2.1 Operating-system services

Introduction

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

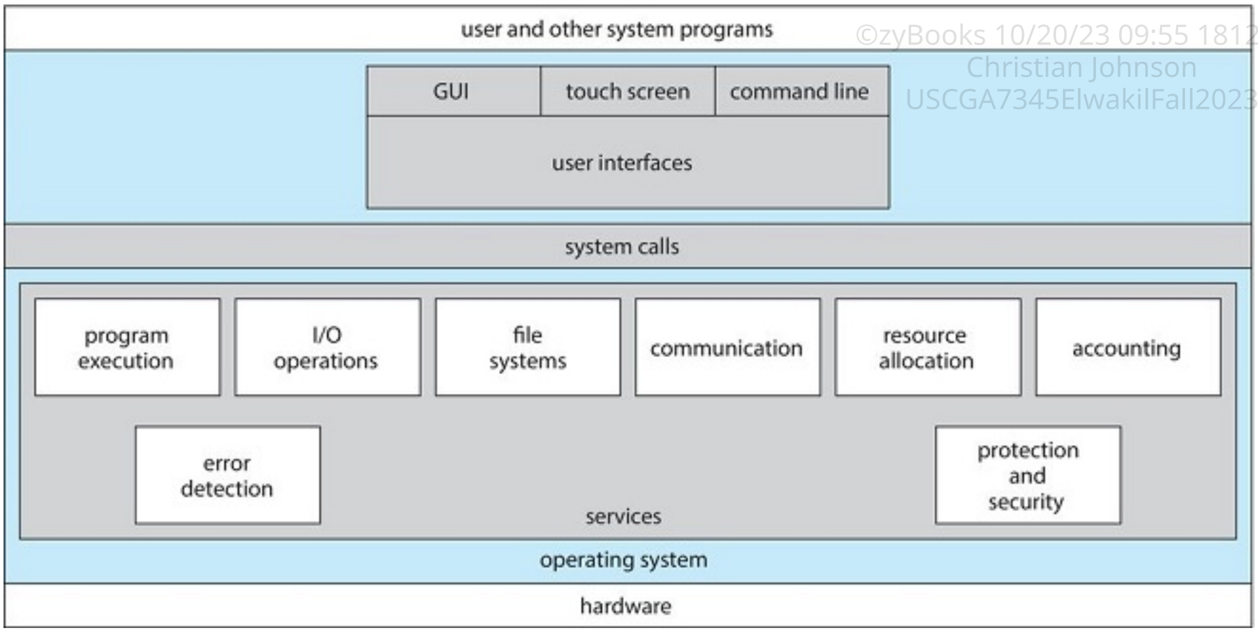
- Identify services provided by an operating system.
- Illustrate how system calls are used to provide operating system services.
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems.
- Illustrate the process for booting an operating system.
- Apply tools for monitoring operating system performance.
- Design and implement kernel modules for interacting with a Linux kernel.

Operating-system services

An operating system provides an environment for the execution of programs. It makes certain services available 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. Figure

2.1.1 shows one view of the various operating-system services and how they interrelate. Note that these services also make the programming task easier for the programmer.

Figure 2.1.1: A view of operating system services.



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. Most commonly, a ***graphical user interface*** (*GUI*) is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Mobile systems such as phones and tablets provide a ***touch-screen interface***, enabling users to slide their fingers across the screen or press buttons on the screen to select choices. Another option is a ***command-line interface*** (*CLI*), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). 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 reading from a network interface or writing to a file system). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **File-system manipulation.** The file system is of particular interest. Obviously, programs need

to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow personal choice and sometimes to provide specific features or performance characteristics.

- **Communications.** There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a network. Communications may be implemented via ***shared memory***, in which two or more processes read and write to a shared section of memory, or ***message passing***, in which packets of information in predefined formats are moved between processes by the operating system.

- **Error detection.** The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow or an attempt to access an illegal memory location). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

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

- **Resource allocation.** When there are multiple processes running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the process that must be executed, the number of processing cores on the CPU, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.
- **Logging.** We want to keep track of which programs 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 system administrators who wish to reconfigure the system to improve computing services.

- **Protection and security.** The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts and 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.

PARTICIPATION ACTIVITY 2.1.1: Section review questions.

1) Which of the following operating system services is related to ensuring the efficient operation of the system and is unrelated to providing services to user programs?

- Program execution
- Resource allocation
- Communication

Section glossary

user interface (UI): A method by which a user interacts with a computer.

graphical user interface (GUI): A computer interface comprising a window system with a pointing device to direct I/O, choose from menus, and make selections and, usually, a keyboard to enter text.

touch-screen interface: A user interface in which touching a screen allows the user to interact with the computer.

command-line interface (CLI): A method of giving commands to a computer based on a text input device (such as a keyboard).

shared memory: In interprocess communication, a section of memory shared by multiple processes and used for message passing.

message passing: In interprocess communication, a method of sharing data in which messages are sent and received by processes. Packets of information in predefined formats are moved between processes or between computers.

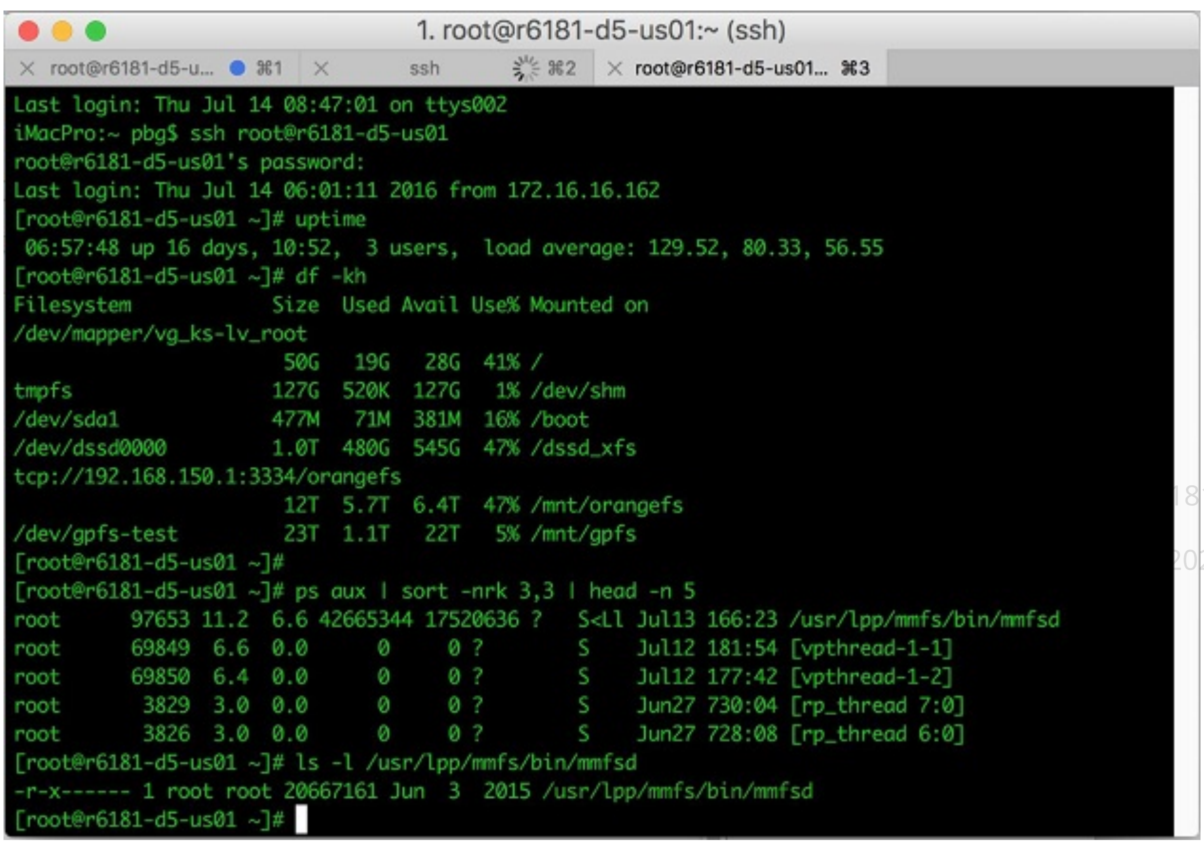
2.2 User and operating-system interface

We mentioned earlier that there are several ways for users to interface with the operating system. Here, we discuss three 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 two allow users to interface with the operating system via a graphical user interface, or GUI.

Command interpreters

Most operating systems, including Linux, UNIX, and Windows, treat the command interpreter as a special program that is running when a process 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 ***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 2.2.1 shows the Bourne-Again (or `bash`) shell command interpreter being used on macOS.

Figure 2.2.1: The bash shell command interpreter in macOS.



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 various shells available on UNIX systems 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 logic 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 program logic. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

Graphical user interface

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 has undergone various changes over the years, the most significant being the adoption of the ***Aqua*** interface that appeared with macOS. 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 significant changes in the appearance of the GUI along with several enhancements in its functionality.

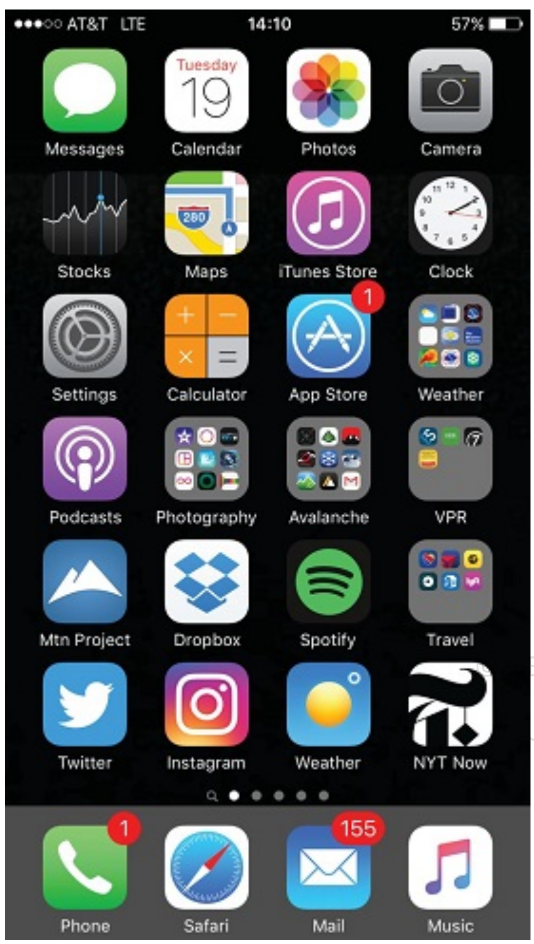
©zyBooks 10/20/23 09:55 1812110
Christian Johnson
USCGA7345ElwakilFall2023

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

Touch-screen interface

Because either a command-line interface or a mouse-and-keyboard system is impractical for most mobile systems, smartphones and handheld tablet computers typically use a touch-screen interface. Here, users interact by making ***gestures*** on the touch screen—for example, pressing and swiping fingers across the screen. Although earlier smartphones included a physical keyboard, most smartphones and tablets now simulate a keyboard on the touch screen. Figure 2.2.2 illustrates the touch screen of the Apple iPhone. Both the iPad and the iPhone use the ***Springboard*** touch-screen interface.

Figure 2.2.2: The iPhone touch screen.

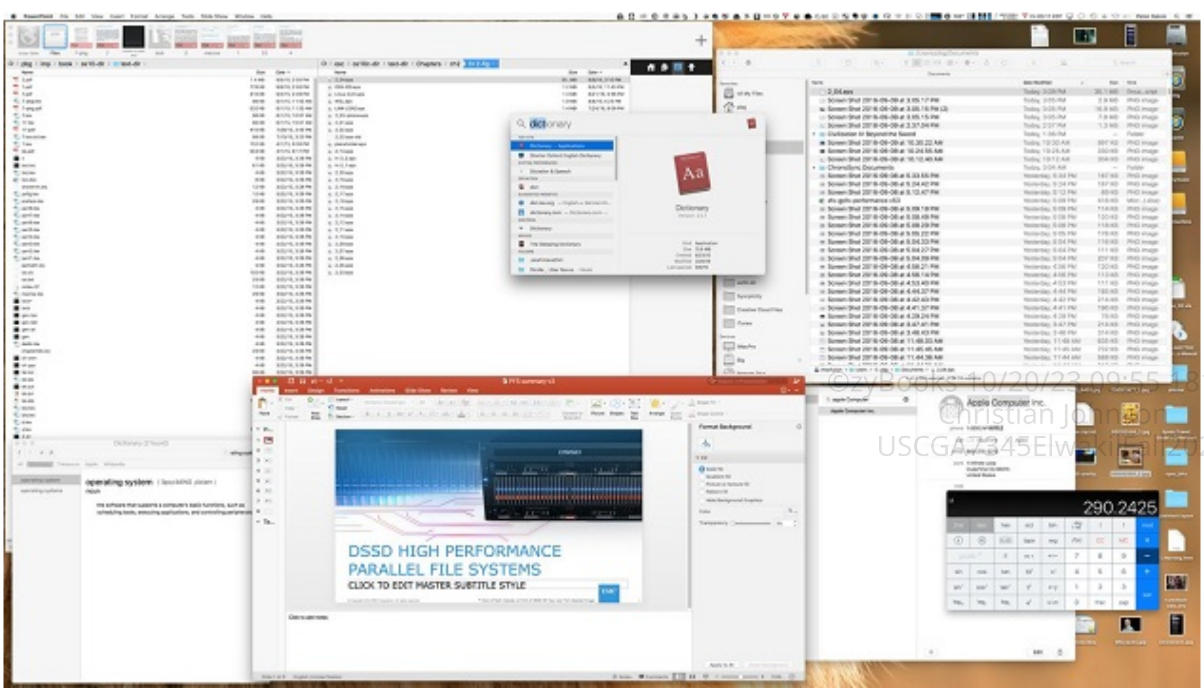


Choice of interface

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

In contrast, most Windows users are happy to use the Windows GUI environment and almost never use the shell interface. Recent versions of the Windows operating system provide both a standard GUI for desktop and traditional laptops and a touch screen for tablets. The various changes undergone by the Macintosh operating systems also 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 macOS (which is in part implemented using a UNIX kernel), the operating system now provides both an Aqua GUI and a command-line interface. Figure 2.2.3 is a screenshot of the macOS GUI.

