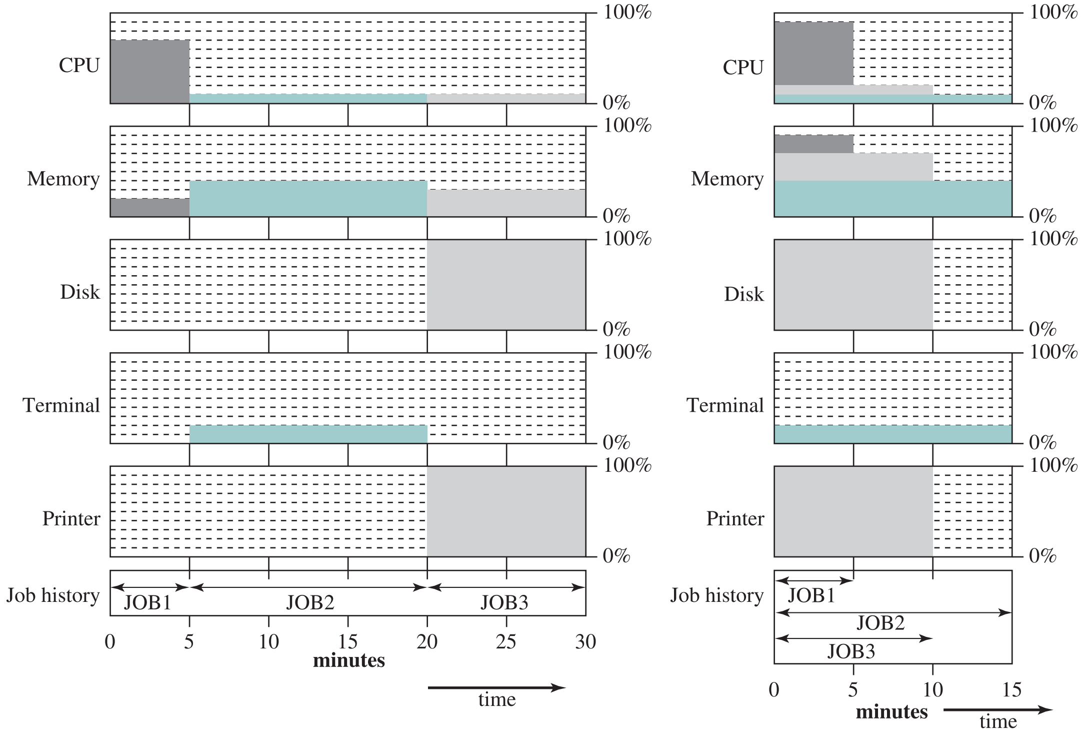
This inefficiency is not necessary. We know there must be enough memory to hold the OS (resident monitor) and one user program. Suppose there is room for the OS and two user programs. When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O (see Figure 2.5b). Furthermore, we might expand memory to hold three, four, or more programs and switch among all of them (see 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, 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.

Now suppose the jobs are run concurrently under a multiprogramming OS. Because there is little resource contention between the jobs, all three can run in

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



(a) Uniprogramming

(b) Multiprogramming

Figure 2.6 **Utilization Histograms**

nearly minimum time while coexisting with the others in the computer (assuming 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 after the interrupt is handled.

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, and this decision requires an algorithm for scheduling. These concepts will be 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.

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 ported to 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 5,000 of those. 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 5,000th 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.

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

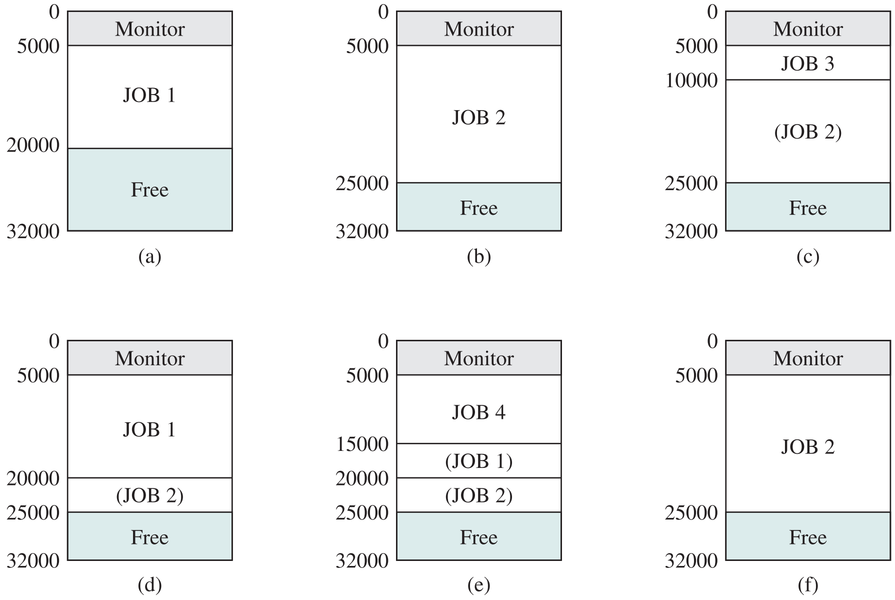


Figure 2.7 CTSS Operation

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 there are four interactive users with the following memory requirements, in words:

- JOB1: 15,000
- JOB2: 20,000
- JOB3: 5,000
- JOB4: 10,000

Initially, the monitor loads JOB1 and transfers control to it (Figure 2.7a). 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 (Figure 2.7b). 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 (Figure 2.7c). 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 (Figure 2.7d). When JOB4 is loaded, part of JOB1 and the portion of JOB2 remaining in memory are retained (Figure 2.7e). 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 the missing portion of JOB2 be read in (Figure 2.7f).

The CTSS approach is primitive compared to present-day time sharing, but it was effective. 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 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 four major theoretical advances in the development of operating systems:

- Processes
- Memory management
- Information protection and security
- Scheduling and resource management

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

The Process

Central to the design of operating systems is the concept of *process.* 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.

A third 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, 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:

1. An executable program
2. The associated data needed by the program (variables, work space, buffers, etc.)
3. 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 the OS needs to manage the process, and 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

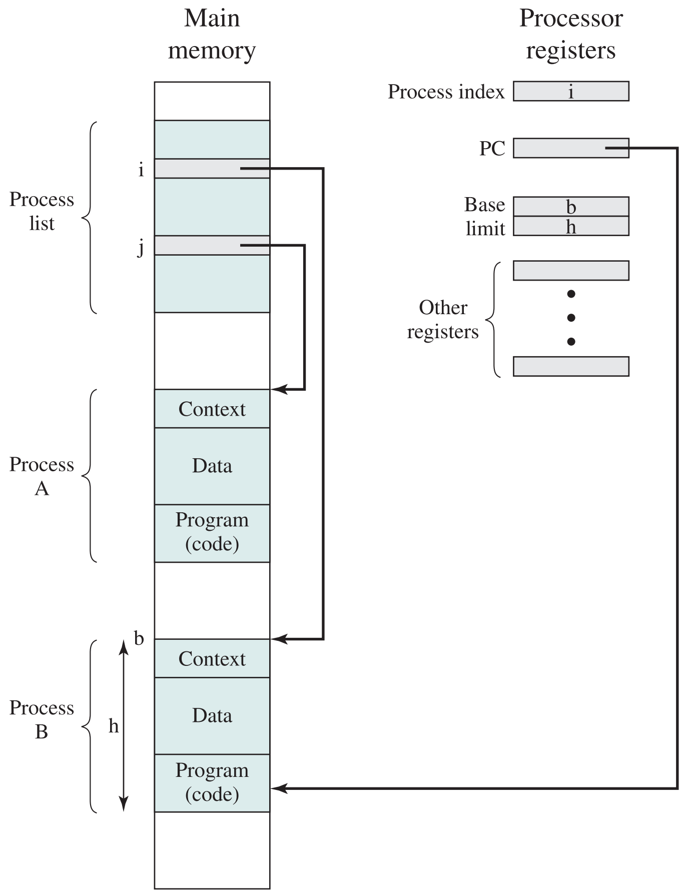


Figure 2.8 Typical Process Implementation

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 the execution of process A. The process switch consists of saving 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, we will see a number of examples where this process structure is employed to solve the problems raised by multiprogramming and resource sharing.

A final point, which we introduce briefly here, is the concept of **thread**. In essence, a single process, which is assigned certain resources, can be broken up into multiple, concurrent threads that execute cooperatively to perform the work of the process. This introduces a new level of parallel activity to be managed by the hardware and software.

Memory Management

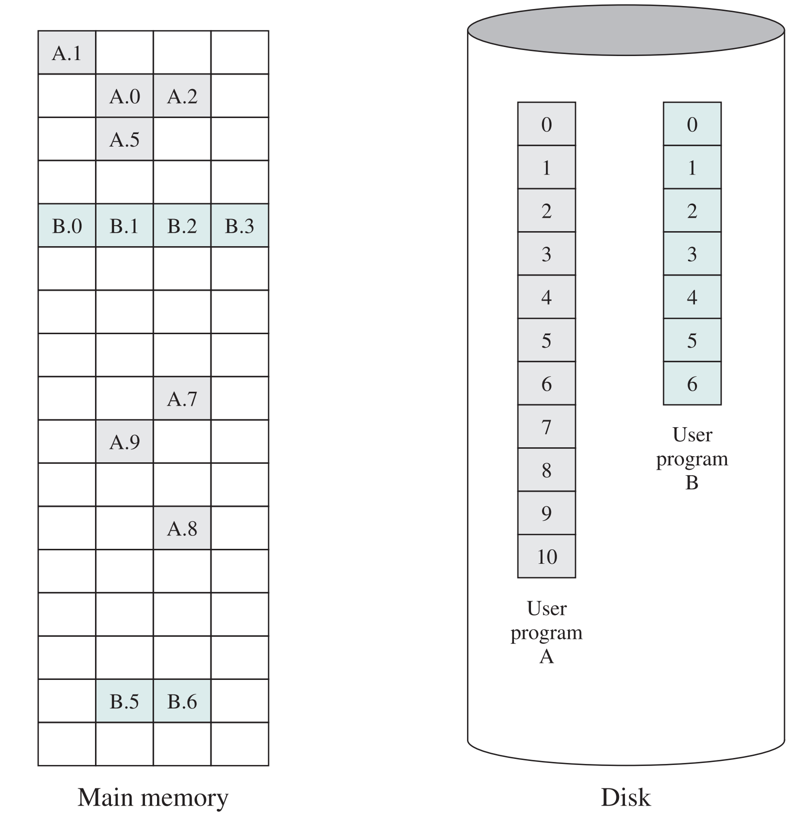
The needs of users can be met best by a computing environment that supports modular programming and the flexible use of data. System managers need efficient and orderly control of storage allocation. The OS, to satisfy these requirements, has five principal storage management responsibilities:

1. **Process isolation:** The OS must prevent independent processes from interfering with each other's memory, both data and instructions.
2. **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.
3. **Support of modular programming:** Programmers should be able to define program modules, and to dynamically create, destroy, and alter the size of modules.
4. **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.
5. **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 concurrently reside in main memory, so 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 simultaneously reside in main memory. 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, in coordination with the OS, arranges for the missing page to be loaded. Such a scheme is referred to as **virtual memory** and is depicted in Figure 2.9.



Main memory consists of a number of fixed-length frames, each equal to the size of a page. For a program to execute, some or all of its pages must be in main memory.

Secondary memory (disk) can hold many fixed-length pages. A user program consists of some number of pages. Pages of all programs plus the OS are on disk, as are files.

Figure 2.9 Virtual Memory Concepts

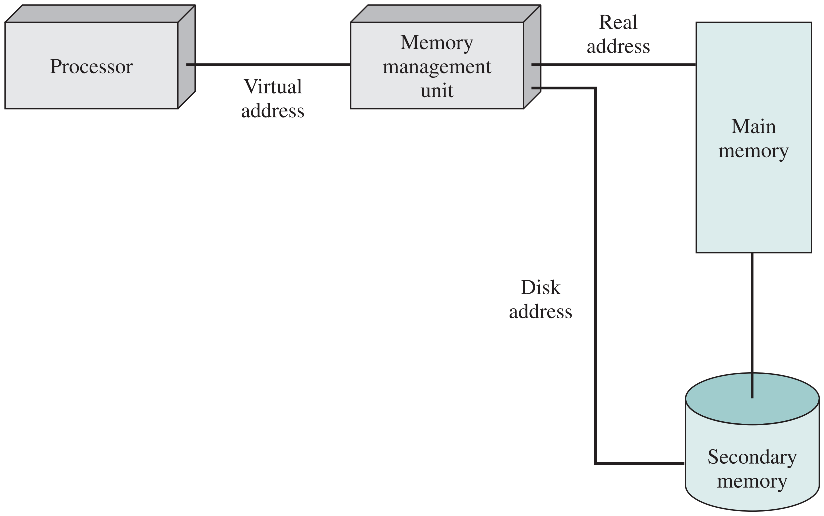


Figure 2.10 Virtual Memory Addressing

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 2.10 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 (a 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.

