Loo [2003]. A discussion of peer-to-peer file-sharing systems can be found in Lee [2003]. Good coverage of cluster computing is provided by Buyya [1999]. Recent advances in cluster computing are described by Ahmed [2000]. A survey of issues relating to operating-system support for distributed systems can be found in Tanenbaum and Van Renesse [1985].

Many general textbooks cover operating systems, including Stallings [2000], Nutt [2004], and Tanenbaum [2001].

Hamacher et al. [2002] describe computer organization, and McDougall and Laudon [2006] discuss multicore processors. Hennessy and Patterson [2007] provide coverage of I/O systems and buses, and of system architecture in general. Blaauw and Brooks [1997] describe details of the architecture of many computer systems, including several from IBM. Stokes [2007] provides an illustrated introduction to microprocessors and computer architecture.

Cache memories, including associative memory, are described and analyzed by Smith [1982]. That paper also includes an extensive bibliography on the subject.

Discussions concerning magnetic-disk technology are presented by Freedman [1983] and by Harker et al. [1981]. Optical disks are covered by Kenville [1982], Fujitani [1984], O'Leary and Kitts [1985], Gait [1988], and Olsen and Kenley [1989]. Discussions of floppy disks are offered by Pechura and Schoeffler [1983] and by Sarisky [1983]. General discussions concerning mass-storage technology are offered by Chi [1982] and by Hoagland [1985].

Kurose and Ross [2005] and Tanenbaum [2003] provides general overviews of computer networks. Fortier [1989] presents a detailed discussion of networking hardware and software. Kozierok [2005] discuss TCP in detail. Mullender [1993] provides an overview of distributed systems. Wolf [2003] discusses recent developments in developing embedded systems. Issues related to handheld devices can be found in Myers and Beigl [2003] and Di Pietro and Mancini [2003].

A full discussion of the history of open sourcing and its benefits and challenges is found in Raymond [1999]. The history of hacking is discussed in Levy [1994]. The Free Software Foundation has published its philosophy on its Web site: `http://www.gnu.org/philosophy/free-software-for-freedom.html`. Detailed instructions on how to build the Ubuntu Linux kernel are on `http://www.howtoforge.com/kernel_compilation_ubuntu`. The open-source components of Mac OS X are available from `http://developer.apple.com/open-source/index.html`.

Wikipedia (`http://en.wikipedia.org/wiki/Richard_Stallman`) has an informative entry about Richard Stallman.

The source code of Multics is available at `http://web.mit.edu/multics-history/source/Multics_Internet_Server/Multics_sources.html`.

CHAPTER 2

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

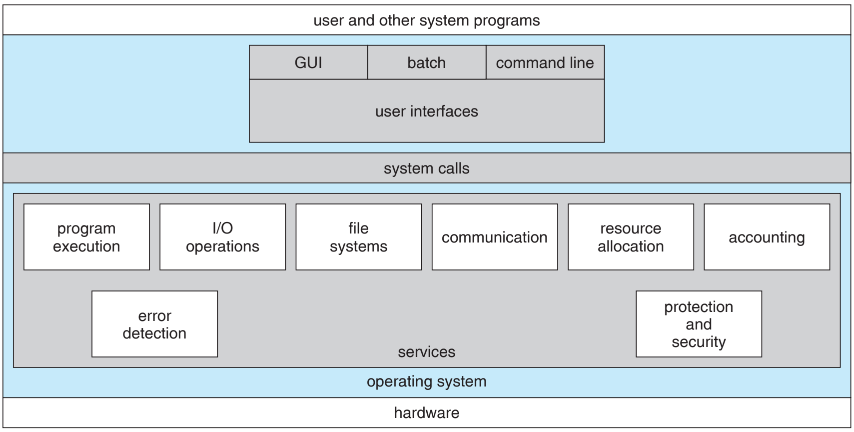

Figure 2.1 A view of operating system services.

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 program to allow entering and editing of commands). 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 programs 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* or through *message passing*, in which packets of information are moved between processes by the operating system.

- **Error detection**. The operating system needs to be constantly aware of possible errors. 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 tape, 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. Of course, there is variation in how operating systems react to and correct errors. Debugging facilities can greatly enhance the user's and programmer's abilities to use the system efficiently.

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. Many different types of resources are managed by the operating system. 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, modems, 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 modems and 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 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 Interpreter

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 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

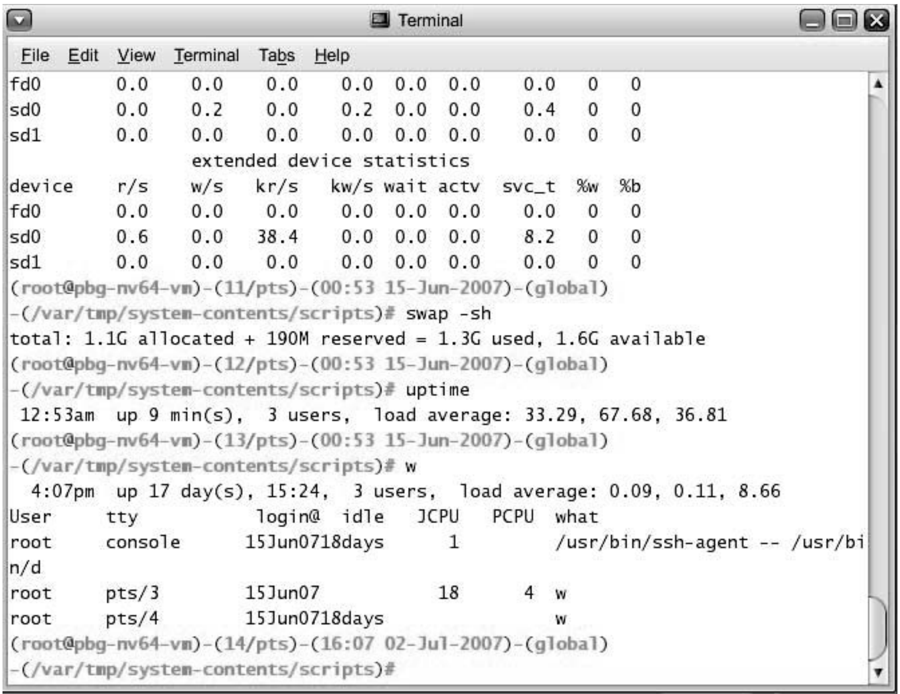

Figure 2.2 The Bourne shell command interpreter in Solaris 10.

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, including Windows Explorer.

Traditionally, UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available, however, including 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.

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. As a very general rule, many UNIX users prefer command-line interfaces, as they often provide powerful shell interfaces. In contrast, most Windows users are pleased 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 new Aqua interface and a command-line interface. Figure 2.3 is a screenshot of 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

Figure 2.3 The Mac OS X GUI.

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 need 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 of the two files. 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 are obtained, the program must open the input file and create the output file. Each of these operations requires another system call. There are also possible error conditions for each operation. When the program tries to open the input file, 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 (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.

Now that 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 (no more disk space, printer out of paper, and so on).

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). As we can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. This system-call sequence is shown in Figure 2.4.

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 Win32 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 designing programs that run on the Java virtual machine. 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 Win32 function `CreateProcess()` (which unsurprisingly is used to create a new process) actually calls 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 of programming according to an API concerns program portability: An application programmer 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. Regardless, there often exists a strong correlation between a function in the API and its associated system call within the kernel.

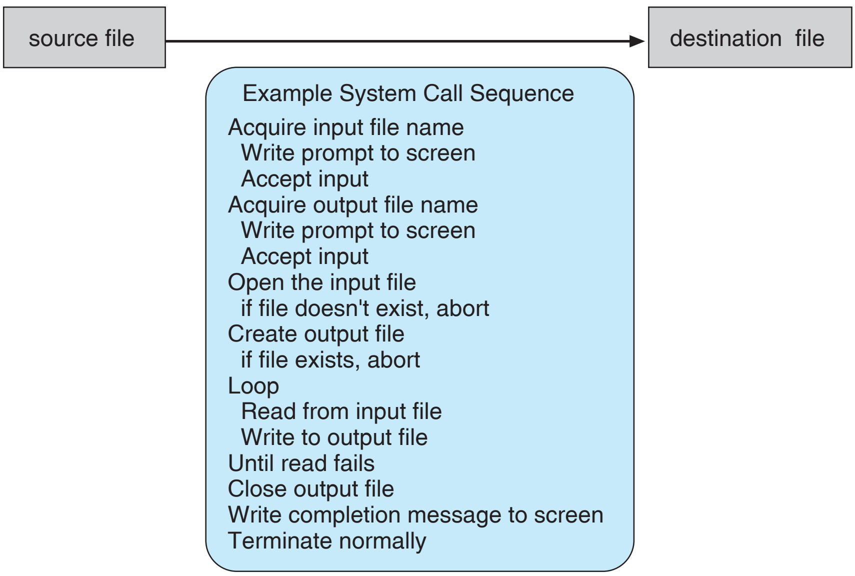

Figure 2.4 Example of how system calls are used.