Figure 2.2.3: The macOS GUI.



Although there are apps that provide a command-line interface for iOS and Android mobile systems, they are rarely used. Instead, almost all users of mobile systems interact with their devices using the touch-screen interface.

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

PARTICIPATION ACTIVITY 2.2.1: Section review questions.

1) A ______ is another term for command interpreter.

- shell
- shell script
- gesture

Section glossary

command interpreter: The operating system component that interprets user commands and causes actions based on them.

shell: One of the command interpreters on a system with multiple command interpreters to choose from.

desktop: In a GUI, the standard workspace represented by the GUI on the screen, in which a user executes tasks.

icons: Images representing objects (such as files or applications) that users can chose via the GUI.

folder: A file system component that allows users to group files together.

gestures: A user interface component in which motions cause computer actions (e.g., "pinching" the screen).

springboard: The iOS touch-screen interface.

system administrators: Computer users that configure, monitor, and manage systems.

power users: Users with unusually deep knowledge of a system.

shell script: A file containing a set series of commands (similar to a batch file) that are specific to the shell being used.

2.3 System calls

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

Example

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 to pass the names of the two files as part of the command—for example, the UNIX `cp` command:

```
cp in.txt out.txt
```

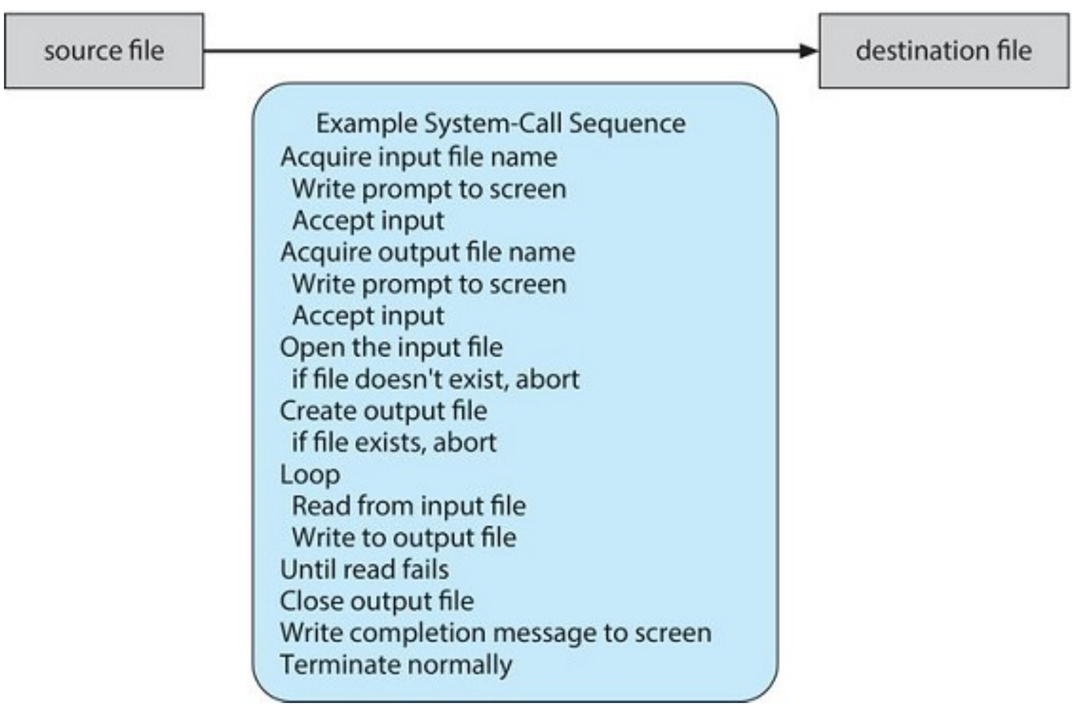
This command copies the input file `in.txt` to the output file `out.txt`. A second approach is for the program to ask the user for the names. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.

Once the two file names have been obtained, the program must open the input file and create and open the output file. Each of these operations requires another system call. Possible error conditions for each system call must be handled. For example, 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 output an error message (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

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

Finally, after the entire file is copied, the program may close both files (two system calls), 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 2.3.1.

Figure 2.3.1: Example of how system calls are used.



PARTICIPATION ACTIVITY 2.3.1: Mid-section review question.

1) ________ provide an interface to the services made available by an operating system.

Check Show answer

Application programming interface

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

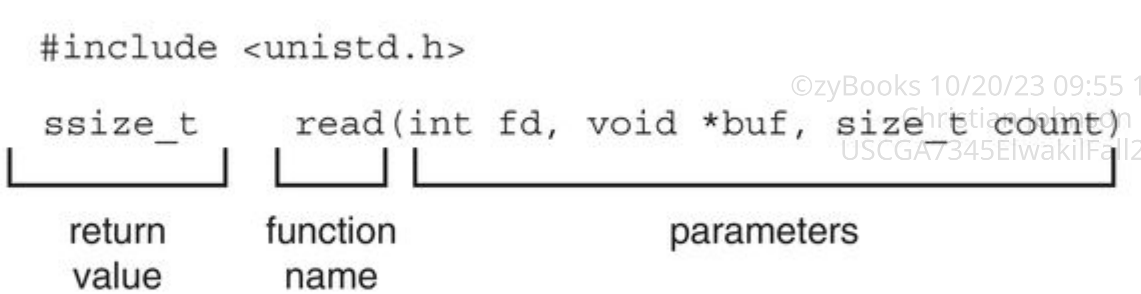
Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Windows function `CreateProcess()` (which, unsurprisingly, is used to create a new process) actually invokes the `NTCreateProcess()` system call in the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

Example of standard API

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

```
man read
```

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



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

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

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

Why would an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit concerns program portability. An application 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. Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Windows APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

Another important factor in handling system calls is the ***run-time environment*** (*RTE*)—the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders. The RTE 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.

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

PARTICIPATION ACTIVITY | 2.3.2: The Handling of a User Application Invoking the open() System Call.

Animation content:

Step 1: Application invokes the system call. Step 2: The system uses a CPU-specific method to transition from user to kernel mode and pass the system-call information to the kernel. Step 3: Each system call has a number, and that number is used as an index in the system call table, to invoke the appropriate routine. Step 4: The routine executes and returns when complete or indicating an error. Step 5: The return transitions the thread from kernel to user mode.

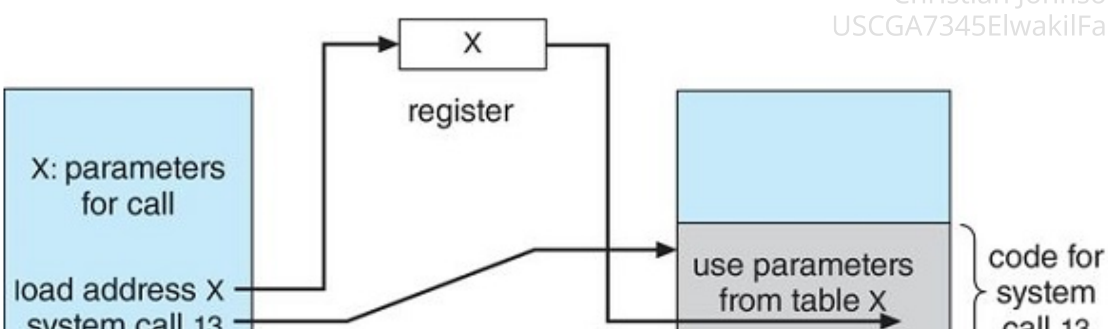
Animation captions:

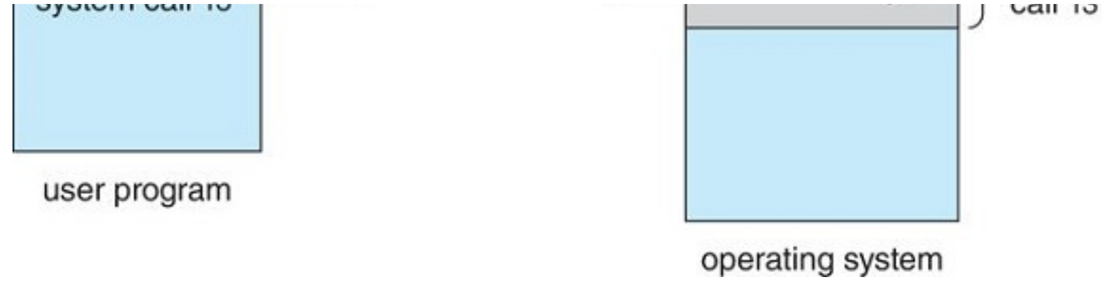
1. Application invokes the system call.
2. The system uses a CPU-specific method to transition from user to kernel mode and pass the system-call information to the kernel.
3. Each system call has a number, and that number is used as an index in the system call table, to invoke the appropriate routine.
4. The routine executes and returns when complete or indicating an error.
5. The return transitions the thread from kernel to user mode.

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 2.3.2). Linux uses a combination of these approaches. If there are five or fewer parameters, registers are used. If there are more than five parameters, the block method is used. Parameters also can be placed, or ***pushed***, onto a ***stack*** by the program and ***popped*** off the stack by the operating system. Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

Figure 2.3.2: Passing of parameters as a table.





Types of system calls

System calls can be grouped roughly into six major categories: ***process control***, ***file management***, ***device management***, ***information maintenance***, ***communications***, and ***protection***. Below, we briefly discuss the types of system calls that may be provided by an operating system. Most of these system calls support, or are supported by, concepts and functions that are discussed in later chapters. Figure 2.3.3 summarizes the types of system calls normally provided by an operating system. As mentioned, in this text, we normally refer to the system calls by generic names. Throughout the text, however, we provide examples of the actual counterparts to the system calls for UNIX, Linux, and Windows systems.

Figure 2.3.3: Types of system calls.

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - 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
- Protection
    - get file permissions
    - set file permissions

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 a special log file on disk and may be examined by a ***debugger***—a system program designed to aid the programmer in finding and correcting errors, or ***bugs***—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error. In a GUI system, a pop-up window might alert the user to the error and ask for guidance. Some systems may allow for special recovery actions in case an error occurs. If the program discovers an error in its input and wants to terminate abnormally, it may also want to define an error level. More severe errors can be indicated by a higher-level error parameter. It is then possible to combine normal and abnormal termination by defining a normal termination as an error at level 0. The command interpreter or a following program can use this error level to determine the next action automatically.

Examples of windows and unix system calls

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	`CreateProcess()` `ExitProcess()`	`fork()` `exit()`

	Windows	Unix
	`WaitForSingleObject()`	`wait()`

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

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

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

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

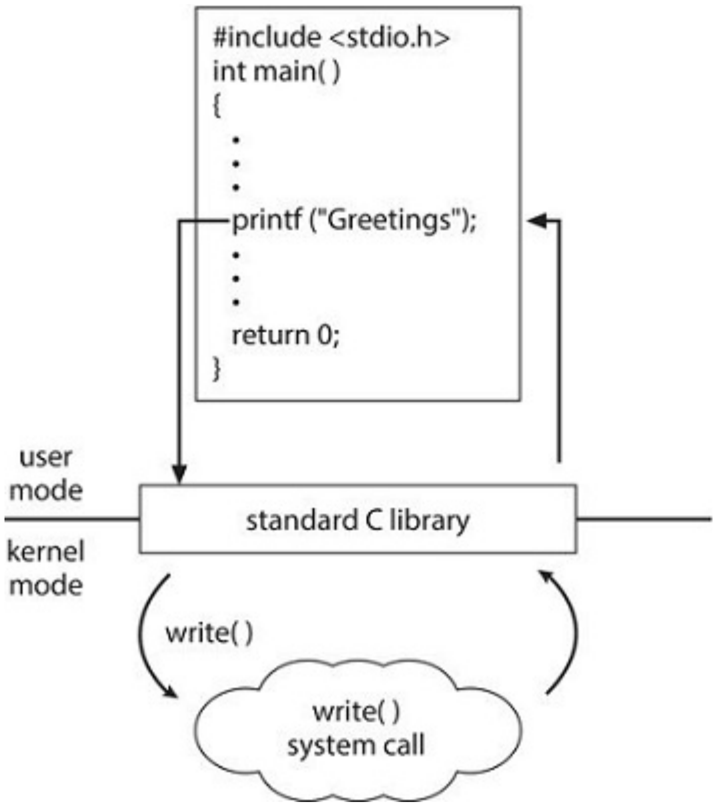
Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

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

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

The 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:



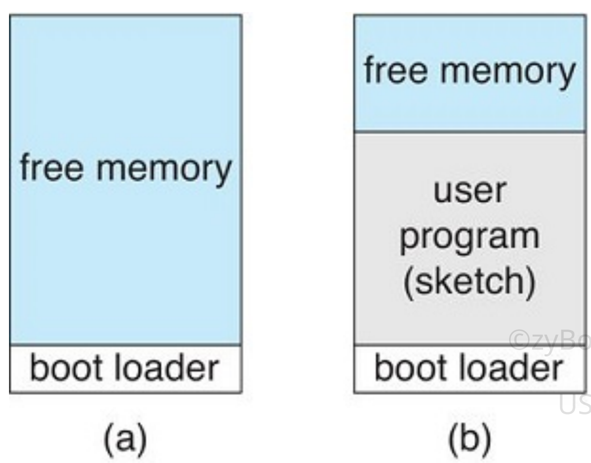
If we create a new process, or perhaps even a set of processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a process, including the process's priority, its maximum allowable execution time, and so on (`get_process_attributes()`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

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

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

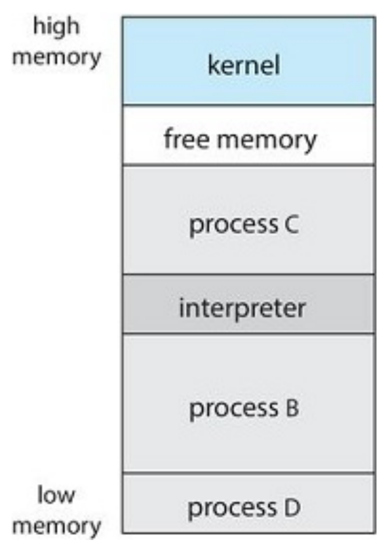
There are so many facets of and variations in process control that we next use two examples—one involving a single-tasking system and the other a multitasking system—to clarify these concepts. The Arduino is a simple hardware platform consisting of a microcontroller along with input sensors that respond to a variety of events, such as changes to light, temperature, and barometric pressure, to just name a few. To write a program for the Arduino, we first write the program on a PC and then upload the compiled program (known as a ***sketch***) from the PC to the Arduino's flash memory via a USB connection. The standard Arduino platform does not provide an operating system; instead, a small piece of software known as a ***boot loader*** loads the sketch into a specific region in the Arduino's memory (Figure 2.3.4). Once the sketch has been loaded, it begins running, waiting for the events that it is programmed to respond to. For example, if the Arduino's temperature sensor detects that the temperature has exceeded a certain threshold, the sketch may have the Arduino start the motor for a fan. An Arduino is considered a single-tasking system, as only one sketch can be present in memory at a time; if another sketch is loaded, it replaces the existing sketch. Furthermore, the Arduino provides no user interface beyond hardware input sensors.

