Live in fragments no longer. Only connect...

—Edward Morgan Forster—

Efficiency is getting the job done right. Effectiveness is getting the right job done.

—Zig Ziglar—

Nothing endures but change.

—Heraclitus—

Open sesame!

—The History of Ali Baba—

Chapter 1

Introduction to Operating Systems

Objectives

After reading this chapter, you should understand:

- *what an operating system is.*
- *a brief history of operating systems.*
- *a brief history of the Internet and the World Wide Web.*
- *core operating system components.*
- *goals of operating systems.*
- *operating system architectures.*

Chapter 1

Chapter Outline

1.13 Operating System Architectures

Operating Systems Thinking: Keep It Simple (KIS)

Anecdote: System Architect vs. System Engineer

Operating Systems Thinking: Architecture

1.13.1 Monolithic Architecture

1.13.2 Layered Architecture

1.13.3 Microkernel Architecture

1.13.4 Networked and Distributed Operating Systems

1.1 Introduction

Welcome to the world of operating systems. During the past several decades, computing has evolved at an unprecedented pace. Computer power continues to increase at phenomenal rates as costs decline dramatically. Today, computer users have desktop workstations that execute billions of instructions per second (BIPS), and supercomputers that execute over a trillion instructions per second have been built,[1, 2] numbers that just a few years ago were inconceivable.

Processors are becoming so inexpensive and powerful that computers can be employed in almost every aspect of our lives. On personal computers, users can edit documents, play games, listen to music, watch videos and manage their personal finances. Portable devices, including laptop computers, personal digital assistants (PDAs), cell phones, and MP3 players all have computers as key components. Wired and wireless network architectures are increasing our interconnectivity—allowing users to communicate instantly over vast distances. The Internet and World Wide Web have revolutionized business, creating demand for networks of large, powerful computers that service vast numbers of transactions per second. Networks of computers have become so powerful that they are used to conduct complex research and simulation projects, such as modeling the Earth's climate, emulating human intelligence and constructing lifelike 3D animations. Such pervasive and powerful computing is reshaping the roles and responsibilities of operating systems.

In this book, we review operating system principles and discuss cutting-edge advances in computing that are redefining operating systems. We investigate the structure and responsibilities of operating systems. Design considerations, such as performance, fault tolerance, security, modularity and cost, are explored in detail. We also address more recent operating system design issues arising from the rapid growth in distributed computing, made possible by the Internet and the World Wide Web.

We have worked hard to create what we hope will be an informative, entertaining and challenging experience for you. As you read this book, you may want to refer to our Web site at `www.deitel.com` for updates and additional information on each topic. You can reach us at `deitel@deitel.com`.

1.2 What Is an Operating System?

In the 1960s, the definition of an operating system might have been *the software that controls the hardware*. But the landscape of computer systems has evolved significantly since then, requiring a richer definition.

Today's hardware executes a great variety of software applications. To increase hardware utilization, applications are designed to execute concurrently. If these applications are not carefully programmed, they might interfere with one another. As a result, a layer of software called an **operating system** separates applications from the hardware they access and provides services that allow each application to execute safely and effectively.

An operating system is software that enables applications to interact with a computer's hardware. The software that contains the core components of the operating system is called the **kernel**. Operating systems can be found in devices ranging from cell phones and automobiles to personal and mainframe computers. In most computer systems, a user requests that the computer perform an action (e.g., execute an application or print a document) and the operating system manages the software and hardware to produce the desired result.

To most users, the operating system is a "black box" between the applications and the hardware they run on that ensures the proper result, given appropriate inputs. Operating systems are primarily resource managers—they manage hardware, including processors, memory, input/output devices and communication devices. They must also manage applications and other software abstractions that, unlike hardware, are not physical objects.

In the next several sections we present a brief history of operating systems from the simple, single-user batch systems of the 1950s to the complex, multiprocessor, distributed, multiuser platforms of today.

Self Review

1. (T/F) Operating systems manage only hardware.
2. What are the primary purposes of an operating system?

Ans: **1)** False. Operating systems manage applications and other software abstractions, such as virtual machines. **2)** The primary purposes of an operating system are to enable applications to interact with a computer's hardware and to manage a system's hardware and software resources.

1.3 Early History: The 1940s and 1950s

Operating systems have evolved over the last 60 years through several distinct phases or generations that correspond roughly to the decades (see the Operating Systems Thinking feature, Innovation).[3] In the 1940s, the earliest electronic digital computers did not include operating systems.[4,5,6] Machines of the time were so primitive that programmers often entered their machine-language programs one bit at a time on rows of mechanical switches. Eventually, programmers entered their machine-language programs on punched cards. Then, assembly languages—which used English-like abbreviations to represent the basic operations of the computer—were developed to speed the programming process.

General Motors Research Laboratories implemented the first operating system in the early 1950s for its IBM 701 computer.[7] The systems of the 1950s generally executed only one job at a time, using techniques that smoothed the transition between jobs to obtain maximum utilization of the computer system.[8] A **job** constituted the set of program instructions corresponding to a particular computational task, such as payroll or inventory. Jobs typically executed without user input for minutes, hours or days. These early computers were called **single-stream batch-pro-**

cessing systems, because programs and data were submitted in groups or batches by loading them consecutively onto tape or disk. A job stream processor read the job control language statements (that defined each job) and facilitated the setup of the next job. When the current job terminated, the job stream reader read in the control-language statements for the next job and performed appropriate housekeeping chores to ease the transition to the next job. Although operating systems of the 1950s reduced interjob transition times, programmers often were required to directly control system resources such as memory and input/output devices. This was slow, difficult and tedious work. Further, these early systems required that an entire program be loaded into memory for the program to run. This limited programmers to creating small programs with limited capabilities.[9]

Self Review

1. Why were assembly languages developed?
2. What limited the size and capabilities of programs in the 1950s?

Ans: **1)** Assembly languages were developed to speed the programming process. They enabled programmers to specify instructions as English-like abbreviations that were easier for humans to work with than machine-language instructions. **2)** The entire program had to be loaded into memory to execute. Because memory was relatively expensive, the amount of memory available to those computers was small.

Operating Systems Thinking

Innovation

Innovation is a fundamental challenge for operating systems designers. If we are going to make the massive investment required to produce new operating systems or new versions of existing operating systems, we must constantly be evaluating new technologies, new applications of computing and communications and new thinking about how systems should be built. We have provided thousands of citations and hundreds of Web resources for you to do additional readings on topics that are of interest to you. You should consider belonging to professional organizations like the ACM (`www.acm.org`), the IEEE (`www.ieee.org`) and USENIX (`www.usenix.org`) that publish journals on the latest research and development efforts in the computer field. You should access the Web frequently to follow important developments in the field. There is always a high degree of risk when innovating, but the rewards can be substantial.

1.4 The 1960s

The systems of the 1960s were also batch-processing systems, but they used the computer's resources more efficiently by running several jobs at once. Systems included many peripheral devices such as card readers, card punches, printers, tape drives and disk drives. Any one job rarely used all the system's resources efficiently. A typical job would use the processor for a certain period of time before performing an input/output (I/O) operation on one of the system's peripheral devices. At this point, the processor would remain idle while the job waited for the I/O operation to complete.

The systems of the 1960s improved resource utilization by allowing one job to use the processor while other jobs used peripheral devices. In fact, running a mixture of diverse jobs—some jobs that mainly used the processor (called **processor-bound jobs** or **compute-bound jobs**) and some jobs that mainly used peripheral devices (called **I/O-bound jobs**)—appeared to be the best way to optimize resource utilization. With these observations in mind, operating systems designers developed **multiprogramming** systems that managed several jobs at once.[10, 11, 12] In a multiprogramming environment, the operating system rapidly switches the processor from job to job, keeping several jobs advancing while also keeping peripheral devices in use. A system's **degree of multiprogramming** (also called its **level of multiprogramming**) indicates how many jobs can be managed at once. Thus, operating systems evolved from managing one job to managing several jobs at a time.

In multiprogrammed computing systems, resource sharing is one of the primary goals. When resources are shared among a set of processes, each process maintaining exclusive control over particular resources allocated to it, a process may be made to wait for a resource that never becomes available. If this occurs, that process will be unable to complete its task, perhaps requiring the user to restart it, losing all work that the process had accomplished to that point. In Chapter 7, Deadlock and Indefinite Postponement, we discuss how operating systems can deal with such problems.

Normally, users of the 1960s were not present at the computing facility when their jobs were run. Jobs were submitted on punched cards or computer tapes and remained on input tables until the system's human operator could load them into the computer for execution. Often, a user's job would sit for hours or even days before it could be processed. The slightest error in a program, even a missing period or comma, would "bomb" the job, at which point the (often frustrated) user would correct the error, resubmit the job and once again wait hours or days for the next attempt at execution. Software development in that environment was painstakingly slow.

In 1964, IBM announced its System/360 family of computers ("360" refers to all points on a compass to denote universal applicability).[13, 14, 15, 16] The various 360 computer models were designed to be hardware compatible, to use the OS/360 operating system and to offer greater computer power as the user moved upward in the series.[17] Over the years, IBM evolved its 360 architecture to the 370 series[18, 19] and, more recently, the 390 series[20] and the zSeries.[21]

More advanced operating systems were developed to service multiple **interactive users** at once. Interactive users communicate with their jobs during execution. In the 1960s, users interacted with the computer via "dumb terminals" (i.e., devices that supplied a user interface but no processor power) which were **online** (i.e., directly attached to the computer via an active connection). Because the user was present and interacting with it, the computer system needed to respond quickly to user requests; otherwise, user productivity could suffer. As we discuss in the Operating Systems Thinking feature, Relative Value of Human and Computer Resources, increased productivity has become an important goal for computers because human resources are extremely expensive compared to computer resources. **Timesharing** systems were developed to support simultaneous interactive users.[22]

Many of the timesharing systems of the 1960s were multimode systems that supported batch-processing as well as real-time applications (such as industrial process control systems).[23] **Real-time systems** attempt to supply a response within a certain bounded time period. For example, a measurement from a petroleum refinery indicating that temperatures are too high might demand immediate attention to

Operating Systems Thinking

Relative Value of Human and Computer Resources

In 1965, reasonably experienced programmers were earning about $4 per hour. Computer time on mainframe computers (which were far less powerful than today's desktop machines) was commonly rented for $500 or more per hour—and that was in 1965 dollars which, because of inflation, would be comparable to thousands of dollars in today's currency! Today, you can buy a top-of-the-line, enormously powerful desktop computer for what it cost to rent a far less powerful mainframe computer for one hour 40 years ago! As the cost of computing has plummeted, the cost of man-hours has risen to the point that today human resources are far more expensive than computing resources.

Computer hardware, operating systems and software applications are all designed to leverage people's time, to help improve efficiency and productivity. A classic example of this was the advent of timesharing systems in the 1960s in which these interactive systems (with almost immediate response times) often enabled programmers to become far more productive than was possible with the batch-processing systems response times of hours or even days. Another classic example was the advent of the graphical user interface (GUI) originally developed at the Xerox Palo Alto Research Center (PARC) in the 1970s. With cheaper and more powerful computing, and with the relative cost of people-time rising rapidly compared to that of computing, operating systems designers must provide capabilities that favor the human over the machine, exactly the opposite of what early operating systems did.

avert an explosion. The resources of a real-time system are often heavily underutilized—it is more important for such systems to respond quickly than it is for them to use their resources efficiently. Servicing both batch and real-time jobs meant that operating systems had to distinguish between types of users and provide each with an appropriate level of service. Batch-processing jobs could suffer reasonable delays, whereas interactive applications demanded a higher level of service and real-time systems demanded extremely high levels of service.

The key timesharing systems development efforts of this period included the **CTSS (Compatible Time-Sharing System)**[24, 25] developed by MIT, the **TSS (Time Sharing System)**[26] developed by IBM, the **Multics** system[27] developed at MIT, GE and Bell Laboratories as the successor to CTSS and the **CP/CMS (Control Program/Conversational Monitor System)**—which eventually evolved into IBM's **VM (Virtual Machine)** operating system—developed by IBM's Cambridge Scientific Center.[28, 29] These systems were designed to perform basic interactive computing tasks for individuals, but their real value proved to be the manner in which they shared programs and data and demonstrated the value of interactive computing in program development environments.

The designers of the Multics system were the first to use the term **process** to describe a program in execution in the context of operating systems. In many cases, users submitted jobs containing multiple processes that could execute concurrently. In Chapter 3, Process Concepts, we discuss how multiprogrammed operating systems manage multiple processes at once.

In general, concurrent processes execute independently, but multiprogrammed systems enable multiple processes to cooperate to perform a common task. In Chapter 5, Asynchronous Concurrent Execution, and Chapter 6, Concurrent Programming, we discuss how processes coordinate and synchronize activities and how operating systems support this capability. We show many examples of concurrent programs, some expressed generally in pseudocode and some in the popular Java™ programming language.

Turnaround time—the time between submission of a job and the return of its results—was reduced to minutes or even seconds. The programmer no longer needed to wait hours or days to correct even the simplest errors. The programmer could enter a program, compile it, receive a list of syntax errors, correct them immediately, recompile and continue this cycle until the program was free of syntax errors. Then the program could be executed, debugged, corrected and completed with similar time savings.

