

# Operating-System Structures



An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

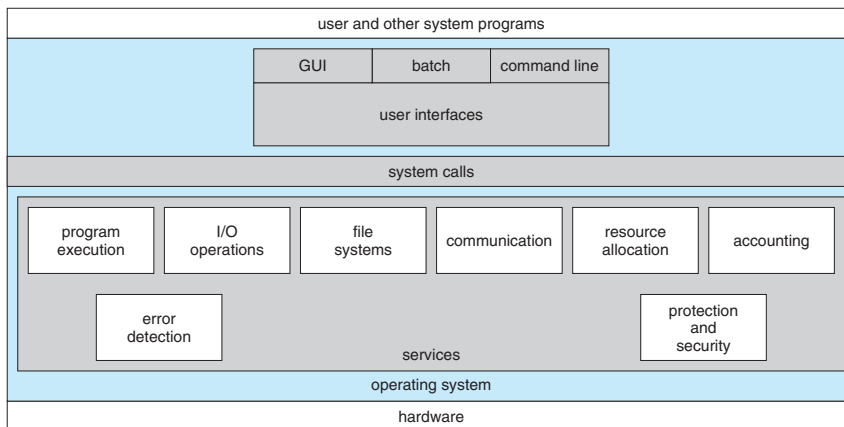
We can view an operating system from several vantage points. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections. In this chapter, we explore all three aspects of operating systems, showing the viewpoints of users, programmers, and operating system designers. We consider what services an operating system provides, how they are provided, how they are debugged, and what the various methodologies are for designing such systems. Finally, we describe how operating systems are created and how a computer starts its operating system.

## CHAPTER OBJECTIVES

- To describe the services an operating system provides to users, processes, and other systems.
- To discuss the various ways of structuring an operating system.
- To explain how operating systems are installed and customized and how they boot.

## 2.1 Operating-System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another, but we can identify common classes. These operating system services are provided for the convenience of the programmer, to make the programming



**Figure 2.1** A view of operating system services.

task easier. Figure 2.1 shows one view of the various operating-system services and how they interrelate.

One set of operating system services provides functions that are helpful to the user.

- **User interface.** Almost all operating systems have a **user interface (UI)**. This interface can take several forms. One is a **command-line interface (CLI)**, which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Another is a **batch interface**, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a **graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all three of these variations.
- **Program execution.** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
- **I/O operations.** A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a display screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **File-system manipulation.** The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow personal choice and sometimes to provide specific features or performance characteristics.

- **Communications.** There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.
- **Error detection.** The operating system needs to be detecting and correcting errors constantly. Errors may occur in the **CPU and memory hardware** (such as a memory error or a power failure), in **I/O devices** (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and **in the user program** (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

Another set of operating system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

- **Resource allocation.** When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.
- **Accounting.** We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.
- **Protection and security.** The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate

himself or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts and to recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

## 2.2 User and Operating-System Interface

We mentioned earlier that there are several ways for users to interface with the operating system. Here, we discuss two fundamental approaches. One provides a command-line interface, or **command interpreter**, that allows users to directly enter commands to be performed by the operating system. The other allows users to interface with the operating system via a graphical user interface, or GUI.

### 2.2.1 Command Interpreters

Some operating systems include the command interpreter in the kernel. Others, such as Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as **shells**. For example, on UNIX and Linux systems, a user may choose among several different shells, including the **Bourne shell, C shell, Bourne-Again shell, Korn shell, and others**. Third-party shells and free user-written shells are also available. Most shells provide similar functionality, and a user's choice of which shell to use is generally based on personal preference. Figure 2.2 shows the Bourne shell command interpreter being used on Solaris 10.

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way. These commands can be implemented in two general ways.

In one approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.

An alternative approach—used by UNIX, among other operating systems—implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file

```
rm file.txt
```

would search for a file called `rm`, load the file into memory, and execute it with the parameter `file.txt`. The function associated with the `rm` command would

```

File Edit View Terminal Tabs Help
fd0      0.0    0.0    0.0    0.0 0.0 0.0 0.0 0 0
sd0      0.0    0.2    0.0    0.2 0.0 0.0 0.4 0 0
sd1      0.0    0.0    0.0    0.0 0.0 0.0 0.0 0 0
extended device statistics
device   r/s    w/s    kr/s    kw/s wait actv  svc_t  %w  %b
fd0      0.0    0.0    0.0    0.0 0.0 0.0 0.0 0 0
sd0      0.6    0.0   38.4    0.0 0.0 0.0 8.2 0 0
sd1      0.0    0.0    0.0    0.0 0.0 0.0 0.0 0 0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# uptime
12:53am up 9 min(s), 3 users, load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-(/var/tmp/system-contents/scripts)# w
4:07pm up 17 day(s), 15:24, 3 users, load average: 0.09, 0.11, 8.66
User      tty          login@ idle   JCPU   PCPU   what
root      console      15Jun0718days 1      /usr/bin/ssh-agent -- /usr/bi
n/d
root      pts/3        15Jun07      18     4     w
root      pts/4        15Jun0718days      w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-(/var/tmp/system-contents/scripts)#

```

**Figure 2.2** The Bourne shell command interpreter in Solrais 10.

be defined completely by the code in the file `rm`. In this way, programmers can add new commands to the system easily by creating new files with the proper names. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

### 2.2.2 Graphical User Interfaces

A second strategy for interfacing with the operating system is through a user-friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window-and-menu system characterized by a **desktop** metaphor. The user moves the mouse to position its pointer on images, or **icons**, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**—or pull down a menu that contains commands.