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:

1. **Availability:** Concerned with protecting the system against interruption.
2. **Confidentiality:** Assures that users cannot read data for which access is unauthorized.
3. **Data integrity:** Protection of data from unauthorized modification.
4. **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:

1. **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.
2. **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; the process can then immediately use the device, then release it for later demands from other processes.
3. **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 OS 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 could use the processor next. It is up to the short-term scheduler,

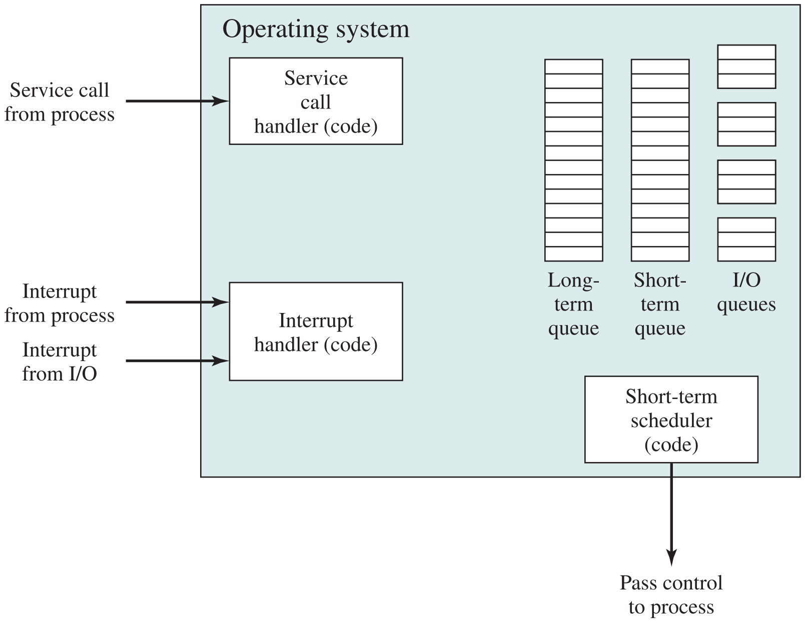


Figure 2.11 **Key Elements of an Operating System for Multiprogramming**

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 OS 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.

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

Until recently, most operating systems 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 space management, 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 interruptible so 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 will be described in subsequent chapters.

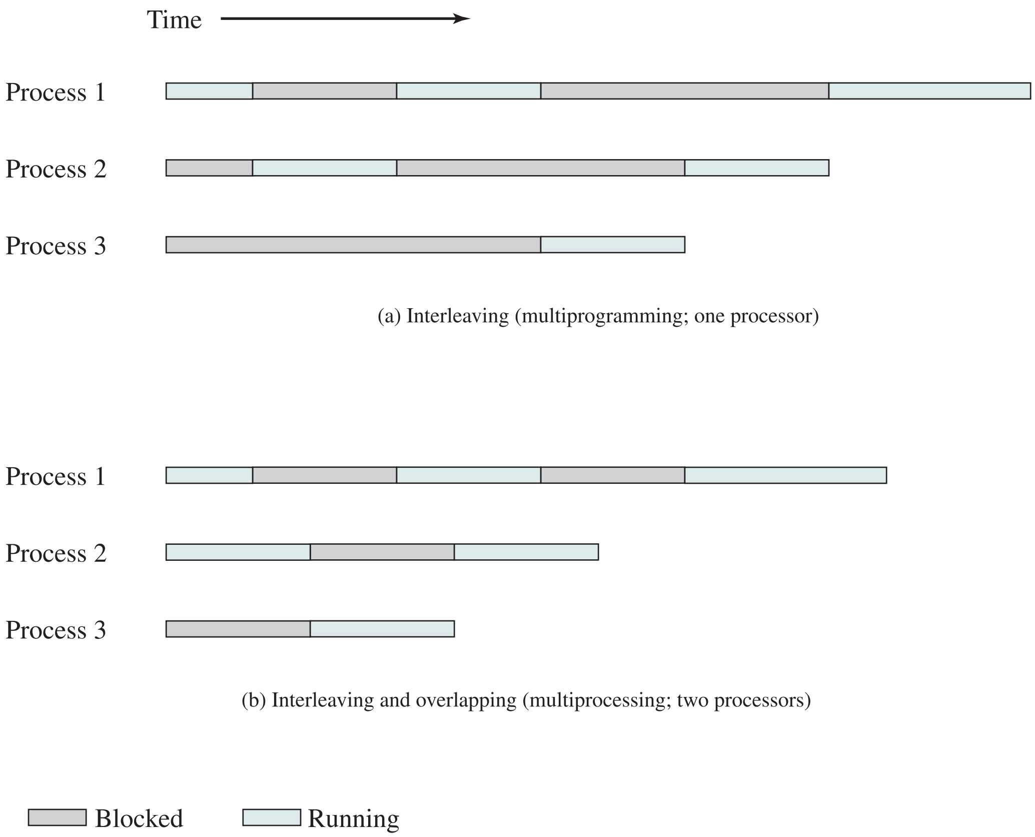
Symmetric multiprocessing (SMP) is a term that refers to a computer hardware architecture (described in Chapter 1) and also to the OS behavior that exploits that architecture. The OS of an SMP schedules processes or threads across all of the processors. SMP has a number of potential advantages over uniprocessor architecture, including the following:

- **Performance:** If the work to be done by a computer can be organized so 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.
- **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



(a) Interleaving (multiprogramming; one processor)

(b) Interleaving and overlapping (multiprocessing; two processors)

Figure 2.12 Multiprogramming and Multiprocessing

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 computers, each with its own main memory, secondary memory, and other I/O modules. A **distributed operating system** provides the illusion of a single main memory space and a single secondary memory space, plus other unified access facilities, such as a distributed file system. Although clusters are becoming increasingly popular, and there are many cluster products on the market, the state of the art for distributed operating systems lags behind that of uniprocessor and SMP operating systems. We will examine such systems in Part Eight.

Another innovation in OS design is the use of object-oriented technologies. **Object-oriented design** lends discipline to the process of adding modular extensions to a small kernel. At the OS level, an object-based structure enables programmers to customize an OS without disrupting system integrity. Object orientation also eases the development of distributed tools and full-blown distributed operating systems.

2.5 FAULT TOLERANCE

Fault tolerance refers to the ability of a system or component to continue normal operation despite the presence of hardware or software faults. This typically involves some degree of redundancy. Fault tolerance is intended to increase the reliability of a system. Typically, increased fault tolerance (and therefore increased reliability) comes with a cost, either in financial terms or performance, or both. Thus, the extent adoption of fault tolerance measures must be determined by how critical the resource is.

Fundamental Concepts

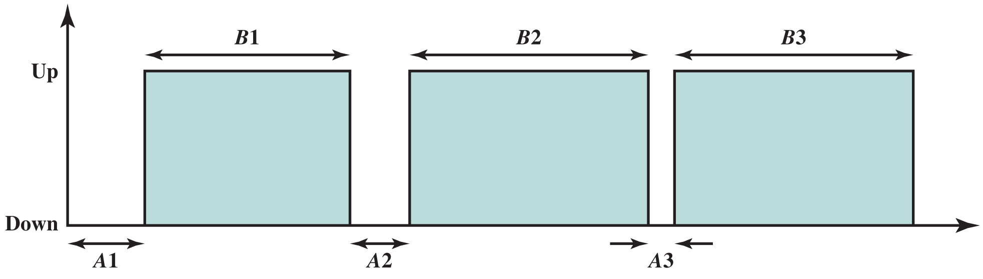
The three basic measures of the quality of the operation of a system that relate to fault tolerance are reliability, mean time to failure (MTTF), and availability. These concepts were developed with specific reference to hardware faults, but apply more generally to hardware and software faults.

The **reliability** $R(t)$ of a system is defined as the probability of its correct operation up to time t given that the system was operating correctly at time $t = 0$. For computer systems and operating systems, the term *correct operation* means the correct execution of a set of programs, and the protection of data from unintended modification. The **mean time to failure (MTTF)** is defined as

$$\text{MTTF} = \int_0^\infty R(t)$$

The **mean time to repair (MTTR)** is the average time it takes to repair or replace a faulty element. Figure 2.13 illustrates the relationship between MTTF and MTTR.

The **availability** of a system or service is defined as the fraction of time the system is available to service users' requests. Equivalently, availability is the probability that an entity is operating correctly under given conditions at a given instant of time. The time during which the system is not available is called **downtime**; the time during



$$MTTF = \frac{B1 + B2 + B3}{3} \qquad MTTR = \frac{A1 + A2 + A3}{3}$$

Figure 2.13 System Operational States

Table 2.4 Availability Classes

Class	Availability	Annual Downtime
Continuous	1.0	0
Fault tolerant	0.99999	5 minutes
Fault resilient	0.9999	53 minutes
High availability	0.999	8.3 hours
Normal availability	0.99–0.995	44–87 hours

which the system is available is called **uptime**. The availability *A* of a system can be expressed as follows:

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Table 2.4 shows some commonly identified availability levels and the corresponding annual downtime.

Often, the mean uptime, which is MTTF, is a better indicator than availability. A small downtime and a small uptime combination may result in a high availability measure, but the users may not be able to get any service if the uptime is less than the time required to complete a service.

Faults

The IEEE Standards Dictionary defines a **fault** as an erroneous hardware or software state resulting from component failure, operator error, physical interference from the environment, design error, program error, or data structure error. The standard also states that a fault manifests itself as (1) a defect in a hardware device or component; for example, a short circuit or broken wire, or (2) an incorrect step, process, or data definition in a computer program.

We can group faults into the following categories:

- **Permanent:** A fault that, after it occurs, is always present. The fault persists until the faulty component is replaced or repaired. Examples include disk head crashes, software bugs, and a burnt-out communications component.
- **Temporary:** A fault that is not present all the time for all operating conditions. Temporary faults can be further classified as follows:

  —**Transient:** A fault that occurs only once. Examples include bit transmission errors due to an impulse noise, power supply disturbances, and radiation that alters a memory bit.

  —**Intermittent:** A fault that occurs at multiple, unpredictable times. An example of an intermittent fault is one caused by a loose connection.

In general, fault tolerance is built into a system by adding redundancy. Methods of redundancy include the following:

- **Spatial (physical) redundancy:** Physical redundancy involves the use of multiple components that either perform the same function simultaneously, or are configured so one component is available as a backup in case of the failure of another component. An example of the former is the use of multiple parallel circuitry with the majority result produced as output. An example of the latter is a backup name server on the Internet.

- **Temporal redundancy:** Temporal redundancy involves repeating a function or operation when an error is detected. This approach is effective with temporary faults, but not useful for permanent faults. An example is the retransmission of a block of data when an error is detected, such as is done with data link control protocols.
- **Information redundancy:** Information redundancy provides fault tolerance by replicating or coding data in such a way that bit errors can be both detected and corrected. An example is the error-control coding circuitry used with memory systems, and error-correction techniques used with RAID disks, as will be described in subsequent chapters.

Operating System Mechanisms

A number of techniques can be incorporated into OS software to support fault tolerance. A number of examples will be evident throughout the book. The following list provides examples:

- **Process isolation:** As was mentioned earlier in this chapter, processes are generally isolated from one another in terms of main memory, file access, and flow of execution. The structure provided by the OS for managing processes provides a certain level of protection for other processes from a process that produces a fault.
- **Concurrency controls:** Chapters 5 and 6 will discuss some of the difficulties and faults that can occur when processes communicate or cooperate. These chapters will also discuss techniques used to ensure correct operation and to recover from fault conditions, such as deadlock.
- **Virtual machines:** Virtual machines, as will be discussed in Chapter 14, provide a greater degree of application isolation and hence fault isolation. Virtual machines can also be used to provide redundancy, with one virtual machine serving as a backup for another.
- **Checkpoints and rollbacks:** A checkpoint is a copy of an application's state saved in some storage that is immune to the failures under consideration. A rollback restarts the execution from a previously saved checkpoint. When a failure occurs, the application's state is rolled back to the previous checkpoint and restarted from there. This technique can be used to recover from transient as well as permanent hardware failures, and certain types of software failures. Database and transaction processing systems typically have such capabilities built in.

A much wider array of techniques could be discussed, but a full treatment of OS fault tolerance is beyond our current scope.

2.6 OS DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE

Symmetric Multiprocessor OS Considerations

In an SMP system, the kernel can execute on any processor, and typically each processor does self-scheduling from the pool of available processes or threads. The kernel can be constructed as multiple processes or multiple threads, allowing portions of the kernel to execute in parallel. The SMP approach complicates the OS. The OS designer must deal with the complexity due to sharing resources (such as data structures) and coordinating actions (such as accessing devices) from multiple parts of the OS executing at the same time. Techniques must be employed to resolve and synchronize claims to resources.