The value of timesharing systems in support of program development was demonstrated when MIT, GE and Bell Laboratories used the CTSS system to develop its own successor, Multics. Multics was notable for being the first major operating system written primarily in a high-level language (EPL—modeled after IBM's PL/1) instead of an assembly language. The designers of **UNIX** learned from this experience; they created the high-level language **C** specifically to implement UNIX. A family of UNIX-based operating systems, including Linux and Berkeley

Software Distribution (BSD) UNIX, have evolved from the original system created by Dennis Ritchie and Ken Thompson at Bell Laboratories in the late 1960s (see the Biographical Note, Ken Thompson and Dennis Ritchie).

TSS, Multics and CP/CMS all incorporated **virtual memory**, which we discuss in detail in Chapter 10, Virtual Memory Organization, and Chapter 11, Virtual Memory Management. In systems with virtual memory, programs are able to address more memory locations than are actually provided in main memory, also called real memory or physical memory.[30,31] (Real memory is discussed in Chapter 9, Real Memory Organization and Management.) Virtual memory systems help remove much of the burden of memory management from programmers, freeing them to concentrate on application development.

Biographical Note

Ken Thompson and Dennis Ritchie

Ken Thompson and Dennis Ritchie are well known in the field of operating systems for their development of the UNIX operating system and the C programming language. They have received several awards and recognition for their accomplishments, including the ACM Turing Award, the National Medal of Technology, the NEC C&C Prize, the IEEE Emmanuel Piore Award, the IEEE Hamming Medal, induction into the United States National Academy of Engineering and the Bell Labs National Fellowship.[32]

Ken Thompson attended the University of California at Berkeley, where he earned a B.S. and M.S. in Computer Science, graduating in 1966.[33] After college Thompson worked at Bell Labs, where he eventually joined Dennis Ritchie on the Multics project.[34] While working on that project, Thompson created the B language that led to Ritchie's C language.[35] The Multics project eventually led to the creation of the UNIX operating system in 1969. Thompson continued to develop UNIX through the early 1970s, rewriting it in Ritchie's C programming language.[36] After Thompson completed UNIX, he made news again in 1980 with Belle. Belle was a chess-playing computer designed by Thompson and Joe Condon that won the World Computing Chess Championship. Thompson worked as a professor at the University of California at Berkeley and at the University of Sydney, Australia. He continued to work at Bell Labs until he retired in 2000.[37]

Dennis Ritchie attended Harvard University, earning a Bachelor's degree in Physics and a Ph.D. in Mathematics. Ritchie went on to work at Bell Labs, where he joined Thompson on the Multics project in 1968. Ritchie is most recognized for his C language, which he completed in 1972.[38] Ritchie added some extra capabilities to Thompson's B language and changed the syntax to make it easier to use. Ritchie still works for Bell Labs and continues to work with operating systems.[39] Within the past 10 years he has created two new operating systems, Plan 9 and Inferno.[40] The Plan 9 system is designed for communication and production quality. [41]Inferno is a system intended for advanced networking.[42]

Once loaded into main memory, programs could execute quickly; however, main memory was far too expensive to contain large numbers of programs at once. Before the 1960s, jobs were largely loaded into memory using punched cards or tape, a tedious and time-consuming task, during which the system could not be used to execute jobs. The systems of the 1960s incorporated devices that reduced system idle time by storing large amounts of rewritable data on relatively inexpensive magnetic storage media such as tapes, disks and drums. Although hard disks enabled relatively fast access to programs and data compared to tape, they were significantly slower than main memory. In Chapter 12, Disk Performance Optimization, we discuss how operating systems can manage disk input/output requests to improve performance. In Chapter 13, File and Database Systems, we discuss how operating systems organize data into named collections called files and manage space on storage devices such as disks. We also discuss how operating systems protect data from access by unauthorized users and prevent data from being lost when system failures or other catastrophic events occur.

Self Review

1. How did interactive computing and its improvement in turnaround time increase programmer productivity?
2. What new concept did TSS, Multics and CP/CMS all incorporate? Why was it so helpful for programmers?

Ans: **1)** The time between submission of a job and the return of its results was reduced from hours or days to minutes or even seconds. This enabled programmers to interactively enter, compile and edit programs until their syntax errors were eliminated, then use a similar cycle to test and debug their programs. **2)** TSS, Multics, and CP/CMS all incorporated virtual memory. Virtual memory allows applications access to more memory than is physically available on the system. This allows programmers to develop larger, more powerful applications. Also, virtual memory systems remove much of the memory management burden from the programmer.

1.5 The 1970s

The systems of the 1970s were primarily multimode multiprogramming systems that supported batch processing, timesharing and real-time applications. Personal computing was in its incipient stages, fostered by early and continuing developments in microprocessor technology.[43] The experimental timesharing systems of the 1960s evolved into solid commercial products in the 1970s. Communications between computer systems throughout the United States increased as the Department of Defense's TCP/IP communications standards became widely used—especially in military and university computing environments.[44, 45, 46] Communication in local area networks (LANs) was made practical and economical by the Ethernet standard developed at Xerox's Palo Alto Research Center (PARC).[47, 48] In Chapter 16, Introduction to Networking, we discuss TCP/IP, Ethernet and fundamental networking concepts.

Security problems increased with the growing volumes of information passing over vulnerable communications lines (see the Anecdote, Abraham Lincoln's Technology Caution). Encryption received much attention—it became necessary to encode proprietary or private data so that, even if the data was compromised, it was of no value to anyone other than the intended receivers. In Chapter 19, Security, we discuss how operating systems secure sensitive information from unauthorized access. During the 1970s, operating systems grew to encompass networking and security capabilities and continued to improve in performance to meet commercial demands.

The personal computing revolution began in the late 1970s with such systems as the Apple II, and exploded in the 1980s.

Self Review

1. What developments in the 1970s improved communication between computer systems?
2. What new problem was introduced by the increased communication between computers? How was this problem addressed?

Ans: **1)** The DoD's TCP/IP standards became widely used in network communications—primarily in university and military computing environments. Also, Xerox's PARC developed the Ethernet standard, which made relatively high-speed local area networks (LANs) practical and economical. **2)** Communication between computers introduced security problems because data was sent over vulnerable communication lines. Encryption was employed to make data unreadable to anyone other than the intended recipient.

Anecdote

Abraham Lincoln's Technology Caution

The story goes that during the Civil War one of President Lincoln's young lieutenants came running up to him eager to speak with the President. "What is it, lieutenant?" "Mr. President, Mr. President, we are wiring the battlefields for this wonderful new technology called the telegraph. Do you know what that means, Mr. President?" "No lieutenant, what does it mean?" "Mr. President, it means we'll be able to make decisions at the speed of light!" The older and wiser President Lincoln looked down at the lieutenant and said calmly, "Yes, lieutenant, but we'll also be able to make wrong decisions at the speed of light!"

Lessons to operating systems designers: Every new technology you will evaluate has its pros and cons. You will inevitably spend a great deal of your time concerned with performance issues. But, making things happen faster may have unpleasant consequences.

1.6 The 1980s

The 1980s was the decade of the personal computer and the workstation.[49] Microprocessor technology evolved to the point where high-end desktop computers called workstations could be built that were as powerful as the mainframes of a decade earlier. The IBM Personal Computer released in 1981 and the Apple Macintosh personal computer released in 1984 made it possible for individuals and small businesses to have their own dedicated computers. Communication facilities could be used to transmit data quickly and economically between systems. Rather than bringing data to a central, large-scale computer installation for processing, computing was distributed to the sites at which it was needed. Software such as spreadsheet programs, word processors, database packages and graphics packages helped drive the personal computing revolution by creating demand from businesses that could use these products to increase their productivity.

Personal computers proved to be relatively easy to learn and use, partially because of **graphical user interfaces (GUI)** that used graphical symbols such as windows, icons and menus to facilitate user interaction with programs. Xerox's Palo Alto Research Center (PARC) developed the mouse and GUI (for more on the origins of the mouse, see the Biographical Note, Doug Engelbart); Apple's release of the Macintosh personal computer in 1984 popularized their use. In Macintosh computers, the GUI was embedded in the operating system so that all applications would have a similar look and feel.[50] Once familiar with the Macintosh GUI, the user could learn to use new applications faster.

As technology costs declined, transferring information between computers in computer networks became more economical and practical. Electronic mail, file transfer and remote database access applications proliferated. **Distributed computing** (i.e., using multiple independent computers to perform a common task) became widespread under the client/server model. **Clients** are user computers that request various services; **servers** are computers that perform the requested services. Servers often are dedicated to one type of task, such as rendering graphics, managing databases or serving Web pages.

The software engineering field continued to evolve, a major thrust coming from the United States government aimed at providing tighter control of Department of Defense software projects.[51] Some goals of the initiative included realized code reusability and the early construction of prototypes so developers and users could suggest modifications early in the software design process.[52]

Self Review

1. What aspect of personal computers, popularized by the Apple Macintosh, made them especially easy to learn and use?
2. (T/F) A server cannot be a client.

Ans: **1)** Graphical User Interfaces (GUIs) facilitated personal computer use by providing an easy-to-use, uniform interface to every application. This enabled users to learn new applications faster. **2)** False. A computer can be a client and server. For example, a Web server can

be both a client and server. When users request a Web page, it is a server; if the server then requests information from a database system, it becomes a client of the database system.

1.7 History of the Internet and World Wide Web

In the late 1960s **ARPA**—the **Advanced Research Projects Agency** of the Department of Defense rolled out the blueprints for networking the main computer systems of about a dozen ARPA-funded universities and research institutions. They were to be connected with communications lines operating at a then-stunning 56 kilobits per second (Kbps)—1 Kbps is equal to 1,000 bits per second—at a time when most people (of the few who could be) were connecting over telephone lines to computers at a rate of 110 bits per second. HMD vividly recalls the excitement at

Biographical Note

Doug Engelbart

Doug Engelbart invented the computer mouse and was one of the primary designers of the original graphical displays and windows.

Engelbart's background was in electronics. During World War II he worked as an electronics technician on a variety of systems including RADAR and SONAR.[53] After leaving the military, he went back to Oregon State to complete a degree in Electrical Engineering in 1948.[54] He went on to receive his Ph.D. from the University of California at Berkeley, then took a job at the Stanford Research Institute (SRI), where he gained his first experience with computers.[55] In 1968, at the Joint Computer Conference in San Francisco, Engelbart and his coworkers displayed their computer system, NLS (oNLine System) which featured Engelbart's computer mouse and a graphical interface with windows.[56] This original mouse, called an X-Y Position Indicator for a Display System, had only one button.[57] The mouse had two wheels on the bottom, one horizontal and one vertical, to detect movement.[58] The mouse and the graphical windows were interdependent. The mouse made it significantly easier to switch between windows, and without windows the mouse was not as useful.

Engelbart has dedicated his life to augmenting human intellect. His original idea behind the NLS system was to create a system that could help people solve problems faster and enhance intelligence. Engelbart founded the Bootstrap Institute to foster worldwide awareness of his mission. Bootstrapping, according to Engelbart, is the idea of improving one's methods of improvement. He believes this is the best way to improve human intelligence.[59]

Today, Engelbart is still working with the Bootstrap Institute. He has received recognition for his work including the Lemelson-MIT Prize, the National Medal of Technology and induction into the National Inventors Hall of Fame.[60]

that conference. Researchers at Harvard talked about communicating with the Univac 1108 "supercomputer" across the country at the University of Utah to handle the massive computations related to their computer graphics research. Academic research was about to take a giant leap forward. Shortly after this conference, ARPA proceeded to implement what quickly became called the **ARPAnet**—the grandparent of today's **Internet**.

Although the ARPAnet did enable researchers to network their computers, its chief benefit proved to be its capability for quick and easy communication via what came to be known as electronic mail (e-mail). This is true even on the Internet today, with e-mail, instant messaging and file transfer facilitating communications among hundreds of millions of people worldwide and growing rapidly.

The ARPAnet was designed to operate without centralized control. This meant that if a portion of the network should fail, the remaining working portions would still be able to route data packets from senders to receivers over alternative paths.

The protocols (i.e., sets of rules) for communicating over the ARPAnet became known as the **Transmission Control Protocol/Internet Protocol (TCP/IP)**. TCP/IP was used to manage communication between applications. The protocols ensured that messages were routed properly from sender to receiver and that those messages arrived intact. The advent of TCP/IP promoted worldwide computing growth. Initially, Internet use was limited to universities and research institutions; later, the military adopted the technology.

Eventually, the government decided to allow access to the Internet for commercial purposes. This decision led to some concern among the research and military communities—it was felt that response times would suffer as "the Net" became saturated with users. In fact, the opposite occurred. Businesses rapidly realized that they could use the Internet to tune their operations and to offer new and better services to their clients. Companies spent vast amounts of money to develop and enhance their Internet presence. This generated intense competition among communications carriers, hardware suppliers and software suppliers to meet the increased infrastructure demand. The result is that **bandwidth** (i.e., the information-carrying capacity of communications lines) on the Internet has increased tremendously, and hardware and communications costs have plummeted.