Graphical user interfaces first appeared due in part to research taking place in the early 1970s at Xerox PARC research facility. The first GUI appeared on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s. The user interface for the Macintosh operating system (Mac OS) has undergone various changes over the years, the most significant being the adoption of the *Aqua* interface that appeared with Mac OS X. Microsoft's first version of Windows—Version 1.0—was based on the addition of a GUI interface to the MS-DOS operating system. Later versions of Windows have made cosmetic

changes in the appearance of the GUI along with several enhancements in its functionality.

Because a mouse is impractical for most mobile systems, smartphones and handheld tablet computers typically use a touchscreen interface. Here, users interact by making **gestures** on the touchscreen—for example, pressing and swiping fingers across the screen. Figure 2.3 illustrates the touchscreen of the Apple iPad. Whereas earlier smartphones included a physical keyboard, most smartphones now simulate a keyboard on the touchscreen.

Traditionally, UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available, however. These include the Common Desktop Environment (CDE) and X-Windows systems, which are common on commercial versions of UNIX, such as Solaris and IBM's AIX system. In addition, there has been significant development in GUI designs from various open-source projects, such as *K Desktop Environment* (or *KDE*) and the *GNOME* desktop by the GNU project. Both the KDE and GNOME desktops run on Linux and various UNIX systems and are available under open-source licenses, which means their source code is readily available for reading and for modification under specific license terms.



Figure 2.3 The iPad touchscreen.



### 2.2.3 Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the **command-line interface**. For them, it is more efficient, giving them faster access to the activities they need to perform. Indeed, on some systems, only a subset of system functions is available via the GUI, leaving the less common tasks to those who are command-line knowledgeable. Further, command-line interfaces usually make repetitive tasks easier, in part because they have their own programmability. For example, if a frequent task requires a set of command-line steps, those steps can be recorded into a file, and that file can be run just like a program. The program is not compiled into executable code but rather is interpreted by the command-line interface. These **shell scripts** are very common on systems that are command-line oriented, such as UNIX and Linux.

In contrast, most Windows users are happy to use the Windows GUI environment and almost never use the MS-DOS shell interface. The various changes undergone by the Macintosh operating systems provide a nice study in contrast. Historically, Mac OS has not provided a command-line interface, always requiring its users to interface with the operating system using its GUI. However, with the release of Mac OS X (which is in part implemented using a UNIX kernel), the operating system now provides both a Aqua interface and a command-line interface. Figure 2.4 is a screenshot of the Mac OS X GUI.



Figure 2.4 The Mac OS X GUI.

The user interface can vary from system to system and even from user to user within a system. It typically is substantially removed from the actual system structure. The design of a useful and friendly user interface is therefore not a direct function of the operating system. In this book, we concentrate on the fundamental problems of providing adequate service to user programs. From the point of view of the operating system, we do not distinguish between user programs and system programs.

## 2.3 System Calls

**System calls** provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is for the program to ask the user for the names. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.

Once the two file names have been obtained, the program must open the input file and create the output file. Each of these operations requires another system call. Possible error conditions for each operation can require additional system calls. When the program tries to open the input file, for example, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should print a message on the console (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (for example, no more disk space).

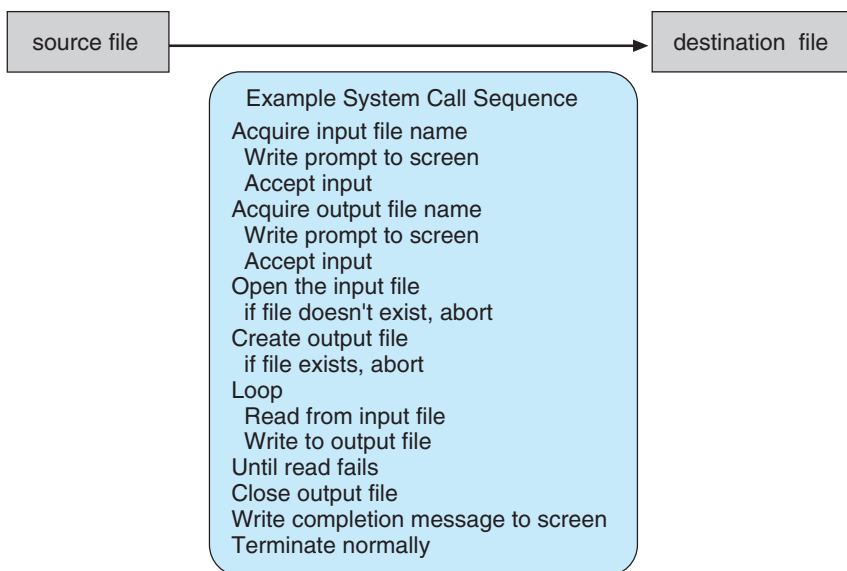


Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call). This system-call sequence is shown in Figure 2.5.

