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

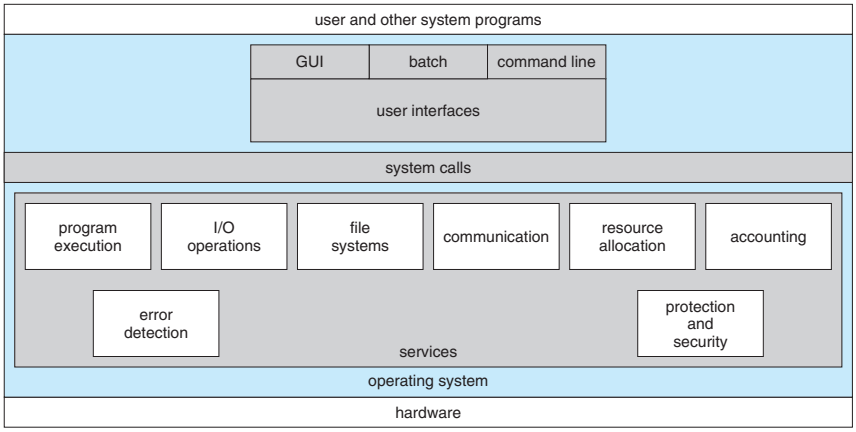


Figure 2.1 A view of operating-system services.

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 XP 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

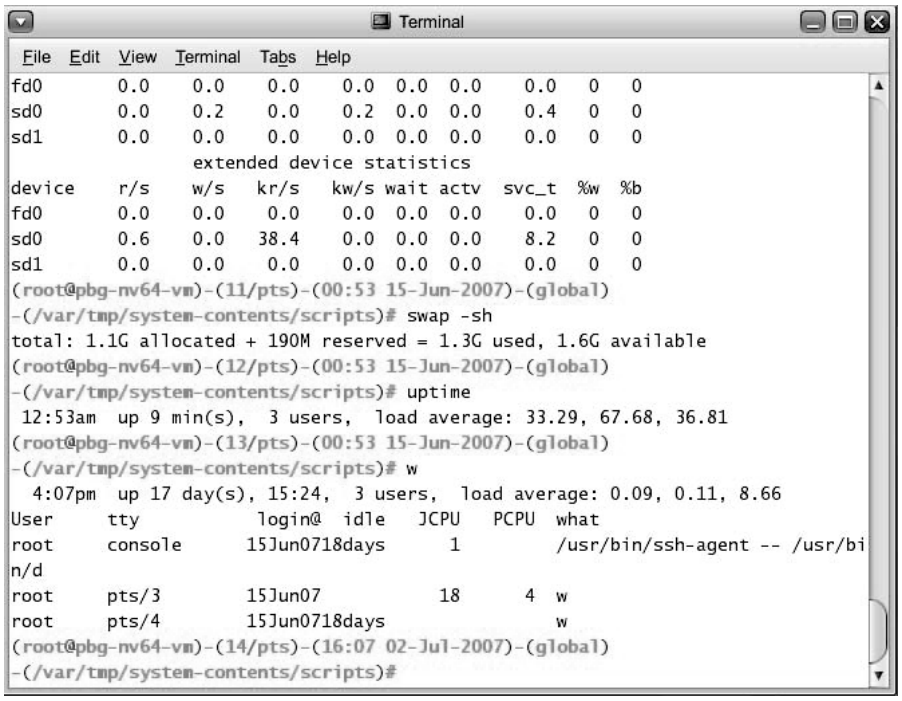


Figure 2.2 The Bourne shell command interpreter in Solaris 10.

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 screen shot 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). This system-

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

Most programmers never see this level of detail, however. Typically, application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the 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 NT`CreateProcess()` 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.

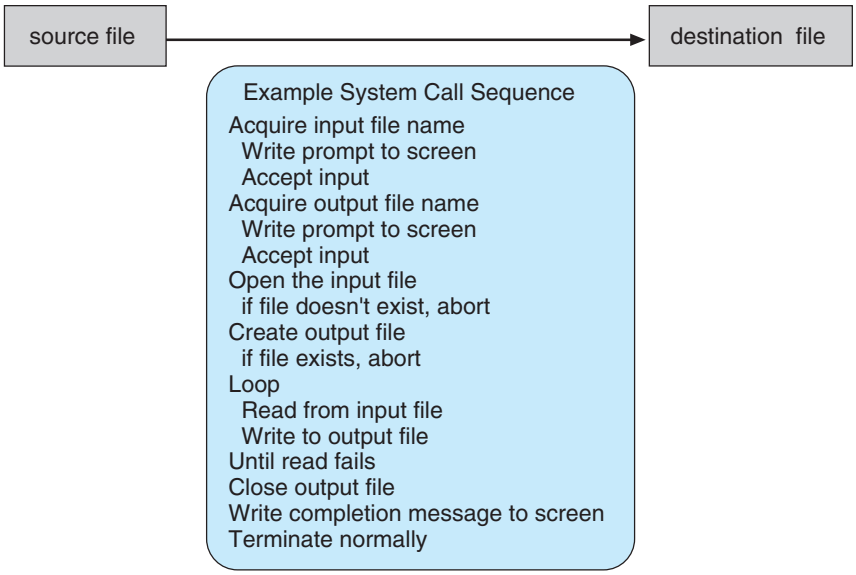


Figure 2.4 Example of using system calls.