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `ReadFile()` function in the Win32 API—a function for reading from a file. The API for this function appears in Figure 2.5.

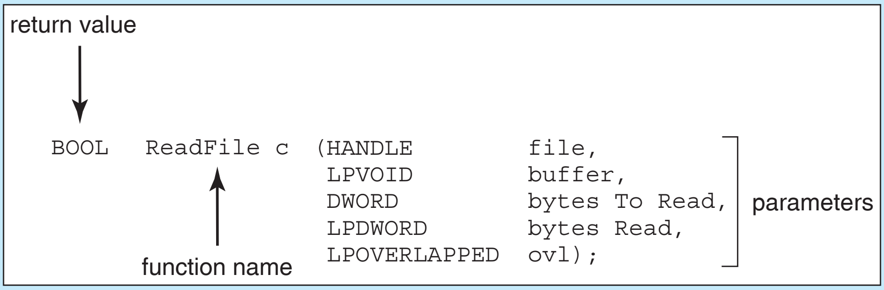

Figure 2.5 The API for the `ReadFile()` function.

A description of the parameters passed to `ReadFile()` is as follows:

- HANDLE file—the file to be read
- LPVOID buffer—a buffer where the data will be read into and written from
- DWORD bytesToRead—the number of bytes to be read into the buffer
- LPDWORD bytesRead—the number of bytes read during the last read
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used

In fact, many of the POSIX and Win32 APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

The run-time support system (a set of functions built into libraries included with a compiler) for most programming languages 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 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, it 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

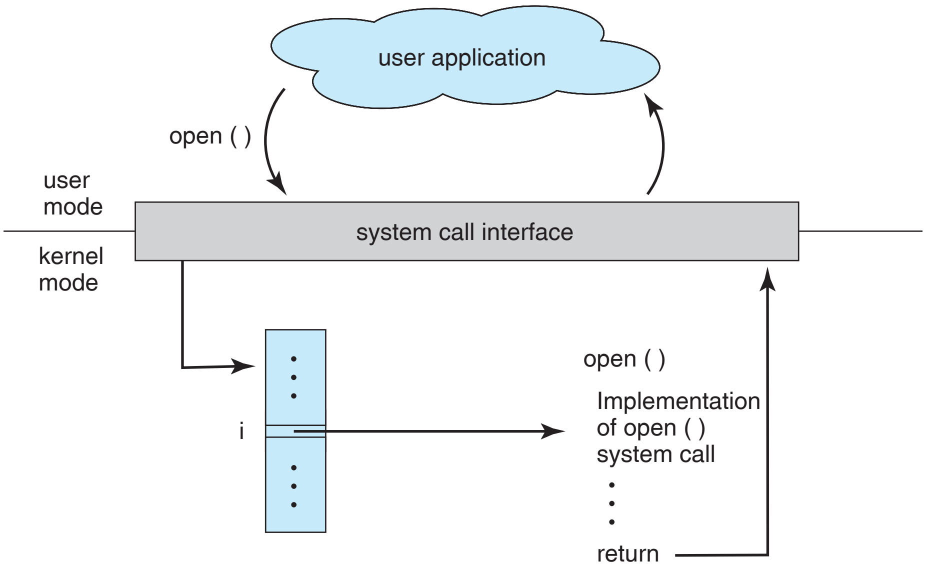

Figure 2.6 The handling of a user application invoking the `open()` system call.

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.

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 discuss briefly 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

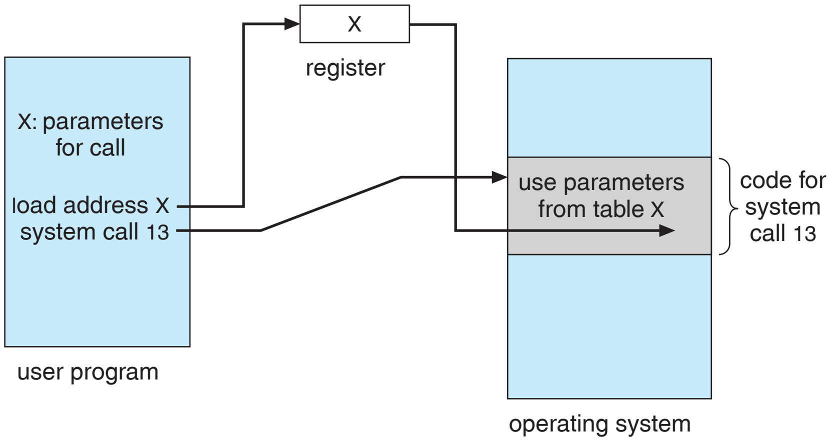

Figure 2.7 Passing of parameters as a table.

that are discussed in later chapters. Figure 2.8 summarizes the types of system calls normally provided by an operating system.

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 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 allow control cards to indicate special recovery actions in case an error occurs. A **control card** is a batch-system concept. It is a command to manage the execution of a process. 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 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 mouse, or a batch command. An interesting question is where to return control when the loaded program terminates. This question is related to the problem of

- 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.

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

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix

Process Control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File Manipulation	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device Manipulation	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shmget()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

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.

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, thus preventing another process from accessing the data while it is locked. Typically such system calls include `acquire_lock` and `release_lock`. System calls of these

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(s) 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 in Figure 2.9.

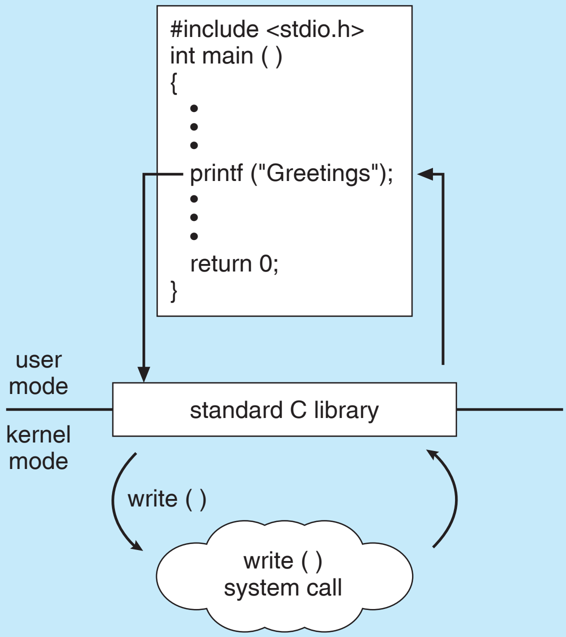

Figure 2.9 Standard C library handling of `write()`.

types, dealing with the coordination of concurrent processes, are discussed in great detail in Chapter 6.

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 2.10(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 give the program as much memory as possible (Figure 2.10(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

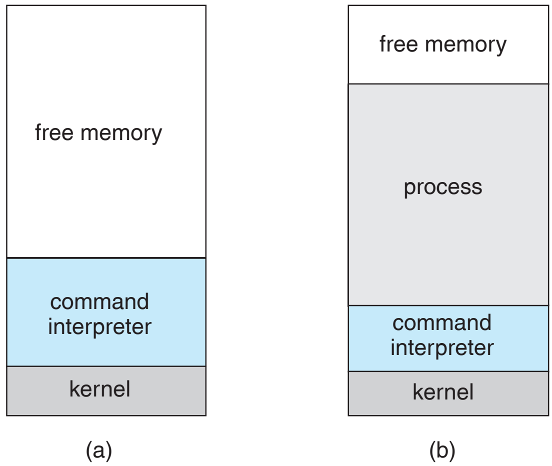

Figure 2.10 MS-DOS execution. (a) At system startup. (b) Running a program.

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 2.11). To start a new process, the shell 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

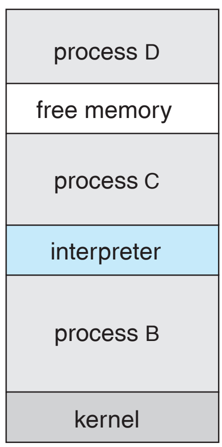

Figure 2.11 FreeBSD running multiple programs.

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 9 and 10. 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` (rewinding or skipping 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 attribute` and `set file attribute`, 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 just 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` the 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.