Figure 2.3.4: Arduino execution. (a) At system startup. (b) Running a sketch.



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, awaiting commands and running programs the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.3.5). To start a new process, the shell executes a `fork()` system call. Then, the selected program is loaded into memory via an `exec()` system call, and the program is executed. Depending on how 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 waits for another command to be entered. 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 the chapter Processes with a program example using the `fork()` and `exec()` system calls.

Figure 2.3.5: FreeBSD running multiple programs.



File management

The file system is discussed in more detail in the chapter File-System Interface through chapter File-System Internals. Here, we identify several common system calls dealing with files.

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

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

Device management

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

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

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.

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 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. The program `strace`, which is available on Linux systems, 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 get and set the process information (`get_process_attributes()` and `set_process_attributes()`). In Section Process control block, we discuss what information is normally kept.

Communication

There are two common models of interprocess communication: the message-passing model and the shared-memory model. In the ***message-passing model***, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a ***host name*** by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a ***process name***, and this name is translated into an identifier by which the operating system can refer to the process. The `get_hostid()` and `get_processid()` system calls do this translation. The identifiers are then passed to the general-purpose `open()` and `close()` calls provided by the file system or to specific `open_connection()` and `close_connection()` system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an `accept_connection()` call. Most processes that will be receiving connections are special-purpose ***daemons***, which are system programs provided for that purpose. They execute a `wait_for_connection()` call and are awakened when a connection is made. The source of the communication, known as the ***client***, and the receiving daemon, known as a ***server***, then exchange messages by using `read_message()` and `write_message()` system calls. The `close_connection()` call terminates the communication.

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

—threads—in which some memory is shared by default.

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

Protection

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

Typically, system calls providing protection include `set_permission()` and `get_permission()`, which manipulate the permission settings of resources such as files and disks. The `allow_user()` and `deny_user()` system calls specify whether particular users can—or cannot—be allowed access to certain resources. We cover protection in the chapter Protection and the much larger issue of security—which involves using protection against external threats—in the chapter Security.

PARTICIPATION ACTIVITY 2.3.3: Section review questions.

1) ____ is not one of the major categories of system calls.

- Process control
- Communications
- Security

2) ____ is/are not a technique for passing parameters from an application to a system call.

- Cache memory
- Registers
- Stack

Section glossary

system call: Software-triggered interrupt allowing a process to request a kernel service.

application programming interface (API): A set of commands, functions, and other tools that can be used by a programmer in developing a program.

C library (libc): The standard UNIX/Linux system API for programs written in the C programming language.

run-time environment (RTE): The full suite of software needed to execute applications written in a given programming language, including its compilers, libraries, and loaders.

system-call interface: An interface that serves as the link to system calls made available by the operating system and that is called by processes to invoke system calls.

push: The action of placing a value on a stack data structure.

stack: A sequentially ordered data structure that uses the last-in, first-out (LIFO) principle for adding and removing items; the last item placed onto a stack is the first item removed.

process control: A category of system calls.

information maintenance: A category of system calls.

communications: A category of system calls.

protection: A category of system calls. Any mechanism for controlling the access of processes or users to the resources defined by a computer system.

debugger: A system program designed to aid programmers in finding and correcting errors.

bug: An error in computer software or hardware.

lock: A mechanism that restricts access by processes or subroutines to ensure integrity of shared data.

sketch: An Arduino program.

single step: A CPU mode in which a trap is executed by the CPU after every instruction (to allow examination of the system state after every instruction); useful in debugging.

message-passing model: A method of interprocess communication in which messages are exchanged.

host name: A human-readable name for a computer.

process name: A human-readable name for a process.

daemon: A service that is provided outside of the kernel by system programs that are loaded into memory at boot time and run continuously.

client: A computer that uses services from other computers (such as a web client). The source of a communication.

server: In general, any computer, no matter the size, that provides resources to other computers.

shared-memory model: An interprocess communication method in which multiple processes share memory and use that memory for message passing.

2.4 System services

Another aspect of a modern system is its collection of system services. ***System services***, 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, list, and generally access and manipulate files and directories.
- **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a ***registry***, which is used to store and retrieve configuration information.
- **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

- **Programming-language support.** Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided with the operating system or available as a separate download.
- **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

- **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.
- **Background services.** All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as ***services***, ***subsystems***, or daemons. One example is the network daemon discussed in Section Communication. In that example, a system needed a service to listen for network connections in order to connect those requests to the correct processes. Other examples include process schedulers that start processes according to a specified schedule, system error monitoring services, and print servers. Typical systems have dozens of daemons. In addition, operating systems that run important activities in user context rather than in kernel context may use daemons to run these activities.

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

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

PARTICIPATION ACTIVITY | 2.4.1: Section review questions.

1) Which of the following is not considered a form of a system utility?

- ○ Application programs
- ○ Programming language support
- ○ Background services

Section glossary

system service: A collection of applications included with or added to an operating system to provide services beyond those provided by the kernel.

system utility: A collection of applications included with or added to an operating system to provide services beyond what are provided by the kernel.

registry: A file, set of files, or service used to store and retrieve configuration information. In Windows, the manager of hives of data.

service: A software entity running on one or more machines and providing a particular type of function to calling clients. In Android, an application component with no user interface; it runs in the background while executing long-running operations or performing work for remote processes.

subsystem: A subset of an operating system responsible for a specific function (e.g., memory management).

application program: A program designed for end-user execution, such as a word processor, spreadsheet, compiler, or Web browser.

2.5 Linkers and loaders

Usually, a program resides on disk as a binary executable file—for example, `a.out` or `prog.exe`. To run on a CPU, the program must be brought into memory and placed in the context of a process. In this section, we describe the steps in this procedure, from compiling a program to placing it in memory, where it becomes eligible to run on an available CPU core. The steps are highlighted in the animation below.

PARTICIPATION ACTIVITY 2.5.1: The role of the linker and loader.

Animation content:

Step1: Source program main.c is read by the C compiler gcc (in general, source program is read by the appropriate compiler). Step 2: Output is object (compiled) code, which is processed by the linker which finds and merges in other object files needed by the program. Step 3: Output is an executable program, ready to run. Step 4: At run time, when program is invoked, it is loaded into memory by the loader, which also loads other code (dynamically linked libraries). The addresses in the ./main process are adjusted to call the now-loaded library functions. Process can now execute.

Animation captions:

1. Source program main.c is read by the C compiler gcc (in general, source program is read by the appropriate compiler).
2. Output is object (compiled) code, which is processed by the linker which finds and merges in other object files needed by the program.
3. Output is an executable program, ready to run.
4. At run time, when program is invoked, it is loaded into memory by the loader, which also loads other code (dynamically linked libraries). The addresses in the ./main process are adjusted to call the now-loaded library functions. Process can now execute.

Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as an ***relocatable object file***. Next, the ***linker*** combines these relocatable object files into a single binary ***executable*** file. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library (specified with the flag `-lm`).

A ***loader*** is used to load the binary executable file into memory, where it is eligible to run on a CPU core. An activity associated with linking and loading is ***relocation***, which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses so that, for example, the code can call library functions and access its variables as it executes. In the animation above, we see that to run the loader, all that is necessary is to enter the name of the executable file on the command line. When a program name is entered on the command line on UNIX systems—for example, `./main`—the shell first creates a new process to run the program using the `fork()` system call. The shell then invokes the loader with the `exec()` system call, passing `exec()` the name of the executable file. The loader then loads the specified program into memory using the address space of the newly created process. (When a GUI interface is used, double-clicking on the icon associated with the executable file invokes the loader using a similar mechanism.)

The process described thus far assumes that all libraries are linked into the executable file and loaded into memory. In reality, most systems allow a program to dynamically link libraries as the program is loaded. Windows, for instance, supports dynamically linked libraries (***DLLs***). The benefit of this approach is that it avoids linking and loading libraries that may end up not being used into an executable file. Instead, the library is conditionally linked and is loaded if it is required during program run time. For example, in the animation above, the math library is not linked into the executable file `main`. Rather, the linker inserts relocation information that allows it to be dynamically linked and loaded as the program is loaded. We shall see in the chapter Main Memory that it is possible for multiple processes to share dynamically linked libraries, resulting in a significant savings in memory use.

Object files and executable files typically have standard formats that include the compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the program. For UNIX and Linux systems, this standard format is known as ELF (for ***Executable and Linkable Format***). There are separate ELF formats for relocatable and executable files. One piece of information in the ELF file for executable files is the program's ***entry point***, which contains the address of the first instruction to be executed when the program runs. Windows systems use the ***Portable Executable*** (PE) format, and macOS uses the ***Mach-O*** format.

ELF format

Linux provides various commands to identify and evaluate ELF files. For example, the `file` command determines a file type. If `main.o` is an object file, and `main` is an executable file, the command

```
file main.o
```

will report that `main.o` is an ELF relocatable file, while the command

```
file main
```

will report that `main` is an ELF executable. ELF files are divided into a number of sections and can be evaluated using the `readelf` command.

PARTICIPATION ACTIVITY | 2.5.2: Section review questions.

1) _______ combines relocatable object files into a single executable file.

- A loader
- A linker
- Relocation

Section glossary

relocatable object file: The output of a compiler in which the contents can be loaded into any location in physical memory.

linker: A system service that combines relocatable object files into a single binary executable file.

executable file: A file containing a program that is ready to be loaded into memory and executed.

loader: A system service that loads a binary executable file into memory, where it is eligible to run on a CPU core.

relocation: An activity associated with linking and loading that assigns final addresses to program parts and adjusts code and data in the program to match those addresses.

dynamically linked libraries (DLLs): System libraries that are linked to user programs when the processes are run, with linking postponed until execution time.

executable and linkable format (ELF): The UNIX standard format for relocatable and executable files.

portable executable (PE): The Windows format for executable files.

Mach-O: The macOS format of executable files.

2.6 Why applications are operating-system specific

Fundamentally, applications compiled on one operating system are not executable on other operating systems. If they were, the world would be a better place, and our choice of what operating system to use would depend on utility and features rather than which applications were available.

Based on our earlier discussion, we can now see part of the problem—each operating system provides a unique set of system calls. System calls are part of the set of services provided by operating systems for use by applications. Even if system calls were somehow uniform, other barriers would make it difficult for us to execute application programs on different operating systems. But if you have used multiple operating systems, you may have used some of the same applications on them. How is that possible?

An application can be made available to run on multiple operating systems in one of three ways:

1. The application can be written in an interpreted language (such as Python or Ruby) that has an interpreter available for multiple operating systems. The interpreter reads each line of the source program, executes equivalent instructions on the native instruction set, and calls native operating system calls. Performance suffers relative to that for native applications, and

the interpreter provides only a subset of each operating system's features, possibly limiting the feature sets of the associated applications.

2. The application can be written in a language that includes a virtual machine containing the running application. The virtual machine is part of the language's full RTE. One example of this method is Java. Java has an RTE that includes a loader, byte-code verifier, and other components that load the Java application into the Java virtual machine. This RTE has been ***ported***, or developed, for many operating systems, from mainframes to smartphones, and in theory any Java app can run within the RTE wherever it is available. Systems of this kind have disadvantages similar to those of interpreters, discussed above.
3. The application developer can use a standard language or API in which the compiler generates binaries in a machine- and operating-system-specific language. The application must be ported to each operating system on which it will run. This porting can be quite time consuming and must be done for each new version of the application, with subsequent testing and debugging. Perhaps the best-known example is the POSIX API and its set of standards for maintaining source-code compatibility between different variants of UNIX-like operating systems.

In theory, these three approaches seemingly provide simple solutions for developing applications that can run across different operating systems. However, the general lack of application mobility has several causes, all of which still make developing cross-platform applications a challenging task. At the application level, the libraries provided with the operating system contain APIs to provide features like GUI interfaces, and an application designed to call one set of APIs (say, those available from IOS on the Apple iPhone) will not work on an operating system that does not provide those APIs (such as Android). Other challenges exist at lower levels in the system, including the following.

- Each operating system has a binary format for applications that dictates the layout of the header, instructions, and variables. Those components need to be at certain locations in specified structures within an executable file so the operating system can open the file and load the application for proper execution.
- CPUs have varying instruction sets, and only applications containing the appropriate instructions can execute correctly.
- Operating systems provide system calls that allow applications to request various activities, such as creating files and opening network connections. Those system calls vary among operating systems in many respects, including the specific operands and operand ordering used, how an application invokes the system calls, their numbering and number, their meanings, and their return of results.

There are some approaches that have helped address, though not completely solve, these architectural differences. For example, Linux—and almost every UNIX system—has adopted the ELF format for binary executable files. Although ELF provides a common standard across Linux and

UNIX systems, the ELF format is not tied to any specific computer architecture, so it does not guarantee that an executable file will run across different hardware platforms.