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `write()` method in the `java.io.OutputStream` class in the Java API, which allows writing to a file or network connection. The API for this method appears in Figure 2.5.

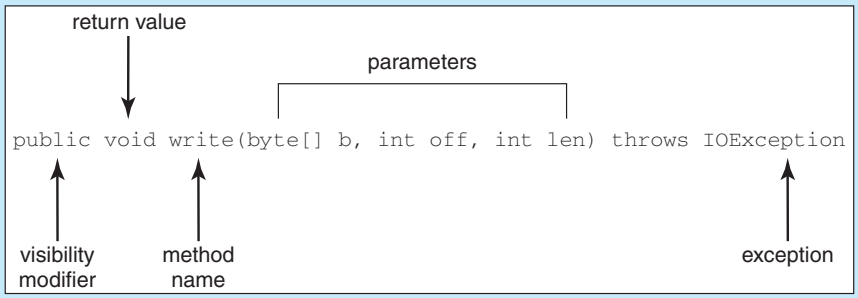


Figure 2.5 The API for the `write()` method.

This method returns a `void`—or no return value. An `IOException` is thrown if an I/O error occurs. The parameters passed to `write()` can be described as follows:

- `byte[] b`—the data to be written.
- `int off`—the starting offset in the array `b` to be written.
- `int len`—the number of bytes to be written.

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 among an API, the system-call interface, and the operating system is shown in Figure 2.6, which illustrates how the operating system handles a user application invoking the `open()` system call.

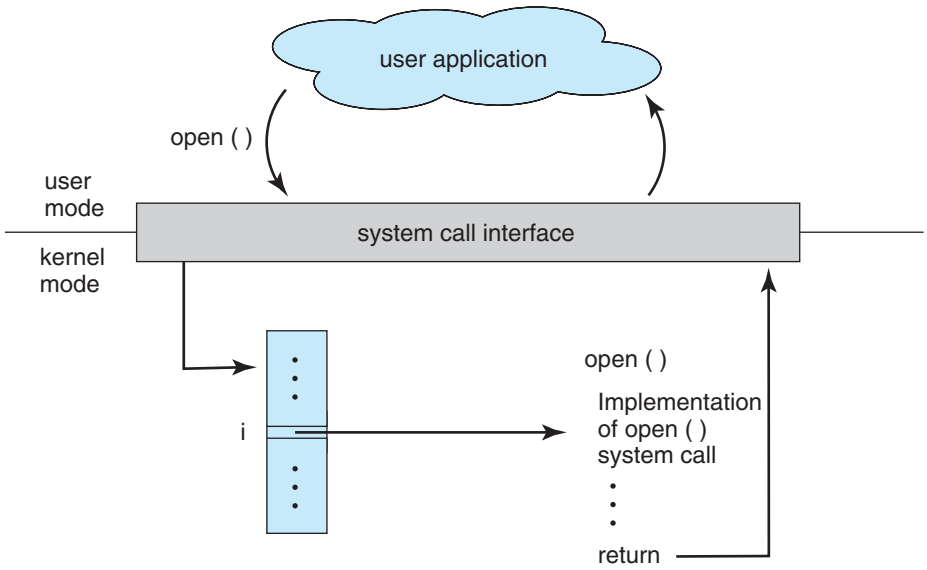


Figure 2.6 The handling of a user application invoking the `open()` system call.

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.

Because Java is intended to run on platform-neutral systems, it is not possible to make system calls directly from a Java program. However, it is possible for a Java method to invoke C or C++ code that is **native** to the underlying platform on which the program is running (for example, Microsoft Vista or Linux). The C/C++ code can invoke a system call on the host system, thus allowing a Java program to make the system call indirectly. This is accomplished through the Java Native Interface (JNI), which allows a Java method to be declared `native`. This `native` Java method is used as a placeholder for the actual C/C++ function. Thus, calling the `native` Java method actually invokes the function written in C or C++. Obviously, a Java program that uses `native` methods is not considered portable from one system to another.

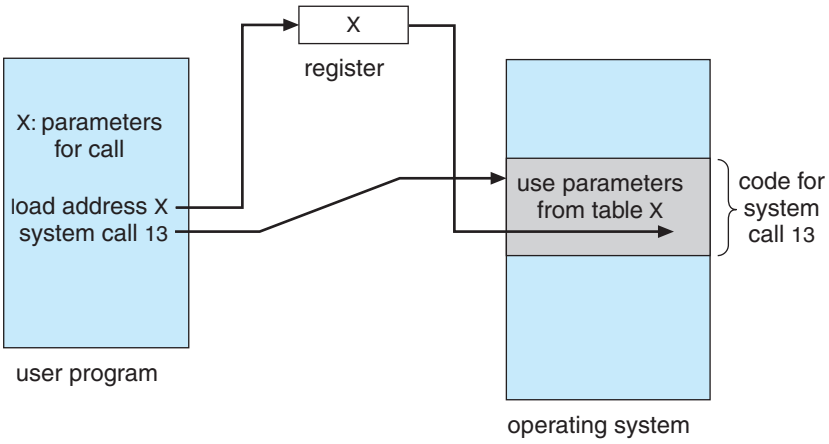


Figure 2.7 Passing of parameters as a table.