The **World Wide Web (WWW)** allows computer users to locate and view multimedia-based documents (i.e., documents with text, graphics, animation, audio or video) on almost any subject. Although the Internet was developed more than three decades ago, the introduction of the World Wide Web (WWW) was a relatively recent event. In 1989, Tim Berners-Lee of CERN (the European Center for Nuclear Research) began to develop a technology for sharing information via hyperlinked text documents (see the Biographical Note, Tim Berners-Lee). To implement this new technology, Berners-Lee created the **HyperText Markup Language (HTML)**. Berners-Lee also implemented the **Hypertext Transfer Protocol (HTTP)** to form the communications backbone of his new hypertext information system, which he called the World Wide Web.

Surely, historians will list the Internet and the World Wide Web among the most important and profound creations of humankind. In the past, most computer applications ran on "stand-alone" computers (computers that were not connected to one another). Today's applications can be written to communicate among the world's hundreds of millions of computers. The Internet and World Wide Web merge computing and communications technologies, expediting and simplifying our work. They make information instantly and conveniently accessible to large numbers of people. They enable individuals and small businesses to achieve worldwide exposure. They are changing the way we do business and conduct our personal lives. And they are changing the way we think of building operating systems. Today's operating systems provide GUIs that enable users to "access the world" over the Internet and the Web as seamlessly as accessing the local system. The operating systems of the 1980s were concerned primarily with managing resources on the local computer. Today's distributed operating systems may utilize resources on computers worldwide. This creates many interesting challenges that we discuss throughout the book, especially in Chapters 16–19, which examine networking, distributed computing and security.

Biographical Note

Tim Berners-Lee

The World Wide Web was invented by Tim Berners-Lee in 1990. The Web allows computer users to locate and view multimedia-based documents (i.e., documents with text, graphics, animation, audio or video) on almost any subject.

Berners-Lee graduated from Queen's College at Oxford University with a degree in Physics in 1976. In 1980 he wrote a program called Enquire, which used hypertext links to help him quickly navigate the numerous documents in a large project. He entered into a fellowship at the European Center for Nuclear Research (CERN) in 1984, where he gained experience in communication software for real-time networked systems.[61, 62, 63]

Berners-Lee invented HTTP (the HyperText Transfer Protocol), HTML (Hypertext Markup Language) and the first World Wide Web server and browser in 1989, while working at CERN.[64, 65] He intended the Web to be a mechanism for open, available access to all shared knowledge and experience.[66]

Until 1993, Berners-Lee individually managed changes and suggestions for HTTP and HTML, sent from the early Web users. By 1994 the Web community had grown large enough that he started the World Wide Web Consortium (W3C; `www.w3.org`) to monitor and establish Web technology standards.[67] As director of the organization, he actively promotes the principle of freely available information accessed by open technologies.[68]

Self Review

1. How did the ARPAnet differ from traditional computer networks? What was its primary benefit?
2. What creations did Berners-Lee develop to facilitate data sharing over the Internet?

Ans: **1)** The ARPAnet was decentralized, so the network continued to be able to pass information even if portions of the network failed. The primary benefit of the ARPAnet was its capability for quick and easy communication via e-mail. **2)** Berners-Lee developed the HyperText Markup Language (HTML) and the Hypertext Transfer Protocol (HTTP), making possible the World Wide Web.

1.8 The 1990s

Hardware performance continued to improve exponentially in the 1990s.[69] By the end of the decade, a typical personal computer could execute several hundred million instructions per second (MIPS) and store over a gigabyte of information on a hard disk; some supercomputers could execute over a trillion operations per second.[70] Inexpensive processing power and storage enabled users to execute large, complex programs on personal computers and enabled small- to mid-size companies to use these economical machines for the extensive database and processing jobs that were once delegated to mainframe systems. Falling technology costs also led to an increase in the number of home computers, which were used both for work and for entertainment.

In the 1990s, the creation of the World Wide Web led to an explosion in the popularity of distributed computing. Originally, operating systems performed isolated resource management inside a single computer. With the creation of the World Wide Web and increasingly fast Internet connections, distributed computing became commonplace among personal computers. Users could request data stored at remote locations or request that programs run on distant processors. Large organizations could use distributed multiprocessors (i.e., networks of computers containing more than one processor) to scale resources and increase efficiency.[71] Distributed applications, however, were still limited by the fact that communication over a network occurred at relatively slow speeds compared to the internal processing speeds of individual computers. Distributed computing is discussed in detail in Chapter 17, Introduction to Distributed Systems, and Chapter 18, Distributed Systems and Web Services.

As demand for Internet connections grew, operating system support for networking tasks became standard. Users at home and in organizations increased productivity by accessing the resources on networks of computers. However, increased connectivity led to a proliferation of computer security threats. Operating system designers developed techniques to protect computers from these malicious attacks. Ever more sophisticated security threats continued to challenge the computer industry's ability to counter such attacks.

Microsoft Corporation became dominant in the 1990s. In 1981, Microsoft released the first version of its DOS operating system for the IBM personal computer. In the mid-1980s, Microsoft developed the Windows operating system, a graphical user interface built on top of the DOS operating system. Microsoft released Windows 3.0 in 1990; this new version featured a user-friendly interface and rich functionality. The Windows operating system became incredibly popular after the 1993 release of Windows 3.1, whose successors, Windows 95 and Windows 98, virtually cornered the desktop operating system market by the late 90s. These operating systems, which borrowed from many concepts (such as icons, menus and windows) popularized by early Macintosh operating systems, enabled users to navigate multiple concurrent applications with ease. Microsoft also entered the corporate operating system market with the 1993 release of Windows NT, which quickly became the operating system of choice for corporate workstations.[72] Windows XP, which is based on the Windows NT operating system, is discussed in Chapter 21, Case Study: Windows XP.

Object Technology

Object technology became popular in many areas of computing, as the number of applications written in object-oriented programming languages, such as C++ or Java, increased steadily. Object concepts also facilitated new approaches to computing. Each software object encapsulates a set of attributes and a set of actions. This allows applications to be built with components that can be reused in many applications, reducing software development time. In **object-oriented operating systems (OOOS)**, objects represent components of the operating system and system resources.[73] Object-oriented concepts such as inheritance and interfaces were exploited to create modular operating systems that were easier to maintain and extend than operating systems built with previous techniques. Modularity facilitates operating system support to new and different architectures. The demand for object integration across multiple platforms and languages led to support for objects in programming languages such as Sun's Java and Microsoft's .NET languages (e.g., Visual Basic .NET, Visual C++ .NET and C#).

Open-Source Movement

Another development in the computing community (particularly in the area of operating systems) during the 1990s was the movement toward **open-source software**. Most software is created by writing source code in a high-level programming language. However, most commercial software is sold as object code (also called machine code or binaries)—the compiled source code that computers can understand. The source code is not included, enabling vendors to hide proprietary information and programming techniques. However, free and open-source software became increasingly common in the 1990s. Open-source software is distributed with the source code, allowing individuals to examine and modify the software before compiling and executing it. For example, the Linux operating system and the Apache Web

server, both of which are free and open source, were downloaded and installed by millions of users during the 1990s, and the number of downloads is increasing rapidly in the new millennium.[74] Linux, created by Linus Torvalds (see the Biographical Note, Linus Torvalds), is discussed in Chapter 20, Case Study: Linux.

In the 1980s, Richard Stallman, a software developer at MIT, launched a project to recreate and extend most of the tools for AT&T's UNIX operating system and to make the code available at no charge. Stallman (see the Biographical Note, Richard Stallman) founded the Free Software Foundation and created the **GNU** project—which is a recursive name that stands for "*G*NU's *N*ot *U*NIX"—because he disagreed with the concept of selling the permission to use software.[75] He believed that granting users the freedom to modify and distribute software would lead to better software, driven by user needs instead of personal or corporate profit. When Linus Torvalds created the original version of the Linux operating system, he employed many of the tools published by GNU for free under the **General Public License (GPL)**. The GPL, published online at `www.gnu.org/licenses/gpl.html`, specifies that anyone can freely modify and redistribute software under its license, provided that the modifications are clearly indicated and any derivative of the software is also distributed under the GPL.[76] Although most GPL-licensed software is available free of charge, the GPL requires only that its software be free in the sense that users can freely modify and redistribute it. Therefore, vendors can charge a fee for providing GPL-licensed software and its source code, but cannot

Biographical Note

Linus Torvalds

Linus Torvalds was born in 1969 in Helsinki, Finland. As a child he taught himself how to program by playing with a Commodore VIC-20. In 1988 he entered the University of Helsinki to study computer science. While there, he wrote a UNIX clone based on Professor Andrew Tanenbaum's Minix to run on his new PC.[77, 78] In 1991 he completed the first version of the basic Linux kernel, which ran on the Intel 80386 processor.[79] He distributed Linux under the GNU Public License (GPL)[80] as open-source code and gladly accepted additions, corrections and free programs from other programmers.[81,82] By 1994 Linux had accrued enough applications to be a complete, usable operating system, and version 1.0 was released.[83] Programmers and professors using UNIX on large systems liked Linux because it brought the features and power of UNIX to inexpensive desktop systems for free.[84, 85]

Torvalds is currently a fellow of the Open Source Development Labs (OSDL), which funds his full-time work on the kernel. He continues to lead the open-source Linux project, managing changes and releasing new versions of the kernel.[86, 87] Linux has become one of the largest and best-known open-source developments in computing history and has become particularly successful in the server market. Linux is discussed in Chapter 20, Case Study: Linux.

prevent end users from modifying and redistributing them. In the late 90s, the **Open Source Initiative (OSI)** was founded to protect open-source software and promote the benefits of open-source programming (see `www.opensource.org`).

Open-source software facilitates enhancements to software products by permitting anyone in the developer community to test, debug and enhance applications. This increases the chance that subtle bugs, which could otherwise be security risks or logic errors, are caught and fixed. Also, individuals and corporations can modify the source to create custom software that meets the needs of a particular environment. Many open-source software vendors remain profitable by charging individuals and organizations for technical support and customizing software.[88] Though most systems in the 1990s still ran proprietary operating systems, such as IBM mainframe operating systems, UNIX systems, Apple's Macintosh and

Biographical Note

Richard Stallman

Richard Stallman was the original developer of the GNU project, started in the 1980s to create free software. Stallman graduated from Harvard in 1974 with a degree in Physics.[89] While at Harvard, he worked at the MIT Artificial Intelligence Lab. After graduating, he continued at the lab, where he and his colleagues worked with shared software.[90] The idea was that someone receiving someone else's executable program would also receive its source code. This was advantageous because a programmer could add more functionality to someone else's program.[91] Stallman's specific job was to modify and improve the ITS operating system the lab used. However, as the 1980s arrived, there was little shared software.[92] The lab's new operating system was not shared. Stallman became frustrated with operating systems, drivers, and the like, that he could no longer modify.[93] In 1984 he left the MIT Artificial Intelligence Lab to work on a new shared operating system, which he called GNU (GNU's Not UNIX).[94]

As interest in Stallman's GNU project grew, he created the Free Software Foundation (FSF) in 1985 to promote free software and continue to develop the GNU operating system.[95] Stallman and his associates at FSF created a number of GNU programs, including GNU Emacs (a text editor), GCC (a C compiler) and GDB (a debugger), to name a few.[96] In 1992 Stallman used the Linux kernel to complete his system. The system, known as GNU/Linux, is a fully functional operating system and includes a variety of programs.[97]

Stallman has received numerous awards and recognition for his work, including the Grace Hopper Award, MacArthur Foundation Fellowship, the Electric Frontier Foundation Pioneer Award, the Yuri Rubinksi Award, the Takeda Award, election to the National Academy of Engineering and two honorary doctorates from the Institute of Technology in Sweden and the University of Glasgow. Stallman continues to promote the free software cause and speaks about free software all over the world.[98]

Microsoft's Windows, open-source operating systems, such as Linux, FreeBSD and OpenBSD, became viable competition. In the future, they will undoubtedly continue to gain ground on proprietary solutions as a result of product improvement, industry standardization, interoperability, product customization and cost savings.

In the 1990s, operating systems became increasingly user friendly. The GUI features that Apple built into its Macintosh operating system in the 1980s became more sophisticated in the 1990s. "Plug-and-play" capabilities were built into operating systems, enabling users to add and remove hardware components dynamically without manually reconfiguring the operating system. Operating systems also maintained user profiles—serving authentication needs and enabling per-user customization of the operating system interface.

Self Review

1. How did object-oriented technology affect operating systems?
2. What are some of the benefits of open-source development?

Ans: **1)** Operating systems designers could reuse objects when developing new components. Increased modularity due to object-oriented technology facilitated operating system support for new and different architectures. **2)** Open-source software can be viewed and modified by anyone in the software development community. Because these people constantly test, debug and use the software, there is a greater chance that bugs will be found and fixed. Also, open-source software enables users and organizations to modify a program to meet their particular needs.