As you can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and Mac OS X), and the Java API for programs that run on the Java virtual machine. A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called **libc**. Note that—unless specified—the system-call names used throughout this text are generic examples. Each operating system has its own name for each system call.

Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function `CreateProcess()` (which unsurprisingly is used to create a new process) actually invokes the `NTCreateProcess()` system call in the Windows kernel.

Why would an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit concerns program portability. An application program-



**Figure 2.5** Example of how system calls are used.

*EXAMPLE OF STANDARD API*

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the man page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

#include <unistd.h>		
<code>ssize_t</code>	<code>read</code>	<code>(int fd, void *buf, size_t count)</code>
return	function	parameters
value	name	

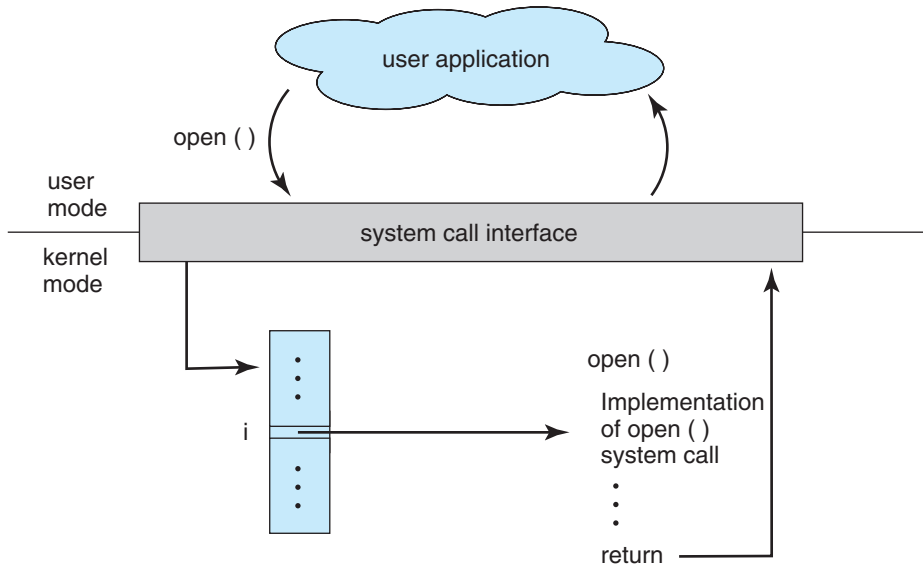
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

mer designing a program using an API can expect her program to compile and run on any system that supports the same API (although, in reality, architectural differences often make this more difficult than it may appear). Furthermore, actual system calls can often be more detailed and difficult to work with than the API available to an application programmer. Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Windows APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

For most programming languages, the run-time support system (a set of functions built into libraries included with a compiler) provides a **system-call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface



**Figure 2.6** The handling of a user application invoking the `open()` system call.

then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.

The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library. The relationship between an API, the system-call interface, and the operating system is shown in Figure 2.6, which illustrates how the operating system handles a user application invoking the `open()` system call.

System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call. For example, to get input, we may need to specify the file or device to use as the source, as well as the address and length of the memory buffer into which the input should be read. Of course, the device or file and length may be implicit in the call.

Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in registers. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register (Figure 2.7). This is the approach taken by Linux and Solaris. Parameters also can be placed, or pushed, onto the stack by the program and popped off the stack by the operating system. Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

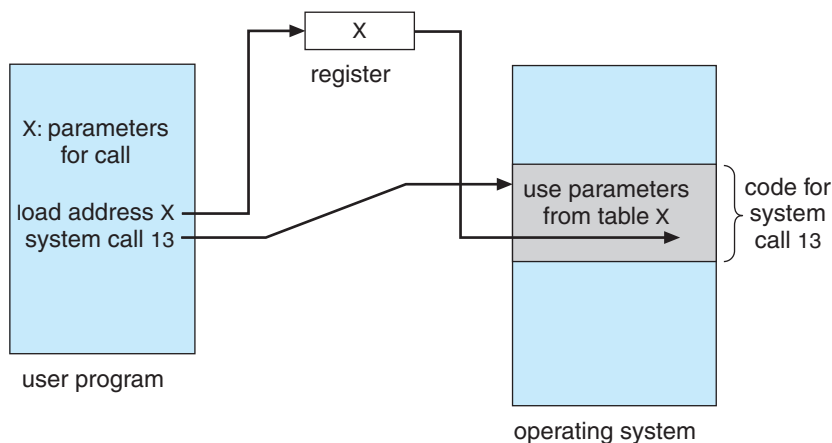


Figure 2.7 Passing of parameters as a table.