2.4 Types of System Calls

System calls can be grouped roughly into six major categories: **process control**, **file manipulation**, **device manipulation**, **information maintenance**, **communications**, and **protection**. In Sections 2.4.1 through 2.4.6, we 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 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

- 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.

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

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

the loaded program terminates. This question is related to the problem of whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program.

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

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

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

EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program. This is shown in Figure 2.9.

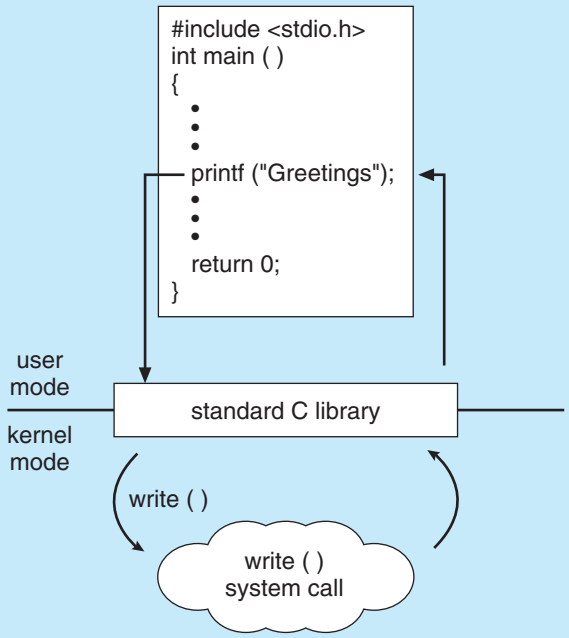


Figure 2.9 Standard C library handling of `write()`.

(`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 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 until the lock is removed. Typically such system calls include `acquire lock` and `release lock`. System calls of these types, dealing with the coordination of concurrent processes, are discussed in great detail in Chapter 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 2.10a). Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a

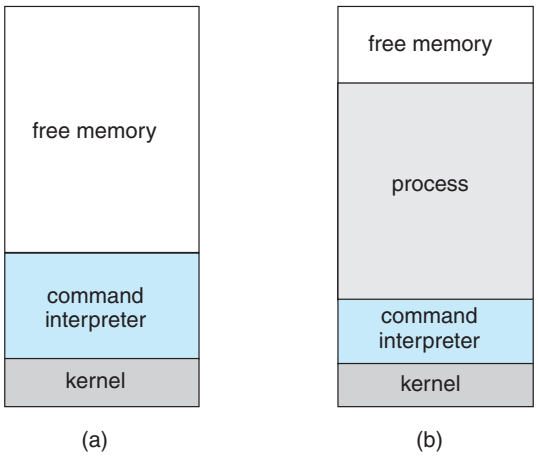


Figure 2.10 MS-DOS execution. (a) At system startup. (b) Running a program.

new process. It loads the program into memory, writing over most of itself to give the program as much memory as possible (Figure 2.10b). Next, it sets the instruction pointer to the first instruction of the program. The program then runs, and either an error causes a trap, or the program executes a system call to terminate. In either case, the error code is saved in the system memory for later use. Following this action, the small portion of the command interpreter that was not overwritten resumes execution. Its first task is to reload the rest of the command interpreter from disk. Then the command interpreter makes the previous error code available to the user or to the next program.

FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user's choice is run. This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.11). To start a new process, the shell

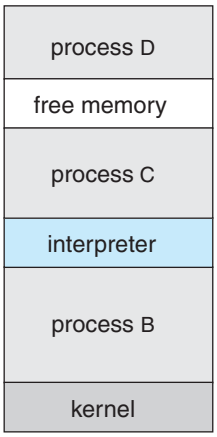


Figure 2.11 FreeBSD running multiple programs.

executes a `fork()` system call. Then, the selected program is loaded into memory via an `exec()` system call, and the program is executed. Depending on the way the command was issued, the shell then either waits for the process to finish or runs the process "in the background." In the latter case, the shell immediately requests another command. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files or through a GUI interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program's priority, and so on. When the process is done, it executes an `exit()` system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs. Processes are discussed in Chapter 3, which includes 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 10 and 11. 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 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 first to `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, which is described in Chapter 7.

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

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

2.4.4 Information Maintenance

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

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

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

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

2.4.5 Communication

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

in a network has a *host name* by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a *process name*, and this name is translated into an identifier by which the operating system can refer to the process. The `get hostid` and `get processid` system calls do this translation. The identifiers are then passed to the general-purpose `open` and `close` calls provided by the file system or to specific `open connection` and `close connection` system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an `accept connection` call. Most processes that will be receiving connections are special-purpose *daemons*, which are 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 among the processes sharing memory.

2.4.6 Protection

Protection provides a mechanism for controlling access to a computer system's resources. 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 14 and the much larger issue of security in Chapter 15.

2.5 System Programs

Another aspect of a modern system is the collection of system programs. Recall Figure 1.1, which depicts 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 e-mail messages, to log in remotely, or to transfer files from one machine to another.