An SMP operating system manages processor and other computer resources so the user may view the system in the same fashion as a multiprogramming uniprocessor system. A user may construct applications that use multiple processes or multiple threads within processes without regard to whether a single processor or multiple processors will be available. Thus, a multiprocessor OS must provide all the functionality of a multiprogramming system, plus additional features to accommodate multiple processors. The key design issues include the following:

- **Simultaneous concurrent processes or threads:** Kernel routines need to be reentrant to allow several processors to execute the same kernel code simultaneously. With multiple processors executing the same or different parts of the kernel, kernel tables and management structures must be managed properly to avoid data corruption or invalid operations.
- **Scheduling:** Any processor may perform scheduling, which complicates the task of enforcing a scheduling policy and assuring that corruption of the scheduler data structures is avoided. If kernel-level multithreading is used, then the opportunity exists to schedule multiple threads from the same process simultaneously on multiple processors. Multiprocessor scheduling will be examined in Chapter 10.
- **Synchronization:** With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization. Synchronization is a facility that enforces mutual exclusion and event ordering. A common synchronization mechanism used in multiprocessor operating systems is locks, and will be described in Chapter 5.
- **Memory management:** Memory management on a multiprocessor must deal with all of the issues found on uniprocessor computers, and will be discussed in Part Three. In addition, the OS needs to exploit the available hardware parallelism to achieve the best performance. The paging mechanisms on different processors must be coordinated to enforce consistency when several processors share a page or segment and to decide on page replacement. The reuse of physical pages is the biggest problem of concern; that is, it must be guaranteed that a physical page can no longer be accessed with its old contents before the page is put to a new use.

- **Reliability and fault tolerance:** The OS should provide graceful degradation in the face of processor failure. The scheduler and other portions of the OS must recognize the loss of a processor and restructure management tables accordingly.

Because multiprocessor OS design issues generally involve extensions to solutions to multiprogramming uniprocessor design problems, we do not treat multiprocessor operating systems separately. Rather, specific multiprocessor issues are addressed in the proper context throughout this book.

Multicore OS Considerations

The considerations for multicore systems include all the design issues discussed so far in this section for SMP systems. But additional concerns arise. The issue is one of the scale of the potential parallelism. Current multicore vendors offer systems with ten or more cores on a single chip. With each succeeding processor technology generation, the number of cores and the amount of shared and dedicated cache memory increases, so we are now entering the era of "many-core" systems.

The design challenge for a many-core multicore system is to efficiently harness the multicore processing power and intelligently manage the substantial on-chip resources. A central concern is how to match the inherent parallelism of a many-core system with the performance requirements of applications. The potential for parallelism in fact exists at three levels in contemporary multicore system. First, there is hardware parallelism within each core processor, known as instruction level parallelism, which may or may not be exploited by application programmers and compilers. Second, there is the potential for multiprogramming and multithreaded execution within each processor. Finally, there is the potential for a single application to execute in concurrent processes or threads across multiple cores. Without strong and effective OS support for the last two types of parallelism just mentioned, hardware resources will not be efficiently used.

In essence, since the advent of multicore technology, OS designers have been struggling with the problem of how best to extract parallelism from computing workloads. A variety of approaches are being explored for next-generation operating systems. We will introduce two general strategies in this section, and will consider some details in later chapters.

Parallelism within Applications Most applications can, in principle, be subdivided into multiple tasks that can execute in parallel, with these tasks then being implemented as multiple processes, perhaps each with multiple threads. The difficulty is that the developer must decide how to split up the application work into independently executable tasks. That is, the developer must decide what pieces can or should be executed asynchronously or in parallel. It is primarily the compiler and the programming language features that support the parallel programming design process. But the OS can support this design process, at minimum, by efficiently allocating resources among parallel tasks as defined by the developer.

One of the most effective initiatives to support developers is Grand Central Dispatch (GCD), implemented in the latest release of the UNIX-based Mac OS X and the iOS operating systems. GCD is a multicore support capability. It does not help the developer decide how to break up a task or application into separate concurrent parts. But once a developer has identified something that can be split off into a separate task, GCD makes it as easy and noninvasive as possible to actually do so.

In essence, GCD is a thread pool mechanism, in which the OS maps tasks onto threads representing an available degree of concurrency (plus threads for blocking on I/O). Windows also has a thread pool mechanism (since 2000), and thread pools have been heavily used in server applications for years. What is new in GCD is the extension to programming languages to allow anonymous functions (called blocks) as a way of specifying tasks. GCD is hence not a major evolutionary step. Nevertheless, it is a new and valuable tool for exploiting the available parallelism of a multicore system.

One of Apple's slogans for GCD is "islands of serialization in a sea of concurrency." That captures the practical reality of adding more concurrency to run-of-the-mill desktop applications. Those islands are what isolate developers from the thorny problems of simultaneous data access, deadlock, and other pitfalls of multithreading. Developers are encouraged to identify functions of their applications that would be better executed off the main thread, even if they are made up of several sequential or otherwise partially interdependent tasks. GCD makes it easy to break off the entire unit of work while maintaining the existing order and dependencies between subtasks. In later chapters, we will look at some of the details of GCD.

Virtual Machine Approach An alternative approach is to recognize that with the ever-increasing number of cores on a chip, the attempt to multiprogram individual cores to support multiple applications may be a misplaced use of resources [JACK10]. If instead, we allow one or more cores to be dedicated to a particular process, then leave the processor alone to devote its efforts to that process, we avoid much of the overhead of task switching and scheduling decisions. The multicore OS could then act as a hypervisor that makes a high-level decision to allocate cores to applications, but does little in the way of resource allocation beyond that.

The reasoning behind this approach is as follows. In the early days of computing, one program was run on a single processor. With multiprogramming, each application is given the illusion that it is running on a dedicated processor. Multiprogramming is based on the concept of a process, which is an abstraction of an execution environment. To manage processes, the OS requires protected space, free from user and program interference. For this purpose, the distinction between kernel mode and user mode was developed. In effect, kernel mode and user mode abstracted the processor into two processors. With all these virtual processors, however, come struggles over who gets the attention of the real processor. The overhead of switching between all these processors starts to grow to the point where responsiveness suffers, especially when multiple cores are introduced. But with many-core systems, we can consider dropping the distinction between kernel and user mode. In this approach, the OS acts more like a hypervisor. The programs themselves take on many of the duties of resource management. The OS assigns an application, a processor and some memory, and the program itself, using metadata generated by the compiler, would best know how to use these resources.

2.7 MICROSOFT WINDOWS OVERVIEW

Background

Microsoft initially used the name Windows in 1985, for an operating environment extension to the primitive MS-DOS operating system, which was a successful OS used on early personal computers. This Windows/MS-DOS combination was ultimately replaced by a new version of Windows, known as Windows NT, first released in 1993, and intended for laptop and desktop systems. Although the basic internal architecture has remained roughly the same since Windows NT, the OS has continued to evolve with new functions and features. The latest release at the time of this writing is Windows 10. Windows 10 incorporates features from the preceding desktop/laptop release, Windows 8.1, as well as from versions of Windows intended for mobile devices for the Internet of Things (IoT). Windows 10 also incorporates software from the Xbox One system. The resulting unified Windows 10 supports desktops, laptops, smart phones, tablets, and Xbox One.

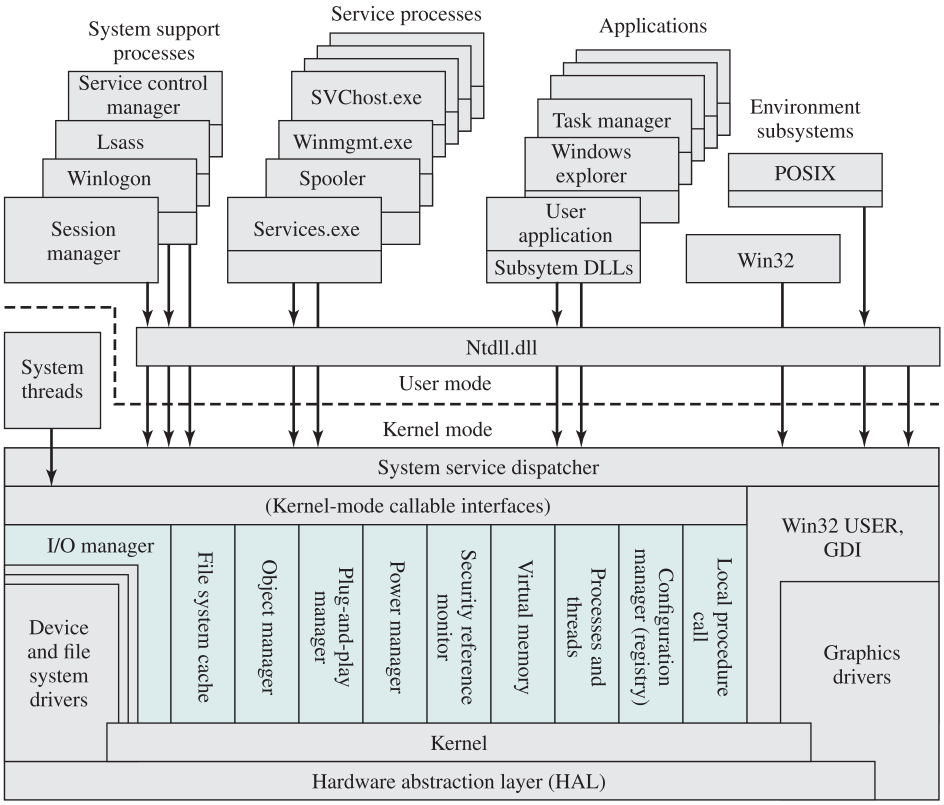
Architecture

Figure 2.14 illustrates the overall structure of Windows. As with virtually all operating systems, Windows separates application-oriented software from the core OS software. The latter, which includes the Executive, the Kernel, device drivers, and the hardware abstraction layer, runs in kernel mode. Kernel-mode software has access to system data and to the hardware. The remaining software, running in user mode, has limited access to system data.

Operating System Organization Windows has a highly modular architecture. Each system function is managed by just one component of the OS. The rest of the OS and all applications access that function through the responsible component using standard interfaces. Key system data can only be accessed through the appropriate function. In principle, any module can be removed, upgraded, or replaced without rewriting the entire system or its standard application program interfaces (APIs).

The kernel-mode components of Windows are the following:

- **Executive:** Contains the core OS services, such as memory management, process and thread management, security, I/O, and interprocess communication.
- **Kernel:** Controls execution of the processors. The Kernel manages thread scheduling, process switching, exception and interrupt handling, and multiprocessor synchronization. Unlike the rest of the Executive and the user levels, the Kernel's own code does not run in threads.
- **Hardware abstraction layer (HAL):** Maps between generic hardware commands and responses and those unique to a specific platform. It isolates the OS from platform-specific hardware differences. The HAL makes each computer's system bus, direct memory access (DMA) controller, interrupt controller,



Lsass = local security authentication server
POSIX = portable operating system interface
GDI = graphics device interface
DLL = dynamic link library

Colored area indicates Executive

Figure 2.14 Windows Internals Architecture [RUSS11]

system timers, and memory controller look the same to the Executive and kernel components. It also delivers the support needed for SMP, explained subsequently.

- **Device drivers:** Dynamic libraries that extend the functionality of the Executive. These include hardware device drivers that translate user I/O function calls into specific hardware device I/O requests, and software components for implementing file systems, network protocols, and any other system extensions that need to run in kernel mode.
- **Windowing and graphics system:** Implements the GUI functions, such as dealing with windows, user interface controls, and drawing.

The Windows Executive includes components for specific system functions and provides an API for user-mode software. Following is a brief description of each of the Executive modules:

- **I/O manager:** Provides a framework through which I/O devices are accessible to applications, and is responsible for dispatching to the appropriate device drivers for further processing. The I/O manager implements all the Windows I/O APIs and enforces security and naming for devices, network protocols, and file systems (using the object manager). Windows I/O will be discussed in Chapter 11.
- **Cache manager:** Improves the performance of file-based I/O by causing recently referenced file data to reside in main memory for quick access, and by deferring disk writes by holding the updates in memory for a short time before sending them to the disk in more efficient batches.
- **Object manager:** Creates, manages, and deletes Windows Executive objects that are used to represent resources such as processes, threads, and synchronization objects. It enforces uniform rules for retaining, naming, and setting the security of objects. The object manager also creates the entries in each process's handle table, which consist of access control information and a pointer to the object. Windows objects will be discussed later in this section.
- **Plug-and-play manager:** Determines which drivers are required to support a particular device and loads those drivers.
- **Power manager:** Coordinates power management among various devices and can be configured to reduce power consumption by shutting down idle devices, putting the processor to sleep, and even writing all of memory to disk and shutting off power to the entire system.
- **Security reference monitor:** Enforces access-validation and audit-generation rules. The Windows object-oriented model allows for a consistent and uniform view of security, right down to the fundamental entities that make up the Executive. Thus, Windows uses the same routines for access validation and for audit checks for all protected objects, including files, processes, address spaces, and I/O devices. Windows security will be discussed in Chapter 15.
- **Virtual memory manager:** Manages virtual addresses, physical memory, and the paging files on disk. Controls the memory management hardware and data structures which map virtual addresses in the process's address space to physical pages in the computer's memory. Windows virtual memory management will be described in Chapter 8.
- **Process/thread manager:** Creates, manages, and deletes process and thread objects. Windows process and thread management will be described in Chapter 4.
- **Configuration manager:** Responsible for implementing and managing the system registry, which is the repository for both system-wide and per-user settings of various parameters.