## 2.4 Types of System Calls

System calls can be grouped roughly into six major categories: **process control**, **file manipulation**, **device manipulation**, **information maintenance**, **communications**, and **protection**. In Sections 2.4.1 through 2.4.6, we briefly discuss the types of system calls that may be provided by an operating system. Most of these system calls support, or are supported by, concepts and functions that are discussed in later chapters. Figure 2.8 summarizes the types of system calls normally provided by an operating system. As mentioned, in this text, we normally refer to the system calls by generic names. Throughout the text, however, we provide examples of the actual counterparts to the system calls for Windows, UNIX, and Linux systems.

### 2.4.1 Process Control

A running program needs to be able to halt its execution either normally (`end()`) or abnormally (`abort()`). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting errors, or **bugs**—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error. In a GUI system, a pop-up window might alert the user to the error and ask for guidance. In a batch system, the command interpreter usually terminates the entire job and continues with the next job. Some systems may allow for special recovery actions in case an error occurs. If the program discovers an error in its input and wants to terminate abnormally, it may also want to define an error level. More severe errors can be indicated by a higher-level error parameter. It is then

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

**Figure 2.8** Types of system calls.

possible to combine normal and abnormal termination by defining a normal termination as an error at level 0. The command interpreter or a following program can use this error level to determine the next action automatically.

A process or job executing one program may want to `load()` and `execute()` another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command, the click of a

*EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS*

	Windows	Unix
<b>Process Control</b>	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
<b>File Manipulation</b>	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
<b>Device Manipulation</b>	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
<b>Information Maintenance</b>	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
<b>Communication</b>	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
<b>Protection</b>	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

mouse, or a batch command. An interesting question is where to return control when the loaded program terminates. This question is related to whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program.

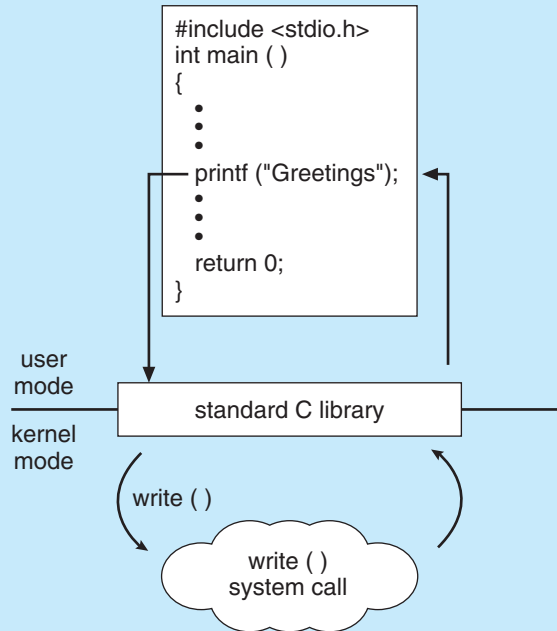
If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new job or process to be multiprogrammed. Often, there is a system call specifically for this purpose (`create_process()` or `submit_job()`).

If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (`get_process_attributes()` and `set_process_attributes()`). We may also want to terminate a job or process that we created (`terminate_process()`) if we find that it is incorrect or is no longer needed.



*EXAMPLE OF STANDARD C LIBRARY*

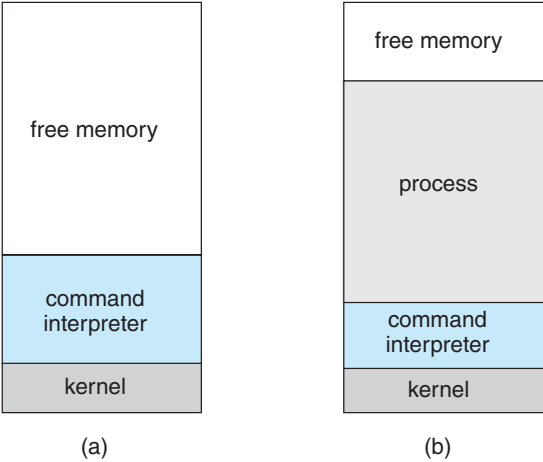
The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program. This is shown below:



Having created new jobs or processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (`wait_time()`). More probably, we will want to wait for a specific event to occur (`wait_event()`). The jobs or processes should then signal when that event has occurred (`signal_event()`).

Quite often, two or more processes may share data. To ensure the integrity of the data being shared, operating systems often provide system calls allowing a process to **lock** shared data. Then, no other process can access the data until the lock is released. Typically, such system calls include `acquire_lock()` and `release_lock()`. System calls of these types, dealing with the coordination of concurrent processes, are discussed in great detail in Chapter 5.

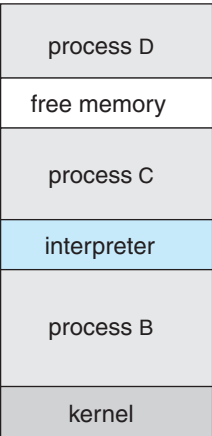
There are so many facets of and variations in process and job control that we next use two examples—one involving a single-tasking system and the other a multitasking system—to clarify these concepts. The MS-DOS operating system is an example of a single-tasking system. It has a command interpreter that is invoked when the computer is started (Figure 2.9(a)). Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a new process. It loads the program into memory, writing over most of itself to



**Figure 2.9** MS-DOS execution. (a) At system startup. (b) Running a program.

give the program as much memory as possible (Figure 2.9(b)). Next, it sets the instruction pointer to the first instruction of the program. The program then runs, and either an error causes a trap, or the program executes a system call to terminate. In either case, the error code is saved in the system memory for later use. Following this action, the small portion of the command interpreter that was not overwritten resumes execution. Its first task is to reload the rest of the command interpreter from disk. Then the command interpreter makes the previous error code available to the user or to the next program.

FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user’s choice is run. This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.10). To start a new process, the shell



**Figure 2.10** FreeBSD running multiple programs.

executes a `fork()` system call. Then, the selected program is loaded into memory via an `exec()` system call, and the program is executed. Depending on the way the command was issued, the shell then either waits for the process to finish or runs the process “in the background.” In the latter case, the shell immediately requests another command. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files or through a GUI interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program’s priority, and so on. When the process is done, it executes an `exit()` system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs. Processes are discussed in Chapter 3 with a program example using the `fork()` and `exec()` system calls.

### 2.4.2 File Management

The file system is discussed in more detail in Chapters 11 and 12. We can, however, identify several common system calls dealing with files.

We first need to be able to `create()` and `delete()` files. Either system call requires the name of the file and perhaps some of the file’s attributes. Once the file is created, we need to `open()` it and to use it. We may also `read()`, `write()`, or `reposition()` (rewind or skip to the end of the file, for example). Finally, we need to `close()` the file, indicating that we are no longer using it.

We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, `get_file_attributes()` and `set_file_attributes()`, are required for this function. Some operating systems provide many more calls, such as calls for file `move()` and `copy()`. Others might provide an API that performs those operations using code and other system calls, and others might provide system programs to perform those tasks. If the system programs are callable by other programs, then each can be considered an API by other system programs.

### 2.4.3 Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.

The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files). A system with multiple users may require us to first `request()` a device, to ensure exclusive use of it. After we are finished with the device, we `release()` it. These functions are similar to the `open()` and `close()` system calls for files. Other operating systems allow unmanaged access to devices.

The hazard then is the potential for device contention and perhaps deadlock, which are described in Chapter 7.

Once the device has been requested (and allocated to us), we can `read()`, `write()`, and (possibly) `reposition()` the device, just as we can with files. In fact, the similarity between I/O devices and files is so great that many operating systems, including UNIX, merge the two into a combined file–device structure. In this case, a set of system calls is used on both files and devices. Sometimes, I/O devices are identified by special file names, directory placement, or file attributes.

The user interface can also make files and devices appear to be similar, even though the underlying system calls are dissimilar. This is another example of the many design decisions that go into building an operating system and user interface.

#### 2.4.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current `time()` and `date()`. Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

Another set of system calls is helpful in debugging a program. Many systems provide system calls to `dump()` memory. This provision is useful for debugging. A program trace lists each system call as it is executed. Even microprocessors provide a CPU mode known as **single step**, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger.

Many operating systems provide a time profile of a program to indicate the amount of time that the program executes at a particular location or set of locations. A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program counter is recorded. With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.

In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (`get_process_attributes()` and `set_process_attributes()`). In Section 3.1.3, we discuss what information is normally kept.

#### 2.4.5 Communication

There are two common models of interprocess communication: the message-passing model and the shared-memory model. In the **message-passing model**, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a **host name** by which it is commonly known. A host also has a

network identifier, such as an IP address. Similarly, each process has a **process name**, and this name is translated into an identifier by which the operating system can refer to the process. The `get_hostid()` and `get_processid()` system calls do this translation. The identifiers are then passed to the general-purpose `open()` and `close()` calls provided by the file system or to specific `open_connection()` and `close_connection()` system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an `accept_connection()` call. Most processes that will be receiving connections are special-purpose **daemons**, which are system programs provided for that purpose. They execute a `wait_for_connection()` call and are awakened when a connection is made. The source of the communication, known as the **client**, and the receiving daemon, known as a **server**, then exchange messages by using `read_message()` and `write_message()` system calls. The `close_connection()` call terminates the communication.

In the **shared-memory model**, processes use `shared_memory_create()` and `shared_memory_attach()` system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data is determined by the processes and is not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously. Such mechanisms are discussed in Chapter 5. In Chapter 4, we look at a variation of the process scheme—threads—in which memory is shared by default.

Both of the models just discussed are common in operating systems, and most systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer. Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

### 2.4.6 Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multiprogrammed computer systems with several users. However, with the advent of networking and the Internet, all computer systems, from servers to mobile handheld devices, must be concerned with protection.

Typically, system calls providing protection include `set_permission()` and `get_permission()`, which manipulate the permission settings of resources such as files and disks. The `allow_user()` and `deny_user()` system calls specify whether particular users can—or cannot—be allowed access to certain resources.

We cover protection in Chapter 14 and the much larger issue of security in Chapter 15.