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

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls. Consider a user's PC. When a user's computer is running the Mac OS X operating system, the user might see the GUI, featuring a mouse-and-windows

interface. Alternatively, or even in one of the windows, the user might have a command-line UNIX shell. Both use the same set of system calls, but the system calls look different and act in different ways. Further confusing the user view, consider the user dual-booting Mac OS X and 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 want certain obvious properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe, and fast. Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.

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

There is, in short, no unique solution to the problem of defining the requirements for an operating system. The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments. For example, the requirements for VxWorks, a real-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 a higher-level language 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 XP 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—more slowly, with more resource use—on other CPUs. **Emulators** are programs that duplicate the functionality of one system in 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 IBM PowerPC.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. This, however, is no longer a major issue 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. (Bottlenecks are discussed further later in this chapter.)

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 systems to crash

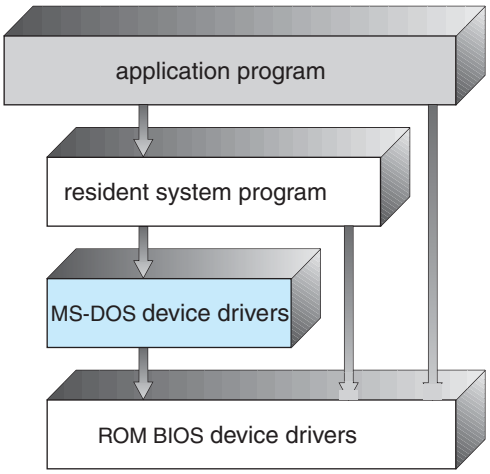


Figure 2.12 MS-DOS layer structure.

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

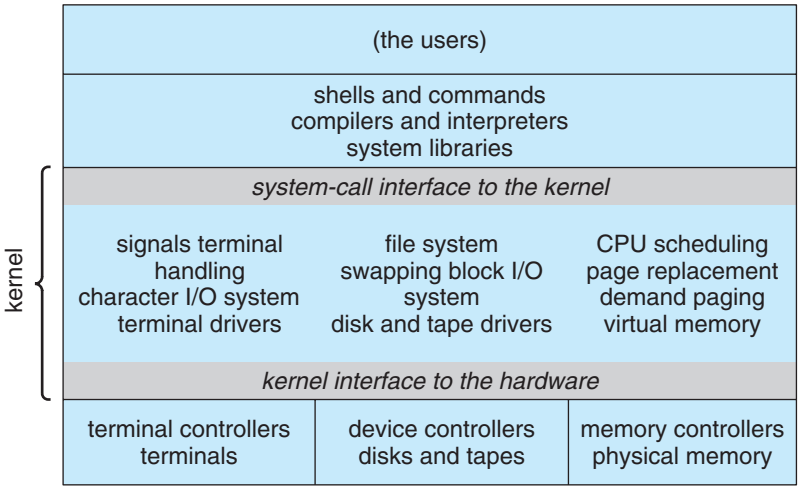


Figure 2.13 Traditional UNIX system structure.

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 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

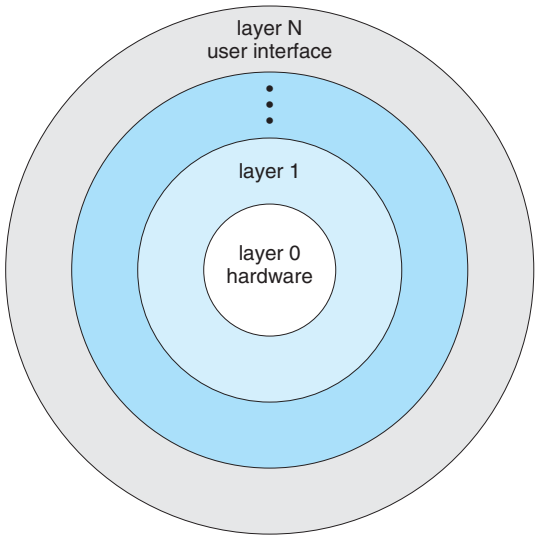


Figure 2.14 A layered operating system.

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

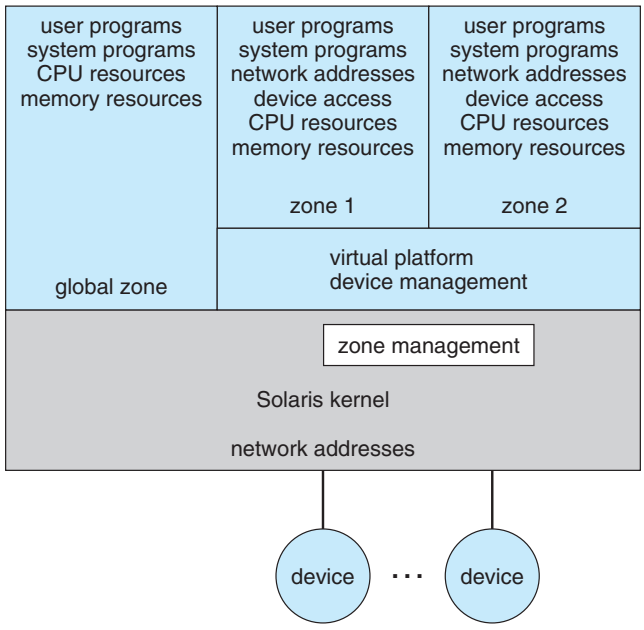


Figure 2.19 Solaris 10 with two containers.