- **Advanced local procedure call (ALPC) facility:** Implements an efficient cross-process procedure call mechanism for communication between local processes implementing services and subsystems. Similar to the remote procedure call (RPC) facility used for distributed processing.

User-Mode Processes Windows supports four basic types of user-mode processes:

1. **Special system processes:** User-mode services needed to manage the system, such as the session manager, the authentication subsystem, the service manager, and the logon process.
2. **Service processes:** The printer spooler, the event logger, user-mode components that cooperate with device drivers, various network services, and many others. Services are used by both Microsoft and external software developers to extend system functionality, as they are the only way to run background user-mode activity on a Windows system.
3. **Environment subsystems:** Provide different OS personalities (environments). The supported subsystems are Win32 and POSIX. Each environment subsystem includes a subsystem process shared among all applications using the subsystem and dynamic link libraries (DLLs) that convert the user application calls to ALPC calls on the subsystem process, and/or native Windows calls.
4. **User applications:** Executables (EXEs) and DLLs that provide the functionality users run to make use of the system. EXEs and DLLs are generally targeted at a specific environment subsystem; although some of the programs that are provided as part of the OS use the native system interfaces (NT API). There is also support for running 32-bit programs on 64-bit systems.

Windows is structured to support applications written for multiple OS personalities. Windows provides this support using a common set of kernel-mode components that underlie the OS environment subsystems. The implementation of each environment subsystem includes a separate process, which contains the shared data structures, privileges, and Executive object handles needed to implement a particular personality. The process is started by the Windows Session Manager when the first application of that type is started. The subsystem process runs as a system user, so the Executive will protect its address space from processes run by ordinary users.

An environment subsystem provides a graphical or command-line user interface that defines the look and feel of the OS for a user. In addition, each subsystem provides the API for that particular environment. This means that applications created for a particular operating environment need only be recompiled to run on Windows. Because the OS interface that applications see is the same as that for which they were written, the source code does not need to be modified.

Client/Server Model

The Windows OS services, the environment subsystems, and the applications are structured using the client/server computing model, which is a common model for distributed computing and will be discussed in Part Six. This same architecture can be adopted for use internally to a single system, as is the case with Windows.

The native NT API is a set of kernel-based services which provide the core abstractions used by the system, such as processes, threads, virtual memory, I/O, and communication. Windows provides a far richer set of services by using the client/server model to implement functionality in user-mode processes. Both the environment subsystems and the Windows user-mode services are implemented as processes that communicate with clients via RPC. Each server process waits for a request from a client for one of its services (e.g., memory services, process creation services, or networking services). A client, which can be an application program or another server program, requests a service by sending a message. The message is routed through the Executive to the appropriate server. The server performs the requested operation and returns the results or status information by means of another message, which is routed through the Executive back to the client.

Advantages of a client/server architecture include the following:

- **It simplifies the Executive.** It is possible to construct a variety of APIs implemented in user-mode servers without any conflicts or duplications in the Executive. New APIs can be added easily.
- **It improves reliability.** Each new server runs outside of the kernel, with its own partition of memory, protected from other servers. A single server can fail without crashing or corrupting the rest of the OS.
- **It provides a uniform means for applications to communicate with services via RPCs without restricting flexibility.** The message-passing process is hidden from the client applications by function stubs, which are small pieces of code which wrap the RPC call. When an application makes an API call to an environment subsystem or a service, the stub in the client application packages the parameters for the call and sends them as a message to the server process that implements the call.
- **It provides a suitable base for distributed computing.** Typically, distributed computing makes use of a client/server model, with remote procedure calls implemented using distributed client and server modules and the exchange of messages between clients and servers. With Windows, a local server can pass a message on to a remote server for processing on behalf of local client applications. Clients need not know whether a request is being serviced locally or remotely. Indeed, whether a request is serviced locally or remotely can change dynamically, based on current load conditions and on dynamic configuration changes.

Threads and SMP

Two important characteristics of Windows are its support for threads and for symmetric multiprocessing (SMP), both of which were introduced in Section 2.4. [RUSS11] lists the following features of Windows that support threads and SMP:

- OS routines can run on any available processor, and different routines can execute simultaneously on different processors.

- Windows supports the use of multiple threads of execution within a single process. Multiple threads within the same process may execute on different processors simultaneously.
- Server processes may use multiple threads to process requests from more than one client simultaneously.
- Windows provides mechanisms for sharing data and resources between processes and flexible interprocess communication capabilities.

Windows Objects

Though the core of Windows is written in C, the design principles followed draw heavily on the concepts of object-oriented design. This approach facilitates the sharing of resources and data among processes, and the protection of resources from unauthorized access. Among the key object-oriented concepts used by Windows are the following:

- **Encapsulation:** An object consists of one or more items of data, called *attributes*, and one or more procedures that may be performed on those data, called *services*. The only way to access the data in an object is by invoking one of the object's services. Thus, the data in the object can easily be protected from unauthorized use and from incorrect use (e.g., trying to execute a nonexecutable piece of data).
- **Object class and instance:** An object class is a template that lists the attributes and services of an object, and defines certain object characteristics. The OS can create specific instances of an object class as needed. For example, there is a single process object class and one process object for every currently active process. This approach simplifies object creation and management.
- **Inheritance:** Although the implementation is hand coded, the Executive uses inheritance to extend object classes by adding new features. Every Executive class is based on a base class which specifies virtual methods that support creating, naming, securing, and deleting objects. Dispatcher objects are Executive objects that inherit the properties of an event object, so they can use common synchronization methods. Other specific object types, such as the device class, allow classes for specific devices to inherit from the base class, and add additional data and methods.
- **Polymorphism:** Internally, Windows uses a common set of API functions to manipulate objects of any type; this is a feature of polymorphism, as defined in Appendix D. However, Windows is not completely polymorphic because there are many APIs that are specific to a single object type.

The reader unfamiliar with object-oriented concepts should review Appendix D.

Not all entities in Windows are objects. Objects are used in cases where data are intended for user-mode access, or when data access is shared or restricted. Among the entities represented by objects are files, processes, threads, semaphores, timers, and graphical windows. Windows creates and manages all types of objects in a uniform way, via the object manager. The object manager is responsible for creating and destroying objects on behalf of applications, and for granting access to an object's services and data.

Each object within the Executive, sometimes referred to as a kernel object (to distinguish from user-level objects not of concern to the Executive), exists as a memory block allocated by the kernel and is directly accessible only by kernel-mode components. Some elements of the data structure are common to all object types (e.g., object name, security parameters, usage count), while other elements are specific to a particular object type (e.g., a thread object's priority). Because these object data structures are in the part of each process's address space accessible only by the kernel, it is impossible for an application to reference these data structures and read or write them directly. Instead, applications manipulate objects indirectly through the set of object manipulation functions supported by the Executive. When an object is created, the application that requested the creation receives back a handle for the object. In essence, a handle is an index into a per-process Executive table containing a pointer to the referenced object. This handle can then be used by any thread within the same process to invoke Win32 functions that work with objects, or can be duplicated into other processes.

Objects may have security information associated with them, in the form of a Security Descriptor (SD). This security information can be used to restrict access to the object based on contents of a token object which describes a particular user. For example, a process may create a named semaphore object with the intent that only certain users should be able to open and use that semaphore. The SD for the semaphore object can list those users that are allowed (or denied) access to the semaphore object along with the sort of access permitted (read, write, change, etc.).

In Windows, objects may be either named or unnamed. When a process creates an unnamed object, the object manager returns a handle to that object, and the handle is the only way to refer to it. Handles can be inherited by child processes or duplicated between processes. Named objects are also given a name that other unrelated processes can use to obtain a handle to the object. For example, if process A wishes to synchronize with process B, it could create a named event object and pass the name of the event to B. Process B could then open and use that event object. However, if process A simply wished to use the event to synchronize two threads within itself, it would create an unnamed event object, because there is no need for other processes to be able to use that event.

There are two categories of objects used by Windows for synchronizing the use of the processor:

- **Dispatcher objects:** The subset of Executive objects which threads can wait on to control the dispatching and synchronization of thread-based system operations. These will be described in Chapter 6.
- **Control objects:** Used by the Kernel component to manage the operation of the processor in areas not managed by normal thread scheduling. Table 2.5 lists the Kernel control objects.

Windows is not a full-blown object-oriented OS. It is not implemented in an object-oriented language. Data structures that reside completely within one Executive component are not represented as objects. Nevertheless, Windows illustrates the power of object-oriented technology and represents the increasing trend toward the use of this technology in OS design.

Table 2.5 Windows Kernel Control Objects

Asynchronous procedure call	Used to break into the execution of a specified thread and to cause a procedure to be called in a specified processor mode.
Deferred procedure call	Used to postpone interrupt processing to avoid delaying hardware interrupts. Also used to implement timers and interprocessor communication.
Interrupt	Used to connect an interrupt source to an interrupt service routine by means of an entry in an Interrupt Dispatch Table (IDT). Each processor has an IDT that is used to dispatch interrupts that occur on that processor.
Process	Represents the virtual address space and control information necessary for the execution of a set of thread objects. A process contains a pointer to an address map, a list of ready threads containing thread objects, a list of threads belonging to the process, the total accumulated time for all threads executing within the process, and a base priority.
Thread	Represents thread objects, including scheduling priority and quantum, and which processors the thread may run on.
Profile	Used to measure the distribution of run time within a block of code. Both user and system codes can be profiled.

2.8 TRADITIONAL UNIX SYSTEMS

History

UNIX was initially developed at Bell Labs and became operational on a PDP-7 in 1970. Work on UNIX at Bell Labs, and later elsewhere, produced a series of versions of UNIX. The first notable milestone was porting the UNIX system from the PDP-7 to the PDP-11. This was the first hint that UNIX would be an OS for all computers. The next important milestone was the rewriting of UNIX in the programming language C. This was an unheard-of strategy at the time. It was generally felt that something as complex as an OS, which must deal with time-critical events, had to be written exclusively in assembly language. Reasons for this attitude include the following:

- Memory (both RAM and secondary store) was small and expensive by today's standards, so effective use was important. This included various techniques for overlaying memory with different code and data segments, and self-modifying code.
- Even though compilers had been available since the 1950s, the computer industry was generally skeptical of the quality of automatically generated code. With resource capacity small, efficient code, both in terms of time and space, was essential.
- Processor and bus speeds were relatively slow, so saving clock cycles could make a substantial difference in execution time.

The C implementation demonstrated the advantages of using a high-level language for most if not all of the system code. Today, virtually all UNIX implementations are written in C.

These early versions of UNIX were popular within Bell Labs. In 1974, the UNIX system was described in a technical journal for the first time [RITC74]. This spurred great interest in the system. Licenses for UNIX were provided to commercial institutions as well as universities. The first widely available version outside Bell Labs was Version 6, in 1976. The follow-on Version 7, released in 1978, is the ancestor of most modern UNIX systems. The most important of the non-AT&T systems to be developed was done at the University of California at Berkeley, called UNIX BSD (Berkeley Software Distribution), running first on PDP and then on VAX computers. AT&T continued to develop and refine the system. By 1982, Bell Labs had combined several AT&T variants of UNIX into a single system, marketed commercially as UNIX System III. A number of features was later added to the OS to produce UNIX System V.

Description

The classic UNIX architecture can be pictured as in three levels: hardware, kernel, and user. The OS is often called the system kernel, or simply the kernel, to emphasize its isolation from the user and applications. It interacts directly with the hardware. It is the UNIX kernel that we will be concerned with in our use of UNIX as an example in this book. UNIX also comes equipped with a number of user services and interfaces that are considered part of the system. These can be grouped into the shell, which supports system calls from applications, other interface software, and the components of the C compiler (compiler, assembler, loader). The level above this consists of user applications and the user interface to the C compiler.

A look at the kernel is provided in Figure 2.15. User programs can invoke OS services either directly, or through library programs. The system call interface is the boundary with the user and allows higher-level software to gain access to specific kernel functions. At the other end, the OS contains primitive routines that interact directly with the hardware. Between these two interfaces, the system is divided into two main parts: one concerned with process control, and the other concerned with file management and I/O. The process control subsystem is responsible for memory management, the scheduling and dispatching of processes, and the synchronization and interprocess communication of processes. The file system exchanges data between memory and external devices either as a stream of characters or in blocks. To achieve this, a variety of device drivers are used. For block-oriented transfers, a disk cache approach is used: A system buffer in main memory is interposed between the user address space and the external device.