APIs, as mentioned above, specify certain functions at the application level. At the architecture level, an ***application binary interface*** (ABI) is used to define how different components of binary code can interface for a given operating system on a given architecture. An ABI specifies low-level details, including address width, methods of passing parameters to system calls, the organization of the run-time stack, the binary format of system libraries, and the size of data types, just to name a few. Typically, an ABI is specified for a given architecture (for example, there is an ABI for the ARM v8 processor). Thus, an ABI is the architecture-level equivalent of an API. If a binary executable file has been compiled and linked according to a particular ABI, it should be able to run on different systems that support that ABI. However, because a particular ABI is defined for a certain operating system running on a given architecture, ABIs do little to provide cross-platform compatibility.

In sum, all of these differences mean that unless an interpreter, RTE, or binary executable file is written for and compiled on a specific operating system on a specific CPU type (such as Intel x86 or ARM v8), the application will fail to run. Imagine the amount of work that is required for a program such as the Firefox browser to run on Windows, macOS, various Linux releases, iOS, and Android, sometimes on various CPU architectures.

PARTICIPATION ACTIVITY | 2.6.1: Section review questions.

1) Which of the following is not a technique that allows a program to be run by different operating systems?

- The program is written in an interpreted language such as Python.
- The program is written in a language that includes a virtual machine as part of the application's runtime such as Java.
- The program has been compiled into a binary executable file.

Section glossary

port: A communication address; a system may have one IP address for network connections but many ports, each for a separate communication. In computer I/O, a connection point for devices to attach to computers. In software development, to move code from its current platform to another platform (e.g., between operating systems or hardware systems). In the Mach OS, a mailbox for communication.

2.7 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.

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: traditional desktop/laptop, mobile, distributed, or real time.

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 the developers 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 Wind River VxWorks, a real-time operating system for embedded systems, must have been substantially different from those for Windows Server, a large multiaccess operating system designed for enterprise applications.

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.

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 Timer) 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 flexible enough to work across a range of policies is preferable. 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 (discussed in Section Microkernels) 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 user programs themselves. In contrast, consider Windows, an enormously popular commercial operating system available for over three decades. Microsoft has closely encoded both mechanism and policy into the system to enforce a global look and feel across all devices that run the Windows operating system. All applications have similar interfaces, because the interface itself is built into the kernel and system libraries. Apple has adopted a similar strategy with its macOS and iOS operating systems.

We can make a similar comparison between commercial and open-source operating systems. For instance, contrast Windows, discussed above, with Linux, an open-source operating system that runs on a wide range of computing devices and has been available for over 25 years. The "standard" Linux kernel has a specific CPU scheduling algorithm (covered in Section Example: Linux scheduling), which is a mechanism that supports a certain policy. However, anyone is free to modify or replace the scheduler to support a different policy.

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.

Implementation

Once an operating system is designed, it must be implemented. Because operating systems are collections of many programs, written by many people over a long period of time, it is difficult to make general statements about how they are implemented.

Early operating systems were written in assembly language. Now, most are written in higher-level languages such as C or C++, with small amounts of the system written in assembly language. In fact, more than one higher-level language is often used. The lowest levels of the kernel might be written in assembly language and C. Higher-level routines might be written in C and C++, and system libraries might be written in C++ or even higher-level languages. Android provides a nice example: its kernel is written mostly in C with some assembly language. Most Android system libraries are written in C or C++, and its application frameworks—which provide the developer interface to the system—are written mostly in Java. We cover Android's architecture in more detail in Section Android.

The advantages of using a higher-level language, or at least a systems-implementation language, for implementing operating systems are the same as those gained 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 other hardware if it is written in a higher-level language. This is particularly important for operating systems that are intended to run on several different hardware systems, such as small embedded devices, Intel x86 systems, and ARM chips running on phones and tablets.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. This, however, is not 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 interrupt handlers, I/O manager, memory manager, and CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottlenecks can be identified and can be refactored to operate more efficiently.

PARTICIPATION ACTIVITY 2.7.1: Section review questions.

1) Policy ____.

- ○ determines how to do something
- ○ determines what will be done
- ○ is not likely to change over time

Section glossary

software engineering: A field of study and a career involving writing software (i.e., programming.)

policy: A rule that defines what will be done.

mechanism: An operation that defines how something will be done.

2.8 Operating-system structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one single system. Each of these modules should be a well-defined portion of the system, with carefully defined interfaces and functions. You may use a similar approach when you structure your programs: rather than placing all of your code in the `main()` function, you instead separate logic into a number of functions, clearly articulate parameters and return values, and then call those functions from `main()`.

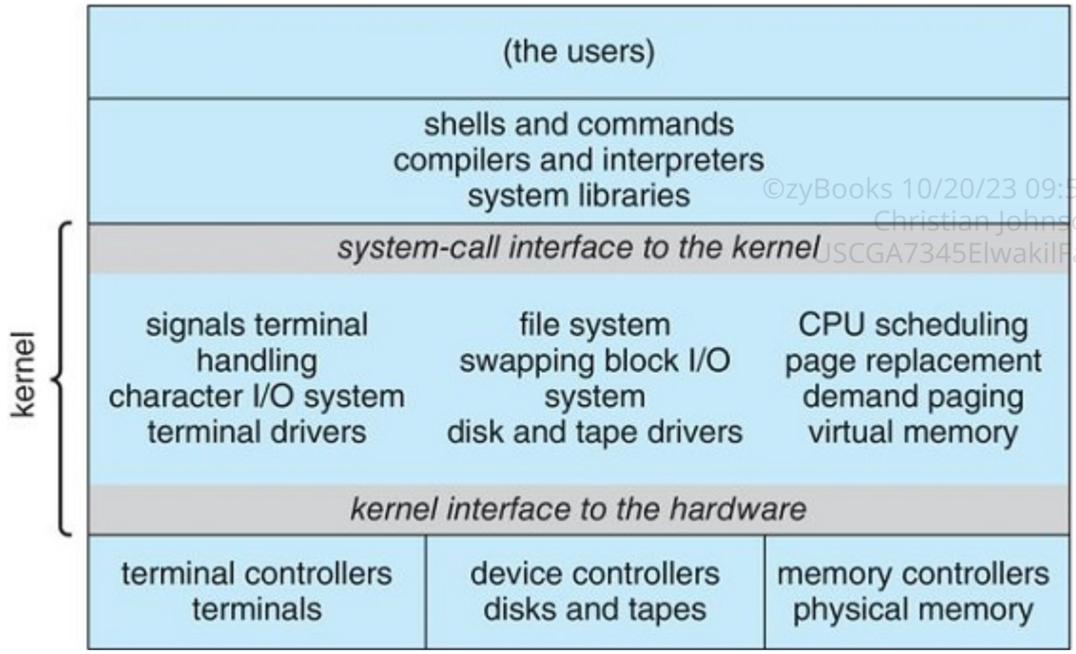
We briefly discussed the common components of operating systems in the chapter Introduction. In this section, we discuss how these components are interconnected and melded into a kernel.

Monolithic structure

The simplest structure for organizing an operating system is no structure at all. That is, place all of the functionality of the kernel into a single, static binary file that runs in a single address space. This approach—known as a ***monolithic*** structure—is a common technique for designing operating systems.

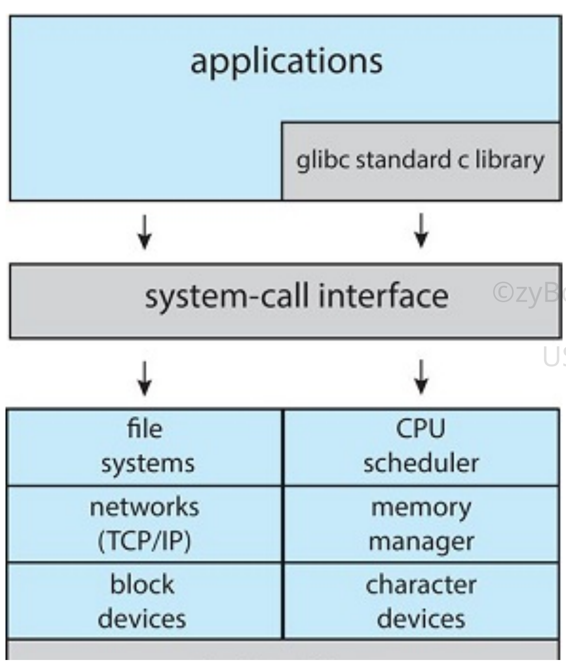
An example of such limited structuring is the original UNIX operating system, which consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.8.1. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one single address space.

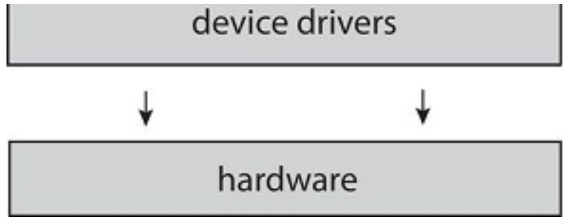
Figure 2.8.1: Traditional UNIX system structure.



The Linux operating system is based on UNIX and is structured similarly, as shown in Figure 2.8.2. Applications typically use the `glibc` standard C library when communicating with the system call interface to the kernel. The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space, but as we shall see in Section Modules, it does have a modular design that allows the kernel to be modified during run time.

Figure 2.8.2: Linux system structure.





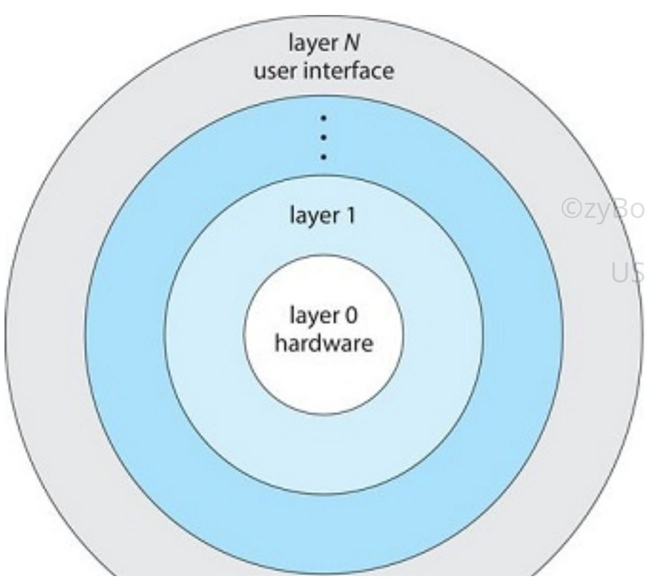
Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend. Monolithic kernels do have a distinct performance advantage, however: there is very little overhead in the system-call interface, and communication within the kernel is fast. Therefore, despite the drawbacks of monolithic kernels, their speed and efficiency explains why we still see evidence of this structure in the UNIX, Linux, and Windows operating systems.

Layered approach

The monolithic approach is often known as a ***tightly coupled*** system because changes to one part of the system can have wide-ranging effects on other parts. Alternatively, we could design a ***loosely coupled*** system. Such a system is divided into separate, smaller components that have specific and limited functionality. All these components together comprise the kernel. The advantage of this modular approach is that changes in one component affect only that component, and no others, allowing system implementers more freedom in creating and changing the inner workings of the system.

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 2.8.3.

Figure 2.8.3: A layered operating system.



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

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

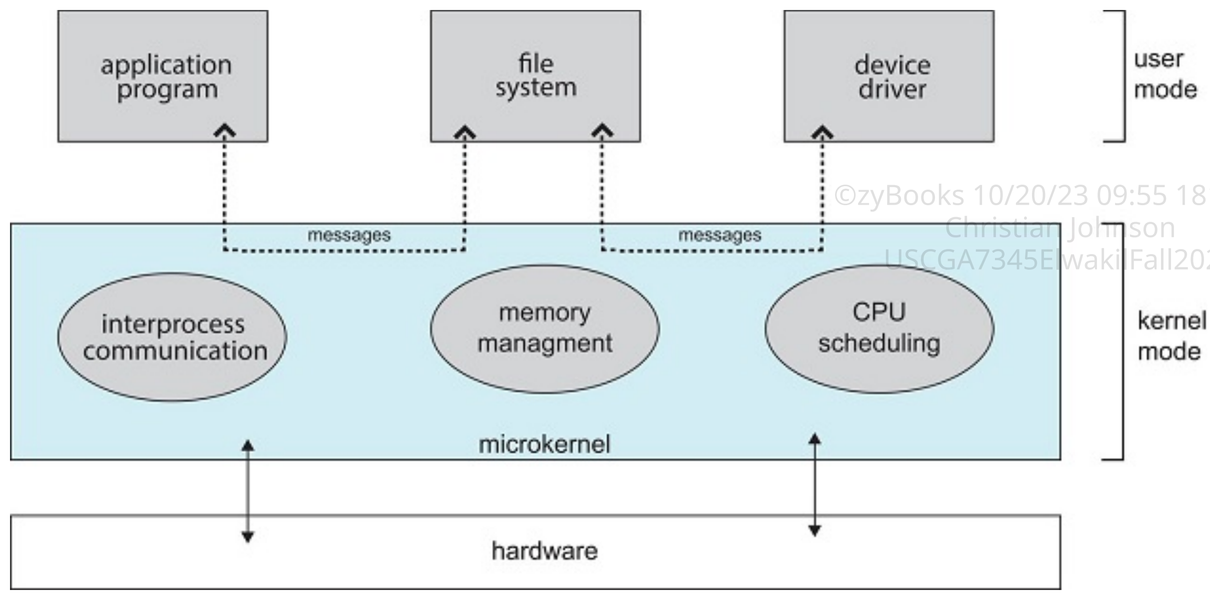
Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level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

Layered systems have been successfully used in computer networks (such as TCP/IP) and web applications. Nevertheless, relatively few operating systems use a pure layered approach. One reason involves the challenges of appropriately defining the functionality of each layer. In addition, the overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service. ***Some*** layering is common in contemporary operating systems, however. Generally, these systems have fewer layers with more functionality, providing most of the advantages of modularized code while avoiding the problems of layer definition and interaction.

Microkernels

We have already seen that the original UNIX system had a monolithic structure. 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 user-level programs that reside in separate address spaces. The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, microkernels provide minimal process and memory management, in addition to a communication facility. Figure 2.8.4 illustrates the architecture of a typical microkernel.

Figure 2.8.4: Architecture of a typical microkernel.