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, giving processes within a container 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 system-wide processes. Figure 2.19 shows a Solaris 10 system with two containers and the standard "global" user space.

2.9 Java

Java is a technology introduced by Sun Microsystems in the mid-1990s. We refer to it as a *technology* rather than just a programming language because it provides more than a conventional programming language. Java technology consists of two essential components:

1. Programming-language specification
2. Virtual-machine specification

We provide an overview of these two components in this section.

2.9.1 The Java Programming Language

Java is a general-purpose, object-oriented programming language with support for distributed programming. Java was originally favored by the Internet programming community because of its support for **applets**, which are

programs with limited resource access that run within a Web browser. Now, Java is a popular language for designing desktop applications, client–server Web applications, and applications that run within embedded systems, such as smartphones.

As mentioned, Java is an object-oriented language, which means that it offers support for the kind of object-oriented programming discussed earlier. Java objects are specified with the `class` construct; a Java program consists of one or more classes. For each Java class, the Java compiler produces an architecture-neutral **bytecode** output (`.class`) file that will run on any implementation of the Java virtual machine. Java also provides high-level support for networking and distributed objects. It is a multithreaded language as well, meaning that a Java program may have several different threads, or flows, of control, thus allowing the development of concurrent applications to take advantage of modern processors with multiple processing cores. We cover distributed objects using Java's remote method invocation (RMI) in Chapter 3, and we discuss multithreaded Java programs in Chapter 4. Java is also considered a secure language. This feature is especially important considering that a Java program may be executing across a distributed network. We look at Java security in Chapter 15.

Java programs are written using the Java Standard Edition API. This is a standard API for designing desktop applications and applets with basic language support for graphics, I/O, security, database connectivity, and networking.

2.9.2 The Java Virtual Machine

The Java virtual machine (JVM) is a specification for an abstract computer. It consists of a **class loader** and a Java interpreter that executes architecture-neutral bytecodes. The class loader loads the compiled `.class` files from both the Java program and the Java API for execution by the Java interpreter, as diagrammed in Figure 2.20. After a class is loaded, it verifies 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

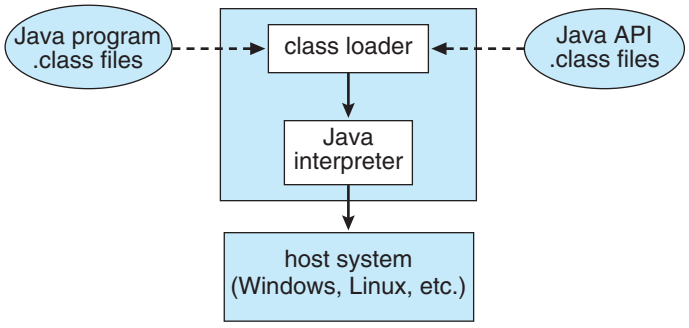


Figure 2.20 The Java virtual machine.

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.

An instance of the JVM is created whenever a Java application or applet is run. This instance of the JVM starts running when the `main()` method of a program is invoked. In the case of applets, the programmer does not define a `main()` method. Rather, the browser executes the `main()` method before creating the applet. If we simultaneously run two Java programs and a Java applet on the same computer, we will have three instances of the JVM.

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 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.

It is the JVM that makes it possible to develop programs that are architecture-neutral and portable. An implementation of the JVM is system-specific, and it abstracts the system in a standard way to the Java program, providing a clean, architecture-neutral interface. This interface allows a `.class` file to run on any system that has implemented the JVM according to its specification. Java virtual machines have been designed for most operating systems, including Windows, Linux, Mac OS X and Solaris. When we use the JVM in this text to illustrate operating system concepts, we refer to the specification of the JVM, rather than to any particular implementation.

2.9.3 The Java Development Kit

The Java development kit, or JDK, consists of (1) development tools, such as a compiler and debugger, and (2) a run-time environment, or JRE. The compiler turns a Java source file program into a bytecode (`.class`) file. The run-time environment provides the JVM as well as the Java API for the host system. The JDK is portrayed in Figure 2.21.

2.9.4 Java Operating Systems

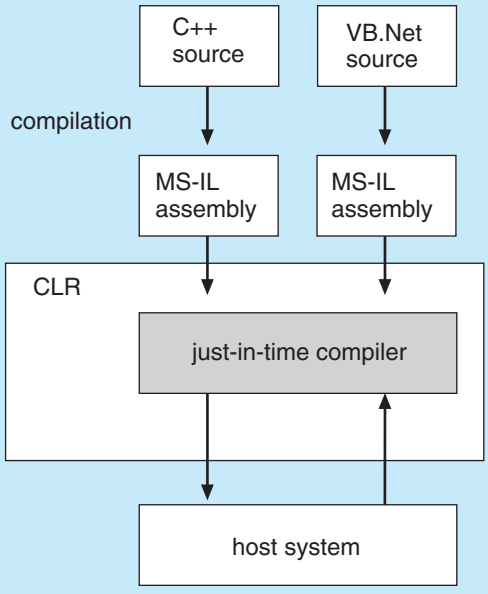
Most operating systems are written in a combination of C and assembly-language code, primarily because of the performance benefits of these languages and the ease of interfacing with hardware. However, recent efforts have been made to write operating systems in Java. Such a system, known as a **language-based extensible system**, runs in a single address space.