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 systems 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 6. 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 PDAs, 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 13 and the much larger issue of security in Chapter 14.

2.5 System Programs

Another aspect of a modern system is the 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, Visual Basic, and PERL) are often provided to the user with the operating system.
- **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 electronic-mail messages, to log in remotely, and to transfer files from one machine to another.

In addition to systems 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 Vista. 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 shared, 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 desire 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-time operating system for embedded systems, must have been substantially different from those for MVS, a large multiuser, multiaccess operating system for IBM mainframes.

Specifying and designing an operating system is a highly creative task. Although no textbook can tell you how to do it, general principles have been developed in the field of **software engineering**, and we turn now to a discussion of some of these principles.

2.6.2 Mechanisms and Policies

One important principle is the separation of **policy** from **mechanism**. Mechanisms determine *how* to do something; policies determine *what* will be done. For example, the timer construct (see Section 1.5.2) is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism insensitive to changes in policy would be more desirable. A change in policy would then require redefinition of only certain parameters of the system. For instance, consider a mechanism for giving priority to certain types of programs over others. If the mechanism is properly separated from policy, it can be used either to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy.

Microkernel-based operating systems (Section 2.7.3) take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks. These blocks are almost policy free, allowing more advanced mechanisms and policies to be added via user-created kernel modules or via user programs themselves. As an example, consider the history of UNIX. At first, it had a time-sharing scheduler. In the latest version of Solaris, scheduling is controlled by loadable tables. Depending on the table currently loaded, the system can be time shared, batch processing, real time, fair share, or any combination. Making the scheduling mechanism general purpose allows vast policy changes to be made with a single `load-new-table` command. At the other extreme is a system such as Windows, in which both mechanism and policy are encoded in the system to enforce a global look and feel. All applications have similar interfaces, because the interface itself is built into the kernel and system libraries. The Mac OS X operating system has similar functionality.

Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made. Whenever the question is *how* rather than *what*, it is a mechanism that must be determined.

2.6.3 Implementation

Once an operating system is designed, it must be implemented. Traditionally, operating systems have been written in assembly language. Now, however, they are most commonly written in higher-level languages such as C or C++.

The first system that was not written in assembly language was probably the Master Control Program (MCP) for Burroughs computers. MCP was written in a variant of ALGOL. MULTICS, developed at MIT, was written mainly in PL/1. The Linux and Windows operating systems are written mostly in C, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers.

The advantages of using a higher-level language, or at least a systems-implementation language, for implementing operating systems are the same as those accrued when the language is used for application programs: the code can be written faster, is more compact, and is easier to understand and debug. In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to *port*—to move to some other hardware—if it is written in a higher-level language. For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it runs natively only on the Intel X86 family of CPUs. (Although MS-DOS runs natively only on Intel X86, emulators of the X86 instruction set allow the operating system to run non-natively—slower, with more resource use—on other CPUs. **Emulators** are programs that duplicate the functionality of one system with another system.) The Linux operating system, in contrast, is written mostly in C and is available natively on a number of different CPUs, including Intel X86, Sun SPARC, and IBMPowerPC.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. These, however, are no longer major issues in today's systems. Although an expert assembly-language programmer can produce efficient small routines, for large programs a modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code. Modern processors have deep pipelining and multiple functional units that can handle the details of complex dependencies much more easily than can the human mind.

As is true in other systems, major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code. In addition, although operating systems are large, only a small amount of the code is critical to high performance; the memory manager and the CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottleneck routines can be identified and can be replaced with assembly-language equivalents.

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 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 commercial 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 divided into modules carefully. Figure 2.12 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

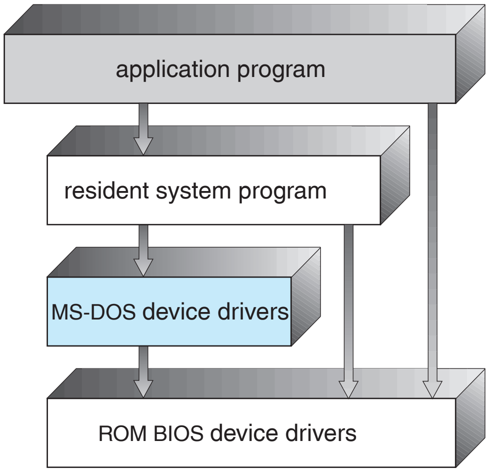

Figure 2.12 MS-DOS layer structure.

is further separated into a series of interfaces and device drivers that have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered, as shown in Figure 2.13. 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.

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

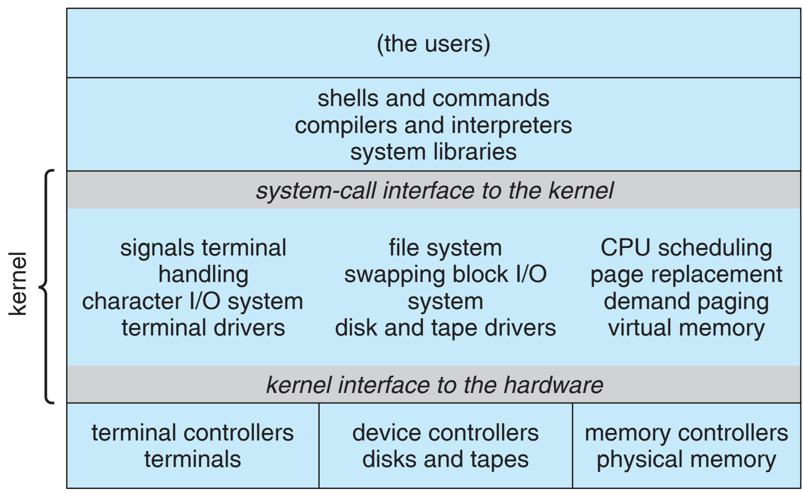

Figure 2.13 Traditional UNIX system structure.

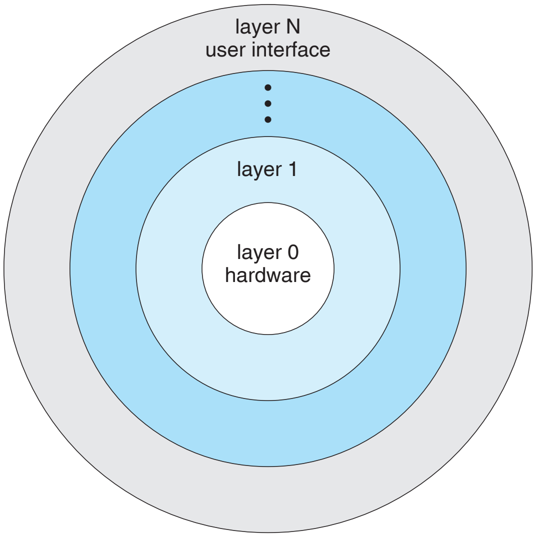

Figure 2.14 A layered operating system.

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 2.14.

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 with only those 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 difficult 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.

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

Several 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 based on the Mach microkernel.

Another example is QNX, a real-time operating system. The QNX 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, microkernels can suffer from performance decreases due to increased system function overhead. Consider the history of Windows NT. The first release had a layered microkernel organization. However, this version delivered low performance compared with that of Windows 95. Windows NT 4.0 partially redressed the performance problem by moving layers from user space to kernel space and integrating them more closely. By the time Windows XP was designed, its architecture was more monolithic than microkernel.

2.7.4 Modules

Perhaps the best current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel. Here, the kernel has a set of core components and links in additional services either during boot time or during run time. Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X. For example, 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

Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically. For example, device and

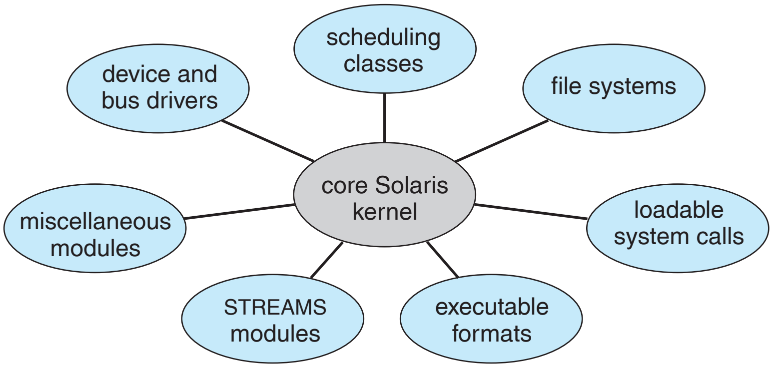

Figure 2.15 Solaris loadable modules.

bus drivers for specific hardware can be added to the kernel, and support for different file systems can be added as 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 in that any module can call any other module. Furthermore, the approach is like 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 is more efficient, because modules do not need to invoke message passing in order to communicate.

