Lecture 2

Structure of operating systems

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.
- 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.

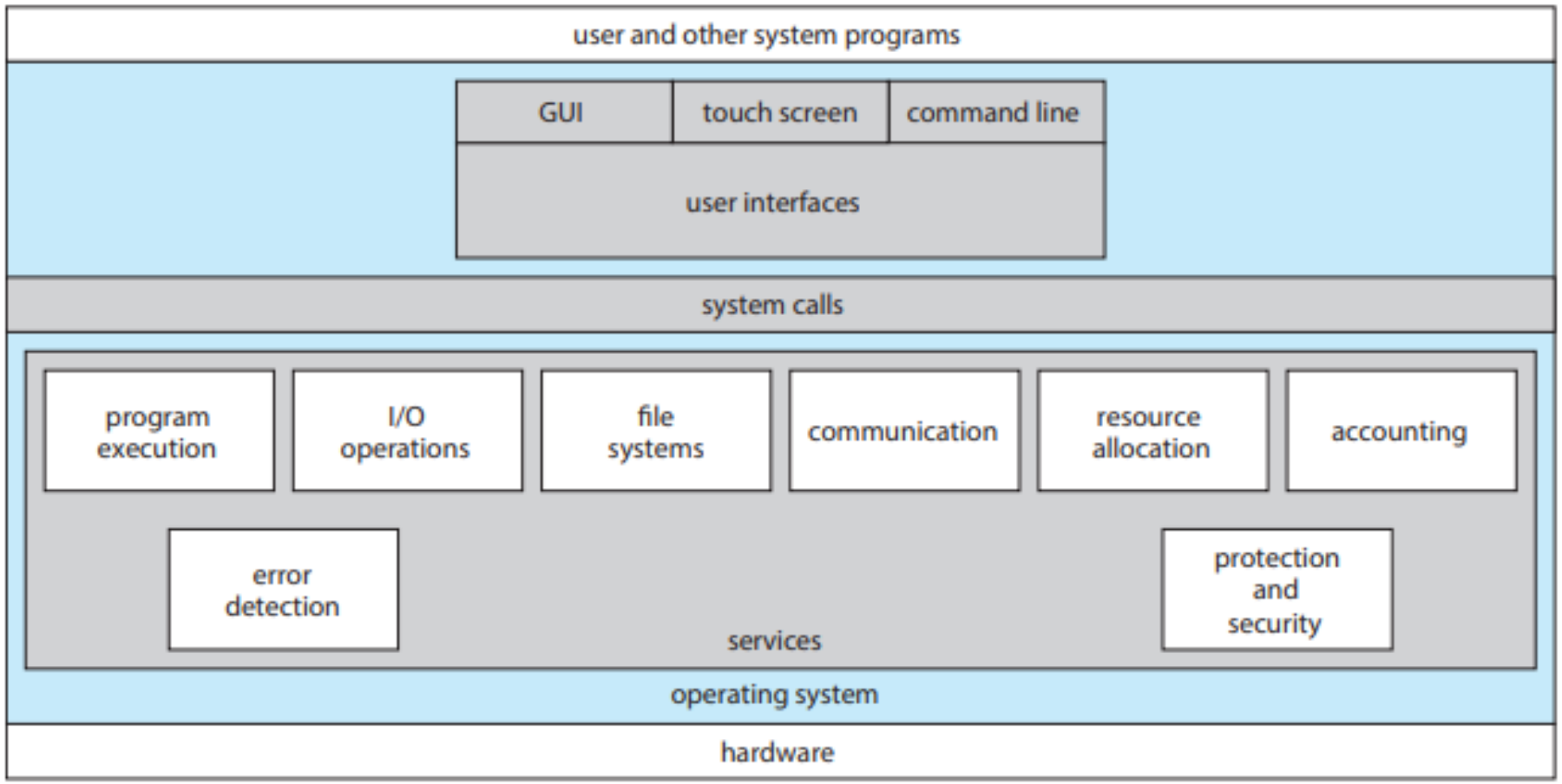

Figure 2.1 A view of operating system services.

User interface

- **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

- **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

- **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

- **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

- **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

- **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.

Resource allocation

- **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

- **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

- **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.