The description in this subsection has dealt with what might be termed *traditional UNIX systems*; [VAHA96] uses this term to refer to System V Release 3 (SVR3), 4.3BSD, and earlier versions. The following general statements may be made about a traditional UNIX system. It is designed to run on a single processor, and lacks the ability to protect its data structures from concurrent access by multiple processors. Its kernel is not very versatile, supporting a single type of file system, process scheduling policy, and executable file format. The traditional UNIX kernel is not designed to be extensible and has few facilities for code reuse. The result is that, as new features were added to the various UNIX versions, much new code had to be added, yielding a bloated and unmodular kernel.

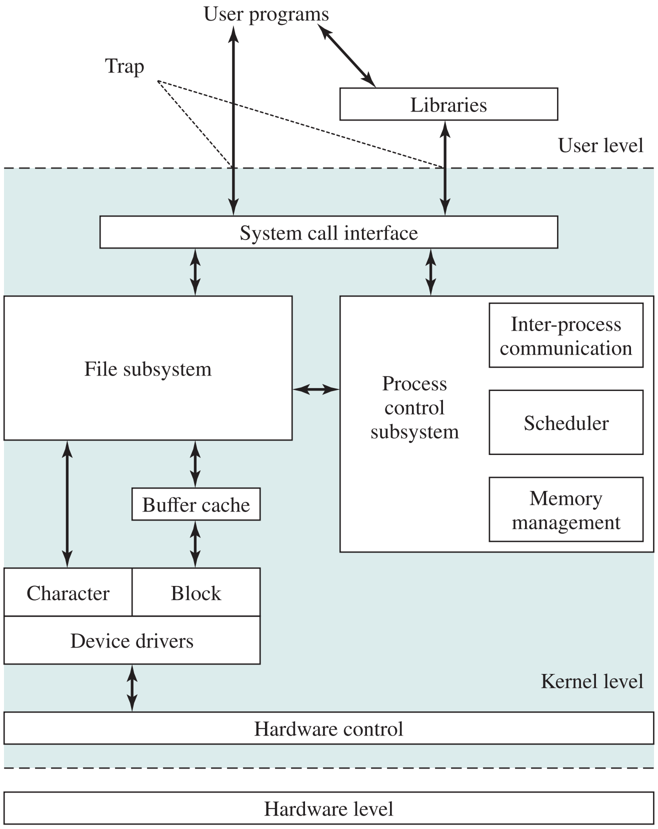


Figure 2.15 Traditional UNIX Architecture

2.9 MODERN UNIX SYSTEMS

As UNIX evolved, the number of different implementations proliferated, each providing some useful features. There was a need to produce a new implementation that unified many of the important innovations, added other modern OS design features, and produced a more modular architecture. Typical of the modern UNIX kernel is the architecture depicted in Figure 2.16. There is a small core of facilities, written in a modular fashion, that provide functions and services needed by a number of OS processes. Each of the outer circles represents functions and an interface that may be implemented in a variety of ways.

We now turn to some examples of modern UNIX systems (see Figure 2.17).

System V Release 4 (SVR4)

SVR4, developed jointly by AT&T and Sun Microsystems, combines features from SVR3, 4.3BSD, Microsoft Xenix System V, and SunOS. It was almost a total rewrite of the System V kernel and produced a clean, if complex, implementation. New features in the release include real-time processing support, process scheduling classes,

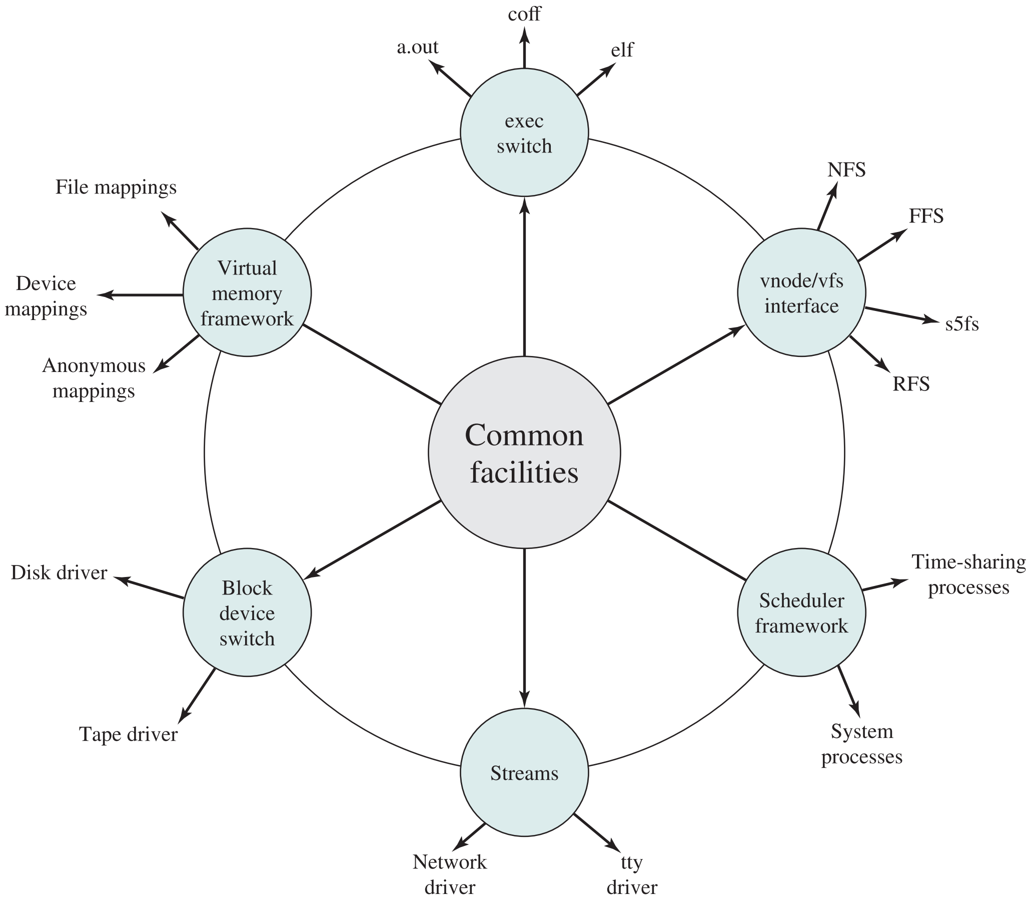


Figure 2.16 **Modern UNIX Kernel**

dynamically allocated data structures, virtual memory management, virtual file system, and a preemptive kernel.

SVR4 draws on the efforts of both commercial and academic designers, and was developed to provide a uniform platform for commercial UNIX deployment. It has succeeded in this objective and is perhaps the most important UNIX variant. It incorporates most of the important features ever developed on any UNIX system and does so in an integrated, commercially viable fashion. SVR4 runs on processors ranging from 32-bit microprocessors up to supercomputers.

BSD

The Berkeley Software Distribution (BSD) series of UNIX releases have played a key role in the development of OS design theory. 4.xBSD is widely used in academic installations and has served as the basis of a number of commercial UNIX products. It is probably safe to say that BSD is responsible for much of the popularity of UNIX, and that most enhancements to UNIX first appeared in BSD versions.

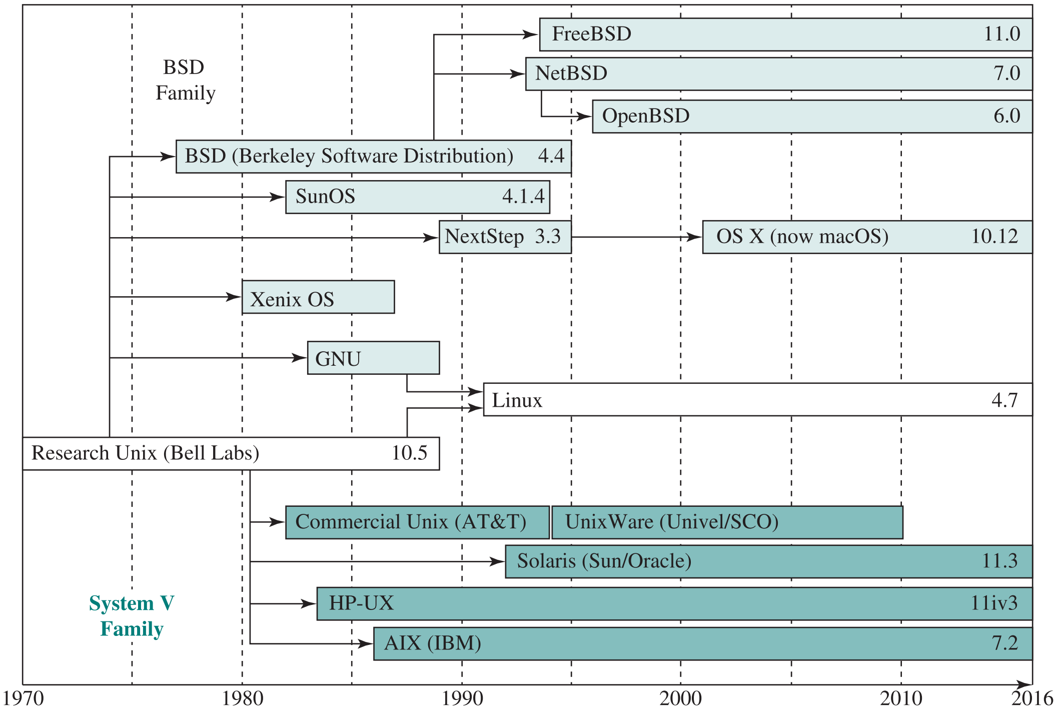


Figure 2.17 UNIX Family Tree

4.4BSD was the final version of BSD to be released by Berkeley, with the design and implementation organization subsequently dissolved. It is a major upgrade to 4.3BSD and includes a new virtual memory system, changes in the kernel structure, and a long list of other feature enhancements.

There are several widely used, open-source versions of BSD. FreeBSD is popular for Internet-based servers and firewalls and is used in a number of embedded systems. NetBSD is available for many platforms, including large-scale server systems, desktop systems, and handheld devices, and is often used in embedded systems. OpenBSD is an open-source OS that places special emphasis on security.

The latest version of the Macintosh OS, originally known as OS X and now called MacOS, is based on FreeBSD 5.0 and the Mach 3.0 microkernel.

Solaris 11

Solaris is Oracle's SVR4-based UNIX release, with the latest version being 11. Solaris provides all of the features of SVR4 plus a number of more advanced features, such as a fully preemptable, multithreaded kernel, full support for SMP, and an object-oriented interface to file systems. Solaris is one the most widely used and most successful commercial UNIX implementations.

2.10 LINUX

History

Linux started out as a UNIX variant for the IBM PC (Intel 80386) architecture. Linus Torvalds, a Finnish student of computer science, wrote the initial version. Torvalds posted an early version of Linux on the Internet in 1991. Since then, a number of people, collaborating over the Internet, have contributed to the development of Linux, all under the control of Torvalds. Because Linux is free and the source code is available, it became an early alternative to other UNIX workstations, such as those offered by Sun Microsystems and IBM. Today, Linux is a full-featured UNIX system that runs on virtually all platforms.

Key to the success of Linux has been the availability of free software packages under the auspices of the Free Software Foundation (FSF). FSF's goal is stable, platform-independent software that is free, high quality, and embraced by the user community. FSF's GNU project[2] provides tools for software developers, and the GNU Public License (GPL) is the FSF seal of approval. Torvalds used GNU tools in developing his kernel, which he then released under the GPL. Thus, the Linux distributions that you see today are the product of FSF's GNU project, Torvald's individual effort, and the efforts of many collaborators all over the world.

In addition to its use by many individual developers, Linux has now made significant penetration into the corporate world. This is not only because of the free software, but also because of the quality of the Linux kernel. Many talented developers have contributed to the current version, resulting in a technically impressive product. Moreover, Linux is highly modular and easily configured. This makes it easy to squeeze optimal performance from a variety of hardware platforms. Plus, with the source code available, vendors can tweak applications and utilities to meet specific requirements. There are also commercial companies such as Red Hat and Canonical, which provide highly professional and reliable support for their Linux-based distributions for long periods of time. Throughout this book, we will provide details of Linux kernel internals based on Linux kernel 4.7, released in 2016.