The Apple Mac OS X operating system uses a hybrid structure. It is a layered system in which one layer consists of the Mach microkernel. The structure of Mac OS X appears in Figure 2.16. The top layers include application environments and a set of services providing a graphical interface to applications. Below these layers is the kernel environment, which consists primarily of the Mach microkernel and the BSD kernel. Mach provides memory management; support for remote procedure calls (RPCs) and interprocess communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads.

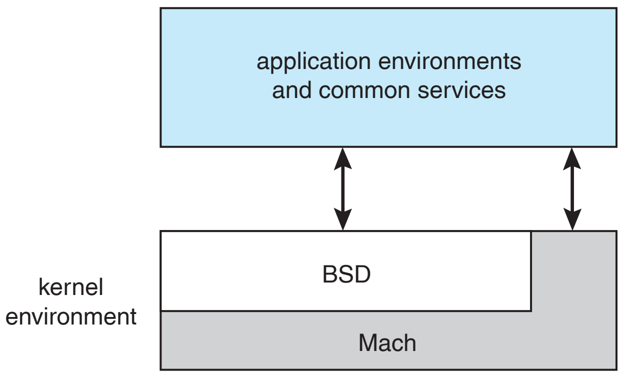

Figure 2.16 The Mac OS X structure.

In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which Mac OS X calls **kernel extensions**). As shown in the figure, applications and common services can make use of either the Mach or BSD facilities directly.

2.8 Virtual Machines

The layered approach described in Section 2.7.2 is taken to its logical conclusion in the concept of a **virtual machine**. The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.

By using CPU scheduling (Chapter 5) and virtual-memory techniques (Chapter 8), an operating system **host** can create the illusion that a process has its own processor with its own (virtual) memory. The virtual machine provides an interface that is *identical* to the underlying bare hardware. Each **guest** process is provided with a (virtual) copy of the underlying computer (Figure 2.17). Usually, the guest process is in fact an operating system, and that is how a single physical machine can run multiple operating systems concurrently, each in its own virtual machine.

2.8.1 History

Virtual machines first appeared commercially on IBM mainframes via the VM operating system in 1972. VM has evolved and is still available, and many of

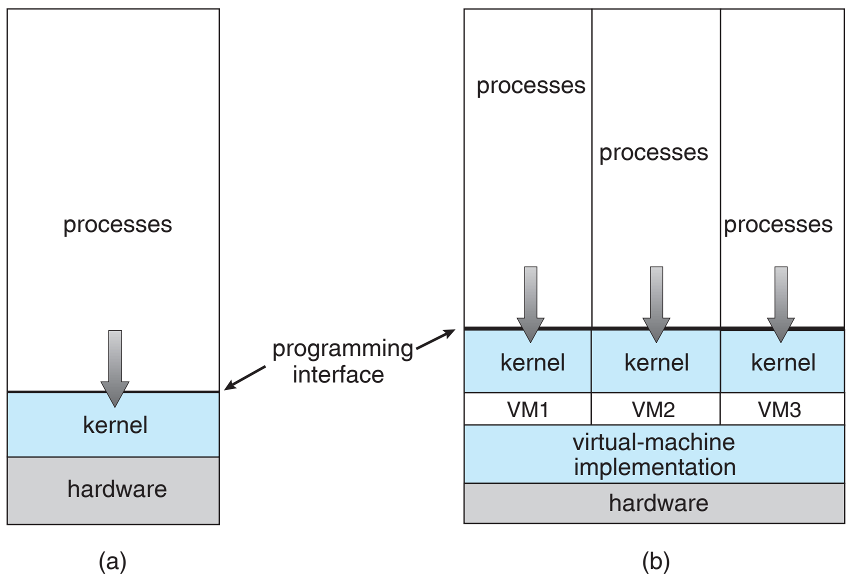

Figure 2.17 System models. (a) Nonvirtual machine. (b) Virtual machine.

the original concepts are found in other systems, making this facility worth exploring.

IBM VM370 divided a mainframe into multiple virtual machines, each running its own operating system. A major difficulty with the VM virtual-machine approach involved disk systems. Suppose that the physical machine had three disk drives but wanted to support seven virtual machines. Clearly, it could not allocate a disk drive to each virtual machine, because the virtual-machine software itself needed substantial disk space to provide virtual memory and spooling. The solution was to provide virtual disks—termed *minidisks* in IBM's VM operating system—that are identical in all respects except size. The system implemented each minidisk by allocating as many tracks on the physical disks as the minidisk needed.

Once these virtual machines were created, users could run any of the operating systems or software packages that were available on the underlying machine. For the IBM VM system, a user normally ran CMS—a single-user interactive operating system.

2.8.2 Benefits

There are several reasons for creating a virtual machine. Most of them are fundamentally related to being able to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently.

One important advantage is that the host system is protected from the virtual machines, just as the virtual machines are protected from each other. A virus inside a guest operating system might damage that operating system but is unlikely to affect the host or the other guests. Because each virtual machine is completely isolated from all other virtual machines, there are no protection problems. At the same time, however, there is no direct sharing of resources. Two approaches to provide sharing have been implemented. First, it is possible to share a file-system volume and thus to share files. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. The network is modeled after physical communication networks but is implemented in software.

A virtual-machine system is a perfect vehicle for operating-systems research and development. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and it is difficult to be sure that a change in one part will not cause obscure bugs to appear in some other part. The power of the operating system makes changing it particularly dangerous. Because the operating system executes in kernel mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

The operating system, however, runs on and controls the entire machine. Therefore, the current system must be stopped and taken out of use while changes are made and tested. This period is commonly called *system-development time*. Since it makes the system unavailable to users, system-development time is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine. Normal system operation seldom needs to be disrupted for system development.

Another advantage of virtual machines for developers is that multiple operating systems can be running on the developer's workstation concurrently. This virtualized workstation allows for rapid porting and testing of programs in varying environments. Similarly, quality-assurance engineers can test their applications in multiple environments without buying, powering, and maintaining a computer for each environment.

A major advantage of virtual machines in production data-center use is system **consolidation**, which involves taking two or more separate systems and running them in virtual machines on one system. Such physical-to-virtual conversions result in resource optimization, as many lightly used systems can be combined to create one more heavily used system.

If the use of virtual machines continues to spread, application deployment will evolve accordingly. If a system can easily add, remove, and move a virtual machine, then why install applications on that system directly? Instead, application developers could pre-install the application on a tuned and customized operating system in a virtual machine. That virtual environment would be the release mechanism for the application. This method would be an improvement for application developers; application management would become easier, less tuning would required, and technical support of the application would be more straightforward. System administrators would find the environment easier to manage as well. Installation would be simple, and redeploying the application to another system would be much easier than the usual steps of uninstalling and reinstalling. For widespread adoption of this methodology to occur, though, the format of virtual machines must be standardized so that any virtual machine will run on any virtualization platform. The "Open Virtual Machine Format" is an attempt to do just that, and it could succeed in unifying virtual-machine formats.

2.8.3 Simulation

System virtualization as discussed so far is just one of many system-emulation methodologies. Virtualization is the most common because it makes guest operating systems and applications "believe" they are running on native hardware. Because only the system's resources need to be virtualized, these guests run at almost full speed.

Another methodology is **simulation**, in which the host system has one system architecture and the guest system was compiled for a different architecture. For example, suppose a company has replaced its outdated computer system with a new system but would like to continue to run certain important programs that were compiled for the old system. The programs could be run in an emulator that translates each of the outdated system's instructions into the native instruction set of the new system. Emulation can increase the life of programs and allow us to explore old architectures without having an actual old machine, but its major challenge is performance. Instruction-set emulation can run an order of magnitude slower than native instructions. Thus, unless the new machine is ten times faster than the old, the program running on the new machine will run more slowly than it did on its native hardware. Another challenge is that it is difficult to create a correct emulator because, in essence, this involves writing an entire CPU in software.

2.8.4 Para-virtualization

Para-virtualization is another variation on this theme. Rather than try to trick a guest operating system into believing it has a system to itself, para-virtualization presents the guest with a system that is similar but not identical to the guest's preferred system. The guest must be modified to run on the paravirtualized hardware. The gain for this extra work is more efficient use of resources and a smaller virtualization layer.