Authentication	Authorization
Determines whether users are who they claim to be	Determines what users can and cannot access
Challenges the user to validate credentials (for example, through passwords, answers to security questions, or facial recognition)	Verifies whether access is allowed through policies and rules
Usually done before authorization	Usually done after successful authentication
Generally, transmits info through an ID Token	Generally, transmits info through an Access Token
Generally governed by the OpenID Connect (OIDC) protocol	Generally governed by the OAuth 2.0 framework
Example: Employees in a company are required to authenticate through the network before accessing their company email	Example: After an employee successfully authenticates, the system determines what information the employees are allowed to access

- Authentication and authorization are two vital information security processes that administrators use to protect systems and information. Authentication verifies the identity of a user or service, and authorization determines their access rights. Although the two terms sound alike, they play separate but equally essential roles in securing applications and data.
- **authentication** is the process of verifying who a user is, while **authorization** is the process of verifying what they have access to

User and Operating-System Interface

- 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 shows the Bourne-Again (or bash) shell command interpreter being used on macOS.

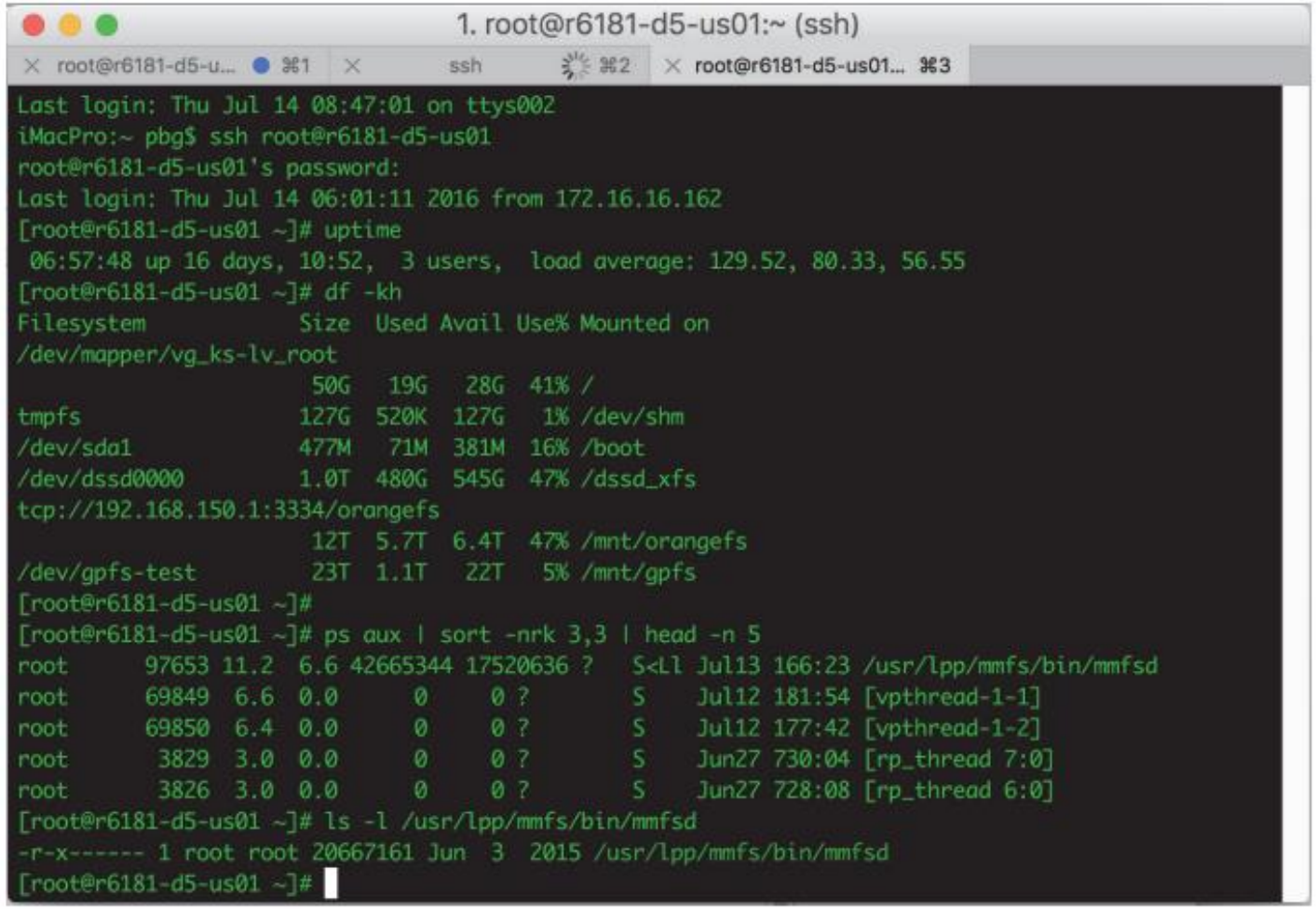

Figure 2.2 The `bash` shell command interpreter in macOS.

- 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
    - `rm file.txt`

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.

Touch-Screen Interface

- Because a 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 2.3 illustrates the touch screen of the Apple iPhone. Both the iPad and the iPhone use the Springboard touch-screen interface.

- The choice of whether to use a command-line or GUI interface is mostly one of personal preference.

Figure 2.3 The iPhone touch screen.

- 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 2.4 is a screenshot of the macOS GUI.