A large part of the success of the Linux Operating System is due to its development model. Code contributions are handled by one main mailing list, called LKML (Linux Kernel Mailing List). Apart from it, there are many other mailing lists, each dedicated to a Linux kernel subsystem (like the netdev mailing list for networking, the linux-pci for the PCI subsystem, the linux-acpi for the ACPI subsystem, and a great many more). The patches which are sent to these mailing lists should adhere to strict rules (primarily the Linux Kernel coding style conventions), and are reviewed by developers all over the world who are subscribed to these mailing lists. Anyone can send patches to these mailing lists; statistics (for example, those published in the lwn.net site from time to time) show that many patches are sent by developers from famous commercial companies like Intel, Red Hat, Google, Samsung, and others. Also, many maintainers are employees of commercial companies (like David

[2] GNU is a recursive acronym for *GNU's Not Unix*. The GNU project is a free software set of packages and tools for developing a UNIX-like operating system; it is often used with the Linux kernel.

Miller, the network maintainer, who works for Red Hat). Many times such patches are fixed according to feedback and discussions over the mailing list, and are resent and reviewed again. Eventually, the maintainer decides whether to accept or reject patches; and each subsystem maintainer from time to time sends a pull request of his tree to the main kernel tree, which is handled by Linus Torvalds. Linus himself releases a new kernel version in about every 7–10 weeks, and each such release has about 5–8 release candidates (RC) versions.

We should mention that it is interesting to try to understand why other open-source operating systems, such as various flavors of BSD or OpenSolaris, did not have the success and popularity which Linux has; there can be many reasons for that, and for sure, the openness of the development model of Linux contributed to its popularity and success. But this topic is out of the scope of this book.

Modular Structure

Most UNIX kernels are monolithic. Recall from earlier in this chapter, a monolithic kernel is one that includes virtually all of the OS functionality in one large block of code that runs as a single process with a single address space. All the functional components of the kernel have access to all of its internal data structures and routines. If changes are made to any portion of a typical monolithic OS, all the modules and routines must be relinked and reinstalled, and the system rebooted, before the changes can take effect. As a result, any modification, such as adding a new device driver or file system function, is difficult. This problem is especially acute for Linux, for which development is global and done by a loosely associated group of independent developers.

Although Linux does not use a microkernel approach, it achieves many of the potential advantages of this approach by means of its particular modular architecture. Linux is structured as a collection of modules, a number of which can be automatically loaded and unloaded on demand. These relatively independent blocks are referred to as **loadable modules** [GOYE99]. In essence, a module is an object file whose code can be linked to and unlinked from the kernel at runtime. Typically, a module implements some specific function, such as a file system, a device driver, or some other feature of the kernel's upper layer. A module does not execute as its own process or thread, although it can create kernel threads for various purposes as necessary. Rather, a module is executed in kernel mode on behalf of the current process.

Thus, although Linux may be considered monolithic, its modular structure overcomes some of the difficulties in developing and evolving the kernel. The Linux loadable modules have two important characteristics:

1. **Dynamic linking:** A kernel module can be loaded and linked into the kernel while the kernel is already in memory and executing. A module can also be unlinked and removed from memory at any time.
2. **Stackable modules:** The modules are arranged in a hierarchy. Individual modules serve as libraries when they are referenced by client modules higher up in the hierarchy, and as clients when they reference modules further down.

Dynamic linking facilitates configuration and saves kernel memory [FRAN97]. In Linux, a user program or user can explicitly load and unload kernel modules using the `insmod` or `modprobe` and `rmmod` commands. The kernel itself monitors the need

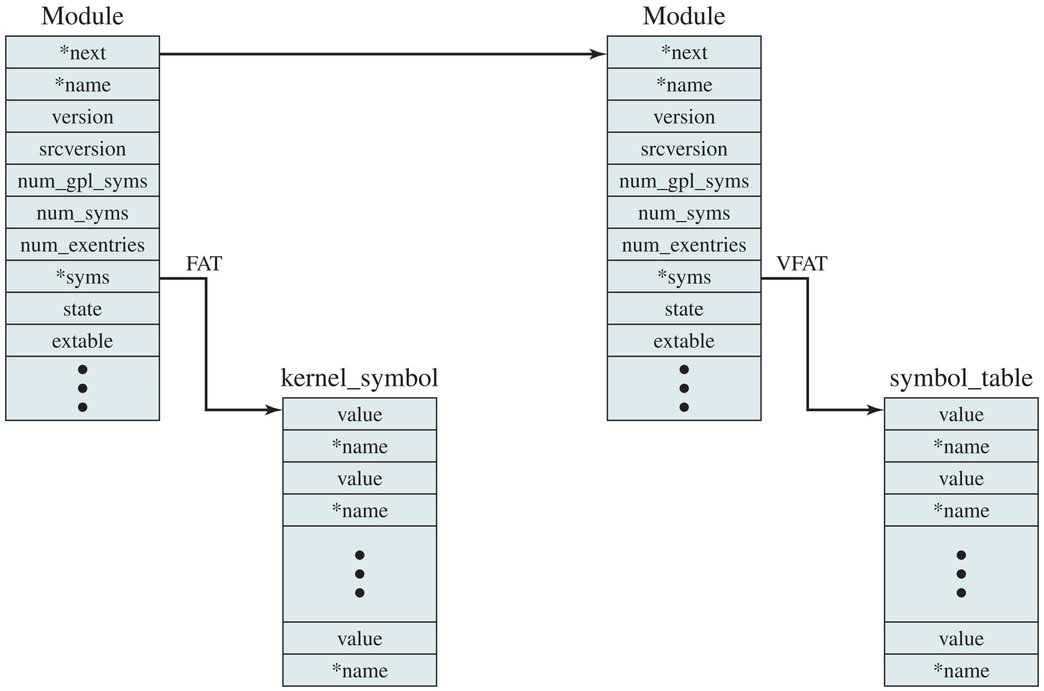


Figure 2.18 Example List of Linux Kernel Modules

for particular functions, and can load and unload modules as needed. With stackable modules, dependencies between modules can be defined. This has two benefits:

1. Code common to a set of similar modules (e.g., drivers for similar hardware) can be moved into a single module, reducing replication.
2. The kernel can make sure that needed modules are present, refraining from unloading a module on which other running modules depend, and loading any additional required modules when a new module is loaded.

Figure 2.18 is an example that illustrates the structures used by Linux to manage modules. The figure shows the list of kernel modules after only two modules have been loaded: FAT and VFAT. Each module is defined by two tables: the module table and the symbol table (kernel_symbol). The module table includes the following elements:

- ***name:** The module name
- **refcnt:** Module counter. The counter is incremented when an operation involving the module's functions is started and decremented when the operation terminates.
- **num_syms:** Number of exported symbols.
- ***syms:** Pointer to this module's symbol table.

The symbol table lists symbols that are defined in this module and used elsewhere.

Kernel Components

Figure 2.19, taken from [MOSB02], shows the main components of a typical Linux kernel implementation. The figure shows several processes running on top of the kernel. Each box indicates a separate process, while each squiggly line with an arrowhead represents a thread of execution. The kernel itself consists of an interacting collection of components, with arrows indicating the main interactions. The underlying hardware is also depicted as a set of components with arrows indicating which kernel components use or control which hardware components. All of the kernel components, of course, execute on the processor. For simplicity, these relationships are not shown.

Briefly, the principal kernel components are the following:

- **Signals:** The kernel uses signals to call into a process. For example, signals are used to notify a process of certain faults, such as division by zero. Table 2.6 gives a few examples of signals.
- **System calls:** The system call is the means by which a process requests a specific kernel service. There are several hundred system calls, which can be roughly grouped into six categories: file system, process, scheduling, interprocess communication, socket (networking), and miscellaneous. Table 2.7 defines a few examples in each category.
- **Processes and scheduler:** Creates, manages, and schedules processes.
- **Virtual memory:** Allocates and manages virtual memory for processes.

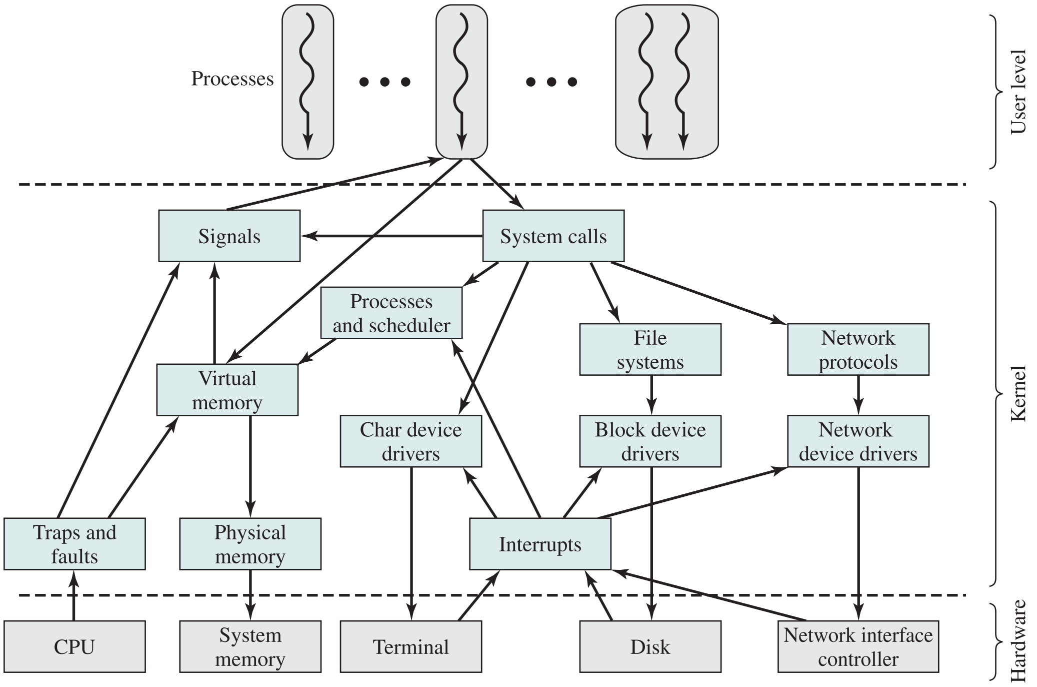


Figure 2.19 Linux Kernel Components

Table 2.6 Some Linux Signals

SIGHUP	Terminal hangup	SIGCONT	Continue
SIGQUIT	Keyboard quit	SIGTSTP	Keyboard stop
SIGTRAP	Trace trap	SIGTTOU	Terminal write
SIGBUS	Bus error	SIGXCPU	CPU limit exceeded
SIGKILL	Kill signal	SIGVTALRM	Virtual alarm clock
SIGSEGV	Segmentation violation	SIGWINCH	Window size unchanged
SIGPIPT	Broken pipe	SIGPWR	Power failure
SIGTERM	Termination	SIGRTMIN	First real-time signal
SIGCHLD	Child status unchanged	SIGRTMAX	Last real-time signal

- **File systems:** Provide a global, hierarchical namespace for files, directories, and other file-related objects and provide file system functions.
- **Network protocols:** Support the Sockets interface to users for the TCP/IP protocol suite.

Table 2.7 Some Linux System Calls

<table>
<tr><th colspan="2">File System Related</th></tr>
<tr><td><b>close</b></td><td>Close a file descriptor.</td></tr>
<tr><td><b>link</b></td><td>Make a new name for a file.</td></tr>
<tr><td><b>open</b></td><td>Open and possibly create a file or device.</td></tr>
<tr><td><b>read</b></td><td>Read from file descriptor.</td></tr>
<tr><td><b>write</b></td><td>Write to file descriptor.</td></tr>
<tr><th colspan="2">Process Related</th></tr>
<tr><td><b>execve</b></td><td>Execute program.</td></tr>
<tr><td><b>exit</b></td><td>Terminate the calling process.</td></tr>
<tr><td><b>getpid</b></td><td>Get process identification.</td></tr>
<tr><td><b>setuid</b></td><td>Set user identity of the current process.</td></tr>
<tr><td><b>ptrace</b></td><td>Provide a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers.</td></tr>
<tr><th colspan="2">Scheduling Related</th></tr>
<tr><td><b>sched_getparam</b></td><td>Set the scheduling parameters associated with the scheduling policy for the process identified by <code>pid</code>.</td></tr>
<tr><td><b>sched_get_priority_max</b></td><td>Return the maximum priority value that can be used with the scheduling algorithm identified by <code>policy</code>.</td></tr>
<tr><td><b>sched_setscheduler</b></td><td>Set both the scheduling policy (e.g., FIFO) and the associated parameters for the process <code>pid</code>.</td></tr>
<tr><td><b>sched_rr_get_interval</b></td><td>Write into the timespec structure pointed to by the parameter to the round-robin time quantum for the process <code>pid</code>.</td></tr>
<tr><td><b>sched_yield</b></td><td>A process can relinquish the processor voluntarily without blocking via this system call. The process will then be moved to the end of the queue for its static priority and a new process gets to run.</td></tr>
</table>

Table 2.7 (*Continued*)

Interprocess Communication (IPC) Related	
msgrcv	A message buffer structure is allocated to receive a message. The system call then reads a message from the message queue specified by `msqid` into the newly created message buffer.
semctl	Perform the control operation specified by `cmd` on the semaphore set `semid`.
semop	Perform operations on selected members of the semaphore set `semid`.
shmat	Attach the shared memory segment identified by `semid` to the data segment of the calling process.
shmctl	Allow the user to receive information on a shared memory segment; set the owner, group, and permissions of a shared memory segment; or destroy a segment.
Socket (networking) Related	
bind	Assign the local IP address and port for a socket. Return 0 for success or −1 for error.
connect	Establish a connection between the given socket and the remote socket associated with sockaddr.
gethostname	Return local host name.
send	Send the bytes contained in buffer pointed to by *msg over the given socket.
setsockopt	Set the options on a socket.
Miscellaneous	
fsync	Copy all in-core parts of a file to disk, and wait until the device reports that all parts are on stable storage.
time	Return the time in seconds since January 1, 1970.
vhangup	Simulate a hangup on the current terminal. This call arranges for other users to have a “clean” tty at login time.