1.9 2000 and Beyond

In the current decade, **middleware**, which is software that links two separate applications (often over a network), has become vital as applications are published on the World Wide Web and consumers use them via affordable, high-speed Internet connections over cable television lines and digital subscriber lines (DSL). Middleware is common in Web applications, in which a Web server (the application that sends data to the user's Web browser) must generate content to satisfy a user's request with the help of a database. The middleware acts as a courier to pass messages between the Web server and the database, simplifying communication between multiple different architectures. **Web services** encompass a set of related standards that can enable any two computer applications to communicate and exchange data via the Internet. A Web service communicates over a network to supply a specific set of operations that other applications can invoke. The data is passed back and forth using standard protocols such as HTTP, the same protocol used to transfer ordinary Web pages. Web services operate using open, text-based standards that enable components written in different languages and on different platforms to communicate. They are ready-to-use pieces of software on the Internet.

Web services will help drive the shift toward true distributed computing. For example, the online retailer Amazon.com allows developers to build online stores that search Amazon's product databases and display detailed product information

via Amazon.com Web Services (`www.amazon.com/gp/aws/landing.html`). The Google search engine also can be integrated with other functionality through the Google Web APIs (`www.google.com/apis`), which connect to Google's indices of Web sites using Web services. We discuss Web services in more detail in Chapter 18, Distributed Systems and Web Services.

Multiprocessor and network architectures are creating numerous opportunities for research and development of new hardware and software design techniques. Sequential programming languages that specify one computation at a time are now complemented by concurrent programming languages, such as Java, that enable the specification of parallel computations; in Java the units of parallel computing are specified via threads. We discuss threads and the technique of multithreading in Chapter 4, Thread Concepts.

An increasing number of systems exhibit **massive parallelism**; they have large numbers of processors so that many independent parts of computations can be performed in parallel. This is dramatically different in concept from the sequential computing of the past 60 years; there are significant and challenging problems in developing the software appropriate for dealing with such parallelism. We discuss parallel computing architectures in Chapter 15, Multiprocessor Management.

Operating systems are standardizing user and application interfaces so that they are easier to use and support a greater number of programs. Microsoft has already merged the consumer and professional lines of its Windows operating system into Windows XP. In its next operating system (code-named Longhorn), Microsoft plans to integrate the formats of different types of files. This will, for example, allow users to search their systems for all files (documents, spreadsheets, e-mails, etc.) containing certain keywords. Longhorn will also include an enhanced 3D user interface, improved security and support for recordable digital versatile discs (DVDs).[99,100] Open-source operating systems, such as Linux, will become more widely used and will employ standard application programming interfaces (APIs) such as the **Portable Operating System Interface (POSIX)** to improve compatibility with other UNIX-based operating systems.

Computing on mobile devices, such as cell phones and PDAs, will become more common as mobile devices are equipped with increasingly powerful processors. Today, these devices are used for such functions as e-mail, Web browsing and digital imaging. Resource-intensive applications, such as full-motion video, will proliferate on these devices. Because mobile device resources are limited by the devices' small size, distributed computing will play an even larger role, as PDAs and cell phones will request increasing amounts of data and processing power from remote computers.

Self Review

1. What technologies can be used to bridge the gap between different operating systems? How would these technologies make it possible to execute the same application on multiple platforms?
2. Why is distributed computing useful for computations performed by mobile devices?

Ans: **1)** Virtual machines and operating system emulators bridge the gap between different operating systems. Applications can be written once to use the functionality of the virtual machine or emulator. The virtual machines or emulators can be implemented to hide the representation of the underlying platform from the applications. **2)** Distributed computing allows a mobile device to delegate jobs to other machines with more resources. The mobile device, having limited resources and battery life, can request data and processing power from larger computers across a network.

1.10 Application Bases

When the IBM Personal Computer (often called simply "the PC") appeared in 1981, it immediately spawned a huge software industry in which **independent software vendors (ISVs)** were able to market packages for the IBM PC to run under the MS-DOS operating system (IBM's version was called DOS). Operating systems free applications software developers from having to deal with the messy details of manipulating computer hardware to manage memory, perform input/output, deal with communication lines, and so on. The operating system provides a series of **application programming interface (API)** calls which applications programmers use to accomplish detailed hardware manipulations and other operations. The API provides **system calls** by which a user program instructs the operating system to do the work; the application developer simply has to know what routines to call to accomplish specific tasks (Fig. 1.1). Note that in Fig. 1.1, the area above the dashed line, user space, indicates software components that are not part of the operating system and cannot directly access the system's physical resources. The area below the dashed line, kernel space, indicates software components that are part of the operating system and have unrestricted access to system resources. We frequently use this convention in our diagrams to indicate the privilege with which software components execute. If an application attempts to misuse system resources, or if the

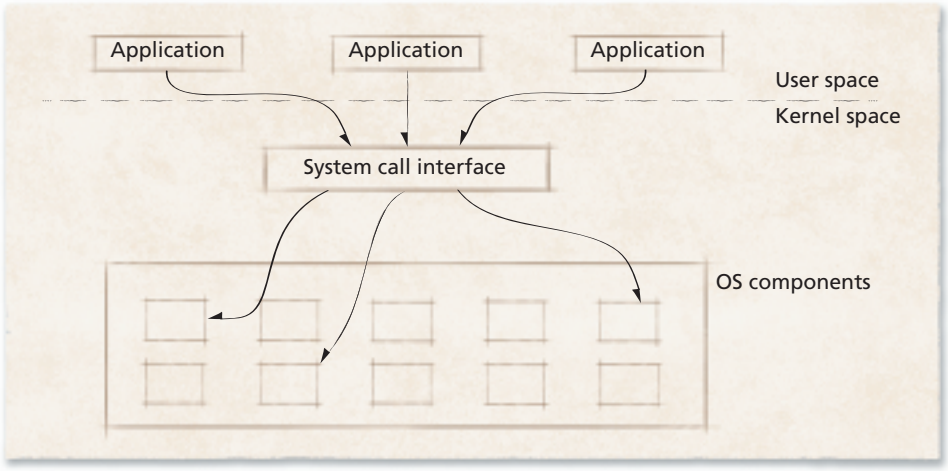

Figure 1.1 | *Interaction between applications and the operating system.*

application attempts to use resources that it has not been granted, the operating system must intervene to prevent the application from damaging the system or interfering with other user applications.

If an operating system presents an environment conducive to developing applications quickly and easily, the operating system and the hardware are more likely to be successful in the marketplace. The applications development environment created by MS-DOS encouraged the development of tens of thousands of application software packages. This in turn encouraged users to buy IBM PCs and compatibles. Windows could well have an application base of a hundred thousand applications.

Once an **application base** (i.e., the combination of the hardware and the operating system environment in which applications are developed) is widely established, it becomes extremely difficult to ask users and software developers to convert to a completely new application development environment provided by a dramatically different operating system. Thus, it is likely that new architectures evolving over the next several years will make every effort to support the existing major application bases.

1.11 Operating System Environments

This book focuses on operating system concepts related to general-purpose computers with a range of resources, including sizable amounts of main memory, high processor speeds, high-capacity disks, various peripheral devices, and so on. Such computers are typically used as personal computers or as workstations.

Many of the concepts that apply to general-purpose computers also apply to high-end Web and database servers, which contain high-performance hardware. Operating systems intended for high-end environments must be designed to support large main memories, special-purpose hardware, and large numbers of processes. We discuss these considerations in Chapter 15, Multiprocessor Management.

Embedded systems provide a different operating system design challenge. They are characterized by a small set of specialized resources that provide functionality to devices such as cell phones and PDAs. In embedded environments, efficient resource management is the key to building a successful operating system. Storage is often limited, so the operating system must provide services using a minimal amount of code. Considerations such as power management and the need for user-friendly interfaces create other challenges in embedded operating system design.

Real-time systems require that tasks be performed within a particular (often short) time frame. For example, the autopilot feature of an aircraft must constantly adjust speed, altitude and direction. Such actions cannot wait indefinitely—and sometimes cannot wait at all—for other nonessential tasks to complete. Real-time operating systems must enable processes to respond immediately to critical events. Soft real-time systems ensure that real-time tasks execute with high priority, but do not guarantee which, if any, of these tasks will complete on time. Hard real-time systems guarantee that all of their tasks complete on time. We discuss how Linux and Windows XP

handle real-time applications in Chapters 20 and 21, respectively. These systems are found in many settings including robotics, avionics and other system control applications. Often, they are used in **mission-critical systems**, where the system fails to meet its objectives (i.e., mission) if any of its tasks are not successfully completed on time. In mission-critical systems such as those for air traffic control, nuclear reactor monitoring and military command and control, people's lives could be at risk.

Business-critical systems, such as Web servers and databases, must consistently meet their objectives. In e-business, this could mean guaranteeing fast response times to users purchasing products over the Internet; in large corporations, it could mean enabling employees to share information efficiently and ensuring that important information is protected from problems such as power failures and disk failures. Unlike mission-critical systems, the business does not necessarily fail if a business-critical system does not always meet its objectives.

Some operating systems must manage hardware that may or may not physically exist in the machine. A **virtual machine (VM)** is a software abstraction of a computer that often executes as a user application on top of the native operating system.[101] A virtual machine operating system manages the resources provided by the virtual machine. One application of virtual machines is to allow multiple instances of an operating system to execute concurrently. Another is emulation—using software or hardware that mimics the functionality of hardware or software not present in the system.

Virtual machines interface with the hardware in a system via the underlying operating system; other user programs can interact with VMs. A VM can create software components that represent the contents of physical systems—such as processors, memory, communication channels, disks and clocks (Fig. 1.2).[102] This allows multiple users to share hardware under the illusion of being serviced by a dedicated machine. By providing this illusion, virtual machines promote **portability**, the ability for software to run on multiple platforms.

The **Java Virtual Machine (JVM)** is one of the most widely used virtual machines. The JVM is the foundation of the Java platform and allows Java applications to execute on any JVM of the correct version, regardless of the platform on which the JVM is installed. The company VMware Software also provides virtual machines, particularly for the Intel architecture, enabling owners of Intel x86-based computers to run operating systems such as Linux and Windows concurrently on one computer (each virtual machine appears in its own window).[103]

Virtual machines tend to be less efficient than real machines because they access the hardware indirectly (or simulate hardware that is not actually connected to the computer). Indirect or simulated hardware access increases the number of software instructions required to perform each hardware action.[104]

Self Review

1. What type of system would a temperature monitor in a nuclear power plant probably be described as? Why?
2. Describe the advantages and disadvantage of virtual machines.

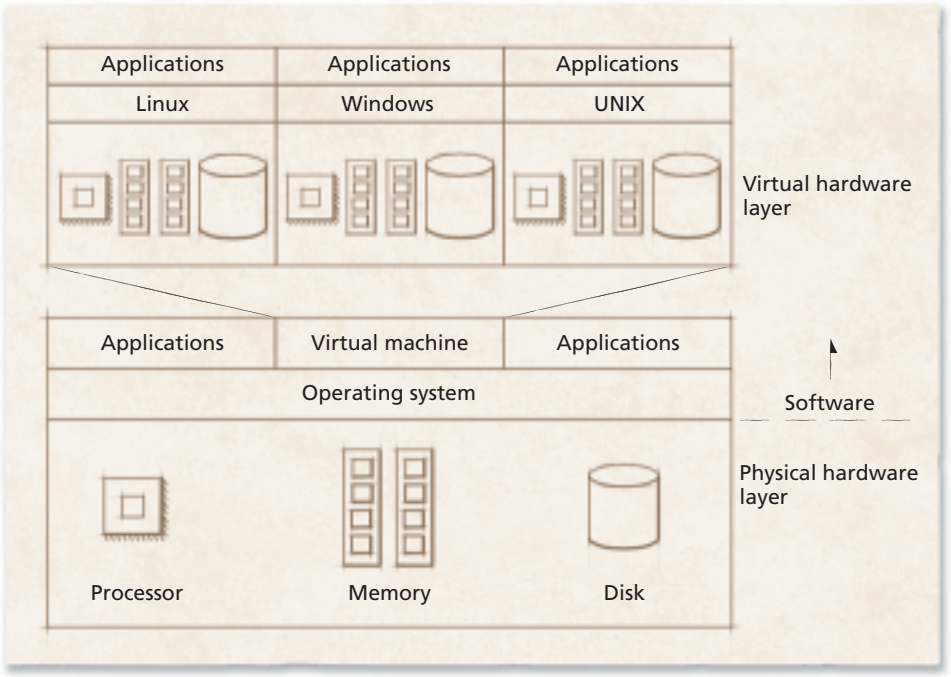

Figure 1.2 | *Schematic of a virtual machine.*

Ans: **1)** A hard real-time system would monitor the temperature in a nuclear power plant to ensure that it is always in an appropriate range, and would notify operators in real time (i.e., instantly) if there was a problem. **2)** Virtual machines promote portability by enabling software to run on multiple platforms, but they tend to be less efficient than real machines, because virtual machines must execute software instructions that simulate hardware operations.

1.12 Operating System Components and Goals

Computer systems have evolved from early systems containing no operating system, to multiprogramming machines, to timesharing machines, to personal computers and finally to truly distributed systems. As the demand for new features and improved efficiency grew and hardware changed, operating systems evolved to fill new roles. This section describes various core operating system components and explains several goals of operating systems.

1.12.1 Core Operating System Components

A user interacts with the operating system via one or more user applications. and often through a special application called a **shell**, or command interpreter.[105] Most of today's shells are implemented as text-based interfaces that enable the user to issue commands from a keyboard or as GUIs that allow the user to point and click and drag and drop icons to request services from the operating system (e.g., to open an application). For example, Microsoft Windows XP provides a GUI through

which users can issue commands; alternatively, the user can open a command prompt window that accepts typed commands.