Solaris 10 includes **containers**, or **zones**, that create a virtual layer between the operating system and the applications. In this system, only one kernel is installed, and the hardware is not virtualized. Rather, the operating system and its devices are virtualized, providing processes within a container with the impression that they are the only processes on the system. One or more containers can be created, and each can have its own applications, network stacks, network address and ports, user accounts, and so on. CPU resources can be divided up among the containers and the systemwide processes. Figure 2.18 shows a Solaris 10 system with two containers and the standard "global" user space.

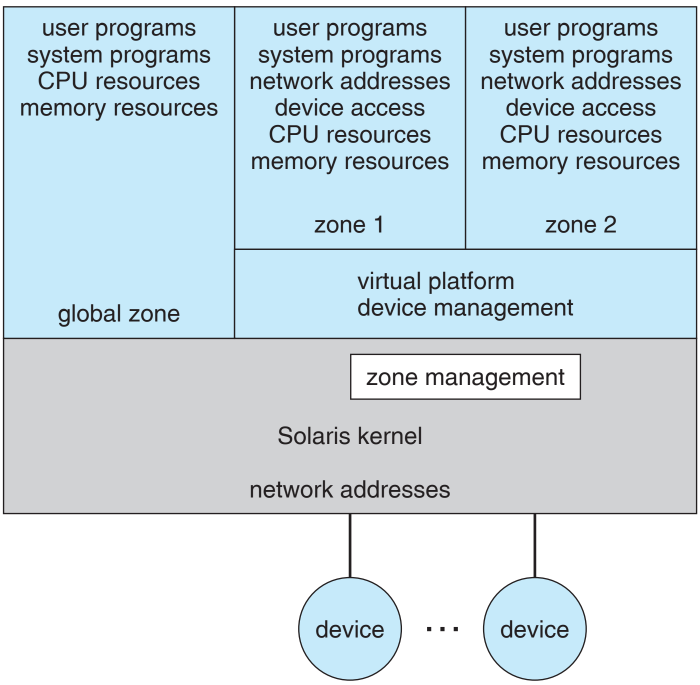

Figure 2.18 Solaris 10 with two containers.

2.8.5 Implementation

Although the virtual-machine concept is useful, it is difficult to implement. Much work is required to provide an *exact* duplicate of the underlying machine. Remember that the underlying machine typically has two modes: user mode and kernel mode. The virtual-machine software can run in kernel mode, since it is the operating system. The virtual machine itself can execute in only user mode. Just as the physical machine has two modes, however, so must the virtual machine. Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in a physical user mode. Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode on a virtual machine.

Such a transfer can be accomplished as follows. When a system call, for example, is made by a program running on a virtual machine in virtual user mode, it will cause a transfer to the virtual-machine monitor in the real machine. When the virtual-machine monitor gains control, it can change the register contents and program counter for the virtual machine to simulate the effect of the system call. It can then restart the virtual machine, noting that it is now in virtual kernel mode.

The major difference, of course, is time. Whereas the real I/O might have taken 100 milliseconds, the virtual I/O might take less time (because it is spooled) or more time (because it is interpreted). In addition, the CPU is being multiprogrammed among many virtual machines, further slowing down the virtual machines in unpredictable ways. In the extreme case, it may be necessary to simulate all instructions to provide a true virtual machine. VM, discussed earlier, works for IBM machines because normal instructions for the virtual machines can execute directly on the hardware. Only the privileged instructions (needed mainly for I/O) must be simulated and hence execute more slowly.

Without some level of hardware support, virtualization would be impossible. The more hardware support available within a system, the more feature rich, stable, and well performing the virtual machines can be. All major general-purpose CPUs provide some amount of hardware support for virtualization. For example, AMD virtualization technology is found in several AMD processors. It defines two new modes of operation—host and guest. Virtual machine software can enable host mode, define the characteristics of each guest virtual machine, and then switch the system to guest mode, passing control of the system to the guest operating system that is running in the virtual machine. In guest mode, the virtualized operating system thinks it is running on native hardware and sees certain devices (those included in the host's definition of the guest). If the guest tries to access a virtualized resource, then control is passed to the host to manage that interaction.

2.8.6 Examples

Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed. Today, however, virtual machines are coming into fashion as a means of solving system compatibility problems. In this section, we explore two popular contemporary virtual machines: the VMware Workstation and the Java virtual machine. As you will see, these virtual machines can typically run on top of operating systems of any of the design types discussed earlier. Thus, operating system design methods—simple layers, microkernels, modules, and virtual machines—are not mutually exclusive.

2.8.6.1 VMware

Most of the virtualization techniques discussed in this section require virtualization to be supported by the kernel. Another method involves writing the virtualization tool to run in user mode as an application on top of the operating system. Virtual machines running within this tool believe they are running on bare hardware but in fact are running inside a user-level application.

VMware Workstation is a popular commercial application that abstracts Intel X86 and compatible hardware into isolated virtual machines. VMware Workstation runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different guest operating systems as independent virtual machines.

The architecture of such a system is shown in Figure 2.19. In this scenario, Linux is running as the host operating system and FreeBSD, Windows NT, and Windows XP are running as guest operating systems. The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

The physical disk the guest owns and manages is really just a file within the file system of the host operating system. To create an identical guest instance, we can simply copy the file. Copying the file to another location protects the guest instance against a disaster at the original site. Moving the file to another

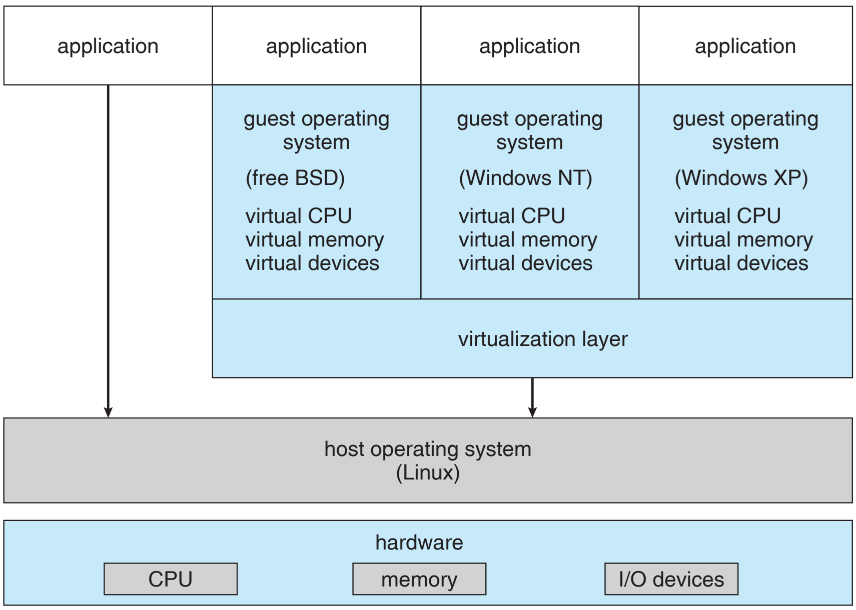

Figure 2.19 VMware architecture.

location moves the guest system. These scenarios show how virtualization can improve the efficiency of system administration as well as system resource use.

2.8.6.2 The Java Virtual Machine

Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995. In addition to a language specification and a large API library, Java also provides a specification for a Java virtual machine—or JVM.

Java objects are specified with the `class` construct; a Java program consists of one or more classes. For each Java class, the compiler produces an architecture-neutral **bytecode** output (`.class`) file that will run on any implementation of the JVM.

The JVM is a specification for an abstract computer. It consists of a **class loader** and a Java interpreter that executes the architecture-neutral bytecodes, as diagrammed in Figure 2.20. The class loader loads the compiled `.class` files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the `.class` file is valid Java bytecode and does not overflow or underflow the stack. It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The JVM also automatically manages memory by performing **garbage collection**—the practice of reclaiming memory from objects no longer in use and returning it to the system. Much research focuses on garbage-collection algorithms for increasing the performance of Java programs in the virtual machine.

The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or Mac OS X, or as part of a Web browser. Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs. If the JVM is implemented in software, the Java interpreter interprets the bytecode operations one at a time. A faster software technique is to use a **just-in-time (JIT)** compiler. Here, the first time a Java method is invoked, the bytecodes for the method are turned into native machine language for the host system. These operations are then cached so that subsequent invocations of a method are performed using the native machine instructions and the bytecode operations need not be interpreted all over again. A technique that is potentially even faster is to run the JVM in hardware on a

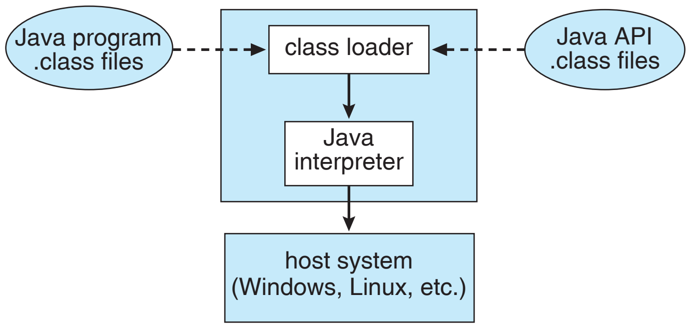

Figure 2.20 The Java virtual machine.