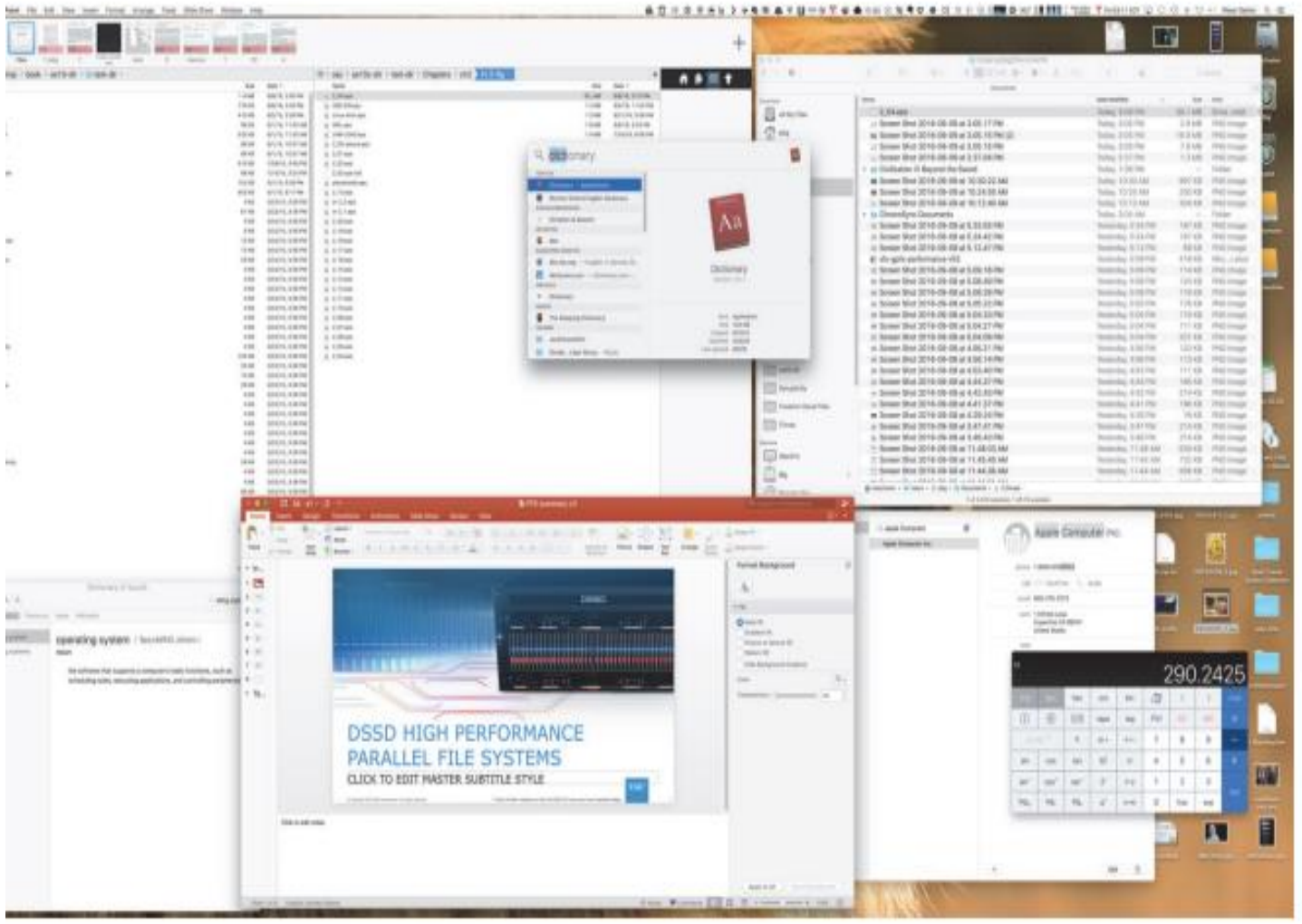

Figure 2.4 The macOS GUI.

- 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.