One of the difficulties in designing language-based systems concerns memory protection—protecting the operating system from malicious user programs as well as protecting user programs from one another. Traditional

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 below.



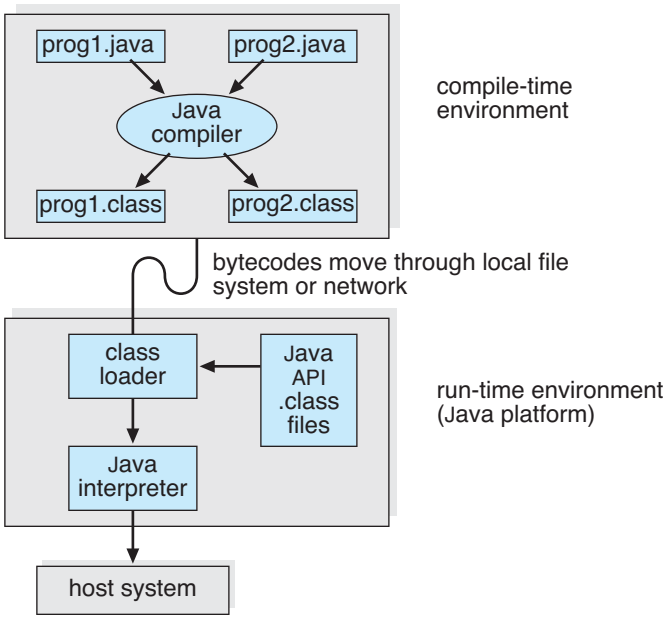


Figure 2.21 Java development kit.

operating systems rely on hardware features to provide memory protection (Section 8.1). Language-based systems instead rely on type-safety features of the language. As a result, language-based systems are desirable on small hardware devices, which may lack hardware features that provide memory protection.

The JX operating system is written almost entirely in Java and provides a run-time system for Java applications as well. JX organizes its system according to **domains**. Each domain represents an independent JVM. Additionally, each domain maintains a heap used for allocating memory during object creation and threads within itself, as well as for garbage collection. Domain zero is a microkernel (Section 2.7.3) responsible for low-level details such as system initialization and saving and restoring the state of the CPU. Domain zero is written in C and assembly language; all other domains are written entirely in Java. Communication between domains occurs through **portals**, communication mechanisms similar to the remote procedure calls (RPCs) used by the Mach microkernel. Protection within and between domains relies on the type safety of the Java language. Since domain zero is not written in Java, it must be considered **trusted**. The architecture of the JX system is illustrated in Figure 2.22.

2.10 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 improves performance by removing **bottlenecks**

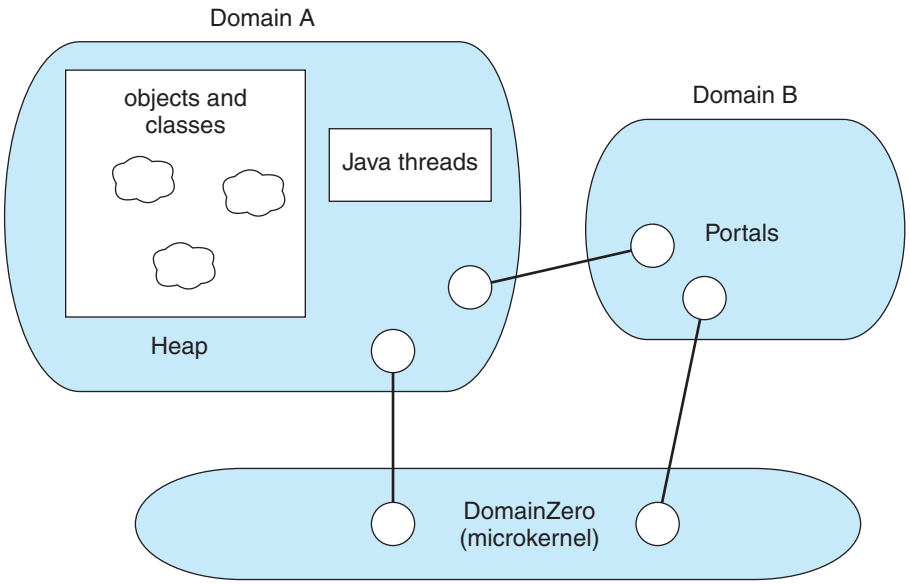


Figure 2.22 The JX operating system.

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.

2.10.1 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 challenging 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.