## 2.5 System Programs

Another aspect of a modern system is its collection of system programs. Recall Figure 1.1, which depicted the logical computer hierarchy. At the lowest level is hardware. Next is the operating system, then the system programs, and finally the application programs. **System programs, also known as system utilities,** provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. Others are considerably more complex. They can be divided into these categories:

- **File management.** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a **registry**, which is used to store and retrieve configuration information.
- **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
- **Programming-language support.** Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or available as a separate download.
- **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.
- **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse Web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.
- **Background services.** All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as **services, subsystems**, or daemons. One example is the network daemon discussed in Section 2.4.5. In that example, a system needed a service to listen for network connections in order to connect those requests to the correct processes. Other examples include process schedulers that start processes according to a specified schedule, system error monitoring services, and print servers. Typical systems have dozens



of daemons. In addition, operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities.

Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such **application programs** include Web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls. Consider a user's PC. When a user's computer is running the Mac OS X operating system, the user might see the GUI, featuring a mouse-and-windows interface. Alternatively, or even in one of the windows, the user might have a command-line UNIX shell. Both use the same set of system calls, but the system calls look different and act in different ways. Further confusing the user view, consider the user dual-booting from Mac OS X into Windows. Now the same user on the same hardware has two entirely different interfaces and two sets of applications using the same physical resources. On the same hardware, then, a user can be exposed to multiple user interfaces sequentially or concurrently.

## 2.6 Operating-System Design and Implementation

In this section, we discuss problems we face in designing and implementing an operating system. There are, of course, no complete solutions to such problems, but there are approaches that have proved successful.

### 2.6.1 Design Goals

The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: batch, time sharing, single user, multiuser, distributed, real time, or general purpose.

Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups: **user goals** and **system goals**.

Users want certain obvious properties in a system. The system should be **convenient to use, easy to learn and to use, reliable, safe, and fast**. Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.

A similar set of requirements can be defined by those people who must **design, create, maintain, and operate the system**. The system should be **easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient**. Again, these requirements are vague and may be interpreted in various ways.

There is, in short, no unique solution to the problem of defining the requirements for an operating system. The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments. For example, the requirements for VxWorks, a real-

## 2.7 Operating-System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions. We have already discussed briefly in Chapter 1 the common components of operating systems. In this section, we discuss how these components are interconnected and melded into a kernel.

### 2.7.1 Simple Structure

Many operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space, so it was not carefully divided into modules. Figure 2.11 shows its structure.

In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

Another example of limited structuring is the original UNIX operating system. Like MS-DOS, UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs. The kernel

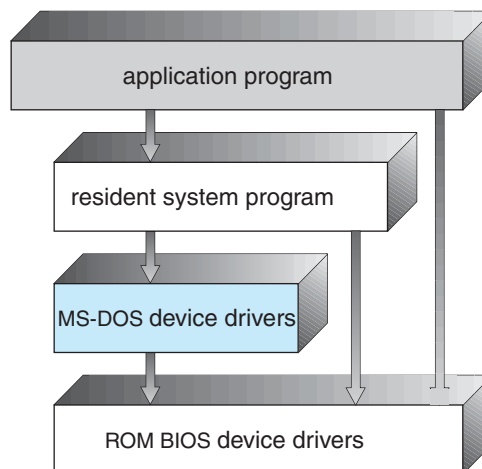


Figure 2.11 MS-DOS layer structure.

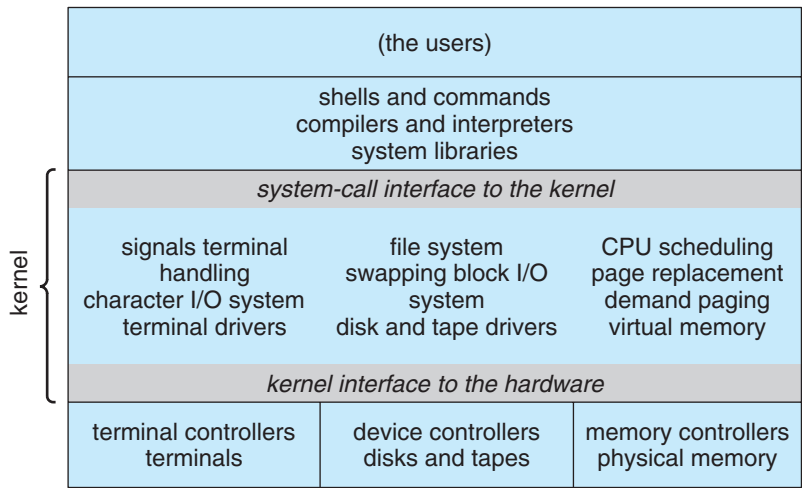


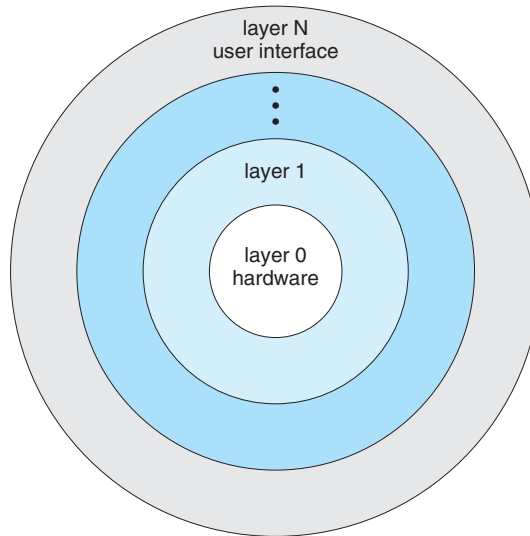
Figure 2.12 Traditional UNIX system structure.

is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.12. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one level. This monolithic structure was difficult to implement and maintain. It had a distinct performance advantage, however: there is very little overhead in the system call interface or in communication within the kernel. We still see evidence of this simple, monolithic structure in the UNIX, Linux, and Windows operating systems.