The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through message passing, which was described in Section Communication. 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 that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

Perhaps the best-known illustration of a microkernel operating system is ***Darwin***, the kernel component of the macOS and iOS operating systems. Darwin, in fact, consists of two kernels, one of which is the Mach microkernel. We will cover the macOS and iOS systems in further detail in Section macOS and iOS.

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

Unfortunately, the performance of microkernels can suffer due to increased system-function overhead. When two user-level services must communicate, messages must be copied between the services, which reside in separate address spaces. In addition, the operating system may have to switch from one process to the next to exchange the messages. The overhead involved in copying messages and switching between processes has been the largest impediment to the growth of microkernel-based operating systems. Consider the history of Windows NT: The first release had a layered microkernel organization. This version's performance was low compared with that of Windows 95. Windows NT 4.0 partially corrected the performance problem by moving layers from user space to kernel space and integrating them more closely. By the time Windows XP was designed, Windows architecture had become more monolithic than microkernel. Section macOS and iOS will describe how macOS addresses the performance issues of the Mach microkernel.

Modules

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

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

The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module. The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

Linux uses loadable kernel modules, primarily for supporting device drivers and file systems. LKMs can be "inserted" into the kernel as the system is started (or ***booted***) or during run time, such as when a USB device is plugged into a running machine. If the Linux kernel does not have the necessary driver, it can be dynamically loaded. LKMs can be removed from the kernel during run time as well. For Linux, LKMs allow a dynamic and modular kernel, while maintaining the performance benefits of a monolithic system. We cover creating LKMs in Linux in several programming exercises at the end of this chapter.

Hybrid systems

In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, Linux is monolithic, because having the operating system in a single address space provides very efficient performance. However, it also modular, so that new functionality can be dynamically added to the kernel. Windows is largely monolithic as well (again primarily for performance reasons), but it retains some behavior typical of microkernel systems, including providing support for separate subsystems (known as operating-system ***personalities***) that run as user-mode processes. Windows systems also provide support for dynamically loadable kernel modules. We provide case studies of Linux and Windows 10 in the chapter The Linux System and chapter Windows 10, respectively. In the remainder of this section, we explore the structure of three hybrid systems: the Apple macOS operating system and the two most prominent mobile operating systems—iOS and Android.

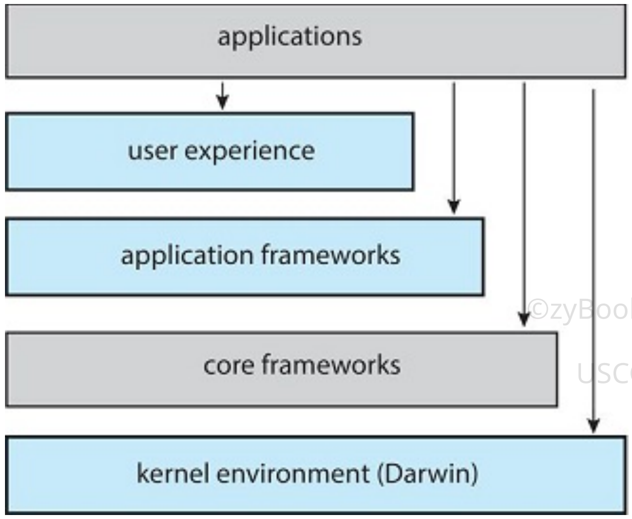
macOS and iOS

Apple's macOS operating system is designed to run primarily on desktop and laptop computer systems, whereas iOS is a mobile operating system designed for the iPhone smartphone and iPad tablet computer. Architecturally, macOS and iOS have much in common, and so we present them together, highlighting what they share as well as how they differ from each other. The general architecture of these two systems is shown in Figure 2.8.5. Highlights of the various layers include the following:

- ***User experience layer***. This layer defines the software interface that allows users to interact with the computing devices. macOS uses the ***Aqua*** user interface, which is designed for a mouse or trackpad, whereas iOS uses the ***Springboard*** user interface, which is designed for touch devices.
- ***Application frameworks layer***. This layer includes the ***Cocoa*** and ***Cocoa Touch*** frameworks, which provide an API for the Objective-C and Swift programming languages. The primary difference between Cocoa and Cocoa Touch is that the former is used for developing macOS applications, and the latter by iOS to provide support for hardware features unique to mobile devices, such as touch screens.
- ***Core frameworks***. This layer defines frameworks that support graphics and media including, Quicktime and OpenGL.
- ***Kernel environment***. This environment, also known as ***Darwin***, includes the Mach microkernel and the BSD UNIX kernel. We will elaborate on Darwin shortly.

As shown in Figure 2.8.5, applications can be designed to take advantage of user-experience features or to bypass them and interact directly with either the application framework or the core framework. Additionally, an application can forego frameworks entirely and communicate directly with the kernel environment. (An example of this latter situation is a C program written with no user interface that makes POSIX system calls.)

Figure 2.8.5: Architecture of Apple's macOS and iOS operating systems.

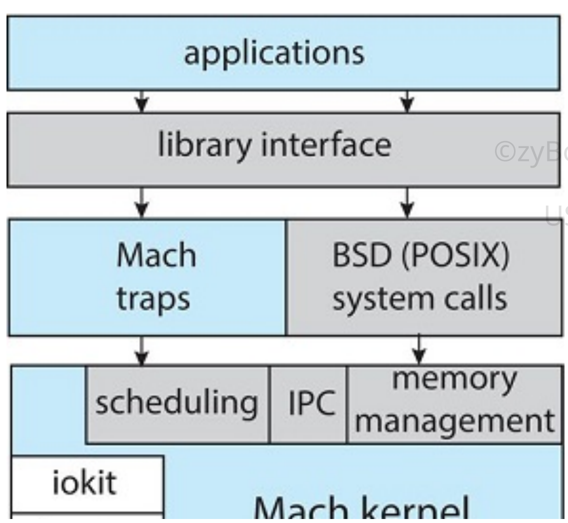


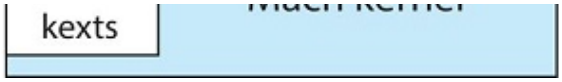
Some significant distinctions between macOS and iOS include the following:

- Because macOS is intended for desktop and laptop computer systems, it is compiled to run on Intel architectures. iOS is designed for mobile devices and thus is compiled for ARM-based architectures. Similarly, the iOS kernel has been modified somewhat to address specific features and needs of mobile systems, such as power management and aggressive memory management. Additionally, iOS has more stringent security settings than macOS.
- The iOS operating system is generally much more restricted to developers than macOS and may even be closed to developers. For example, iOS restricts access to POSIX and BSD APIs on iOS, whereas they are openly available to developers on macOS.

We now focus on Darwin, which uses a hybrid structure. Darwin is a layered system that consists primarily of the Mach microkernel and the BSD UNIX kernel. Darwin's structure is shown in Figure 2.8.6.

Figure 2.8.6: The structure of Darwin.





Whereas most operating systems provide a single system-call interface to the kernel—such as through the standard C library on UNIX and Linux systems—Darwin provides ***two*** system-call interfaces: Mach system calls (known as ***traps***) and BSD system calls (which provide POSIX functionality). The interface to these system calls is a rich set of libraries that includes not only the standard C library but also libraries that provide networking, security, and progamming language support (to name just a few).

Beneath the system-call interface, Mach provides fundamental operating-system services, including memory management, CPU scheduling, and interprocess communication (IPC) facilities such as message passing and remote procedure calls (RPCs). Much of the functionality provided by Mach is available through ***kernel abstractions***, which include tasks (a Mach process), threads, memory objects, and ports (used for IPC). As an example, an application may create a new process using the BSD POSIX `fork()` system call. Mach will, in turn, use a task kernel abstraction to represent the process in the kernel.

In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which macOS refers to as ***kernel extensions***, or ***kexts***).

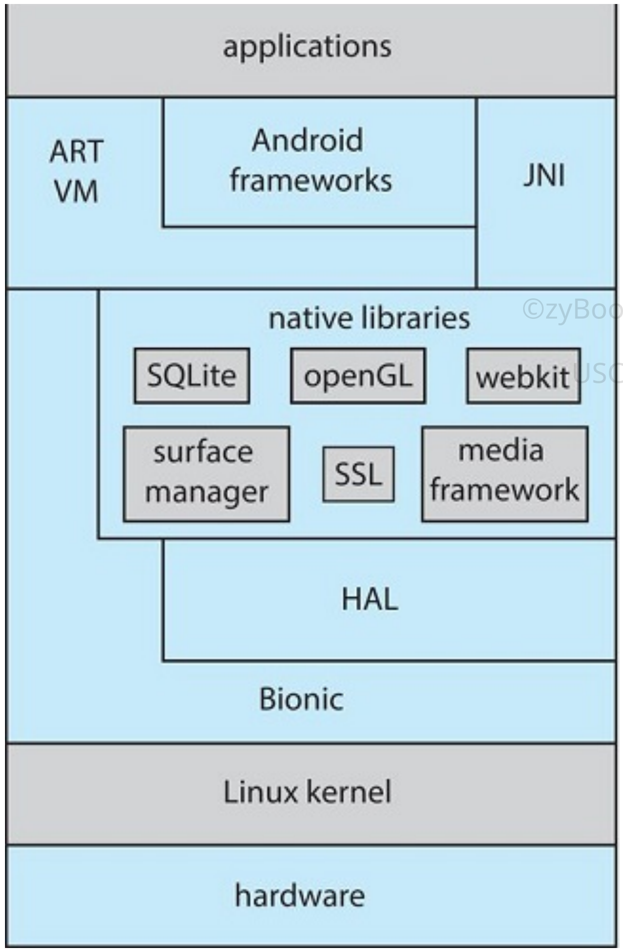
In Section Microkernels, we described how the overhead of message passing between different services running in user space compromises the performance of microkernels. To address such performance problems, Darwin combines Mach, BSD, the I/O kit, and any kernel extensions into a single address space. Thus, Mach is not a pure microkernel in the sense that various subsystems run in user space. Message passing within Mach still does occur, but no copying is necessary, as the services have access to the same address space.

Apple has released the Darwin operating system as open source. As a result, various projects have added extra functionality to Darwin, such as the X-11 windowing system and support for additional file systems. Unlike Darwin, however, the Cocoa interface, as well as other proprietary Apple frameworks available for developing macOS applications, are closed.

Android

The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers. Whereas iOS is designed to run on Apple mobile devices and is close-sourced, Android runs on a variety of mobile platforms and is open-sourced, partly explaining its rapid rise in popularity. The structure of Android appears in Figure 2.8.7.

Figure 2.8.7: Architecture of Google's Android.



Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks supporting graphics, audio, and hardware features. These features, in turn, provide a platform for developing mobile applications that run on a multitude of Android-enabled devices.

Software designers for Android devices develop applications in the Java language, but they do not generally use the standard Java API. Google has designed a separate Android API for Java development. Java applications are compiled into a form that can execute on the Android RunTime ART, a virtual machine designed for Android and optimized for mobile devices with limited memory and CPU processing capabilities. Java programs are first compiled to a Java bytecode `.class` file and then translated into an executable `.dex` file. Whereas many Java virtual machines perform just-in-time (JIT) compilation to improve application efficiency, ART performs ***ahead-of-time*** (*AOT*) compilation. Here, `.dex` files are compiled into native machine code when they are installed on a device, from which they can execute on the ART. AOT compilation allows more efficient application execution as well as reduced power consumption, features that are crucial for mobile systems.

Android developers can also write Java programs that use the Java native interface—or JNI—which allows developers to bypass the virtual machine and instead write Java programs that can access specific hardware features. Programs written using JNI are generally not portable from one hardware device to another.

The set of native libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and network support, such as secure sockets (SSLs).

Because Android can run on an almost unlimited number of hardware devices, Google has chosen to abstract the physical hardware through the hardware abstraction layer, or HAL. By abstracting all hardware, such as the camera, GPS chip, and other sensors, the HAL provides applications with a consistent view independent of specific hardware. This feature, of course, allows developers to write programs that are portable across different hardware platforms.

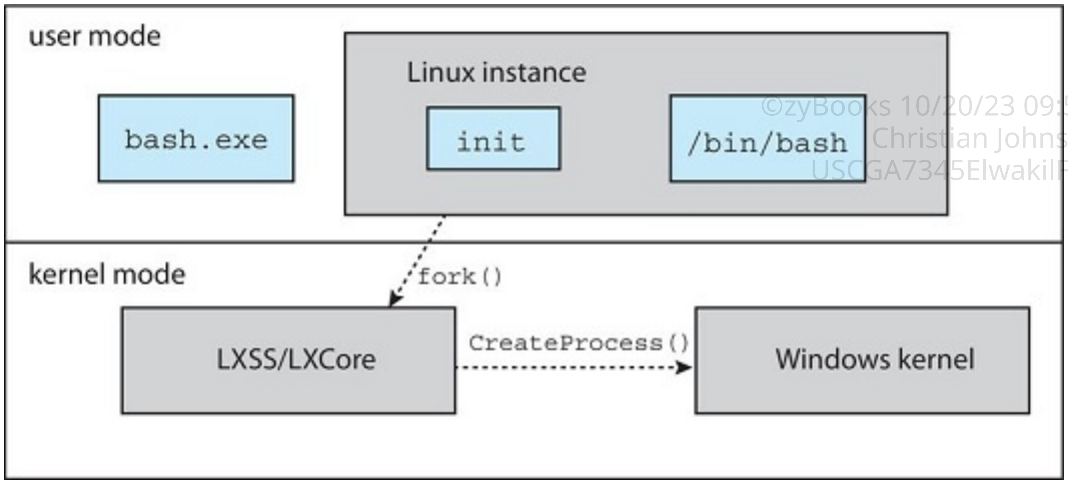
The standard C library used by Linux systems is the GNU C library (`glibc`). Google instead developed the ***Bionic*** standard C library for Android. Not only does Bionic have a smaller memory footprint than `glibc`, but it also has been designed for the slower CPUs that characterize mobile devices. (In addition, Bionic allows Google to bypass GPL licensing of `glibc`.)