THE .NET FRAMEWORK

The .NET Framework is a collection of technologies, including a set of class libraries, and an execution environment that come together to provide a platform for developing software. This platform allows programs to be written to target the .NET Framework instead of a specific architecture. A program written for the .NET Framework need not worry about the specifics of the hardware or the operating system on which it will run. Thus, any architecture implementing .NET will be able to successfully execute the program. This is because the execution environment abstracts these details and provides a virtual machine as an intermediary between the executing program and the underlying architecture.

At the core of the .NET Framework is the Common Language Runtime (CLR). The CLR is the implementation of the .NET virtual machine. It provides an environment for execution of programs written in any of the languages targeted at the .NET Framework. Programs written in languages such as C# (pronounced *C-sharp*) and VB.NET are compiled into an intermediate, architecture-independent language called Microsoft Intermediate Language (MS-IL). These compiled files, called assemblies, include MS-IL instructions and metadata. They have file extensions of either .EXE or .DLL. Upon execution of a program, the CLR loads assemblies into what is known as the **Application Domain**. As instructions are requested by the executing program, the CLR converts the MS-IL instructions inside the assemblies into native code that is specific to the underlying architecture using just-in-time compilation. Once instructions have been converted to native code, they are kept and will continue to run as native code for the CPU. The architecture of the CLR for the .NET framework is shown in Figure 2.21.

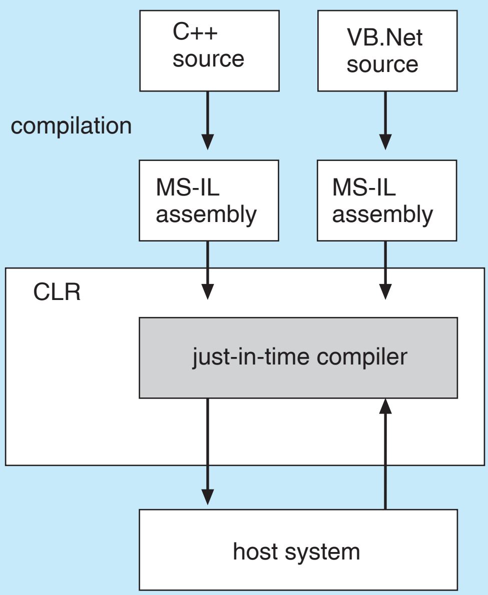

Figure 2.21 Architecture of the CLR for the .NET Framework.

special Java chip that executes the Java bytecode operations as native code, thus bypassing the need for either a software interpreter or a just-in-time compiler.

2.9 Operating-System Debugging

Broadly, **debugging** is the activity of finding and fixing errors, or **bugs**, in a system. Debugging seeks to find and fix errors in both hardware and software. Performance problems are considered bugs, so debugging can also include **performance tuning**, which seeks to improve performance by removing **bottlenecks** in the processing taking place within a system. A discussion of hardware debugging is outside of the scope of this text. In this section, we explore debugging kernel and process errors and performance problems.

2.9.1 Failure Analysis

If a process fails, most operating systems write the error information to a **log file** to alert system operators or users that the problem occurred. The operating system can also take a **core dump**—a capture of the memory (referred to as the "core" in the early days of computing) of the process. This core image is stored in a file for later analysis. Running programs and core dumps can be probed by a **debugger**, a tool designed to allow a programmer to explore the code and memory of a process.

Debugging user-level process code is a challenge. Operating-system kernel debugging is even more complex because of the size and complexity of the kernel, its control of the hardware, and the lack of user-level debugging tools. A kernel failure is called a **crash**. As with a process failure, error information is saved to a log file, and the memory state is saved to a **crash dump**.

Operating system debugging frequently uses different tools and techniques from process debugging due to the very different nature of these two tasks. Consider that a kernel failure in the file-system code would make it risky for the kernel to try to save its state to a file on the file system before rebooting. A common technique is to save the kernel's memory state to a section of disk set aside for this purpose that contains no file system. If the kernel detects an unrecoverable error, it writes the entire contents of memory, or at least the kernel-owned parts of the system memory, to the disk area. When the system reboots, a process runs to gather the data from that area and write it to a crash dump file within a file system for analysis.

2.9.2 Performance Tuning

To identify bottlenecks, we must be able to monitor system performance. Code must be added to compute and display measures of system behavior. In a number of systems, the operating system does this task by producing trace listings of system behavior. All interesting events are logged with their time and important parameters and are written to a file. Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies. These same traces can be run as input for a simulation of a suggested improved system. Traces also can help people to find errors in operating-system behavior.

Kernighan's Law

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Another approach to performance tuning is to include interactive tools with the system that allow users and administrators to question the state of various components of the system to look for bottlenecks. The UNIX command `top` displays resources used on the system, as well as a sorted list of the "top" resource-using processes. Other tools display the state of disk I/O, memory allocation, and network traffic. The authors of these single-purpose tools try to guess what a user would want to see while analyzing a system and to provide that information.

Making running operating systems easier to understand, debug, and tune is an active area of operating-system research and implementation. The cycle of enabling tracing as system problems occur and analyzing the traces later is being broken by a new generation of kernel-enabled performance analysis tools. Further, these tools are not single-purpose or merely for sections of code that were written to emit debugging data. The Solaris 10 DTrace dynamic tracing facility is a leading example of such a tool.

2.9.3 DTrace

DTrace is a facility that dynamically adds probes to a running system, both in user processes and in the kernel. These probes can be queried via the D programming language to determine an astonishing amount about the kernel, the system state, and process activities. For example, Figure 2.22 follows an application as it executes a system call (`ioctl`) and further shows the functional calls within the kernel as they execute to perform the system call. Lines ending with "U" are executed in user mode, and lines ending in "K" in kernel mode.

Debugging the interactions between user-level and kernel code is nearly impossible without a toolset that understands both sets of code and can instrument the interactions. For that toolset to be truly useful, it must be able to debug any area of a system, including areas that were not written with debugging in mind, and do so without affecting system reliability. This tool must also have a minimum performance impact—ideally it should have no impact when not in use and a proportional impact during use. The DTrace tool meets these requirements and provides a dynamic, safe, low-impact debugging environment.

Until the DTrace framework and tools became available with Solaris 10, kernel debugging was usually shrouded in mystery and accomplished via happenstance and archaic code and tools. For example, CPUs have a breakpoint feature that will halt execution and allow a debugger to examine the state of the system. Then execution can continue until the next breakpoint or termination. This method cannot be used in a multiuser operating-system kernel without negatively affecting all of the users on the system. **Profiling**, which periodically samples the instruction pointer to determine which code is being executed, can show statistical trends but not individual activities. Code can be included in the kernel to emit specific data under specific circumstances, but that code