The software that contains the core components of the operating system is referred to as the kernel. Typical operating system core components include:

- the **process scheduler**, which determines when and for how long a process executes on a processor.
- the **memory manager**, which determines when and how memory is allocated to processes and what to do when main memory becomes full.
- the **I/O manager**, which services input and output requests from and to hardware devices, respectively.
- the **interprocess communication (IPC) manager**, which allows processes to communicate with one another.
- the **file system manager**, which organizes named collections of data on storage devices and provides an interface for accessing data on those devices.

Almost all modern operating systems support a multiprogrammed environment in which multiple applications can execute concurrently. One of the most fundamental responsibilities of an operating system is to determine which processor executes a process and for how long that process executes.

A program may contain several elements that share data and that can be executed concurrently. For example, a Web browser may contain separate components to read a Web page's HTML, retrieve the page's media (e.g., images, text and video) and render the page by laying out its content in the browser window. Such program components, which execute independently but perform their work in a common memory space, are called **threads**. Threads are discussed in Chapter 4, Thread Concepts.

Typically, many processes compete to use the processor. The process scheduler can base its decisions on several criteria, such as importance of a process, its estimated running time, or how long it has waited to obtain the processor. We discuss processor scheduling in Chapter 8, Processor Scheduling.

The memory manager allocates memory to the operating system and to processes. To ensure that processes to do not interfere with the operating system or with one another, the memory manager prevents each process from accessing memory that has not been allocated to it. Almost all of today's operating systems support virtual memory, as discussed in Chapters 10 and 11.

Another core function of the operating system is to manage the computer's input/output (I/O) devices. Input devices include keyboards, mice, microphones and scanners; output devices include monitors, printers and speakers. Storage devices (e.g., hard disks, rewritable optical discs and tape) and network cards function as both input and output devices. When a process wishes to access an I/O device, it must issue a system call to the operating system. That system call is subsequently handled by a **device driver**, which is a software component that interacts directly with hardware, often containing device-specific commands and other instructions to perform the requested input/output operations.

Most computer systems can store data persistently (i.e., after the computer is turned off). Because main memory is often relatively small and loses its data when the power is turned off, persistent secondary storage devices are used, most commonly hard disks. Disk I/O—one of the most common forms of I/O—occurs when a process requests access to information on a disk device.

Secondary storage, however, is much slower than processors and main memory. The **disk scheduler** component of an operating system is responsible for reordering disk I/O requests to maximize performance and minimize the amount of time a process waits for disk I/O. Redundant Array of Independent Disks (RAID) systems attempt to reduce the time a process waits for disk I/O by using multiple disks at once to service I/O requests. We discuss disk scheduling algorithms and RAID systems in Chapter 12, Disk Performance Optimization.

Operating systems use file systems to organize and efficiently access named collections of data called files located on storage devices. File system concepts are addressed in Chapter 13, File and Database Systems.

Often, processes (or threads) cooperate to accomplish a common goal. Thus, many operating systems provide interprocess communication (IPC) and synchronization mechanisms to simplify such concurrent programming. Interprocess communication enables processes to communicate via messages sent between the processes (and threads); synchronization provides structures that can be used to ensure that processes (and threads) share data properly. Processes and threads are discussed in Chapters 3 through 8.

Self Review

1. Which operating system components perform each of the following operations?

- **a.** Write to disk.
- **b.** Determine which process will run next.
- **c.** Determine where in memory a new process should be placed.
- **d.** Organize files on disk.
- **e.** Enable one process to send data to another.

2. Why is it dangerous to allow users to perform read or write operations to any region of disk at will?

Ans: **1)** a) I/O manager; b) processor scheduler; c) memory manager; d) file system manager; e) interprocess communication (IPC) manager. **2)** It is dangerous because users could accidentally or maliciously overwrite critical data (such as operating system files) or read sensitive information (such as confidential documents) without authorization.

1.12.2 Operating System Goals

Users have come to expect certain characteristics of operating systems, such as:

- efficiency
- robustness

- scalability
- extensibility
- portability
- security
- interactivity
- usability

An **efficient operating system** achieves high **throughput** and low average turnaround time. Throughput measures the amount of work a processor can complete within a certain time period. Recall that one role of an operating system is to provide services to many applications. An efficient operating system minimizes the time spent providing these services (see the Operating Systems Thinking feature, Performance).

A **robust operating system** is fault tolerant and reliable—the system will not fail due to isolated application or hardware errors, and if it fails, it does so gracefully (i.e., by minimizing loss of work and by preventing damage to the system's hard-

Operating Systems Thinking

Performance

One of the most important goals of an operating system is to maximize system performance. We are performance conscious in our everyday lives. We measure our cars' gasoline mileage, we record various speed records, professors assign grades to students, employees receive performance evaluations from their employers, a corporate executive's performance is measured by company profits, politicians' performance is measured in frequent polls of their constituents and so on.

High performance is essential to successful operating systems. However, performance is often "in the eye of the beholder"—there are many ways to classify operating system performance. For batch-processing systems, throughput is an important measure; for interactive timesharing systems, fast response times are more important.

Throughout the book we present many performance improvement techniques. For example, Chapter 8, Processor Scheduling, discusses allocating processor time to processes to improve system performance as measured by interactivity and throughput. Chapter 11, Virtual Memory Management, discusses allocating memory to processes to reduce their execution times. Chapter 12, Disk Performance Optimization, focuses on improving disk performance by reordering I/O requests. In Chapter 14, Performance and Processor Design, we discuss evaluating systems according to several important performance criteria. Chapters 20, and 21 discuss performance issues in the Linux and Windows XP operating systems, respectively.

ware). Such an operating system will provide services to each application unless the hardware it relies on fails.

A **scalable operating system** is able to use resources as they are added. If an operating system is not scalable, then it will quickly reach a point where additional resources will not be fully utilized. A scalable operating system can readily adjust its degree of multiprogramming. Scalability is a particularly important attribute of multiprocessor systems—as more processors are added to a system, ideally the processing capacity should increase in proportion to the number of processes, though, in practice, that does not happen. Multiprocessing is discussed in Chapter 15, Multiprocessor Management.

An **extensible operating system** will adapt well to new technologies and provide capabilities to extend the operating system to perform tasks beyond its original design.

A **portable operating system** is designed such that it can operate on many hardware configurations. Application portability is also important, because it is costly to develop applications, so the same application should run on a variety of hardware configurations to reduce development costs. The operating system is crucial to achieving this kind of portability.

A **secure operating system** prevents users and software from accessing services and resources without authorization. **Protection** refers to the mechanisms that implement the system's security policy.

An **interactive operating system** allows applications to respond quickly to user actions, or events. A **usable operating system** is one that has the potential to serve a significant user base. These operating systems generally provide an easy-to-use user interface. Operating systems such as Linux, Windows XP and MacOS X are characterized as usable operating systems, because each supports a large set of applications and provides standard user interfaces. Many experimental and academic operating systems do not support a large number of applications or provide user-friendly interfaces and therefore are not considered to be usable.

Self Review

1. Which operating system goals correspond to each of the following characteristics?

- **a.** Users cannot access services or information without proper authorization.
- **b.** The operating system runs on a variety of hardware configurations.
- **c.** System performance increases steadily when additional memory and processors are added.
- **d.** The operating system supports devices that were not available at the time of its design.
- **e.** Hardware failure does not necessarily cause the system to fail.

2. How does device driver support contribute to an operating system's extensibility?

Ans: **1)** a) security; b) portability; c) scalability; d) extensibility; e) robustness. **2)** Device drivers enable developers to add support for hardware that did not exist when the operating system was designed. With each new type of device that is added to a system a corresponding device driver must be installed.

1.13 Operating System Architectures

Today's operating systems tend to be complex because they provide many services and support a variety of hardware and software resources (see the Operating Systems Thinking feature, Keep It Simple (KIS) and the Anecdote,). Operating system architectures can help designers manage this complexity by organizing operating system components and specifying the privilege with which each component executes. In the monolithic design, every component of the operating system is contained in the kernel; in the microkernel design, only the essential components are included. In the sections that follow, we survey several important architectures (see the Operating Systems Thinking feature, Architecture).

1.13.1 Monolithic Architecture

The **monolithic operating system** is the earliest and most common operating system architecture. Every component of the operating system is contained in the kernel and can directly communicate with any other (i.e., simply by using function calls). The kernel typically executes with unrestricted access to the computer system (Fig. 1.3). OS/360, VMS and Linux are broadly characterized as monolithic operating systems.[106] Direct intercommunication between components makes monolithic operating systems highly efficient. Because monolithic kernels group components together,

Operating Systems Thinking

Keep It Simple (KIS)

Complex systems are costly to design, implement, test, debug and maintain. Often, operating systems designers will choose the simplest of several approaches to solving a particular problem. Sometimes, though, a more complex approach can yield performance benefits or other improvements that make such an approach worthwhile. Such trade-offs are common in computing. A simple linear search of an array is trivial to program, but runs slowly compared to a more elegant and complicated binary search. Tree data structures can be more complex to work with than arrays, but make it easier and faster to perform certain types of insertions and deletions. We typically consider alternate approaches to solving operating systems problems and developing resource management strategies. As you read these discussions, you will see the trade-offs between simplicity and complexity. As you read these solutions, you might be inclined to favor certain approaches. The systems you work with in the future may demand different approaches. Our philosophy is to present the pros and cons of the popular approaches to help you prepare to make your own best judgment calls in industry.

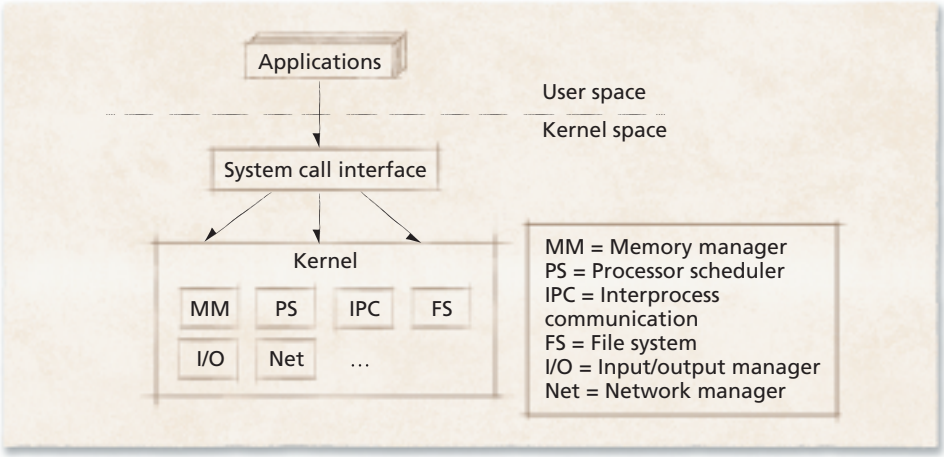

Figure 1.3 | *Monolithic operating system kernel architecture.*

however it is difficult to isolate the source of bugs and other errors. Further, because all code executes with unrestricted access to the system, systems with monolithic kernels are particularly susceptible to damage from errant or malicious code.

Self Review

1. What is the defining characteristic of a monolithic operating system?
2. Why do monolithic operating systems tend to be efficient? What is a key weakness of monolithic kernels?

Anecdote

System Architect vs. System Engineer

If you enter the field of operating systems development and you become more senior, you may be given a title like systems architect or systems engineer. A professional software colleague was a guest speaker at a conference some years ago. He was introduced as a systems architect. He said: "You may wonder how that differs from being a systems engineer." He humbly explained the difference with a building analogy: "When an engineer builds a building, it's very well built, but it's so ugly that the people tear it down; when an architect builds a building, it's very beautiful, but it falls down!"

Lesson to operating systems designers: You need to combine aspects of both architecture and engineering to insure that your systems are both well built and elegant. The latter goal is less important.

Ans: **1)** In a monolithic operating system every component of the operating system is contained in the kernel. **2)** Monolithic kernels tend to be efficient because few calls cross from user space to kernel space. Because all OS code in monolithic kernels operates with unrestricted access to the computer's hardware and software, these systems are particularly susceptible to damage from errant code.

1.13.2 Layered Architecture

As operating systems became larger and more complex, purely monolithic designs became unwieldy. The **layered** approach to operating systems attempts to address this issue by grouping components that perform similar functions into layers. Each layer communicates exclusively with those immediately above and below it. Lower-level layers provide services to higher-level ones using an interface that hides their implementation.

Layered operating systems are more modular than monolithic operating systems, because the implementation of each layer can be modified without requiring any modification to other layers. A modular system has self-contained componen that can be reused throughout the system. Each component hides how it performs

Operating Systems Thinking

Architecture

Just as architects use different approaches to designing buildings, operating systems designers employ different architectural approaches to designing operating systems. Sometimes these approaches are pure in that one architectural approach is used throughout the system. Sometimes hybridized approaches are used, mixing the advantages of several architectural styles. The approach the designer chooses will have monumental consequences on the initial implementation and the evolution of the operating system. It becomes increasingly difficult to change approaches the further into the development you proceed, so it is important to choose the proper architecture early in system development. More generally, it is much easier to build the building correctly in the first place than it is to modify the building after it has been built.