At the bottom of Android's software stack is the Linux kernel. Google has modified the Linux kernel used in Android in a variety of areas to support the special needs of mobile systems, such as power management. It has also made changes in memory management and allocation and has added a new form of IPC known as ***Binder*** (which we will cover in Section Android RPC).

Windows subsystem for linux

Windows uses a hybrid architecture that provides subsystems to emulate different operating-system environments. These user-mode subsystems communicate with the Windows kernel to provide actual services. Windows 10 adds a Windows subsystem for Linux (***WSL***), which allows native Linux applications (specified as ELF binaries) to run on Windows 10. The typical operation is for a user to start the Windows application `bash.exe`, which presents the user with a `bash` shell running Linux. Internally, the WSL creates a ***Linux instance*** consisting of the `init` process, which in turn creates the `bash` shell running the native Linux application `/bin/bash`. Each of these processes runs in a Windows ***Pico*** process. This special process loads the native Linux binary into the process's own address space, thus providing an environment in which a Linux application can execute.

Pico processes communicate with the kernel services LXCore and LXSS to translate Linux system calls, if possible using native Windows system calls. When the Linux application makes a system call that has no Windows equivalent, the LXSS service must provide the equivalent functionality. When there is a one-to-one relationship between the Linux and Windows system calls, LXSS forwards the Linux system call directly to the equivalent call in the Windows kernel. In some situations, Linux and Windows have system calls that are similar but not identical. When this occurs, LXSS will provide some of the functionality and will invoke the similar Windows system call to provide the remainder of the functionality. The Linux `fork()` provides an illustration of this: The Windows `CreateProcess()` system call is similar to `fork()` but does not provide exactly the same functionality. When `fork()` is invoked in WSL, the LXSS service does some of the initial work of `fork()` and then calls `CreateProcess()` to do the remainder of the work. The figure below illustrates the basic behavior of WSL.



PARTICIPATION ACTIVITY 2.8.1: Section review questions.

1) A monolithic kernel ________ .

- provides most services as separate user-level programs that run outside of the kernel
- has very-limited or no structure at all
- is loosely-coupled

2) The major difficulty in designing a layered operating system approach is ___.

- appropriately defining the various layers
- making sure that each layer hides certain data structures, hardware, and operations from higher-level layers
- debugging a particular layer

3) A microkernel is a kernel ___.

- containing many components that are optimized to reduce resident memory size
- that is compiled to produce the smallest size possible when loading the operating system into main memory
- that is stripped of all nonessential components

Section glossary

monolithic: In kernel architecture, describes a kernel without structure (such as layers or modules).

tightly coupled systems: Systems with two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

loosely coupled: Describes a kernel design in which the kernel is composed of components that have specific and limited functions.

layered approach: A kernel architecture in which the operating system is separated into a number of layers (levels); typically, the bottom layer (layer 0) is the hardware, and the highest (layer N) is the user interface.

Mach: An operating system with microkernel structure and threading; developed at Carnegie Mellon University.

microkernel: An operating-system structure that removes all nonessential components from the kernel and implements them as system and user-level programs.

loadable kernel module (LKM): A kernel structure in which the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time.

user experience layer: in the layered macOS and iOS operating system design, the layer that defines the software interface that allows users to interact with computing devices.

application frameworks layer: in the layered macOS and iOS operating system design, the layer that includes Cocoa and Cocoa Touch frameworks, providing an API for Objective-C and Swift programming languages.

core frameworks: in the layered macOS and iOS operating system design, the layer that defines frameworks that support graphics and media, including quicktime and opengl.

kernel environment: in the layered macOS and iOS operating system design, the darwin layer that includes the mach microkernel and the bsd unix kernel.

Darwin: The Apple code name for its open-source kernel.

trap: A software interrupt. The interrupt can be caused either by an error (e.g., division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.

kernel abstractions: Components provided with the Mach microkernel to add functionality beyond the microkernel, such as tasks, threads, memory objects, and ports.

kernel extensions (kexts): Third-party components added to the kernel (usually to support third-party devices or services).

kexts: Third-party components added to the kernel (usually to support third-party devices or services).

ahead-of-time (AOT) compilation: A feature of the Android RunTime (ART) virtual machine environment in which Java applications are compiled to native machine code when they are installed on a system (rather than just in time, when they are executed).

Bionic: A type of standard C library used by Android; it has a smaller memory footprint than glibc and is more efficient on slower (mobile) CPUs.

Windows Subsystem for Linux (WSL): A Windows 10 component allowing native Linux applications (ELF binaries) to run on Windows.

Linux instance: A set of Pico processes running in Windows created by WSL containing an init process and a bash shell (allowing executing of Linux commands within Windows)

Pico: In WSL, a special Linux-enabling process that translates Linux system calls to the LXCore and LXSS services.

2.9 Building and booting an operating system

It is possible to design, code, and implement an operating system specifically for one specific machine configuration. More commonly, however, operating systems are designed to run on any of a class of machines with a variety of peripheral configurations.

Operating-system generation

Most commonly, a computer system, when purchased, has an operating system already installed. For example, you may purchase a new laptop with Windows or macOS preinstalled. But suppose you wish to replace the preinstalled operating system or add additional operating systems. Or suppose you purchase a computer without an operating system. In these latter situations, you have a few options for placing the appropriate operating system on the computer and configuring it for use.

If you are generating (or building) an operating system from scratch, you must follow these steps:

1. Write the operating system source code (or obtain previously written source code).
2. Configure the operating system for the system on which it will run.
3. Compile the operating system.
4. Install the operating system.
5. Boot the computer and its new operating system.

Configuring the system involves specifying which features will be included, and this varies by operating system. Typically, parameters describing how the system is configured is stored in a configuration file of some type, and once this file is created, it can be used in several ways.

At one extreme, a system administrator can use it to modify a copy of the operating-system source code. Then the operating system is completely compiled (known as a ***system build***). Data declarations, initializations, and constants, along with compilation, produce an output-object version of the operating system that is tailored to the system described in the configuration file.

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

At the other extreme, it is possible to construct a system that is completely modular. Here, selection occurs at execution time rather than at compile or link time. System generation involves simply setting the parameters that describe the system configuration.

The major differences among these approaches are the size and generality of the generated system and the ease of modifying it as the hardware configuration changes. For embedded systems, it is not uncommon to adopt the first approach and create an operating system for a specific, static hardware configuration. However, most modern operating systems that support desktop and laptop computers as well as mobile devices have adopted the second approach. That is, the operating system is still generated for a specific hardware configuration, but the use of techniques such as loadable kernel modules provides modular support for dynamic changes to the system.

We now illusrate how to build a Linux system from scratch, where it is typically necessary to perform the following steps:

1. Download the Linux source code from http://www.kernel.org.
2. Configure the kernel using the "`make menuconfig`" command. This step generates the `.config` configuration file.
3. Compile the main kernel using the "`make`" command. The `make` command compiles the kernel based on the configuration parameters identified in the `.config` file, producing the file `vmlinuz`, which is the kernel image.
4. Compile the kernel modules using the "`make modules`" command. Just as with compiling the kernel, module compilation depends on the configuration parameters specified in the `.config` file.
5. Use the command "`make modules_install`" to install the kernel modules into `vmlinuz`.
6. Install the new kernel on the system by entering the "`make install`" command.

When the system reboots, it will begin running this new operating system.

Alternatively, it is possible to modify an existing system by installing a Linux virtual machine. This will allow the host operating system (such as Windows or macOS) to run Linux. (We introduced virtualization in Section 1.7 and cover the topic more fully in the chapter Virtual Machines.)

There are a few options for installing Linux as a virtual machine. One alternative is to build a virtual machine from scratch. This option is similar to building a Linux system from scratch; however, the operating system does not need to be compiled. Another approach is to use a Linux virtual machine appliance, which is an operating system that has already been built and configured. This option simply requires downloading the appliance and installing it using virtualization software such as VirtualBox or VMware. For example, to build the operating system used in the virtual machine provided with this text, the authors did the following:

1. Downloaded the Ubuntu ISO image from https://www.ubuntu.com/
2. Instructed the virtual machine software VirtualBox to use the ISO as the bootable medium and booted the virtual machine
3. Answered the installation questions and then installed and booted the operating system as a virtual machine

System boot

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The process of starting a computer by loading the kernel is known as ***booting*** the system. On most systems, the boot process proceeds as follows:

1. A small piece of code known as the ***bootstrap program*** or ***boot loader*** locates the kernel.
2. The kernel is loaded into memory and started.
3. The kernel initializes hardware.
4. The root file system is mounted.

In this section, we briefly describe the boot process in more detail.

Some computer systems use a multistage boot process: When the computer is first powered on, a small boot loader located in nonvolatile firmware known as ***BIOS*** is run. This initial boot loader usually does nothing more than load a second boot loader, which is located at a fixed disk location called the ***boot block***. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it must fit in a single disk block) and knows only the address on disk and the length of the remainder of the bootstrap program.

Many recent computer systems have replaced the BIOS-based boot process with ***UEFI*** (Unified Extensible Firmware Interface). UEFI has several advantages over BIOS, including better support for 64-bit systems and larger disks. Perhaps the greatest advantage is that UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.

Whether booting from BIOS or UEFI, the bootstrap program can perform a variety of tasks. In addition to loading the file containing the kernel program into memory, it also runs diagnostics to determine the state of the machine—for example, inspecting memory and the CPU and discovering devices. If the diagnostics pass, the program can continue with the booting steps. The bootstrap can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system and mounts the root file system. It is only at this point is the system said to be ***running***.

GRUB is an open-source bootstrap program for Linux and UNIX systems. Boot parameters for the system are set in a GRUB configuration file, which is loaded at startup. GRUB is flexible and allows changes to be made at boot time, including modifying kernel parameters and even selecting among different kernels that can be booted. As an example, the following are kernel parameters from the special Linux file `/proc/cmdline`, which is used at boot time:


```
BOOT_IMAGE=/boot/vmlinuz-4.4.0-59-generic
root=UUID=5f2e2232-4e47-4fe8-ae94-45ea749a5c92
```

`BOOT_IMAGE` is the name of the kernel image to be loaded into memory, and `root` specifies a unique identifier of the root file system.

To save space as well as decrease boot time, the Linux kernel image is a compressed file that is extracted after it is loaded into memory. During the boot process, the boot loader typically creates a temporary RAM file system, known as `initramfs`. This file system contains necessary drivers and kernel modules that must be installed to support the ***real*** root file system (which is not in main memory). Once the kernel has started and the necessary drivers are installed, the kernel switches the root file system from the temporary RAM location to the appropriate root file system location. Finally, Linux creates the `systemd` process, the initial process in the system, and then starts other services (for example, a web server and/or database). Ultimately, the system will present the user with a login prompt. In Section Boot block, we describe the boot process for Windows.

It is worthwhile to note that the booting mechanism is not independent from the boot loader. Therefore, there are specific versions of the GRUB boot loader for BIOS and UEFI, and the firmware must know as well which specific bootloader is to be used.

The boot process for mobile systems is slightly different from that for traditional PCs. For example, although its kernel is Linux-based, Android does not use GRUB and instead leaves it up to vendors to provide boot loaders. The most common Android boot loader is LK (for "little kernel"). Android systems use the same compressed kernel image as Linux, as well as an initial RAM file system. However, whereas Linux discards the `initramfs` once all necessary drivers have been loaded, Android maintains `initramfs` as the root file system for the device. Once the kernel has been loaded and the root file system mounted, Android starts the `init` process and creates a number of services before displaying the home screen.

Finally, boot loaders for most operating systems—including Windows, Linux, and macOS, as well as both iOS and Android—provide booting into ***recovery mode*** or ***single-user mode*** for diagnosing hardware issues, fixing corrupt file systems, and even reinstalling the operating system. In addition to hardware failures, computer systems can suffer from software errors and poor operating-system performance, which we consider in the following section.

PARTICIPATION ACTIVITY | 2.9.1: Section review questions.

1) A bootstrap program ____.

- starts the operating system
- consists of the entire operating system
- cannot be used for diagnosing system issues

Section glossary

system build: Creation of an operating-system build and configuration for a specific computer site.

booting: The procedure of starting a computer by loading the kernel.

bootstrap program: The program that allows the computer to start running by initializing hardware and loading the kernel.

BIOS: Code stored in firmware and run at boot time to start system operation.

boot block: A block of code stored in a specific location on disk with the instructions to boot the kernel stored on that disk. The UFS boot control block.

unified extensible firmware interface (UEFI): The modern replacement for BIOS containing a complete boot manager.

running: The state of the operating system after boot when all kernel initialization has completed and system services have started. In general, the system state after booting and before crashing or being shut down.

GRUB: A common open-source bootstrap loader that allows selection of boot partitions and options to be passed to the selected kernel.

recovery mode: A system boot state providing limited services and designed to enable the system admin to repair system problems and debug system startup.

single-user mode: A system boot state providing limited services and designed to enable the system admin to repair system problems and debug system startup.

2.10 Operating-system debugging

We have mentioned debugging from time to time in this chapter. Here, we take a closer look. Broadly, ***debugging*** is the activity of finding and fixing errors in a system, both in hardware and in software. Performance problems are considered bugs, so debugging can also include ***performance tuning***, which seeks to improve performance by removing processing ***bottlenecks***. In this section, we explore debugging process and kernel errors and performance problems. Hardware debugging is outside the scope of this text.

Failure analysis

If a process fails, most operating systems write the error information to a ***log file*** to alert system administrators or users that the problem occurred. The operating system can also take a ***core***

dump—a capture of the memory of the process—and store it in a file for later analysis. (Memory was referred to as the "core" in the early days of computing.) Running programs and core dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process at the time of failure.

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

Operating-system debugging and process debugging frequently use different tools and techniques 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. Obviously, such strategies would be unnecessary for debugging ordinary user-level processes.

Performance monitoring and tuning

We mentioned earlier that performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance. Thus, the operating system must have some means of computing and displaying measures of system behavior. Tools may be characterized as providing either ***per-process*** or ***system-wide*** observations. To make these observations, tools may use one of two approaches—***counters*** or ***tracing***. We explore each of these in the following sections.