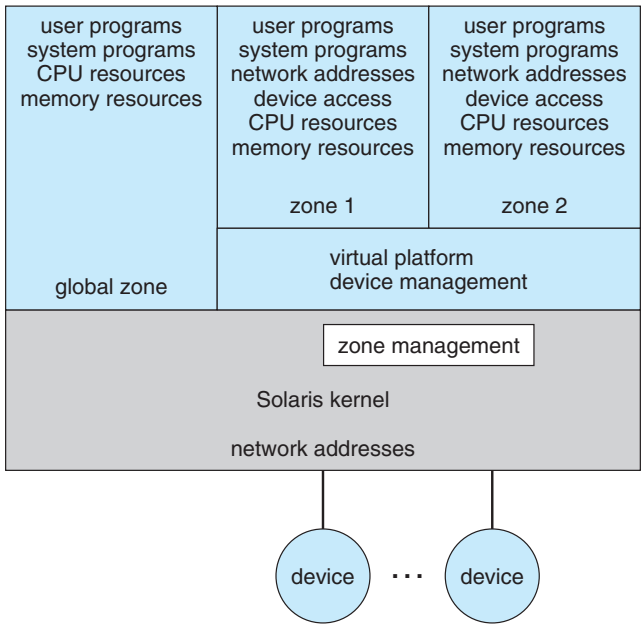


Figure 2.19 Solaris 10 with two containers.

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, giving processes within a container 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 system-wide processes. Figure 2.19 shows a Solaris 10 system with two containers and the standard "global" user space.

2.9 Java

Java is a technology introduced by Sun Microsystems in the mid-1990s. We refer to it as a *technology* rather than just a programming language because it provides more than a conventional programming language. Java technology consists of two essential components:

1. Programming-language specification
2. Virtual-machine specification

We provide an overview of these two components in this section.

2.9.1 The Java Programming Language

Java is a general-purpose, object-oriented programming language with support for distributed programming. Java was originally favored by the Internet programming community because of its support for **applets**, which are

programs with limited resource access that run within a Web browser. Now, Java is a popular language for designing desktop applications, client–server Web applications, and applications that run within embedded systems, such as smartphones.

As mentioned, Java is an object-oriented language, which means that it offers support for the kind of object-oriented programming discussed earlier. Java objects are specified with the `class` construct; a Java program consists of one or more classes. For each Java class, the Java compiler produces an architecture-neutral **bytecode** output (`.class`) file that will run on any implementation of the Java virtual machine. Java also provides high-level support for networking and distributed objects. It is a multithreaded language as well, meaning that a Java program may have several different threads, or flows, of control, thus allowing the development of concurrent applications to take advantage of modern processors with multiple processing cores. We cover distributed objects using Java's remote method invocation (RMI) in Chapter 3, and we discuss multithreaded Java programs in Chapter 4. Java is also considered a secure language. This feature is especially important considering that a Java program may be executing across a distributed network. We look at Java security in Chapter 15.

Java programs are written using the Java Standard Edition API. This is a standard API for designing desktop applications and applets with basic language support for graphics, I/O, security, database connectivity, and networking.

2.9.2 The Java Virtual Machine

The Java virtual machine (JVM) is a specification for an abstract computer. It consists of a **class loader** and a Java interpreter that executes architecture-neutral bytecodes. The class loader loads the compiled `.class` files from both the Java program and the Java API for execution by the Java interpreter, as diagrammed in Figure 2.20. After a class is loaded, it verifies 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

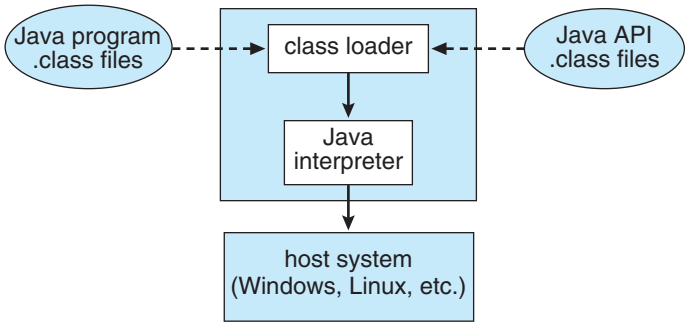


Figure 2.20 The Java virtual machine.

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.

An instance of the JVM is created whenever a Java application or applet is run. This instance of the JVM starts running when the `main()` method of a program is invoked. In the case of applets, the programmer does not define a `main()` method. Rather, the browser executes the `main()` method before creating the applet. If we simultaneously run two Java programs and a Java applet on the same computer, we will have three instances of the JVM.

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 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.

It is the JVM that makes it possible to develop programs that are architecture-neutral and portable. An implementation of the JVM is system-specific, and it abstracts the system in a standard way to the Java program, providing a clean, architecture-neutral interface. This interface allows a `.class` file to run on any system that has implemented the JVM according to its specification. Java virtual machines have been designed for most operating systems, including Windows, Linux, Mac OS X and Solaris. When we use the JVM in this text to illustrate operating system concepts, we refer to the specification of the JVM, rather than to any particular implementation.

2.9.3 The Java Development Kit

The Java development kit, or JDK, consists of (1) development tools, such as a compiler and debugger, and (2) a run-time environment, or JRE. The compiler turns a Java source file program into a bytecode (`.class`) file. The run-time environment provides the JVM as well as the Java API for the host system. The JDK is portrayed in Figure 2.21.