One of the most common architectural approaches employed in software systems such as operating systems is called layering. This software is divided into modules called layers that each perform certain tasks. Each layer calls the services provided by the layer below it, while the implementation of that layer is hidden from the layer above. Layering combines the virtues of the software engineering techniques of modularity and information hiding to provide a solid basis for building quality systems. We discuss layered software approaches throughout the book, starting with a historic mention of Dijkstra's THE system (see Section 1.13.2, Layered Architecture) and continuing to explanations of how layering is used in Linux and Windows XP in Chapters 20, and 21, respectively.

its job and presents a standard interface that other components can use to request its services. Modularity imposes structure and consistency on the operating system—often simplifying validation, debugging and modification. However, in a layered approach, a user process's request may need to pass through many layers before it is serviced. Because additional methods must be invoked to pass data from one layer to the next, performance degrades compared to that of a monolithic kernel, which may require only a single call to service a similar request. Also, because all layers have unrestricted access to the system, layered kernels are also susceptible to damage from errant or malicious code. The THE operating system is an early example of a layered operating system (Fig. 1.4).[107] Many of today's operating systems, including Windows XP and Linux, implement some level of layering.

Self Review

1. How are layered operating systems more modular than monolithic operating systems?
2. Why do layered operating systems tend to be less efficient than monolithic operating systems?

Ans: **1)** In layered operating systems, the implementation and interface are separate for each layer. This allows each layer to be tested and debugged separately. It also enables designers to change each layer's implementation without needing to modify the other layers. **2)** In layered operating systems, several calls may be required to communicate between the layers, whereas this overhead does not exist in monolithic kernels.

1.13.3 Microkernel Architecture

A **microkernel operating system** architecture provides only a small number of services in an attempt to keep the kernel small and scalable. These services typically include low-level memory management, interprocess communication and basic pro-

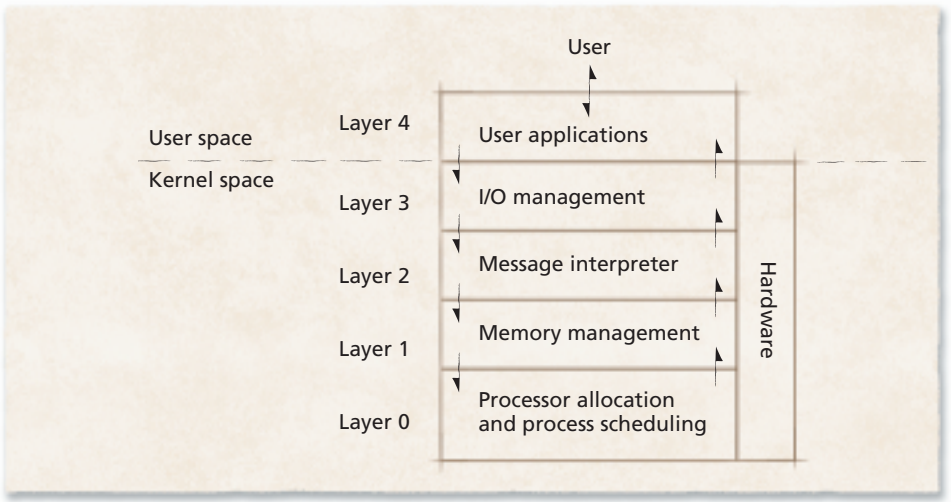

Figure 1.4 | *Layers of the THE operating system.*

cess synchronization to enable processes to cooperate. In microkernel designs, most operating system components—such as process management, networking, file system interaction and device management—execute outside the kernel with a lower privilege level (Fig. 1.5).[108, 109, 110, 111]

Microkernels exhibit a high degree of modularity, making them extensible, portable and scalable.[112] Further, because the microkernel does not rely on each component to execute, one or more components can fail, without causing the operating system to fail. However, such modularity comes at the cost of an increased level of intermodule communication, which can degrade system performance. Although few of today's popular operating systems fully embrace the microkernel design, Linux and Windows XP, for example, contain modular components.[113]

Self Review

1. What is the difference between a purely layered architecture and a microkernel architecture?
2. How do microkernels promote portability?

Ans: **1)** A layered architecture enables communication exclusively between operating system components in adjacent layers. A microkernel architecture enables communication between all operating system components via the microkernel. **2)** The microkernel does not depend on a particular hardware platform; support for new hardware can be provided by loading a new module.

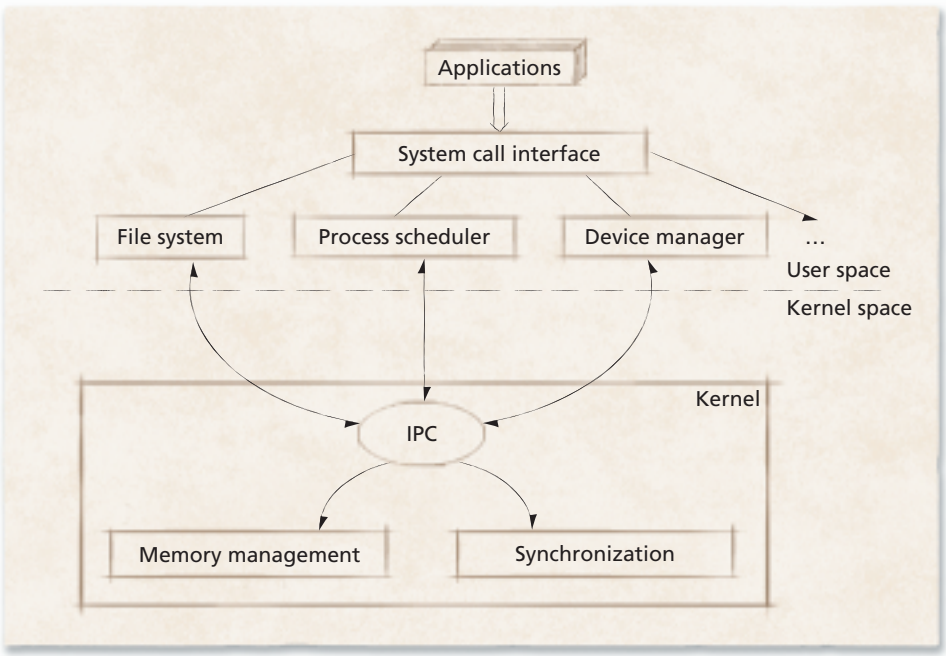

Figure 1.5 | *Microkernel operating system architecture.*

1.13.4 Networked and Distributed Operating Systems

Advances in telecommunications technology have profoundly affected operating systems. A **network operating system** enables its processes to access resources (e.g., files) that reside on other independent computers on a network.[114] The structure of many networked and distributed operating systems is often based on the client/server model (Fig. 1.6). The client computers in such a network request resources—such as files and processor time—via the appropriate network protocol. The servers respond with the appropriate resources. In such networks, operating system designers must carefully consider how to manage data and communication among computers.

Some operating systems are more "networked" than others. In a networked environment, a process can execute on the computer on which it is created or on another computer on the network. In some network operating systems, users can specify exactly where their processes run; in others, the operating system determines where processes are executed. For example, the system may determine that a process can be more efficiently executed on a computer experiencing a light load.[115]

Networked file systems are an important component of networked operating systems. At the lowest level, users acquire resources on another machine by explicitly connecting to that machine and retrieving files. Higher-level network file systems enable users to access remote files as if they were on the local system. Examples of network file systems include Sun's Network File System (NFS) and

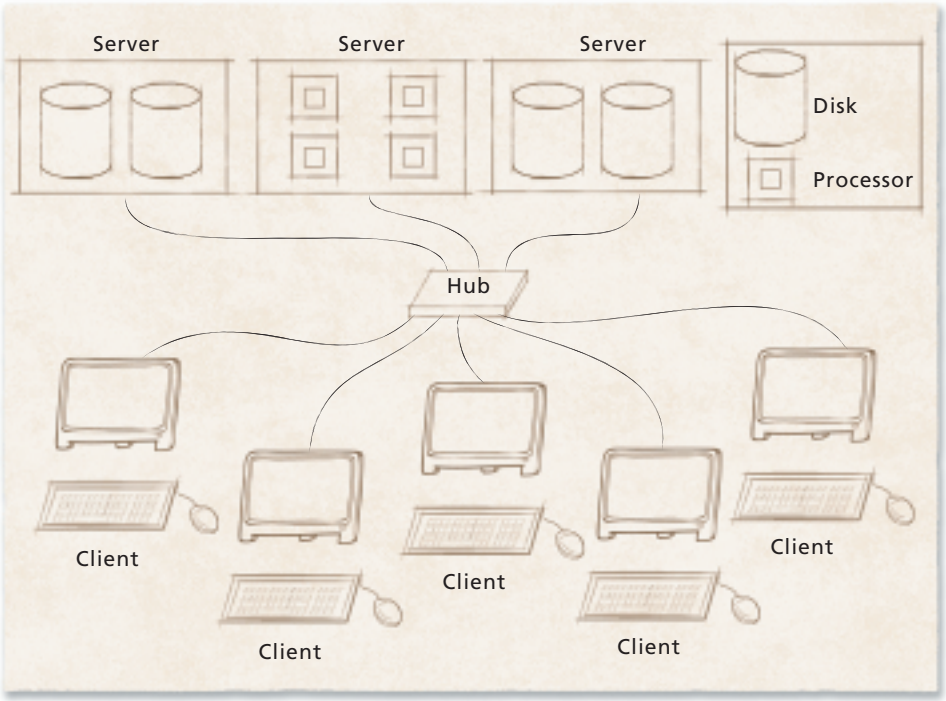

Figure 1.6 | *Client/server networked operating system model.*

CMU's Andrew and Coda file systems. Networked file systems are discussed in detail in Chapter 18, Distributed Systems and Web Services.

A **distributed operating system** is a single operating system that manages resources on more than one computer system. **Distributed systems** provide the illusion that multiple computers are a single powerful computer, so that a process can access all of the system's resources regardless of the process's location within the distributed system's network of computers.[116] Distributed operating systems are often difficult to implement and require complicated algorithms to enable processes to communicate and share data. Examples of distributed operating systems are MIT's Chord operating system and the Amoeba operating system from the Vrije Universiteit (VU) in Amsterdam.[117, 118] We discuss distributed systems in Chapter 17, Introduction to Distributed Systems.

Now that we have presented a seemingly endless stream of facts, issues, and acronyms, we proceed with a discussion of the basic principles of computer hardware and software in Chapter 2, Hardware and Software Concepts.

Self Review

1. What is the major difference between networked and distributed operating systems?
2. What is the primary advantage of a distributed operating system? What is the primary challenge of designing one?

Ans: **1)** A networked operating system controls one computer but cooperates with other computers on the network. In a distributed operating system, one operating system controls many computers in a network. **2)** The primary advantage is that processes do not need to know the locations of the resources they use, which simplifies applications programming. This comes at the expense of the systems programmer, who must implement complicated algorithms to enable processes to communicate and share data among many computers, creating the illusion of there being only a single larger computer.

Web Resources

`www.bell-labs.com/history/unix/`
Provides a history of the UNIX operating system, from its origins in the Multics system to the mature UNIX operating systems of today. Discusses many of the design and architectural considerations in the evolution of UNIX.

`www.softpanorama.org/History/os_history.shtml`
Provides a wealth of information on open-source software from a historical perspective.

`www.microsoft.com/windows/WinHistoryIntro.mspx`
Provides a history of the Microsoft Windows family of operating systems.

`www.viewz.com/shoppingguide/os.shtml`
Compares several popular operating systems, including Windows, Linux and MacOS, and provides a historical perspective.

`developer.apple.com/darwin/history.html`
Covers the evolution of Darwin, the core of Apple's OS X operating system.

`www.cryptonomicon.com/beginning.html`
Contains a link to the article, "In the Beginning Was the Command Line," by Neal Stephenson. This text is a narrative account of the recent history of operating systems, making colorful use of anecdote and metaphor.

`www.acm.org/sigcse/cc2001/225.html`
Lists the operating systems course curriculum recommendation from the ACM/IEEE Joint Task Force that completed the Computing Curricula 2001 (CC2001) project. The curriculum indicates key areas of coverage for a typical operating systems course.

whatis.techtarget.com/
Provides definitions for computer-related terms.

www.webopedia.com/
Provides an online dictionary and a search engine for computer- and Internet-related terms.

www.wikipedia.org/
Wikipedia is a project aimed at creating a free, open and accurate encyclopedia online. Any user may modify the entries in the encyclopedia, and all entries are licensed under the GNU Free Documentation License (GNUFDL).

Summary

Some years ago an operating system was defined as the software that controls the hardware, but the landscape of computer systems has evolved significantly since then, requiring a more complicated description. To increase hardware utilization, applications are designed to execute concurrently. However, if these applications are not carefully programmed, they might interfere with one another. As a result, a layer of software called an operating system separates applications (the software layer) from the hardware they access.

When a user requests that the computer perform an action (e.g., execute an application or print a document), the operating system manages the software and hardware to produce the desired result. Operating systems are primarily resource managers—they manage hardware, including processors, memory, input/output devices and communication devices. The operating system must also manage applications and other software abstractions that, unlike hardware, are not physical objects.