Counters

Operating systems keep track of system activity through a series of counters, such as the number of system calls made or the number of operations performed to a network device or disk. The following are examples of Linux tools that use counters:

Per-process

- `ps`—reports information for a single process or selection of processes
- `top`—reports real-time statistics for current processes

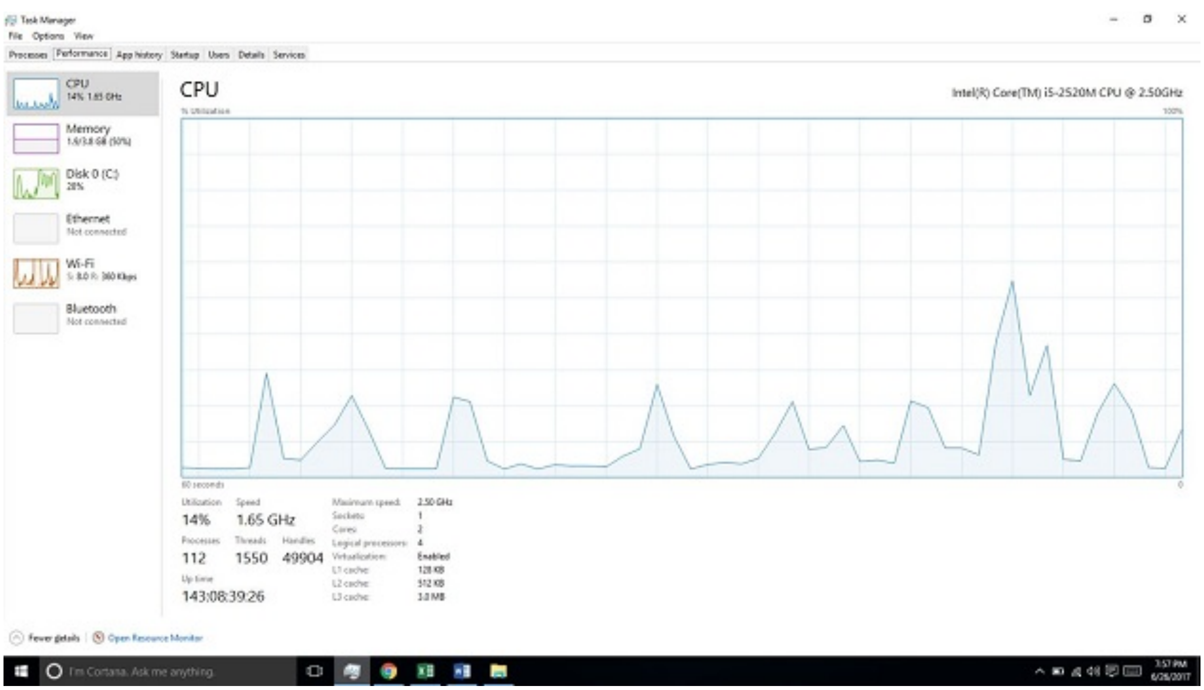
System-wide

- `vmstat`—reports memory-usage statistics
- `netstat`—reports statistics for network interfaces
- `iostat`—reports I/O usage for disks

Most of the counter-based tools on Linux systems read statistics from the `/proc` file system. `/proc` is a "pseudo" file system that exists only in kernel memory and is used primarily for querying various per-process as well as kernel statistics. The `/proc` file system is organized as a directory hierarchy, with the process ID (a unique integer value assigned to each process) appearing as a subdirectory below `/proc`. For example, the directory entry `/proc/2155` would contain per-process statistics for the process with an ID of 2155. There are `/proc` entries for various kernel statistics as well. In both this chapter and the next chapter Processes, we provide programming projects where you will create and access the `/proc` file system.

Windows systems provide the ***Windows Task Manager***, a tool that includes information for current applications as well as processes, CPU and memory usage, and networking statistics. A screen shot of the task manager in Windows 10 appears in Figure 2.10.1.

Figure 2.10.1: The Windows 10 task manager.



Tracing

Whereas counter-based tools simply inquire on the current value of certain statistics that are maintained by the kernel, tracing tools collect data for a specific event—such as the steps involved in a system-call invocation.

The following are examples of Linux tools that trace events:

Per-process

- `strace`—traces system calls invoked by a process

- gdb—a source-level debugger

System-wide

- `perf`—a collection of Linux performance tools
- `tcpdump`—collects network packets

Kernighan's law

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

Making operating systems easier to understand, debug, and tune as they run is an active area of research and practice. A new generation of kernel-enabled performance analysis tools has made significant improvements in how this goal can be achieved. Next, we discuss BCC, a toolkit for dynamic kernel tracing in Linux.

BCC

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

BCC (BPF Compiler Collection) is a rich toolkit that provides tracing features for Linux systems. BCC is a front-end interface to the eBPF (extended Berkeley Packet Filter) tool. The BPF technology was developed in the early 1990s for filtering traffic across a computer network. The "extended" BPF (eBPF) added various features to BPF. eBPF programs are written in a subset of C and are compiled into eBPF instructions, which can be dynamically inserted into a running Linux system. The eBPF instructions can be used to capture specific events (such as a certain system call being invoked) or to monitor system performance (such as the time required to perform disk I/O). To ensure that eBPF instructions are well behaved, they are passed through a ***verifier*** before being inserted into the running Linux kernel. The verifier checks to make sure that the instructions do not affect system performance or security.

Although eBPF provides a rich set of features for tracing within the Linux kernel, it traditionally has been very difficult to develop programs using its C interface. BCC was developed to make it easier to write tools using eBPF by providing a front-end interface in Python. A BCC tool is written in Python and it embeds C code that interfaces with the eBPF instrumentation, which in turn interfaces with the kernel. The BCC tool also compiles the C program into eBPF instructions and inserts it into the kernel using either probes or tracepoints, two techniques that allow tracing events in the Linux kernel.

The specifics of writing custom BCC tools are beyond the scope of this text, but the BCC package (which is installed on the Linux virtual machine we provide) provides a number of existing tools that monitor several areas of activity in a running Linux kernel. As an example, the BCC `disksnoop` tool traces disk I/O activity. Entering the command

```
./disksnoop.py
```

generates the following example output:

TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

This output tells us the timestamp when the I/O operation occurred, whether the I/O was a Read or Write operation, and how many bytes were involved in the I/O. The final column reflects the duration (expressed as latency or LAT) in milliseconds of the I/O.

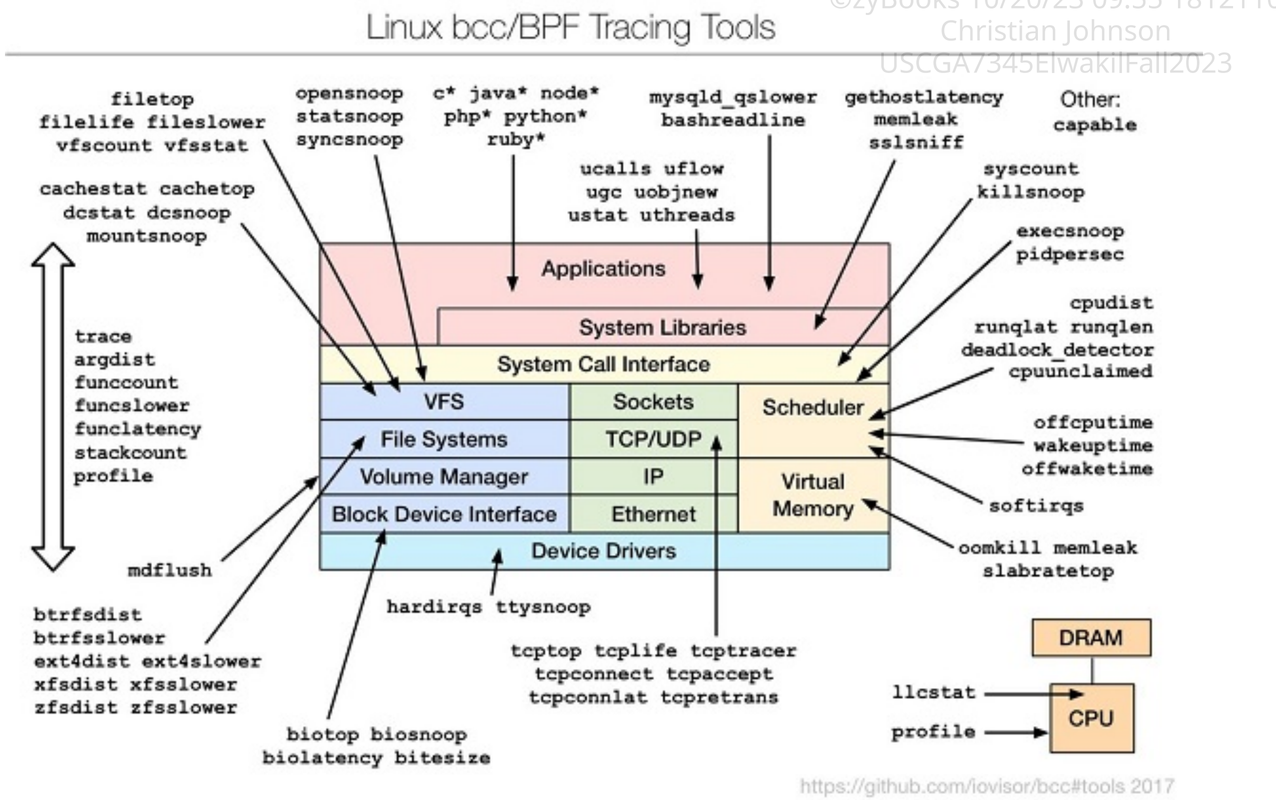
Many of the tools provided by BCC can be used for specific applications, such as MySQL databases, as well as Java and Python programs. Probes can also be placed to monitor the activity of a specific process. For example, the command

```
./opensnoop -p 1225
```


will trace `open()` system calls performed only by the process with an identifier of 1225.

What makes BCC especially powerful is that its tools can be used on live production systems that are running critical applications without causing harm to the system. This is particularly useful for system administrators who must monitor system performance to identify possible bottlenecks or security exploits. Figure 2.10.2 illustrates the wide range of tools currently provided by BCC and eBPF and their ability to trace essentially any area of the Linux operating system. BCC is a rapidly changing technology with new features constantly being added.

Figure 2.10.2: The BCC and eBPF tracing tools.



PARTICIPATION ACTIVITY | 2.10.1: Section review questions.

1) If a program terminates abnormally, a dump of memory may be examined by a ____ to determine the cause of the problem.

- crash
- debugger
- shell

2) What best describes the difference between counters and tracing for performance monitoring?

- Counters keep track of system-wide events and tracing records per-process events.
- The BCC toolkit provides both counters and tracing.
- Counters count the number of events and tracing follows the steps of a specific event.

Section glossary

debugging: The activity of finding and removing errors.

performance tuning: The activity of improving performance by removing bottlenecks.

bottleneck: A performance-limiting aspect of computing (e.g., poorly written code or a hardware component that is not as fast as others in the system).

log file: A file containing error or "logging" information; used for debugging or understanding the state or activities of the system.

core dump: A copy of the state of a process written to disk when a process crashes; used for debugging.

crash: Termination of execution due to a problem. A failure in the kernel results in a system crash and reboot, while a process failure results in a process crash.

crash dump: A copy of the state of the kernel written to disk during a crash; used for debugging.

BPF compiler collection (BCC): A rich toolkit for tracing system activity on Linux for debugging and performance-tuning purposes.

2.11 Summary

- An operating system provides an environment for the execution of programs by providing services to users and programs.
- The three primary approaches for interacting with an operating system are (1) command

interpreters, (2) graphical user interfaces, and (3) touch-screen interfaces.
- System calls provide an interface to the services made available by an operating system. Programmers use a system call's application programming interface (API) for accessing system-call services.
- System calls can be divided into six major categories: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection.
- The standard C library provides the system-call interface for UNIX and Linux systems.
- Operating systems also include a collection of system programs that provide utilities to users.
- A linker combines several relocatable object modules into a single binary executable file. A loader loads the executable file into memory, where it becomes eligible to run on an available CPU.
- There are several reasons why applications are operating-system specific. These include different binary formats for program executables, different instruction sets for different CPUs, and system calls that vary from one operating system to another.
- An operating system is designed with specific goals in mind. These goals ultimately determine the operating system's policies. An operating system implements these policies through specific mechanisms.
- A monolithic operating system has no structure; all functionality is provided in a single, static binary file that runs in a single address space. Although such systems are difficult to modify, their primary benefit is efficiency.
- A layered operating system is divided into a number of discrete layers, where the bottom layer is the hardware interface and the highest layer is the user interface. Although layered software systems have had some success, this approach is generally not ideal for designing operating systems due to performance problems.
- The microkernel approach for designing operating systems uses a minimal kernel; most services run as user-level applications. Communication takes place via message passing.
- A modular approach for designing operating systems provides operating-system services through modules that can be loaded and removed during run time. Many contemporary operating systems are constructed as hybrid systems using a combination of a monolithic kernel and modules.
- A boot loader loads an operating system into memory, performs initialization, and begins system execution.
- The performance of an operating system can be monitored using either counters or tracing. Counters are a collection of system-wide or per-process statistics, while tracing follows the execution of a program through the operating system.

2.12 Practice exercises

EXERCISE 2.12.1: (Problem 2.1 in the 10th edition).

(a) What is the purpose of system calls?

EXERCISE 2.12.2: (Problem 2.2 in the 10th edition).

(a) What is the purpose of the command interpreter? Why is it usually separate from the kernel?

EXERCISE 2.12.3: (Problem 2.3 in the 10th edition).

(a) What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?

EXERCISE 2.12.4: (Problem 2.4 in the 10th edition).

(a) What is the purpose of system programs?

EXERCISE 2.12.5: (Problem 2.5 in the 10th edition).

(a) What is the main advantage of the layered approach to system design? What are the disadvantages of the layered approach?

EXERCISE 2.12.6: (Problem 2.6 in the 10th edition).

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

EXERCISE 2.12.7: (Problem 2.7 in the 10th edition).

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

EXERCISE 2.12.8: (Problem 2.8 in the 10th edition).

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

2.13 Exercises

EXERCISE 2.13.1: (Problem 2.9 in the 10th edition).

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

EXERCISE 2.13.2: (Problem 2.10 in the 10th edition).