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`

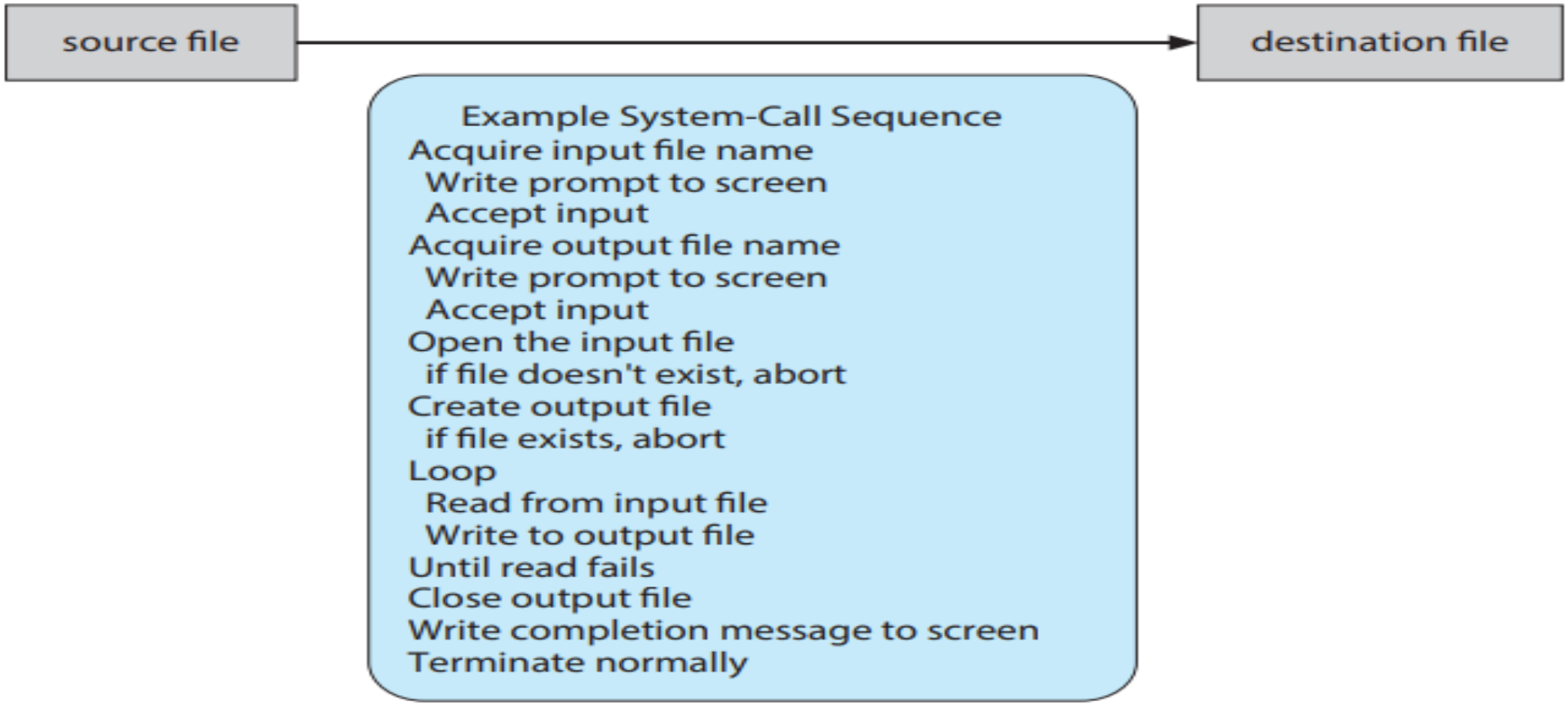

Figure 2.5 Example of how system calls are used.