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                     U
  0   -> _XEventsQueued                  U
  0     -> _X11TransBytesReadable        U
  0     <- _X11TransBytesReadable        U
  0     -> _X11TransSocketBytesReadable U
  0     <- _X11TransSocketBytesreadable U
  0     -> ioctl                         U
  0       -> ioctl                       K
  0         -> getf                      K
  0           -> set_active_fd           K
  0           <- set_active_fd           K
  0         <- getf                      K
  0         -> get_udatamodel            K
  0         <- get_udatamodel            K
...
  0         -> releasef                  K
  0           -> clear_active_fd         K
  0           <- clear_active_fd         K
  0           -> cv_broadcast            K
  0           <- cv_broadcast            K
  0         <- releasef                  K
  0       <- ioctl                       K
  0     <- ioctl                         U
  0   <- _XEventsQueued                  U
  0 <- XEventsQueued                     U
```

Figure 2.22 Solaris 10 dtrace follows a system call within the kernel.

slows down the kernel and tends not to be included in the part of the kernel where the specific problem being debugged is occurring.

In contrast, DTrace runs on production systems—systems that are running important or critical applications—and causes no harm to the system. It slows activities while enabled, but after execution it resets the system to its pre-debugging state. It is also a broad and deep tool. It can broadly debug everything happening in the system (both at the user and kernel levels and between the user and kernel layers). DTrace can also delve deeply into code, showing individual CPU instructions or kernel subroutine activities.

DTrace is composed of a compiler, a framework, **providers** of **probes** written within that framework, and **consumers** of those probes. DTrace providers create probes. Kernel structures exist to keep track of all probes that the providers have created. The probes are stored in a hash table data structure that is hashed by name and indexed according to unique probe identifiers. When a probe is enabled, a bit of code in the area to be probed is rewritten to call `dtrace_probe(probe identifier)` and then continue with the code's original operation. Different providers create different kinds of probes. For example, a kernel system-call probe works differently from a user-process probe, and that is different from an I/O probe.

DTrace features a compiler that generates a byte code that is run in the kernel. This code is assured to be "safe" by the compiler. For example, no loops are allowed, and only specific kernel state modifications are allowed when specifically requested. Only users with the DTrace "privileges" (or "root" users) are allowed to use DTrace, as it can retrieve private kernel data (and modify data if requested). The generated code runs in the kernel and enables probes. It also enables consumers in user mode and enables communications between the two.

A DTrace consumer is code that is interested in a probe and its results. A consumer requests that the provider create one or more probes. When a probe fires, it emits data that are managed by the kernel. Within the kernel, actions called **enabling control blocks**, or **ECBs**, are performed when probes fire. One probe can cause multiple ECBs to execute if more than one consumer is interested in that probe. Each ECB contains a predicate ("if statement") that can filter out that ECB. Otherwise, the list of actions in the ECB is executed. The most usual action is to capture some bit of data, such as a variable's value at that point of the probe execution. By gathering such data, a complete picture of a user or kernel action can be built. Further, probes firing from both user space and the kernel can show how a user-level action caused kernel-level reactions. Such data are invaluable for performance monitoring and code optimization.

Once the probe consumer terminates, its ECBs are removed. If there are no ECBs consuming a probe, the probe is removed. That involves rewriting the code to remove the dtrace_probe call and put back the original code. Thus, before a probe is created and after it is destroyed, the system is exactly the same, as if no probing occurred.

DTrace takes care to assure that probes do not use too much memory or CPU capacity, which could harm the running system. The buffers used to hold the probe results are monitored for exceeding default and maximum limits. CPU time for probe execution is monitored as well. If limits are exceeded, the consumer is terminated, along with the offending probes. Buffers are allocated per CPU to avoid contention and data loss.

An example of D code and its output shows some of its utility. The following program shows the DTrace code to enable scheduler probes and record the amount of CPU time for each process running with user ID 101 while those probes are enabled (that is, while the program runs):

```
sched:::on-cpu
uid == 101
{
   self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
   @time[execname] = sum(timestamp - self->ts);
   self->ts = 0;
}
```

The output of the program, showing the processes and how much time (in nanoseconds) they spend running on the CPUs, is shown in Figure 2.23.

```
# 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 2.23 Output of the D code.

Because DTrace is part of the open-source Solaris 10 operating system, it is being added to other operating systems when those systems do not have conflicting license agreements. For example, DTrace has been added to Mac OS X 10.5 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.

2.10 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.

2.11 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, PDAs, 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 11.5.1) 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**.

2.12 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 when in an interactive or time-shared mode. System programs are provided to satisfy many common user requests.

The types of requests vary according to level. The system-call level must provide the basic functions, such as process control and file and device manipulation. Higher-level requests, satisfied by the command interpreter or system programs, are translated into a sequence of system calls. System services can be classified into several categories: program control, status requests, and I/O requests. Program errors can be considered implicit requests for service.

Once the system services are defined, the structure of the operating system can be developed. Various tables are needed to record the information that defines the state of the computer system and the status of the system's jobs.

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. The type of system desired is the foundation for choices among various algorithms and strategies that will be needed.

Since an operating system is large, modularity is important. Designing a system as a sequence of layers or using a microkernel is considered a good technique. The virtual-machine concept takes the layered approach and treats both the kernel of the operating system and the hardware as though they were hardware. Even other operating systems may be loaded on top of this virtual machine.

Throughout the entire operating-system design cycle, we must be careful to separate policy decisions from implementation details (mechanisms). This separation allows maximum flexibility if policy decisions are to be changed later.

Operating systems are now almost always written in a systems-implementation language or in a higher-level language. This feature improves their implementation, maintenance, and portability. To create an operating system for a particular machine configuration, we must perform system generation.

Debugging process and kernel failures can be accomplished through the use of debuggers and other tools that analyze core dumps. Tools such as DTrace analyze production systems to find bottlenecks and understand other system behavior.

For a computer system to begin running, the CPU must initialize and start executing the bootstrap program in firmware. The bootstrap can execute the operating system directly if the operating system is also in the firmware, or it can complete a sequence in which it loads progressively smarter programs from firmware and disk until the operating system itself is loaded into memory and executed.

Practice Exercises

2.1 What is the purpose of system calls?

2.2 What are the five major activities of an operating system with regard to process management?

2.3 What are the three major activities of an operating system with regard to memory management?

2.4 What are the three major activities of an operating system with regard to secondary-storage management?

2.5 What is the purpose of the command interpreter? Why is it usually separate from the kernel?

2.6 What system calls have to be executed by a command interpreter or shell in order to start a new process?

2.7 What is the purpose of system programs?

2.8 What is the main advantage of the layered approach to system design? What are the disadvantages of the layered approach?

2.9 List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.

2.10 Why do some systems store the operating system in firmware, while others store it on disk?

2.11 How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?

Exercises

2.12 The services and functions provided by an operating system can be divided into two main categories. Briefly describe the two categories and discuss how they differ.

2.13 Describe three general methods for passing parameters to the operating system.

2.14 Describe how you could obtain a statistical profile of the amount of time spent by a program executing different sections of its code. Discuss the importance of obtaining such a statistical profile.

2.15 What are the five major activities of an operating system with regard to file management?

2.16 What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?

2.17 Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?

2.18 What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?

2.19 Why is the separation of mechanism and policy desirable?

2.20 It is sometimes difficult to achieve a layered approach if two components of the operating system are dependent on each other. Identify a scenario in which it is unclear how to layer two system components that require tight coupling of their functionalities.

2.21 What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of the microkernel approach?

2.22 In what ways is the modular kernel approach similar to the layered approach? In what ways does it differ from the layered approach?

2.23 What is the main advantage for an operating-system designer of using a virtual-machine architecture? What is the main advantage for a user?

2.24 Why is a just-in-time compiler useful for executing Java programs?

2.25 What is the relationship between a guest operating system and a host operating system in a system like VMware? What factors need to be considered in choosing the host operating system?

2.26 The experimental Synthesis operating system has an assembler incorporated in the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended to make building the operating system easier. Discuss the pros and cons of the Synthesis approach to kernel design and system-performance optimization.

Programming Problems

2.27 In Section 2.3, we described a program that copies the contents of one file to a destination file. This program works by first prompting the user for the name of the source and destination files. Write this program using either the Win32 or POSIX API. Be sure to include all necessary error checking, including ensuring that the source file exists.
Once you have correctly designed and tested the program, if you used a system that supports it, run the program using a utility that traces system calls. Linux systems provide the `ptrace` utility, and Solaris systems use the `truss` or `dtrace` command. On Mac OS X, the `ktrace` facility provides similar functionality. As Windows systems do not provide such features, you will have to trace through the Win32 version of this program using a debugger.

Programming Projects

Adding a system call to the Linux kernel.

In this project, you will study the system-call interface provided by the Linux operating system and learn how user programs communicate with the operating system kernel via this interface. Your task is to incorporate a new system call into the kernel, thereby expanding the functionality of the operating system.

Part 1: Getting Started

A user-mode procedure call is performed by passing arguments to the called procedure either on the stack or through registers, saving the current state and the value of the program counter, and jumping to the beginning of the code corresponding to the called procedure. The process continues to have the same privileges as before.

System calls appear as procedure calls to user programs but result in a change in execution context and privileges. In Linux on the Intel 386 architecture, a system call is accomplished by storing the system-call number into the EAX register, storing arguments to the system call in other hardware registers, and executing a trap instruction (which is the `INT 0x80` assembly instruction). After the trap is executed, the system-call number is used to index into a table of code pointers to obtain the starting address for the handler code implementing the system call. The process then jumps to this address, and the privileges of the process are switched from user to kernel mode. With the expanded privileges, the process can now execute kernel code, which may include privileged instructions that cannot be executed in user mode. The kernel code can then carry out the requested services, such as interacting with I/O devices, and can perform process management and other activities that cannot be performed in user mode.