2.7.2 Layered Approach

With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS and UNIX systems. The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under a top-down approach, the overall functionality and features are determined and are separated into components. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface. This layering structure is depicted in Figure 2.13.



**Figure 2.13** A layered operating system.

An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer  $M$ —consists of data structures and a set of routines that can be invoked by higher-level layers. Layer  $M$ , in turn, can invoke operations on lower-level layers.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified.

Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. For example, the device driver for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than the memory-management routines, because memory management requires the ability to use the backing store.

Other requirements may not be so obvious. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. However, on a large

system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU scheduler.

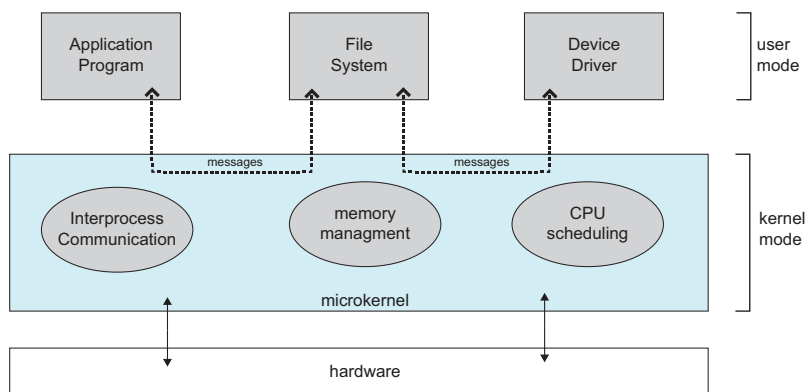
A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call. The net result is a system call that takes longer than does one on a nonlayered system.

These limitations have caused a small backlash against layering in recent years. Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the problems of layer definition and interaction.

### 2.7.3 Microkernels

We have already seen that as UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, microkernels provide minimal process and memory management, in addition to a communication facility. Figure 2.14 illustrates the architecture of a typical microkernel.

The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through **message passing**, which was described in Section 2.4.5. For example, if the client program wishes to access a file, it



**Figure 2.14** Architecture of a typical microkernel.

must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

Some contemporary operating systems have used the microkernel approach. Tru64 UNIX (formerly Digital UNIX) provides a UNIX interface to the user, but it is implemented with a Mach kernel. The Mach kernel maps UNIX system calls into messages to the appropriate user-level services. The Mac OS X kernel (also known as Darwin) is also partly based on the Mach microkernel.

Another example is QNX, a real-time operating system for embedded systems. The QNX Neutrino microkernel provides services for message passing and process scheduling. It also handles low-level network communication and hardware interrupts. All other services in QNX are provided by standard processes that run outside the kernel in user mode.

Unfortunately, the performance of microkernels can suffer due to increased system-function overhead. Consider the history of Windows NT. The first release had a layered microkernel organization. This version's performance was low compared with that of Windows 95. Windows NT 4.0 partially corrected the performance problem by moving layers from user space to kernel space and integrating them more closely. By the time Windows XP was designed, Windows architecture had become more monolithic than microkernel.

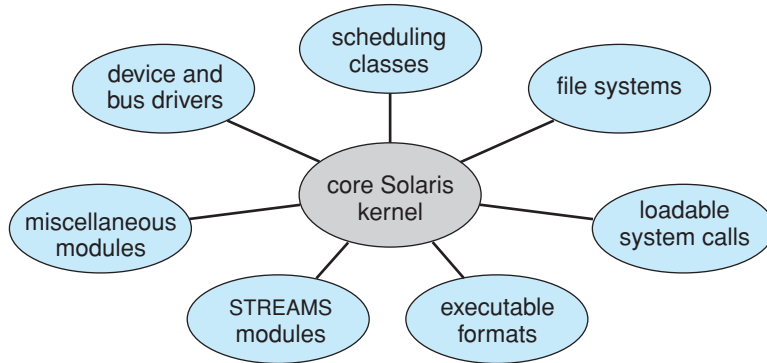
#### 2.7.4 Modules

Perhaps the best current methodology for operating-system design involves using loadable kernel modules. Here, the kernel has a set of core components and links in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X, as well as Windows.

The idea of the design is for the kernel to provide core services while other services are implemented dynamically, as the kernel is running. Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made. Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.

The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module. The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it





**Figure 2.15** Solaris loadable modules.

is more efficient, because modules do not need to invoke message passing in order to communicate.