(a) Describe three general methods for passing parameters to the operating system.

EXERCISE 2.13.3: (Problem 2.11 in the 10th edition).

(a) Describe how you could obtain a statistical profile of the amount of time a program spends executing different sections of its code. Discuss the importance of obtaining such a statistical profile.

EXERCISE 2.13.4: (Problem 2.12 in the 10th edition).

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

EXERCISE 2.13.5: (Problem 2.13 in the 10th edition).

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

EXERCISE 2.13.6: (Problem 2.14 in the 10th edition).

(a) Describe why Android uses ahead-of-time (AOT) rather than just-in-time (JIT) compilation.

EXERCISE 2.13.7: (Problem 2.15 in the 10th edition).

(a) What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?

EXERCISE 2.13.8: (Problem 2.16 in the 10th edition).

(a) Contrast and compare an application programming interface (API) and an application binary interface (ABI).

EXERCISE 2.13.9: (Problem 2.17 in the 10th edition).

(a) Why is the separation of mechanism and policy desirable?

EXERCISE 2.13.10: (Problem 2.18 in the 10th edition).

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

EXERCISE 2.13.11: (Problem 2.19 in the 10th edition).

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

EXERCISE | 2.13.12: (Problem 2.20 in the 10th edition).

(a) What are the advantages of using loadable kernel modules?

EXERCISE | 2.13.13: (Problem 2.21 in the 10th edition).

(a) How are iOS and Android similar? How are they different?

EXERCISE | 2.13.14: (Problem 2.22 in the 10th edition).

(a) Explain why Java programs running on Android systems do not use the standard Java API and virtual machine.

EXERCISE | 2.13.15: (Problem 2.23 in the 10th edition).

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

2.14 Programming problems

EXERCISE | 2.14.1: (Problem 2.24 in the 10th edition).

(a) In Section 2.3, we described a program that copies the contents of one file to a destination file. This program works by first prompting the user for the name of the source and destination files. Write this program using either the POSIX or Windows API. Be sure to include all necessary error checking, including ensuring that the source file exists.

Once you have correctly designed and tested the program, if you used a system that supports it, run the program using a utility that traces system calls. Linux systems provide the `strace` utility, and macOS systems use the `dtruss` command. (The `dtruss` command, which actually is a front end to `dtrace`, requires `admin` privileges, so it must be run using `sudo`.) These tools can be used as follows (assume that the name of the executable file is `FileCopy`:

- ***Linux:***
  `strace ./FileCopy`
- ***Macos:***
  `sudo dtruss ./FileCopy`

Since Windows systems do not provide such a tool, you will have to trace through the Windows version of this program using a debugger.

2.15 Programming projects

Introduction to Linux kernel modules

In this project, you will learn how to create a kernel module and load it into the Linux kernel. You will then modify the kernel module so that it creates an entry in the `/proc` file system. The project can be completed using the Linux virtual machine that is available with this text. Although you may use any text editor to write these C programs, you will have to use the ***terminal*** application to compile the programs, and you will have to enter commands on the command line to manage the modules in the kernel.

As you'll discover, the advantage of developing kernel modules is that it is a relatively easy method of interacting with the kernel, thus allowing you to write programs that directly invoke kernel functions. It is important for you to keep in mind that you are indeed writing ***kernel code*** that directly interacts with the kernel. That normally means that any errors in the code could crash the system! However, since you will be using a virtual machine, any failures will at worst only require rebooting the system.

I. Kernel modules overview

The first part of this project involves following a series of steps for creating and inserting a module into the Linux kernel.

You can list all kernel modules that are currently loaded by entering the command

```
lsmod
```

This command will list the current kernel modules in three columns: name, size, and where the module is being used.

The program in the figure below (named `simple.c` and available with the source code for this text) illustrates a very basic kernel module that prints appropriate messages when it is loaded and unloaded.

Figure 2.15.1: Kernel module simple.c.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

/* This function is called when the module is loaded.
*/
int simple_init(void)
{
   printk(KERN_INFO "Loading Kernel Module\n");

   return 0;
}

/* This function is called when the module is removed.
*/
void simple_exit(void)
{
   printk(KERN_INFO "Removing Kernel Module\n");
}

/* Macros for registering module entry and exit points.
*/
module_init(simple_init);
module_exit(simple_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```


The function `simple_init()` is the ***module entry point***, which represents the function that is invoked when the module is loaded into the kernel. Similarly, the `simple_exit()` function is the ***module exit point***—the function that is called when the module is removed from the kernel.

The module entry point function must return an integer value, with 0 representing success and any other value representing failure. The module exit point function returns `void`. Neither the module entry point nor the module exit point is passed any parameters. The two following macros are used for registering the module entry and exit points with the kernel:

```
module_init(simple_init)
module_exit(simple_exit)
```

Notice in the figure how the module entry and exit point functions make calls to the `printk()` function. `printk()` is the kernel equivalent of `printf()`, but its output is sent to a kernel log buffer whose contents can be read by the `dmesg` command. One difference between `printf()` and `printk()` is that `printk()` allows us to specify a priority flag, whose values are given in the `<linux/printk.h>` include file. In this instance, the priority is `KERN_INFO`, which is defined as an ***informational*** message.

The final lines—`MODULE_LICENSE()`, `MODULE_DESCRIPTION()`, and `MODULE_AUTHOR()` —represent details regarding the software license, description of the module, and author. For our purposes, we do not require this information, but we include it because it is standard practice in developing kernel modules.

This kernel module `simple.c` is compiled using the `Makefile` accompanying the source code with this project. To compile the module, enter the following on the command line:

```
make
```

The compilation produces several files. The file `simple.ko` represents the compiled kernel module. The following step illustrates inserting this module into the Linux kernel.

II. Loading and removing kernel modules

Kernel modules are loaded using the `insmod` command, which is run as follows:

```
sudo insmod simple.ko
```

To check whether the module has loaded, enter the `lsmod` command and search for the module `simple`. Recall that the module entry point is invoked when the module is inserted into the kernel. To check the contents of this message in the kernel log buffer, enter the command

```
dmesg
```

You should see the message "`Loading Module.`"

Removing the kernel module involves invoking the `rmmod` command (notice that the `.ko` suffix is unnecessary):

```
sudo rmmod simple
```

Be sure to check with the `dmesg` command to ensure the module has been removed.

Because the kernel log buffer can fill up quickly, it often makes sense to clear the buffer periodically. This can be accomplished as follows:

```
sudo dmesg -c
```

Proceed through the steps described above to create the kernel module and to load and unload the module. Be sure to check the contents of the kernel log buffer using `dmesg` to ensure that you have followed the steps properly.

As kernel modules are running within the kernel, it is possible to obtain values and call functions that are available only in the kernel and not to regular user applications. For example, the Linux include file <`linux/hash.h`> defines several hashing functions for use within the kernel. This file also defines the constant value `GOLDEN_RATIO_PRIME` (which is defined as an `unsigned long`). This value can be printed out as follows:

```
printk(KERN_INFO "%lu\n", GOLDEN_RATIO_PRIME);
```

As another example, the include file <`linux/gcd.h`> defines the following function

```
unsigned long gcd(unsigned long a, unsigned b);
```

which returns the greatest common divisor of the parameters a and b.

Once you are able to correctly load and unload your module, complete the following additional steps:

1. Print out the value of `GOLDEN_RATIO_PRIME` in the `simple_init()` function.
2. Print out the greatest common divisor of 3,300 and 24 in the `simple_exit()` function.

As compiler errors are not often helpful when performing kernel development, it is important to compile your program often by running `make` regularly. Be sure to load and remove the kernel module and check the kernel log buffer using `dmesg` to ensure that your changes to `simple.c` are working properly.

In Section Timer, we described the role of the timer as well as the timer interrupt handler. In Linux, the rate at which the timer ticks (the ***tick rate***) is the value `HZ` defined in <`asm/param.h`>. The value of `HZ` determines the frequency of the timer interrupt, and its value varies by machine type and architecture. For example, if the value of `HZ` is 100, a timer interrupt occurs 100 times per second, or every 10 milliseconds. Additionally, the kernel keeps track of the global variable `jiffies`, which maintains the number of timer interrupts that have occurred since the system was booted. The `jiffies` variable is declared in the file <`linux/jiffies.h`>.

1. Print out the values of `jiffies` and `HZ` in the `simple_init()` function.
2. Print out the value of `jiffies` in the `simple_exit()` function.

Before proceeding to the next set of exercises, consider how you can use the different values of `jiffies` in `simple_init()` and `simple_exit()` to determine the number of seconds that have elapsed since the time the kernel module was loaded and then removed.

III. The `/proc` file system

The `/proc` file system is a "pseudo" file system that exists only in kernel memory and is used primarily for querying various kernel and per-process statistics. This exercise involves designing kernel modules that create additional entries in the `/proc` file system involving both kernel statistics and information related to specific processes. The entire program is included in the figures below.

Figure 2.15.2: The /proc file-system kernel module, Part 1

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128
#define PROC_NAME "hello"

ssize_t proc_read(struct file *file, char __user
*usr_buf,
  size_t count, loff_t *pos);

static struct file_operations proc_ops = {
    .owner = THIS_MODULE,
    .read = proc_read,
};

/* This function is called when the module is loaded.
*/
int proc_init(void)
{
/* creates the /proc/hello entry */
proc_create(PROC_NAME, 0666, NULL, &proc_ops);

return 0;
}

/* This function is called when the module is removed.
*/
void proc_exit(void)
{
  /* removes the /proc/hello entry */
  remove_proc_entry(PROC_NAME, NULL);
}
```

Figure 2.15.3: The /proc file system kernel module, Part 2

```
/* This function is called each time /proc/hello is
read */
ssize_t proc_read(struct file *file, char __user
*usr_buf,
  size_t count, loff_t *pos)
{
  int rv = 0;
  char buffer[BUFFER_SIZE];
  static int completed = 0;

  if (completed) {
   completed = 0;
   return 0;
  }

  completed = 1;

  rv = sprintf(buffer, "Hello World\n");

  /* copies kernel space buffer to user space usr_buf
*/
  copy_to_user(usr_buf, buffer, rv);
  return rv;
}
module_init(proc_init);
module_exit(proc_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello Module");
MODULE_AUTHOR("SGG");
```


We begin by describing how to create a new entry in the `/proc` file system. The following program example (named `hello.c` and available with the source code for this text) creates a `/proc` entry named `/proc/hello`. If a user enters the command

```
cat /proc/hello
```

the infamous `Hello World` message is returned.

In the module entry point `proc_init()`, we create the new `/proc/hello` entry using the `proc_create()` function. This function is passed `proc_ops`, which contains a reference to a `struct file_operations`. This struct initializes the `.owner` and `.read` members. The value of `.read` is the name of the function `proc_read()` that is to be called whenever `/proc/hello` is read.

Examining this `proc_read()` function, we see that the string `"Hello World\n"` is written to the variable `buffer` where `buffer` exists in kernel memory. Since `/proc/hello` can be accessed from user space, we must copy the contents of `buffer` to user space using the kernel function `copy_to_user()`. This function copies the contents of kernel memory `buffer` to the variable `usr_buf`, which exists in user space.

Each time the `/proc/hello` file is read, the `proc_read()` function is called repeatedly until it returns 0, so there must be logic to ensure that this function returns 0 once it has collected the data (in this case, the string `"Hello World\n"`) that is to go into the corresponding `/proc/hello` file.

Finally, notice that the `/proc/hello` file is removed in the module exit point `proc_exit()` using the function `remove_proc_entry()`.

IV. Assignment

This assignment will involve designing two kernel modules:

1. Design a kernel module that creates a `/proc` file named `/proc/jiffies` that reports the current value of `jiffies` when the `/proc/jiffies` file is read, such as with the command

   ```
   cat /proc/jiffies
   ```

   Be sure to remove `/proc/jiffies` when the module is removed.

2. Design a kernel module that creates a `proc` file named `/proc/seconds` that reports the number of elapsed seconds since the kernel module was loaded. This will involve using the value of `jiffies` as well as the `HZ` rate. When a user enters the command

   ```
   cat /proc/seconds
   ```

   your kernel module will report the number of seconds that have elapsed since the kernel module was first loaded. Be sure to remove `/proc/seconds` when the module is removed.

2.16 Further reading

[Bryant and O'Hallaron (2015)] provide an overview of computer systems, including the role of the linker and loader. [Atlidakis et al. (2016)] discuss POSIX system calls and how they relate to modern operating systems. [Levin (2013)] covers the internals of bothmacOS and iOS, and [Levin (2015)] describes details of the Android system.Windows 10 internals are covered in [Russinovich et al. (2017)]. BSD UNIX is described in [McKusick et al. (2015)]. [Love (2010)] and [Mauerer (2008)] thoroughly discuss the Linux kernel. Solaris is fully described in [McDougall and Mauro (2007)].

Linux source code is available at http://www.kernel.org. The Ubuntu ISO image is available from https://www.ubuntu.com/.

Comprehensive coverage of Linux kernel modules can be found at http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf. [Ward (2015)] and http://www.ibm.com/developerworks/linux/library/l-linuxboot/ describe the Linux boot process using GRUB. Performance tuning—with a focus on Linux and Solaris systems—is covered in [Gregg (2014)]. Details for the BCC toolkit can be found at https://github.com/iovisor/bcc/#tools.

2.17 Bibliography

[Atlidakis et al. (2016)] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, "*POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing*" (2016), pages 19:1-19:17.

[Bryant and O'Hallaron (2015)] R. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition (2015).

[Gregg (2014)] B. Gregg, *Systems Performance-Enterprise and the Cloud*, Pearson (2014).

[Levin (2013)] J. Levin, *Mac OS X and iOS Internals to the Apple's Core*, Wiley (2013).

[Levin (2015)] J. Levin, *Android Internals-A Confectioner's Cookbook*. *Volume I* (2015).

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

[McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).

[McKusick et al. (2015)] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIXOperating System-Second Edition*, Pearson (2015).

[Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals-Part 1*, Seventh Edition, Microsoft Press (2017).

[Ward (2015)] B. Ward, *How LINUX Works-What Every Superuser Should Know*, Second Edition, No Starch Press (2015).