- **Character device drivers:** Manage devices that require the kernel to send or receive data one byte at a time, such as terminals, modems, and printers.
- **Block device drivers:** Manage devices that read and write data in blocks, such as various forms of secondary memory (magnetic disks, CD-ROMs, etc.).
- **Network device drivers:** Manage network interface cards and communications ports that connect to network devices, such as bridges and routers.
- **Traps and faults:** Handle traps and faults generated by the processor, such as a memory fault.
- **Physical memory:** Manages the pool of page frames in real memory and allocates pages for virtual memory.
- **Interrupts** Handle interrupts from peripheral devices.

2.11 ANDROID

The Android operating system is a Linux-based system originally designed for mobile phones. It is the most popular mobile OS by a wide margin: Android handsets outsell Apple’s iPhones globally by about 4 to 1 [MORR16]. But, this is just one element in the increasing dominance of Android. Increasingly, it is the OS behind virtually any device with a computer chip other than servers and PCs. Android is a widely used OS for the Internet of things.

Initial Android OS development was done by Android, Inc., which was bought by Google in 2005. The first commercial version, Android 1.0, was released in 2008. As of this writing, the most recent version is Android 7.0 (Nougat). Android has an active community of developers and enthusiasts who use the Android Open Source Project (AOSP) source code to develop and distribute their own modified versions of the operating system. The open-source nature of Android has been the key to its success.

Android Software Architecture

Android is defined as a software stack that includes a modified version of the Linux kernel, middleware, and key applications. Figure 2.20 shows the Android software architecture in some detail. Thus, Android should be viewed as a complete software stack, not just an OS.

Applications All the applications with which the user interacts directly are part of the application layer. This includes a core set of general-purpose applications, such as an e-mail client, SMS program, calendar, maps, browser, contacts, and other applications commonly standard with any mobile device. Applications are typically implemented in Java. A key goal of the open-source Android architecture is to make it easy for developers to implement new applications for specific devices and specific end-user requirements. Using Java enables developers to be relieved of hardware-specific considerations and idiosyncrasies, as well as tap into Java's higher-level language features, such as predefined classes. Figure 2.20 shows examples of the types of base applications found on the Android platform.

Application Framework The Application Framework layer provides high-level building blocks, accessible through standardized APIs, that programmers use to create new apps. The architecture is designed to simplify the reuse of components. Some of the key Application Framework components are:

- **Activity Manager:** Manages lifecycle of applications. It is responsible for starting, pausing, and resuming the various applications.
- **Window Manager:** Java abstraction of the underlying Surface Manager. The Surface Manager handles the frame buffer interaction and low-level drawing, whereas the Window Manager provides a layer on top of it, to allow applications to declare their client area and use features like the status bar.
- **Package Manager:** Installs and removes applications.
- **Telephony Manager:** Allows interaction with phone, SMS, and MMS services.
- **Content Providers:** These functions encapsulate application data that need to be shared between applications, such as contacts.
- **Resource Manager:** Manages application resources, such as localized strings and bitmaps.

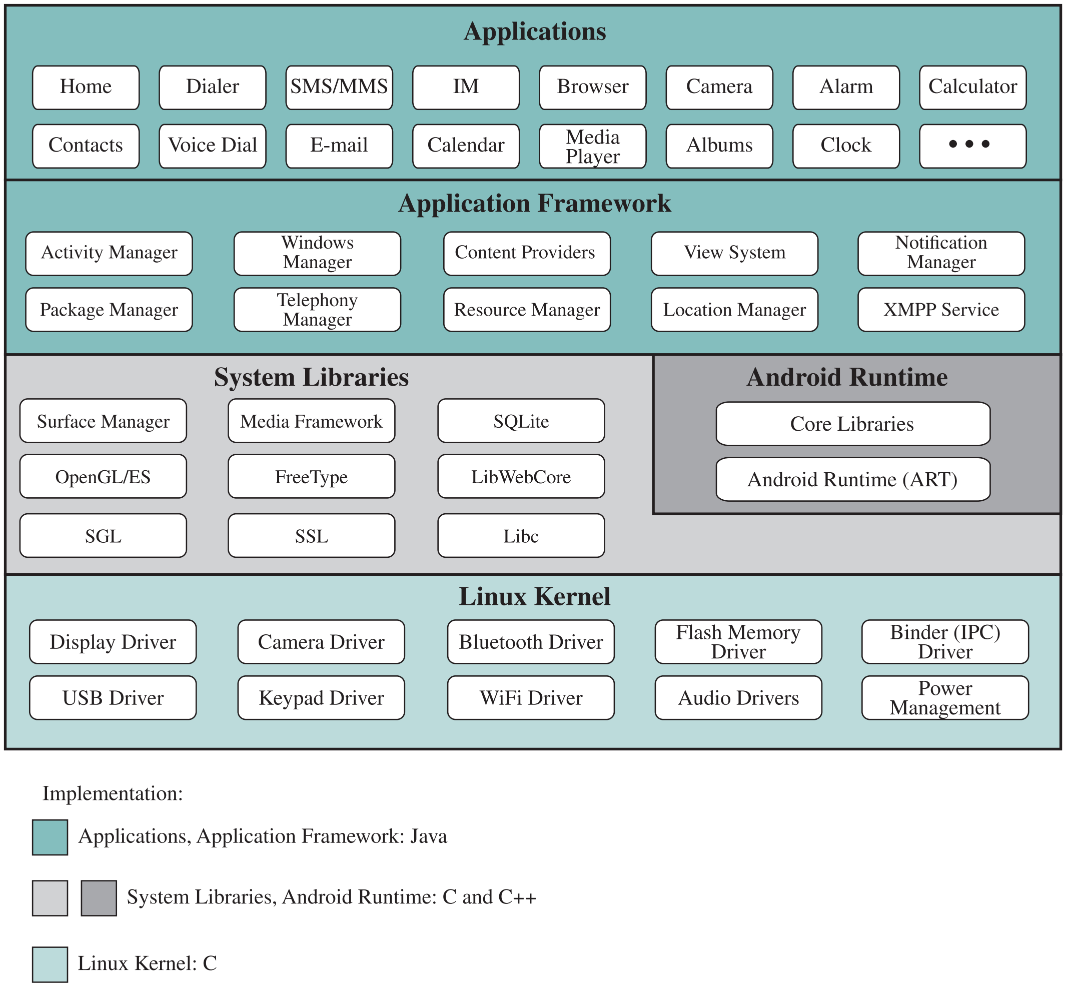


Figure 2.20 **Android Software Architecture**

- **View System:** Provides the user interface (UI) primitives, such as buttons, listboxes, date pickers, and other controls, as well as UI Events (such as touch and gestures).
- **Location Manager:** Allows developers to tap into location-based services, whether by GPS, cell tower IDs, or local Wi-Fi databases. (recognized Wi-Fi hotspots and their status)
- **Notification Manager:** Manages events, such as arriving messages and appointments.
- **XMPP:** Provides standardized messaging (also, Chat) functions between applications.

System Libraries The layer below the Application Framework consists of two parts: System Libraries, and Android Runtime. The System Libraries component is a collection of useful system functions, written in C or C++ and used by various components of the Android system. They are called from the application framework and applications through a Java interface. These features are exposed to developers through the Android application framework. Some of the key system libraries include the following:

- **Surface Manager:** Android uses a compositing window manager similar to Vista or Compiz, but it is much simpler. Instead of drawing directly to the screen buffer, your drawing commands go into off-screen bitmaps that are then combined with other bitmaps to form the screen content the user sees. This lets the system create all sorts of interesting effects, such as see-through windows and fancy transitions.
- **OpenGL:** OpenGL (Open Graphics Library) is a cross-language, multiplatform API for rendering 2D and 3D computer graphics. OpenGL/ES (OpenGL for embedded systems) is a subset of OpenGL designed for embedded systems.
- **Media Framework:** The Media Framework supports video recording and playing in many formats, including AAC, AVC (H.264), H.263, MP3, and MPEG-4.
- **SQL Database:** Android includes a lightweight SQLite database engine for storing persistent data. SQLite is discussed in a subsequent section.
- **Browser Engine:** For fast display of HTML content, Android uses the WebKit library, which is essentially the same library used in Safari and iPhone. It was also the library used for the Google Chrome browser until Google switched to Blink.
- **Bionic LibC:** This is a stripped-down version of the standard C system library, tuned for embedded Linux-based devices. The interface is the standard Java Native Interface (JNI).

Linux Kernel The OS kernel for Android is similar to, but not identical with, the standard Linux kernel distribution. One noteworthy change is the Android kernel lacks drivers not applicable in mobile environments, making the kernel smaller. In addition, Android enhances the Linux kernel with features that are tailored to the mobile environment, and generally not as useful or applicable on a desktop or laptop platform.

Android relies on its Linux kernel for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack, and enables Android to use the wide range of hardware drivers that Linux supports.

Android Runtime

Most operating systems used on mobile devices, such as iOS and Windows, use software that is compiled directly to the specific hardware platform. In contrast, most Android software is mapped into a bytecode format, which is then transformed into native instructions on the device itself. Earlier releases of Android used a scheme known as Dalvik. However, Dalvik has a number of limitations in terms of scaling up to larger memories and multicore architectures, so more recent releases of Android rely on a scheme known as Android runtime (ART). ART is fully compatible with Dalvik's existing bytecode format, dex (Dalvik Executable), so application developers do not need to change their coding to be executable under ART. We will first look at Dalvik, then examine ART.

The Dalvik Virtual Machine The Dalvik VM (DVM) executes files in the .dex format, a format that is optimized for efficient storage and memory-mappable execution. The VM can run classes compiled by a Java language compiler that have been transformed into its native format using the included "dx" tool. The VM runs on top of Linux kernel, which it relies on for underlying functionality (such as threading and low-level memory management). The Dalvik core class library is intended to provide a familiar development base for those used to programming with Java Standard Edition, but it is geared specifically to the needs of a small mobile device.

Each Android application runs in its own process, with its own instance of the Dalvik VM. Dalvik has been written so a device can efficiently run multiple VMs efficiently.

The Dex File Format The DVM runs applications and code written in Java. A standard Java compiler turns source code (written as text files) into bytecode. The bytecode is then compiled into a .dex file that the DVM can read and use. In essence, class files are converted into .dex files (much like a .jar file if one were using the standard Java VM) and then read and executed by the DVM. Duplicate data used in class files are included only once in the .dex file, which saves space and uses less overhead. The executable files can be modified again when an application is installed to make things even more optimized for mobile.

Android Runtime Concepts ART is the current application runtime used by Android, introduced with Android version 4.4 (KitKat). When Android was designed initially, it was designed for single core (with minimal multithreading support in hardware) and low-memory devices, for which Dalvik seemed a suitable runtime. However, in recent times, the devices that run Android have multicore processors and more memory (at a relatively cheaper cost), which made Google to re-think the runtime design to provide developers and users a richer experience by making use of the available high-end hardware.

For both Dalvik and ART, all Android applications written in Java are compiled to dex bytecode. While Dalvik uses dex bytecode format for portability, it has to be converted (compiled) to machine code to be actually run by a processor. The Dalvik runtime did this conversion from dex bytecode to native machine code when the application ran, and this process was called JIT (just-in-time) compilation. Because JIT compiles only a part of the code, it has a smaller memory footprint and uses less physical space on the device. (Only the dex files are stored in the permanent storage as opposed to the actual machine code.) Dalvik identifies the section of code that runs frequently and caches the compiled code for this once, so the subsequent executions of this section of code are faster. The pages of physical memory that store the cached code are not swappable/pageable, so this also adds a bit to the memory pressure if the system is already in such a state. Even with these optimizations, Dalvik has to do JIT-compilation every time the app is run, which consumes a considerable amount of processor resources. Note the processor is not only being used for actually running the app, but also for converting the dex bytecode to native code, thereby draining more power. This processor usage was also the reason for poor user interface experience in some heavy usage applications when they start.