2.9.4 Java Operating Systems

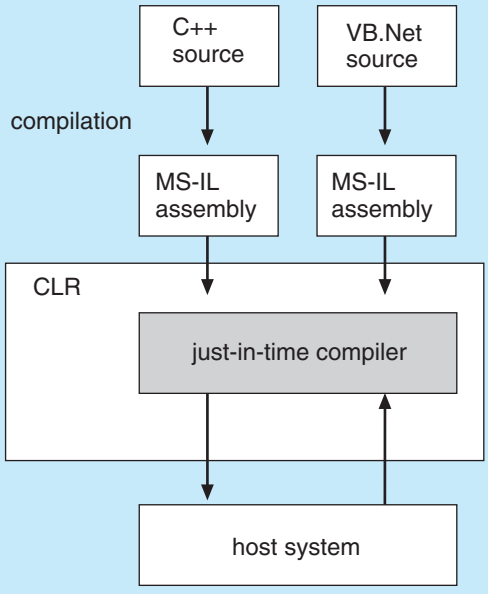
Most operating systems are written in a combination of C and assembly-language code, primarily because of the performance benefits of these languages and the ease of interfacing with hardware. However, recent efforts have been made to write operating systems in Java. Such a system, known as a **language-based extensible system**, runs in a single address space.

One of the difficulties in designing language-based systems concerns memory protection—protecting the operating system from malicious user programs as well as protecting user programs from one another. Traditional

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 below.



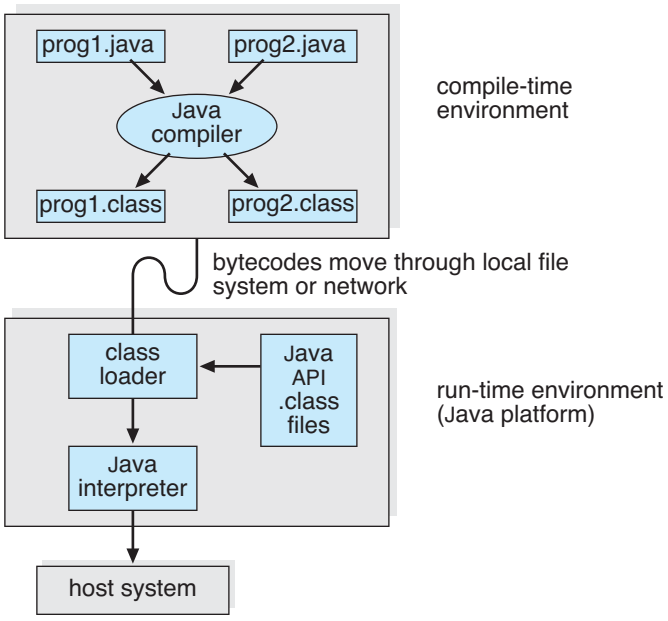


Figure 2.21 Java development kit.

operating systems rely on hardware features to provide memory protection (Section 8.1). Language-based systems instead rely on type-safety features of the language. As a result, language-based systems are desirable on small hardware devices, which may lack hardware features that provide memory protection.

The JX operating system is written almost entirely in Java and provides a run-time system for Java applications as well. JX organizes its system according to **domains**. Each domain represents an independent JVM. Additionally, each domain maintains a heap used for allocating memory during object creation and threads within itself, as well as for garbage collection. Domain zero is a microkernel (Section 2.7.3) responsible for low-level details such as system initialization and saving and restoring the state of the CPU. Domain zero is written in C and assembly language; all other domains are written entirely in Java. Communication between domains occurs through **portals**, communication mechanisms similar to the remote procedure calls (RPCs) used by the Mach microkernel. Protection within and between domains relies on the type safety of the Java language. Since domain zero is not written in Java, it must be considered **trusted**. The architecture of the JX system is illustrated in Figure 2.22.

2.10 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 improves performance by removing **bottlenecks**

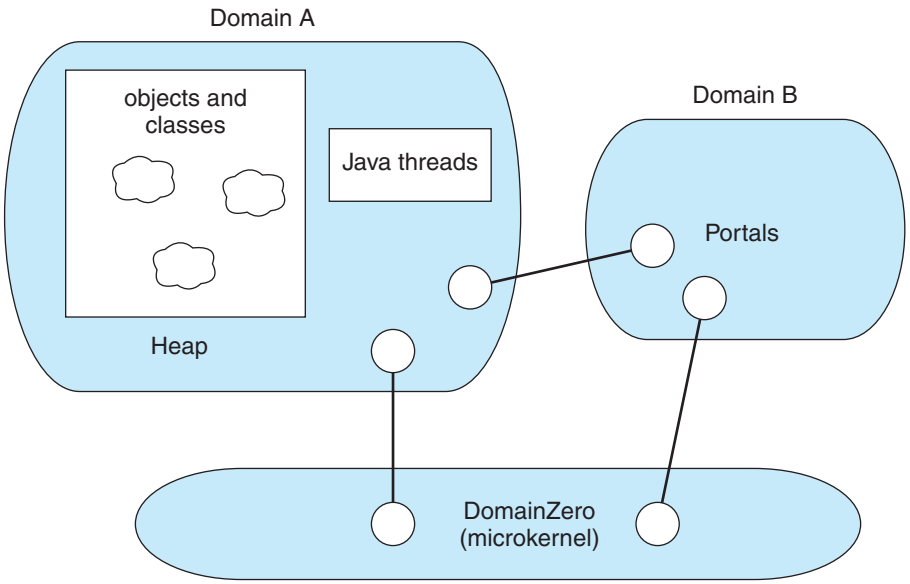


Figure 2.22 The JX operating system.

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.

2.10.1 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 challenging 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.