Operating systems have evolved over the last 60 years through several distinct phases or generations that correspond roughly to the decades. In the 1940s, the earliest electronic digital computers did not include operating systems. The systems of the 1950s generally executed only one job at a time, but used techniques that smoothed the transition between jobs to obtain maximum utilization of the computer system. A job constituted the set of instructions that a program would execute. These early computers were called single-stream batch-processing systems, because programs and data were submitted in groups or batches by loading them consecutively onto tape or disk.

The systems of the 1960s were also batch-processing systems, but they used the computer's resources more efficiently by running several jobs at once. The systems of the 1960s improved resource utilization by allowing one job to use the processor while other jobs used peripheral devices. With these observations in mind, operating systems designers developed multiprogramming systems that managed a number of jobs at once, that number being indicated by the system's degree of multiprogramming.

In 1964, IBM announced its System/360 family of computers. The various 360 computer models were designed to be hardware compatible, to use the OS/360 operating system and to offer greater computer power as the user moved upward in the series. More advanced operating systems were developed to service multiple interactive users at once. Timesharing systems were developed to support large numbers of simultaneous interactive users.

Real-time systems attempt to supply a response within a certain bounded time period. The resources of a real-time system are often heavily under-utilized. It is more important for real-time systems to respond quickly when needed than to use their resources efficiently.

Turnaround time—the time between submission of a job and the return of its results—was reduced to minutes or even seconds. The value of timesharing systems in support of program development was demonstrated when MIT used the CTSS system to develop its own successor, Multics. TSS, Multics and CP/CMS all incorporated virtual memory, which enables programs to address more memory locations than are actually provided in main memory, which is also called real memory or physical memory.

The systems of the 1970s were primarily multimode timesharing systems that supported batch processing, timesharing and real-time applications. Personal computing was in its incipient stages, fostered by early and continuing developments in microprocessor technology. Communications between computer systems throughout the United States increased as the Department of Defense's TCP/IP communications standards became widely used—especially in military and university computing environments. Security problems increased as growing volumes of information passed over vulnerable communications lines.

The 1980s was the decade of the personal computer and the workstation. Rather than data being brought to a central, large-scale computer installation for processing, computing was distributed to the sites at which it was needed. Personal computers proved to be relatively easy to learn and use, partially because of graphical user interfaces (GUI), which used graphical symbols such as windows,

icons and menus to facilitate user interaction with programs. As technology costs declined, transferring information between computers in computer networks became more economical and practical. The client/server distributed computing model became widespread. Clients are user computers that request various services; servers are computers that perform the requested services.

The software engineering field continued to evolve, a major thrust by the United States government being aimed especially at providing tighter control of Department of Defense software projects. Some goals of the initiative included realizing code reusability and a greater degree of abstraction in programming languages. Another software engineering development was the implementation of processes containing multiple threads of instructions that could execute independently.

In the late 1960s, ARPA, the Advanced Research Projects Agency of the Department of Defense, rolled out the blueprints for networking the main computer systems of about a dozen ARPA-funded universities and research institutions. ARPA proceeded to implement what was dubbed the ARPAnet—the grandparent of today's Internet. ARPAnet's chief benefit proved to be its capability for quick and easy communication via what came to be known as electronic mail (e-mail). This is true even on today's Internet, with e-mail, instant messaging and file transfer facilitating communications among hundreds of millions of people worldwide.

The ARPAnet was designed to operate without centralized control. The protocols (i.e., set of rules) for communicating over the ARPAnet became known as the Transmission Control Protocol/Internet Protocol (TCP/IP). TCP/IP was used to manage communication between applications. The protocols ensured that messages were routed properly from sender to receiver and arrived intact. Eventually, the government decided to allow access to the Internet for commercial purposes.

The World Wide Web allows computer users to locate and view multimedia-based documents (i.e., documents with text, graphics, animations, audios or videos) on almost any subject. Even though the Internet was developed more than three decades ago, the introduction of the World Wide Web (WWW) was a relatively recent event. In 1989, Tim Berners-Lee of CERN (the European Center for Nuclear Research) began to develop a technology for sharing information via hyperlinked text documents. To implement this new technology, He created the HyperText Markup Language (HTML). Berners-Lee also implemented the Hypertext Transfer Protocol (HTTP) to form the communications backbone of his new hypertext information system, which he called the World Wide Web.

Hardware performance continued to improve exponentially in the 1990s. Inexpensive processing power and storage allowed users to execute large, complex programs on personal computers and enabled small to mid-size companies to use these economical machines for the extensive database and processing jobs that earlier had been delegated to mainframe systems. In the 1990s, the shift toward distributed computing (i.e., using multiple independent computers to perform a common task) rapidly accelerated. As demand for Internet connections grew, operating system support for networking tasks became standard. Users at home and in large corporations increased productivity by accessing the resources on networks of computers.

Microsoft Corporation became dominant in the 1990s. Its Windows operating systems, which borrowed from many concepts popularized by early Macintosh operating systems (such as icons, menus and windows), enabled users to navigate multiple concurrent applications with ease.

Object technology became popular in many areas of computing. Many applications were written in object-oriented programming languages, such as C++ or Java. In object-oriented operating systems (OOOS), objects represent components of the operating system. Object-oriented concepts such as inheritance and interfaces were exploited to create modular operating systems that were easier to maintain and extend than operating systems built with previous techniques.

Most commercial software is sold as object code. The source code is not included, enabling vendors to hide proprietary information and programming techniques. Free and open-source software became increasingly common in the 1990s. Open-source software is distributed with the source code, allowing individuals to examine and modify the software before compiling and executing it. The Linux operating system and the Apache Web server are both free and open source.

In the 1980s, Richard Stallman, a developer at MIT, launched the GNU project to recreate and extend most of the tools for AT&T's UNIX operating system. Stallman created the GNU project because he disagreed with the concept of paying for permission to use software. The Open Source Initiative (OSI) was founded to further the benefits of open-source programming. Open-source software facilitates enhancements to software products by permitting anyone in the developer community to test, debug and enhance applications. This increases the chance that subtle

bugs, which could otherwise be security risks or logic errors, will be caught and fixed. Also, individuals and corporations can modify the source to create custom software that meets the needs of a particular environment.

In the 1990s, operating systems became increasingly user friendly. The GUI features that Apple had built into its Macintosh operating system in the 1980s were widely used in many operating systems and became more sophisticated. "Plug-and-play" capabilities were built into operating systems, enabling users to add and remove hardware components dynamically without manually reconfiguring the operating system.

Middleware is software that links two separate applications, often over a network and often between incompatible machines. It is particularly important for Web services because it simplifies communication across multiple architectures. Web services encompass a set of related standards that can enable any two computer applications to communicate and exchange data via the Internet. They are ready-to-use pieces of software on the Internet.

When the IBM PC appeared, it immediately spawned a huge software industry in which independent software vendors (ISVs) were able to market software packages for the IBM PC to run under the MS-DOS operating system. If an operating system presents an environment conducive to developing applications quickly and easily, the operating system and the hardware are more likely to be successful in the marketplace. Once an application base (i.e., the combination of the hardware and the operating system environment in which applications are developed) is widely established, it becomes extremely difficult to ask users and software developers to convert to a completely new applications development environment provided by a dramatically different operating system.

Operating systems intended for high-end environments must be designed to support large main memories, special-purpose hardware, and large numbers of processes. Embedded systems are characterized by a small set of specialized resources that provide functionality to devices such as cell phones and PDAs. In these environments, efficient resource management is the key to building a successful operating system.

Real-time systems require that tasks be performed within a particular (often short) time frame. For example, the autopilot feature of an aircraft must constantly adjust speed, altitude and direction. Such actions cannot wait indefinitely—and sometimes cannot wait at all—for other nonessential tasks to complete.

Some operating systems must manage hardware that may or may not physically exist in the machine. A virtual machine (VM) is a software abstraction of a computer that often executes as a user application on top of the native operating system. A virtual machine operating system manages the resources provided by the virtual machine. One application of virtual machines is to allow multiple instances of an operating system to execute concurrently. Another use for virtual machines is emulation—the ability to use software or hardware that mimics the functionality of hardware or software not present in the system. By providing the illusion that applications are running on different hardware or operating systems, virtual machines promote portability—the ability for software to run on multiple platforms—and many other benefits.

A user interacts with the operating system via one or more user applications. Often, the user interacts with an operating system through a special application called a shell. The software that contains the core components of the operating system is referred to as the kernel. Typical operating system components include the processor scheduler, memory manager, I/O manager, interprocess communication (IPC) manager, and file system manager.

Almost all modern operating systems support a multiprogrammed environment in which multiple applications can execute concurrently. The kernel manages the execution of processes. Program components, which execute independently but use a single memory space to share data, are called threads.

When a process wishes to access an I/O device, it must issue a system call to the operating system. That system call is subsequently handled by a device driver—a software component that interacts directly with hardware—often containing device-specific commands and other instructions to perform the requested input and output operations.

Users have come to expect certain characteristics of operating systems, such as efficiency, robustness, scalability, extensibility, portability, security and protection, interactivity and usability.

In a monolithic operating system, every component is contained in the kernel. As a result, any component can directly communicate with any other. Monolithic operating systems tend to be highly efficient. A disadvantage of monolithic designs is that it is difficult to determine the source of subtle errors.

The layered approach to operating systems attempts to address this issue by grouping components that perform similar functions into layers. Each layer communicates exclusively with the layers immediately above and below it.

In a layered approach, a user process's request may need to pass through many layers before completion. Because additional methods must be invoked to pass data and control from one layer to the next, system throughput decreases compared to that with a monolithic kernel, which may require only a single call to service a similar request.

A microkernel operating system architecture provides only a small number of services in an attempt to keep the kernel small and scalable. Microkernels exhibit a high degree of modularity, making them extensible, portable and scalable. However, such modularity comes at the cost of an increased level of intermodule communication, which can degrade system performance.

A network operating system runs on one computer and allows its processes to access resources such as files and processors on a remote computer. A distributed operating system is a single operating system that manages resources on more than one computer system. The goals of a distributed operating system include transparent performance, scalability, fault tolerance and consistency.

Key Terms

Advanced Research Projects Agency (ARPA)—Government agency under the Department of Defense that laid the groundwork for the Internet; it is now called the Defense Advanced Research Projects Agency (DARPA).

application base—Combination of the hardware and the operating system environment in which applications are developed. It is difficult for users and application developers to convert from an established application base to another.

application programming interface (API)—Specification that allows applications to request services from the kernel by making system calls.

ARPAnet—Predecessor to the Internet that enabled researchers to network their computers. ARPAnet's chief benefit proved to be quick and easy communication via what came to be known as electronic mail (e-mail).

bandwidth—Information-carrying capacity of a communications line.

business-critical system—System that must function properly, but whose failure of which leads to reduced productivity and profitability; not as crucial as a mission-critical system, where failure could put human lives could be at risk.

C—High-level programming language that was designed and used to implement the UNIX operating system.

client—Process that requests a service from another process (a server). The machine on which the client process runs is also called a client.

compute-bound—See processor-bound.

CP/CMS—Timesharing operating system developed by IBM in the 1960s.

CTSS—Timesharing operating system developed at MIT in the 1960s.

degree of multiprogramming—Number of programs a system can manage at a time.

device driver—Software through which the kernel interacts with hardware devices. Device drivers are intimately familiar with the specifics of the devices they manage—such as the arrangement of data on those devices—and they deal with device-specific operations such as reading data, writing data and opening and closing a DVD drive's tray. Drivers are modular, so they can be added and removed as a system's hardware changes, enabling users to add new types of devices easily; in this way they contribute to a system's extensibility.

disk scheduler—Operating system component that determines the order in which disk I/O requests are serviced to improve performance.

distributed computing—Using multiple independent computers to perform a common task.

distributed operating system—Single operating system that provides transparent access to resources spread over multiple computers.

distributed system—Collection of computers that cooperate to perform a common task.

efficient operating system—Operating system that exhibits high throughput and small turnaround time.

embedded system—Small computer containing limited resources and specialized hardware to run devices such as PDAs or cellular phones.

extensible operating system—An operating system that can incorporate new features easily.

fault tolerance—Operating system's ability to handle software or hardware errors.

file system manager—Operating system component that organizes named collections of data on storage devices and provides an interface for accessing data on those devices.

General Public License (GPL)—Open-source software license which specifies that software distributed under it must contain the complete source code, must clearly indicate any modifications to the original code and must be accompanied by the GPL. End users are free to modify and redistribute any software under the GPL.

GNU—Project initiated by Stallman in the 1980s aimed at producing an open-source operating system with the features and utilities of UNIX.

graphical user interface (GUI)—User-friendly point of access to an operating system that uses graphical symbols such as windows, icons and menus to facilitate program and file manipulation.

HyperText Markup Language (HTML)—Language that specifies the content and arrangement of information on a Web page and provides hyperlinks to access other pages.

Hypertext Transfer Protocol (HTTP)—Network protocol used for transferring HTML documents and other data formats between a client and a server. This is the key protocol of the World Wide Web.