The system call numbers for recent versions of the Linux kernel are listed in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. (For instance, `__NR_close` corresponds to the system call `close()`, which is invoked for closing a file descriptor and is defined as value 6.) The list of pointers to system-call handlers is typically stored in the file `/usr/src/linux-2.x/arch/i386/kernel/entry.S` under the heading `ENTRY(sys_call_table)`. Notice that `sys_close` is stored at entry number 6 in the table to be consistent with the system-call number defined in the `unistd.h` file. (The keyword `.long` denotes that the entry will occupy the same number of bytes as a data value of type `long`.)

Part 2: Building a New Kernel

Before adding a system call to the kernel, you must familiarize yourself with the task of building the binary for a kernel from its source code and booting the machine with the newly built kernel. This activity comprises the following tasks, some of which depend on the particular installation of the Linux operating system in use.

- Obtain the kernel source code for the Linux distribution. If the source code package has already been installed on your machine, the corresponding files may be available under `/usr/src/linux` or `/usr/src/linux-2.x` (where the suffix corresponds to the kernel version number). If the package has not yet been installed, it can be downloaded from the provider of your Linux distribution or from `http://www.kernel.org`.
- Learn how to configure, compile, and install the kernel binary. This will vary among the different kernel distributions, but some typical commands for building the kernel (after entering the directory where the kernel source code is stored) include:

- ○ `make xconfig`
- ○ `make dep`
- ○ `make bzImage`

- Add a new entry to the set of bootable kernels supported by the system. The Linux operating system typically uses utilities such as `lilo` and `grub` to maintain a list of bootable kernels from which the user can choose during machine boot-up. If your system supports `lilo`, add an entry to `lilo.conf`, such as:

```
image=/boot/bzImage.mykernel
label=mykernel
root=/dev/hda5
read-only
```

  where `/boot/bzImage.mykernel` is the kernel image and `mykernel` is the label associated with the new kernel. This step will allow you to choose the new kernel during the boot-up process. You will then have the option of either booting the new kernel or booting the unmodified kernel if the newly built kernel does not function properly.

Part 3: Extending the Kernel Source

You can now experiment with adding a new file to the set of source files used for compiling the kernel. Typically, the source code is stored in the `/usr/src/linux-2.x/kernel` directory, although that location may differ in your Linux distribution. There are two options for adding the system call. The first is to add the system call to an existing source file in this directory. The second is to create a new file in the source directory and modify `/usr/src/linux-2.x/kernel/Makefile` to include the newly created file in the compilation process. The advantage of the first approach is that when you modify an existing file that is already part of the compilation process, the Makefile need not be modified.

Part 4: Adding a System Call to the Kernel

Now that you are familiar with the various background tasks corresponding to building and booting Linux kernels, you can begin the process of adding a new system call to the Linux kernel. In this project, the system call will have limited functionality; it will simply transition from user mode to kernel mode, print a message that is logged with the kernel messages, and transition back to user mode. We will call this the *helloworld* system call. While it has only limited functionality, it illustrates the system-call mechanism and sheds light on the interaction between user programs and the kernel.

- Create a new file called `helloworld.c` to define your system call. Include the header files `linux/linkage.h` and `linux/kernel.h`. Add the following code to this file:

```
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage int sys_helloworld() {
   printk(KERN_EMERG "hello world!");

   return 1;
}
```

This creates a system call with the name `sys_helloworld()`. If you choose to add this system call to an existing file in the source directory, all that is necessary is to add the `sys_helloworld()` function to the file you choose. In the code, `asmlinkage` is a remnant from the days when Linux used both C++ and C code and is used to indicate that the code is written in C. The `printk()` function is used to print messages to a kernel log file and therefore may be called only from the kernel. The kernel messages specified in the parameter to `printk()` are logged in the file `/var/log/kernel/warnings`. The function prototype for the `printk()` call is defined in `/usr/include/linux/kernel.h`.

- Define a new system call number for `__NR_helloworld` in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. A user program can use this number to identify the newly added system call. Also be sure to increment the value for `__NR_syscalls`, which is stored in the same file. This constant tracks the number of system calls currently defined in the kernel.

- Add an entry `.long sys_helloworld` to the `sys_call_table` defined in the `/usr/src/linux-2.x/arch/i386/kernel/entry.S` file. As discussed earlier, the system-call number is used to index into this table to find the position of the handler code for the invoked system call.

- Add your file `helloworld.c` to the Makefile (if you created a new file for your system call.) Save a copy of your old kernel binary image (in case there are problems with your newly created kernel). You can now build the new kernel, rename it to distinguish it from the unmodified kernel, and add an entry to the loader configuration files (such as `lilo.conf`). After completing these steps, you can boot either the old kernel or the new kernel that contains your system call.

Part 5: Using the System Call from a User Program

When you boot with the new kernel, it will support the newly defined system call; you now simply need to invoke this system call from a user program. Ordinarily, the standard C library supports an interface for system calls defined for the Linux operating system. As your new system call is not linked into the standard C library, however, invoking your system call will require manual intervention.

As noted earlier, a system call is invoked by storing the appropriate value in a hardware register and performing a trap instruction. Unfortunately, these low-level operations cannot be performed using C language statements and instead require assembly instructions. Fortunately, Linux provides macros for instantiating wrapper functions that contain the appropriate assembly instructions. For instance, the following C program uses the `_syscall0()` macro to invoke the newly defined system call:

```
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

_syscall0(int, helloworld);

main()
{
   helloworld();
}
```

- The `_syscall0` macro takes two arguments. The first specifies the type of the value returned by the system call; the second is the name of the system call. The name is used to identify the system-call number that is stored in the hardware register before the trap instruction is executed. If your system call requires arguments, then a different macro (such as `_syscall0`, where the suffix indicates the number of arguments) could be used to instantiate the assembly code required for performing the system call.
- Compile and execute the program with the newly built kernel. There should be a message "`hello world!`" in the kernel log file `/var/log/kernel/warnings` to indicate that the system call has executed.

As a next step, consider expanding the functionality of your system call. How would you pass an integer value or a character string to the system call and have it printed into the kernel log file? What are the implications of passing pointers to data stored in the user program's address space as opposed to simply passing an integer value from the user program to the kernel using hardware registers?

Bibliographical Notes

Dijkstra [1968] advocated the layered approach to operating-system design. Brinch-Hansen [1970] was an early proponent of constructing an operating system as a kernel (or nucleus) on which more complete systems can be built.

System instrumentation and dynamic tracing are described in Tamches and Miller [1999]. DTrace is discussed in Cantrill et al. [2004]. The DTrace source code is available at `http://src.opensolaris.org/source/`. Cheung and Loong [1995] explore issues of operating-system structure from microkernel to extensible systems.

MS-DOS, Version 3.1, is described in Microsoft [1986]. Windows NT and Windows 2000 are described by Solomon [1998] and Solomon and Russinovich [2000]. WinSEVEN internals are described in Russinovich and Solomon [2009]. Hart [2005] covers Windows systems programming in detail. BSD UNIX is described in McKusick et al. [1996]. Bovet and Cesati [2006] thoroughly discuss the Linux kernel. Several UNIX systems—including Mach—are treated in detail in Vahalia [1996]. Mac OS X is presented at `http://www.apple.com/macosx` and in Singh [2007]. Solaris is fully described in McDougall and Mauro [2007].

The first operating system to provide a virtual machine was the CP/67 on an IBM 360/67. The commercially available IBM VM/370 operating system was derived from CP/67. Details regarding Mach, a microkernel-based operating system, can be found in Young et al. [1987]. Kaashoek et al. [1997] present details regarding exokernel operating systems, wherein the architecture separates management issues from protection, thereby giving untrusted software the ability to exercise control over hardware and software resources.

The specifications for the Java language and the Java virtual machine are presented by Gosling et al. [1996] and by Lindholm and Yellin [1999], respectively. The internal workings of the Java virtual machine are fully described by Venners [1998]. Golm et al. [2002] highlight the JX operating system; Back et al. [2000] cover several issues in the design of Java operating systems. More information on Java is available on the Web at `http://www.javasoft.com`. Details about the implementation of VMware can be found in Sugerman et al. [2001]. Information about the Open Virtual Machine Format can be found at `http://www.vmware.com/appliances/learn/ovf.html`.

Part Two

Process Management

A *process* can be thought of as a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices—to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

A process is the unit of work in most systems. Systems consist of a collection of processes: Operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

Although traditionally a process contained only a single *thread* of control as it ran, most modern operating systems now support processes that have multiple threads.

The operating system is responsible for the following activities in connection with process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.