To overcome some of these issues, and to make more effective use of the available high-end hardware, Android introduced ART. ART also executes dex bytecode but instead of compiling the bytecode at runtime, ART compiles the bytecode to native machine code during install time of the app. This is called ahead-of-time (AOT) compilation. ART uses the "`dex2oat`" tool to do this compilation at install time. The output of the tool is a file that is then executed when the application runs.

Figure 2.21 shows the life cycle of an APK, an application package that comes from the developer to the user. The cycle begins with source code being compiled into .dex format and combined with any appropriate support code to form an APK. On the user side, the received APK is unpacked. The resources and native code are generally installed directly into the application directory. The .dex code, however, requires further processing, both in the case of Dalvik and of ART. In Dalvik, a function called `dexopt` is applied to the dex file to produce an optimized version of dex (odex) referred to as quickened dex; the objective is to make the `dex` code execute more quickly on the dex interpreter. In ART, the `dex2oat` function does the same sort of optimization as `dexopt`; it also compiles the dex code to produce native code on the target device. The output of the dex2oat function is an Executable and Linkable Format (ELF) file, which runs directly without an interpreter.

Advantages and Disadvantages The benefits of using ART include the following:

- Reduces startup time of applications as native code is directly executed.
- Improves battery life because processor usage for JIT is avoided.
- Lesser RAM footprint is required for the application to run (as there is no storage required for JIT cache). Moreover, because there is no JIT code cache, which is non-pageable, this provides flexibility of RAM usage when there is a low-memory scenario.
- There are a number of Garbage Collection optimizations and debug enhancements that went into ART.

Some potential disadvantages of ART:

- Because the conversion from bytecode to native code is done at install time, application installation takes more time. For Android developers who load an app a number of times during testing, this time may be noticeable.

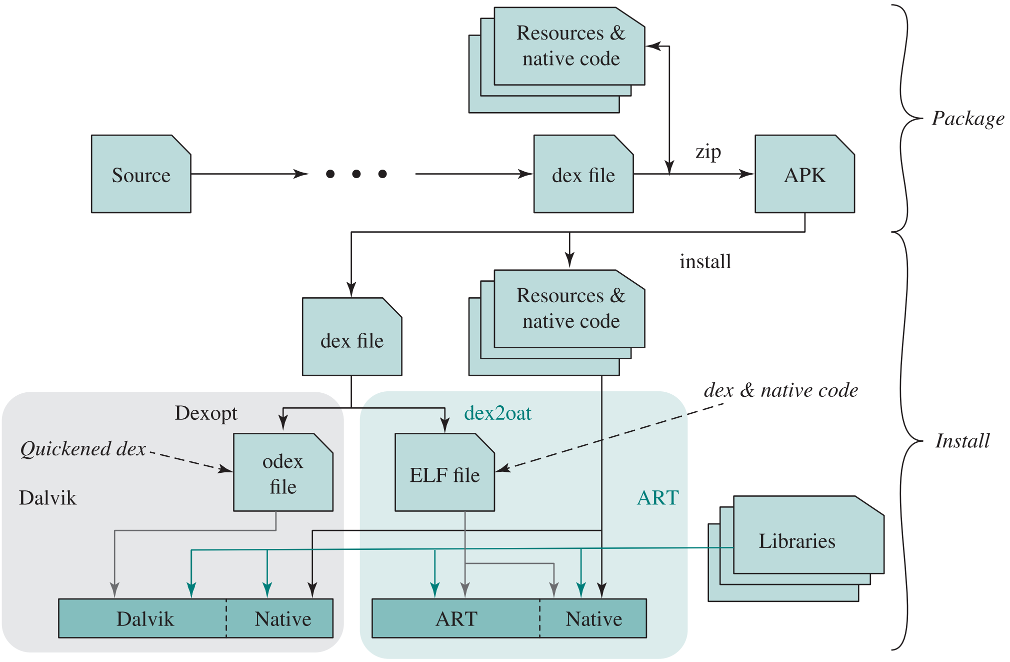


Figure 2.21 The Life Cycle of an APK

- On the first fresh boot or first boot after factory reset, all applications installed on a device are compiled to native code using dex2opt. Therefore, the first boot can take significantly longer (in the order of 3–5 seconds) to reach Home Screen compared to Dalvik.
- The native code thus generated is stored on internal storage that requires a significant amount of additional internal storage space.

Android System Architecture

It is useful to illustrate Android from the perspective of an application developer, as shown in Figure 2.22. This system architecture is a simplified abstraction of the software architecture shown in Figure 2.20. Viewed in this fashion, Android consists of the following layers:

- **Applications and Framework:** Application developers are primarily concerned with this layer and the APIs that allow access to lower-layer services.
- **Binder IPC:** The Binder Inter-Process Communication mechanism allows the application framework to cross process boundaries and call into the Android system services code. This basically allows high-level framework APIs to interact with Android's system services.

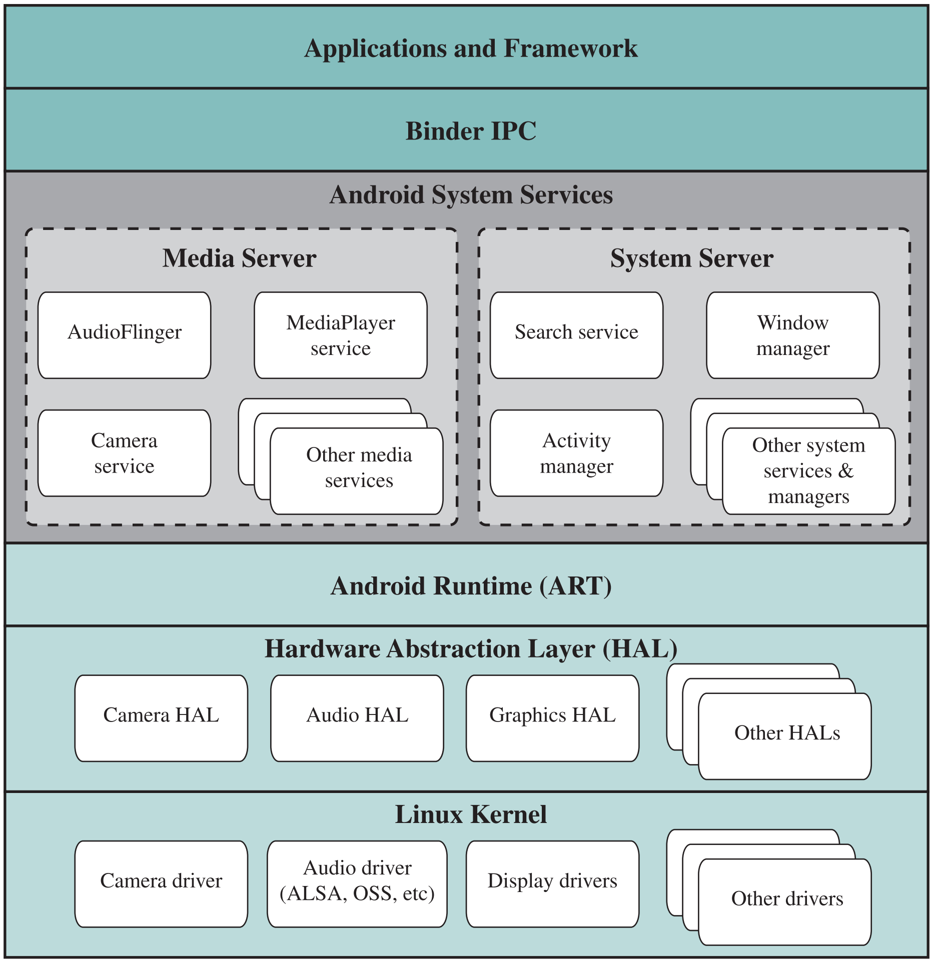


Figure 2.22 **Android System Architecture**

- **Android System Services:** Most of the functionality exposed through the application framework APIs invokes system services that in turn access the underlying hardware and kernel functions. Services can be seen as being organized in two groups: Media services deal with playing and recording media and system services handle system-level functionalities such as power management, location management, and notification management.
- **Hardware Abstraction Layer (HAL):** The HAL provides a standard interface to kernel-layer device drivers, so upper-layer code need not be concerned with the details of the implementation of specific drivers and hardware. The HAL is virtually unchanged from that in a standard Linux distribution. This

layer is used to abstract the device-specific capabilities (which are supported by hardware and exposed by the Kernel) from the user space. The user space could either be Android's Services or Applications. The purpose of HAL is to keep the user space consistent with respect to various devices. Also, vendors can make their own enhancements and put it in their HAL layer without impacting the user space. An example for this is the HwC (Hardware Composer), which is a vendor-specific HAL implementation that understands the rendering capabilities of the underlying hardware. Surface manager seamlessly works with various implementations of the HwC from different vendors.

- **Linux Kernel:** Linux kernel is tailored to meet the demands of a mobile environment.

Activities

An activity is a single visual user interface component, including objects such as menu selections, icons, and checkboxes. Every screen in an application is an extension of the Activity class. Activities use Views to form graphical user interfaces that display information and respond to user actions. We will discuss Activities in Chapter 4.

Power Management

Android adds two features to the Linux kernel to enhance the ability to perform power management: alarms, and wakelocks.

The Alarms capability is implemented in the Linux kernel, and is visible to the app developer through the AlarmManager in the RunTime core libraries. Through the AlarmManager, an app can request a timed wake-up service. The Alarms facility is implemented in the kernel so an alarm can trigger even if the system is in sleep mode. This allows the system to go into sleep mode, saving power, even though there is a process that requires a wake up.

The wakelock facility prevents an Android system from entering into sleep mode. An application can hold one of the following wakelocks:

- **Full_Wake_Lock:** Processor on, full screen brightness, keyboard bright
- **Partial_Wake_Lock:** Processor on, screen off, keyboard off
- **Screen_Dim_Wake_Lock:** Processor on, screen dim, keyboard off
- **Screen_Bright_Wake_Lock:** Processor on, screen bright, keyboard off

These locks are requested through the API whenever an application requires one of the managed peripherals to remain powered on. If no wakelock exists, which locks the device, then it is powered off to conserve battery life.

These kernel objects are made visible to apps in user space by means of /sys/power/wavelock files. The wake_lock and wake_unlock files can be used to define and toggle a lock by writing to the corresponding file.

2.12 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

batch processing
batch system
execution context
distributed operating system
downtime
fault
interrupt
job
job control language (JCL)
kernel
kernel mode
loadable modules
mean time to failure (MTTF)
mean time to repair (MTTR)
memory management
microkernel
monitor
monolithic kernel
multiprogrammed batch system
multiprogramming
multitasking
multithreading
nucleus
object-oriented design
operating system
physical address
privileged instruction
process
process state
real address
reliability
resident monitor
round-robin
scheduling
serial processing
state
symmetric multiprocessing (SMP)
task
thread
time sharing
time-sharing system
time slicing
uniprogramming
uptime
user mode
virtual address
virtual machine
virtual memory

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. What is time slicing?
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?
2.11. What do you understand by a distributed operating system?

Problems

2.1. Suppose we have four jobs in a computer system, in the order JOB1, JOB2, JOB3 and JOB4. JOB1 requires 8 s of CPU time and 8 s of I/O time; JOB2 requires 4 s of CPU time and 14 s of disk time; JOB3 requires 6 s of CPU time; and, JOB4 requires 4 s of CPU time and 16 s of printer time. Define the following quantities for system utilization:

- 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 (with illustrations if needed) in each of the following systems:

a. A uniprogramming system, whereby each job executes to completion before the next job can start its execution.

b. A multiprogramming system that follows a simple round-robin scheduling. Each process gets 2 s of CPU time turn-wise in a circular manner

2.2. In a batch operating system, three jobs are submitted for execution. Each job involves an I/O activity, CPU time and another I/O activity of the same time span as the first. Job JOB1 requires a total of 23 ms, with 3 ms CPU time; JOB2 requires a total time of 29 ms with 5 ms CPU time; JOB3 requires a total time of 14 ms with 4 ms CPU time. Illustrate their execution and find CPU utilization for uniprogramming and multiprogramming systems.

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. A computer system boots and starts a user application when an interrupt occurs. In which modes does the operating system work in this scenario?

2.5. In IBM's mainframe OS, OS/390, one of the major modules in the kernel is the System Resource Manager. 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?

2.6. A multiprocessor with ten processors has 24 attached tape drives. There are a large number of jobs submitted to the system that each require a maximum of six tape drives to complete execution. Assume that each job starts running with only four tape drives for a long period before requiring the other two tape drives for a short period toward the end of its operation. Also assume an endless supply of such jobs.

a. Assume the scheduler in the OS will not start a job unless there are six tape drives available. When a job is started, six drives are assigned immediately and are not released until the job finishes. What is the maximum number of jobs that can be in progress at once? What are the maximum and minimum number of tape drives that may be left idle as a result of this policy?

b. Suggest an alternative policy to improve tape drive utilization and at the same time avoid system deadlock. What is the maximum number of jobs that can be in progress at once? What are the bounds on the number of idling tape drives?