The Solaris operating system structure, shown in Figure 2.15, is organized around a core kernel with seven types of loadable kernel modules:

1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable formats
5. STREAMS modules
6. Miscellaneous
7. Device and bus drivers

Linux also uses loadable kernel modules, primarily for supporting device drivers and file systems. We cover creating loadable kernel modules in Linux as a programming exercise at the end of this chapter.

### 2.7.5 Hybrid Systems

In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, both Linux and Solaris are monolithic, because having the operating system in a single address space provides very efficient performance. However, they are also modular, so that new functionality can be dynamically added to the kernel. Windows is largely monolithic as well (again primarily for performance reasons), but it retains some behavior typical of microkernel systems, including providing support for separate subsystems (known as operating-system *personalities*) that run as user-mode processes. Windows systems also provide support for dynamically loadable kernel modules. We provide case studies of Linux and Windows 7 in Chapters 18 and 19, respectively. In the remainder of this section, we explore the structure of

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
    gnome-settings-d          142354
    gnome-vfs-daemon          158243
    dsdm                       189804
    wnck-applet                200030
    gnome-panel                277864
    clock-applet               374916
    mapping-daemon             385475
    xscreensaver               514177
    metacity                   539281
    Xorg                       2579646
    gnome-terminal             5007269
    mixer_applet2              7388447
    java                       10769137
```

**Figure 2.21** Output of the D code.

systems do not have conflicting license agreements. For example, DTrace has been added to Mac OS X and FreeBSD and will likely spread further due to its unique capabilities. Other operating systems, especially the Linux derivatives, are adding kernel-tracing functionality as well. Still other operating systems are beginning to include performance and tracing tools fostered by research at various institutions, including the Paradyn project.

## 2.9 Operating-System Generation

It is possible to design, code, and implement an operating system specifically for one machine at one site. More commonly, however, operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations. **The system must then be configured or generated for each specific computer site, a process sometimes known as system generation SYSGEN.**

The operating system is normally distributed on disk, on CD-ROM or DVD-ROM, or as an **“ISO” image**, which is a file in the format of a CD-ROM or DVD-ROM. To generate a system, we use a special program. **This SYSGEN** program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there. The following kinds of information must be determined.

- What CPU is to be used? What options (extended instruction sets, floating-point arithmetic, and so on) are installed? For multiple CPU systems, each CPU may be described.
- How will the boot disk be formatted? How many sections, or “partitions,” will it be separated into, and what will go into each partition?

- How much memory is available? Some systems will determine this value themselves by referencing memory location after memory location until an “illegal address” fault is generated. This procedure defines the final legal address and hence the amount of available memory.
- What devices are available? The system will need to know how to address each device (the device number), the device interrupt number, the device’s type and model, and any special device characteristics.
- What operating-system options are desired, or what parameter values are to be used? These options or values might include how many buffers of which sizes should be used, what type of CPU-scheduling algorithm is desired, what the maximum number of processes to be supported is, and so on.

Once this information is determined, it can be used in several ways. At one extreme, a system administrator can use it to modify a copy of the source code of the operating system. The operating system then is completely compiled. Data declarations, initializations, and constants, along with conditional compilation, produce an output-object version of the operating system that is tailored to the system described.

At a slightly less tailored level, the system description can lead to the creation of tables and the selection of modules from a precompiled library. These modules are linked together to form the generated operating system. Selection allows the library to contain the device drivers for all supported I/O devices, but only those needed are linked into the operating system. Because the system is not recompiled, system generation is faster, but the resulting system may be overly general.

At the other extreme, it is possible to construct a system that is completely table driven. All the code is always part of the system, and selection occurs at execution time, rather than at compile or link time. System generation involves simply creating the appropriate tables to describe the system.

The major differences among these approaches are the size and generality of the generated system and the ease of modifying it as the hardware configuration changes. Consider the cost of modifying the system to support a newly acquired graphics terminal or another disk drive. Balanced against that cost, of course, is the frequency (or infrequency) of such changes.

## 2.10 System Boot

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The procedure of starting a computer by loading the kernel is known as **booting** the system. On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory

location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of **read-only memory (ROM)**, because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus.

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.

Some systems—such as cellular phones, tablets, and game consoles—store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using **erasable programmable read-only memory (EPROM)**, which is read-only except when explicitly given a command to become writable. All forms of ROM are also known as **firmware**, since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM. Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available.

For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader is stored in firmware, and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that **boot block**. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it fits in a single disk block) and knows only the address on disk and length of the remainder of the bootstrap program. **GRUB** is an example of an open-source bootstrap program for Linux systems. All of the disk-bound bootstrap, and the operating system itself, can be easily changed by writing new versions to disk. A disk that has a boot partition (more on that in Section 10.5.1) is called a **boot disk** or **system disk**.

Now that the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be **running**.

## 2.11 Summary

Operating systems provide a number of services. At the lowest level, system calls allow a running program to make requests from the operating system directly. At a higher level, the command interpreter or shell provides a mechanism for a user to issue a request without writing a program. Commands may come from files during batch-mode execution or directly from a terminal or desktop GUI when in an interactive or time-shared mode. System programs are provided to satisfy many common user requests.