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.

run-time environment (RTE)

- 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.

Types of System Calls

- System calls can be grouped roughly into ***six major categories***: **process control, fil management, device management, information maintenance, communications, and protection**.

- 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

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 generalpurpose 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.

shared-memory model

- 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

Protection

- Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multi programmed 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

Linkers and Loaders0

- 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 Figure 2.11.

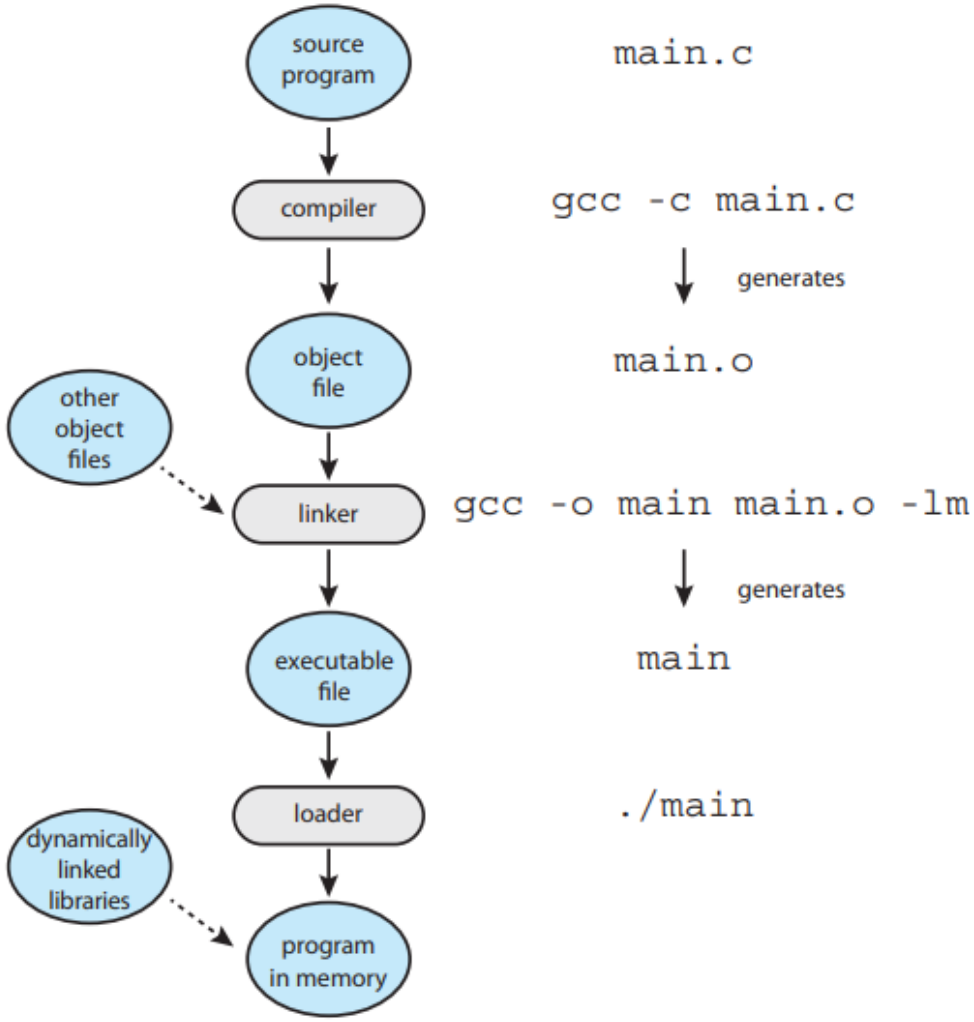

Figure 2.11 The role of the linker and loader.

- 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 fil** . 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 Figure 2.11, 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.)

Thank you for your attention!