An **operating system** is the software that controls the overall operation of a computer. It provides the means by which a user can store and retrieve files, provides the interface by which a user can request the execution of programs, and provides the environment necessary to execute the programs requested.

Perhaps the best-known example of an operating system is Windows, which is provided in numerous versions by Microsoft and widely used in the PC arena. Another well-established example is UNIX, which is a popular choice for larger computer systems as well as PCs. In fact, UNIX is the core of two other popular operating systems: Mac OS, which is the operating system provided by Apple for its range of Mac machines, and Solaris, which was developed by Sun Microsystems (now owned by Oracle). Still another example of an operating system found on both large and small machines is Linux, which was originally developed non-commercially by computer enthusiasts and is now available through many commercial sources, including IBM.

For casual computer users, the differences between operating systems are largely cosmetic. For computing professionals, different operating systems can represent major changes in the tools they work with or the philosophy they follow in disseminating and maintaining their work. Nevertheless, at their core, all mainstream operating systems address the same kinds of problems that computing experts have faced for more than half a century.

3.1 The History of Operating Systems

Today's operating systems are large, complex software packages that have grown from humble beginnings. The computers of the 1940s and 1950s were not very flexible or efficient. Machines occupied entire rooms. Program execution required significant preparation of equipment in terms of mounting magnetic tapes, placing punched cards in card readers, setting switches, and so on. The execution of each program, called a **job**, was handled as an isolated activity—the machine was prepared for executing the program, the program was executed, and then all the tapes, punched cards, etc. had to be retrieved before the next program preparation could begin. When several users needed to share a machine, sign-up sheets were provided so that users could reserve the machine for blocks of time. During the time period allocated to a user, the machine was totally under that user's control. The session usually began with program setup, followed by short periods of program execution. It was often completed in a hurried effort to do just one more thing ("It will only take a minute") while the next user was impatiently starting to set up.

In such an environment, operating systems began as systems for simplifying program setup and for streamlining the transition between jobs. One early development was the separation of users and equipment, which eliminated the physical transition of people in and out of the computer room. For this purpose, a computer operator was hired to operate the machine. Anyone wanting a program run was required to submit it, along with any required

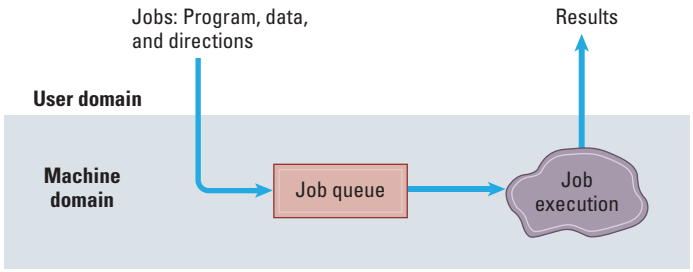

Figure 3.1 Batch processing

data and special directions about the program's requirements, to the operator and return later for the results. The operator, in turn, loaded these materials into the machine's mass storage where a program called the operating system could read and execute them one at a time. This was the beginning of batch processing—the execution of jobs by collecting them in a single batch, then executing them without further interaction with the user.

In batch processing systems, the jobs residing in mass storage wait for execution in a **job queue** (Figure 3.1). A **queue** is a storage organization in which objects (in this case, jobs) are ordered in **first-in, first-out** (abbreviated FIFO and pronounced "FI-foe") fashion. That is, the objects are removed from the queue in the order in which they arrived. In reality, most job queues do not rigorously follow the FIFO structure, since most operating systems provide for consideration of job priorities. As a result, a job waiting in the job queue can be bumped by a higher-priority job.

In early batch processing systems, each job was accompanied by a set of instructions explaining the steps required to prepare the machine for that particular job. These instructions were encoded, using a system known as a job control language (JCL), and stored with the job in the job queue. When the job was selected for execution, the operating system printed these instructions at a printer where they could be read and followed by the computer operator. This communication between the operating system and the computer operator is still seen today, as witnessed by PC operating systems that report such errors as "network not available" and "printer not responding."

A major drawback to using a computer operator as an intermediary between a computer and its users is that the users have no interaction with their jobs once they are submitted to the operator. This approach is acceptable for some applications, such as payroll processing, in which the data and all processing decisions are established in advance. However, it is not acceptable when the user must interact with a program during its execution. Examples include reservation systems in which reservations and cancellations must be reported as they occur; word processing systems in which documents are developed in a dynamic write and rewrite manner; and computer games in which interaction with the machine is the central feature of the game.

To accommodate these needs, new operating systems were developed that allowed a program being executed to carry on a dialogue with the user through

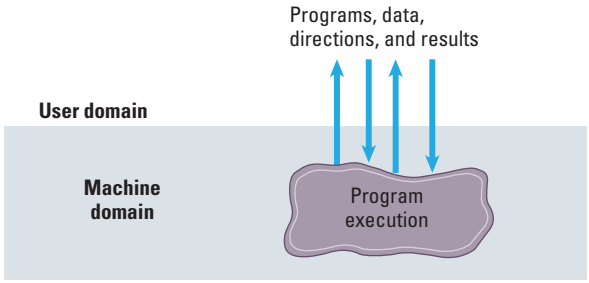

Figure 3.2 Interactive processing

remote terminals—a feature known as **interactive processing** (Figure 3.2). (A terminal consisted of little more than an electronic typewriter by which the user could type input and read the computer's response that was printed on paper. While today's PCs, laptops, and smartphones are vastly more powerful than the computers and terminals of old, ironically, we still frequently use them merely to interact with yet more powerful Internet cloud services.)

Paramount to successful interactive processing is that the actions of the computer be sufficiently fast to coordinate with the needs of the user rather than forcing the user to conform to the machine's timetable. (The task of processing payroll can be scheduled to conform to the amount of time required by the computer, but using a word processor would be frustrating if the machine did not respond promptly as characters are typed.) In a sense, the computer is forced to execute tasks under a deadline, a process that became known as **real-time processing**, in which the actions performed are said to occur in real time. That is, to say that a computer performs a task in real time means that the computer performs the task in accordance with deadlines in its (external real-world) environment.

If interactive systems had been required to serve only one user at a time, real-time processing would have been no problem. But computers in the 1960s and 1970s were expensive, so each machine had to serve more than one user. In turn, it was common for several users, working at remote terminals, to seek interactive service from a machine at the same time, and real-time considerations presented obstacles. If the operating system insisted on executing only one job at a time, only one user would receive satisfactory real-time service.

The solution to this problem was to design operating systems that provided service to multiple users at the same time: a feature called **time-sharing**. One means of implementing time-sharing is to apply the technique called **multiprogramming**, in which time is divided into intervals and then the execution of each job is restricted to only one interval at a time. At the end of each interval, the current job is temporarily set aside and another is allowed to execute during the next interval. By rapidly shuffling the jobs back and forth in this manner, the illusion of several jobs executing simultaneously is created. Depending on the types of jobs being executed, early time-sharing systems were able to provide acceptable real-time processing to as many as 30 users simultaneously. Today, multiprogramming techniques are used in single-user

as well as multiuser systems, although in the former the result is usually called **multitasking**. That is, time-sharing refers to multiple users sharing access to a common computer, whereas multitasking refers to one user executing numerous tasks simultaneously.

With the development of multiuser, time-sharing operating systems, a typical computer installation was configured as a large central computer connected to numerous workstations. From these workstations, users could communicate directly with the computer from outside the computer room rather than submitting requests to a computer operator. Commonly used programs were stored in the machine's mass storage devices and operating systems were designed to execute these programs as requested from the workstations. In turn, the role of a computer operator as an intermediary between the users and the computer began to fade.

Today, the existence of a computer operator has essentially disappeared, especially in the arena of personal computers, where the computer user assumes all of the responsibilities of computer operation. Even most large computer installations run essentially unattended. Indeed, the job of computer operator has given way to that of a system administrator who manages the computer system—obtaining and overseeing the installation of new equipment and software, enforcing local regulations such as the issuing of new accounts and establishing mass storage space limits for the various users, and coordinating efforts to resolve problems that arise in the system—rather than operating the machines in a hands-on manner.

In short, operating systems have grown from simple programs that retrieved and executed programs one at a time into complex systems that coordinate time-sharing, maintain programs and data files in the machine's mass storage devices, and respond directly to requests from the computer's users.

But the evolution of operating systems continues. The development of multiprocessor machines has led to operating systems that provide time-sharing/multitasking capabilities by assigning different tasks to different processors as well as by sharing the time of each single processor. These operating systems must wrestle with such problems as **load balancing** (dynamically allocating tasks to the various processors so that all processors are used efficiently) and **scaling** (breaking tasks into a number of subtasks compatible with the number of processors available).

Moreover, the advent of computer networks in which numerous machines are connected over great distances has led to the creation of software systems to coordinate the network's activities. Thus, the field of networking (which we will study in Chapter 4) is in many ways an extension of the subject of operating systems—the goal being to manage resources across many users on many machines rather than a single, isolated computer.

Still another direction of research in operating systems focuses on devices that are dedicated to specific tasks such as medical devices, vehicle electronics, home appliances, cell phones, or other hand-held computers. The computer

systems found in these devices are known as **embedded systems**. Embedded operating systems are often expected to conserve battery power, meet demanding, real-time deadlines, or operate continuously with little or no human oversight. Successes in this endeavor are marked by systems such as VxWORKS, developed by Wind River Systems and used in the Mars Exploration Rovers named Spirit and Opportunity; Windows CE (also known as Windows Embedded Compact), developed by Microsoft; and Blackberry Ltd's QNX, developed especially for use in hand-held devices and vehicles.

What's in a Smartphone?

As cell phones have become more powerful, it has become possible for them to offer services well beyond simply processing voice calls. A typical **smartphone** can now be used to text message, browse the Web, provide directions, view multimedia content—in short, it can be used to provide many of the same services as a traditional PC. As such, smartphones require full-fledged operating systems, not only to manage the limited resources of the smartphone hardware, but also to provide features that support the rapidly expanding collection of smartphone application software. The battle for dominance in the smartphone operating system marketplace promises to be fierce and will likely be settled on the basis of which system can provide the most imaginative features at the best price. Competitors in the smartphone operating system arena include Apple's iOS, Blackberry Ltd's Blackberry OS, Microsoft's Windows Mobile, and Google's Android.

3.1 Questions & Exercises

1. Identify examples of queues. In each case, indicate any situations that violate the FIFO structure.
2. Which of the following activities require real-time processing?
   a. Printing mailing labels
   b. Playing a computer game
   c. Displaying numbers on a smartphone screen as they are dialed
   d. Executing a program that predicts the state of next year's economy
   e. Playing an MP3 recording
3. What is the difference between embedded systems and PCs?
4. What is the difference between time-sharing and multitasking?