I/O-bound—Process (or job) that tends to use a processor for a short time before generating an I/O request and relinquishing a processor.

I/O manager—Operating system component that receives, interprets and performs I/O requests.

independent software vendor (ISV)—Organization that develops and sells software. ISVs prospered after the release of the IBM PC.

interactive operating system—Operating system that allows applications to respond quickly to user input.

interactive users—Users that are present when the system processes their jobs. Interactive users communicate with their jobs during execution.

Internet—Network of communication channels that provides the backbone for telecommunication and the World Wide Web. Each computer on the Internet determines which services it uses and which it makes available to other computers connected to the Internet.

interprocess communication (IPC) manager—Operating system component that governs communication between processes.

Java Virtual Machine (JVM)—Virtual machine that enables Java programs to execute on many different architectures without recompiling Java programs into the native machine language of the computer on which they execute. The JVM promotes application portability and simplifies programming by freeing the programmer from architecture-specific considerations.

job—Set of work to be done by a computer.

kernel—Software that contains the core components of an operating system.

layered operating system—Modular operating system that places similar components in isolated layers. Each layer accesses the services of the layer below and returns results to the layer above.

level of multiprogramming—See degree of multiprogramming.

massive parallelism—Property of a system containing large numbers of processors so that many parts of computations can be performed in parallel.

memory manager—Operating system component that controls physical and virtual memory.

microkernel operating system—Scalable operating system that puts a minimal number of services in the kernel and requires user-level programs to implement services generally delegated to the kernel in other types of operating systems.

middleware—Layer of software that enables communication between different applications. Middleware simplifies application programming by performing work such as network communication and translation between different data formats.

mission-critical system—System that must function properly; its failure could lead to loss of property, money or even human life.

monolithic operating system—Operating system whose kernel contains every component of the operating system. The kernel typically operates with unrestricted access to the computer system.

Multics—One of the first operating systems to implement virtual memory. Developed by MIT, GE and Bell Laboratories as the successor to MIT's CTSS.

multiprogramming—Ability to store multiple programs in memory at once so that they can be executed concurrently.

network operating system—Operating system that can manipulate resources at remote locations but does not hide the location of these resources from applications (as distributed systems can).

object-oriented operating system (OOOS)—Operating system in which components and resources are represented as objects. Object-oriented concepts such as inheritance and interfaces help create modular operating systems that are easier to maintain and extend than operating systems built with previous techniques. Many operating systems use objects, but few are written entirely using object-oriented languages.

online—State describing a computer that is turned on (i.e., active) and directly connected to a network.

open-source software—Software that includes the application's source code and is often distributed under the General Public License (GPL) or a similar license. Open-source software is typically developed by teams of independent programmers worldwide.

open-source initiative (OSI)—Group that supports and promotes open-source software (see `www.opensource.com`).

operating system—Software that manages system resources to provide services that allow applications to execute properly. An operating system may manage both hardware and software resources. Operating systems provide an application programming interface to facilitate application development. They also help make system resources conveniently available to users while providing a reliable, secure and responsive environment to applications and users.

portability—Property of software that can run on different platforms.

portable operating system—Operating system that is designed to operate on many hardware configurations.

Portable Operating Systems Interface (POSIX)—API based on early UNIX operating systems.

priority of a process—Importance or urgency of a process relative to other processes.

process scheduler—Operating system component that determines which process can gain access to a processor and for how long.

process—An executing program.

processor-bound—Process (or job) that consumes its quantum when executing. These processes (or jobs) tend to be calculation intensive and issue few, if any, I/O requests.

protection—Mechanism that implements a system's security policy by preventing applications from accessing resources and services without authorization.

real-time system—System that attempts to service requests within a specified (usually short) time period. In mission-critical real-time systems (e.g., air traffic control and petroleum refinery monitors), money, property or even human life could be lost if requests are not serviced on time.

robust operating system—Operating system that is fault tolerant and reliable—the system will not fail due to unexpected application or hardware errors (but if it must fail, it does so gracefully). Such an operating system will provide services to each application unless the hardware those services requires fails to function.

scalable operating system—Operating system that is able to employ resources as they are added to the system. It can readily adapt its degree of multiprogramming to meet the needs of its users.

secure operating system—Operating system that prevents users and software from gaining unauthorized access to services and data.

server—Process that provides services to other processes (called clients). The machine on which these processes run is also called a server.

shell—Application (typically GUI or text based) that enables a user to interact with an operating system

single-stream batch-processing system—Early computer system that executed a series of noninteractive jobs sequentially, one at a time.

system call—Call from a user process that invokes a service of the kernel.

thread—Entity that describes an independently executable stream of program instructions (also called a thread of execution or thread of control). Threads facilitate parallel execution of concurrent activities within a process.

throughput—Amount of work performed per unit time.

timesharing system—Operating system that enables multiple simultaneous interactive users.

Transmission Control Protocol/Internet Protocol (TCP/IP)—Family of protocols that provide a framework for networking on the Internet.

TSS—Operating system designed by IBM in the 1960s that offered timesharing and virtual memory capabilities. Although it was never released commercially, many of its capabilities appeared in later IBM systems.

turnaround time—Time it takes from the submission of a request until the system returns the result.

UNIX—Operating system developed at Bell Laboratories that was written using the C high-level programming language.

usable operating system—Operating system that has the potential to serve a significant user base by providing an easy-to-use interface and supporting a large set of user-oriented applications.

virtual machine—Application that emulates the functionality of a computer system. A virtual machine can execute applications that are not directly compatible with the physical system that runs the virtual machine. The user "sees" the computer not as the virtual machine, but as the underlying physical machine.

virtual memory—Capability of operating systems that enables programs to address more memory locations than are

actually provided in main memory. Virtual memory systems help remove much of the burden of memory management from programmers, freeing them to concentrate on application development.

VM operating system—One of the first virtual machine operating systems; developed at IBM in the 1960s and still used widely today; its latest version is the z/VM.

Web services—Set of services and related standards that can allow any two computer applications to communicate and exchange data over the Internet. Web services operate using open, text-based standards that enable components written in different languages and on different platforms to communicate. They are ready-to-use pieces of software on the Internet.

World Wide Web (WWW)—Collection of hyperlinked documents accessible via the Internet using the Hypertext Transfer Protocol (HTTP). Web documents are typically written in languages such as HyperText Markup Language (HTML) and Extensible Markup Language (XML).

Exercises

1.1 Distinguish between multiprogramming and multiprocessing. What were the key motivations for the development of each?

1.2 Briefly discuss the significance of each of the following systems mentioned in this chapter:

- **a.** MS-DOS
- **b.** CTSS
- **c.** Multics
- **d.** OS/360
- **e.** TSS
- **f.** UNIX
- **g.** Macintosh

1.3 What developments made personal computing feasible?

1.4 Why is it impractical to use a virtual machine for a hard real-time system?

1.5 What role did the development of graphical user interfaces play in the personal computer revolution?

1.6 The GNU Public License (GPL) promotes software that is free, as in "freedom." How does the GPL provide such freedom?

1.7 How has distributed computing affected operating system design?

1.8 What are the advantages and disadvantages of communication between computers?

1.9 Define, compare, and contrast each of the following terms:

- **a.** online
- **b.** real time
- **c.** interactive computing
- **d.** timesharing

1.10 How do middleware and Web services promote interoperability?

1.11 Evaluate monolithic, layered and microkernel architectures according to

- **a.** efficiency.
- **b.** robustness.
- **c.** extensibility.
- **d.** security.

Suggested Projects

1.12 Prepare a research paper on the Linux operating system. In what ways does it support Stallman's "free as in freedom" doctrine for software? In what ways does Linux conflict with this philosophy?

1.13 Prepare a research paper on the Internet and how its pervasive use affects operating system design.

1.14 Prepare a research paper on the open-source software movement. Discuss whether all open-source software is free, as in both "freedom" and "price." How do the GPL and similar licenses promote open-source software?

1.15 Prepare a research paper on the evolution of operating systems. Be sure to mention the key hardware, software and communications technologies that encouraged each new operating system innovation.

1.16 Prepare a research paper on the future of operating systems.

1.17 Prepare a research paper giving a thorough taxonomy of past and present operating systems.

1.18 Prepare a research paper on Web services. Discuss the key technologies on which the Web services infrastructure is being built. How will the availability of Web services affect applications development?

1.19 Prepare a research paper on business-critical and mission-critical applications. Discuss the key attributes of hardware, communications software and operating systems that are essential to building systems to support these types of applications.

1.20 Prepare a research paper on virtual machine systems. Be sure to investigate IBM's VM operating system and Sun's Java Virtual Machine (JVM).

1.21 Prepare a research paper on operating systems and the law. Survey legislation related to operating systems.

1.22 Prepare a research paper on the impact of operating systems on business and the economy.

1.23 Prepare a research paper on operating systems and security and privacy. Be sure to consider the issues of worms and viruses.

1.24 Prepare a research paper on the ethical issues with which operating systems designers must be concerned. Be sure to deal with issues such as the use of computer systems in warfare and in life-threatening situations, viruses and worms, and other important topics you discover as you do your research for your paper.

1.25 List several trends leading the way to future operating systems designs. How will each affect the nature of future systems?

1.26 Prepare a research paper discussing the design of massively parallel systems. Be sure to compare large-scale multiprocessor systems (e.g., the Hewlitt-Packard Superdome supercomputer, which contains up to 64 processors; `www.hp.com/products1/servers/scalableservers/superdome/`) to clustered systems and server farms that contain hundreds or thousands of low-end computers that cooperate to perform common tasks (see, for example, `www.beowulf.org`). Use `www.top500.org`, a listing of the world's most powerful supercomputers, to determine the type of tasks that each of these massively parallel systems performs.

1.27 What trends are leading the way to dramatic increases in parallel computation? What challenges must be addressed by hardware designers and software designers before parallel computation will become widely used?

1.28 Prepare a research paper that compares MIT's Exokernel (`www.pdos.lcs.mit.edu/exo.html`) and CMU's Mach microkernel (`www-2.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html`) research operating systems.[119] What is the primary focus of each operating system? Be sure to mention how the researchers organized components such as memory management, disk scheduling and process management. Has either or both of these systems become commercially successful? Has either or both of these systems influenced the designs of commercially successful operating systems?

1.29 Why have UNIX and UNIX-based systems continued to be popular in recent decades? How does Linux impact the future of UNIX systems?

Recommended Reading

This section will be used throughout the book to provide the reader with references to seminal books and papers and current research on the topic of each chapter. Many of the papers that are cited in each chapter can be found in one of several journals associated with either the Association of Computing Machinery (ACM) or the Institute of Electrical and Electronics Engineers (IEEE). *Communications of the ACM* is the ACM's flagship journal, containing research papers, editorials and features about contemporary topics in computer science. The ACM also sponsors several special-interest groups (SIGs), each dedicated to particular fields of computer science. SIGOPS, the SIG for the field of operating systems (`www.acm.org/sigops/`), holds annual conferences and publishes the *Operating Systems Review*. The IEEE Computer Society (`www.computer.org`), the largest IEEE society, publishes several journals related to computer science, including the popular *IEEE Computer* (`www.computer.org/computer/`). The reader is encouraged to join the ACM, SIGOPS and the IEEE Computer Society. In the Web Resources section, we have listed several links to sites that recount the history of operating systems. An enlightening account of the design and development of the OS/360 operating system can be found in Frederick P. Brooks, Jr.'s *The Mythical Man-Month*.[120] Per Brinch Hansen's *Classic Operating Systems: From Batch to Distributed Systems* provides an anthology of papers describing the design and development of innovative and pioneering operating systems.[121] The bibliography for this chapter is located on our Web site at `www.deitel.com/books/os3e/Bibliography.pdf`.

Works Cited

1. "Evolution of the Intel Microprocessor: 1971–2007," <www.berghell.com/whitepapers/Evolution%20of%20Intel%20Microprocessors%201971%202007.pdf>.
2. "Top 500 List for November 2002," <www.top500.org/list/2002/11/>.
3. Weizer, N., "A History of Operating Systems," *Datamation,* January 1981, pp. 119–126.
4. Goldstein, H., *The Computer from Pascal to von Neumann,* Princeton: Princeton University Press, 1972.
5. Stern, N., *From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers* Bedford: Digital Press, 1981.
6. Bashe, C., et al., *IBM's Early Computers* (Cambridge: MIT Press, 1986).
7. Weizer, N., "A History of Operating Systems," *Datamation,* January 1981, pp. 119–126.
8. Grosch, H., "The Way It Was in 1957," *Datamation,* September 1977.
9. Denning, P., "Virtual Memory," *ACM CSUR*, Vol. 2, No. 3, September 1970, pp. 153–189.
10. Codd, E.; E. Lowry; E. McDonough; and C. Scalzi, "Multiprogramming STRETCH: Feasibility Considerations," *Communications of the ACM,* Vol. 2, 1959, pp. 13–17.
11. Critchlow, A., "Generalized Multiprocessor and Multiprogramming Systems," *Proc. AFIPS, FJCC,* Vol. 24, 1963, pp. 107–125.
12. Belady, L., et al., "The IBM History of Memory Management Technology," *IBM Journal of Research and Development,* Vol. 25, No. 5, September 1981, pp. 491–503.
13. "The Evolution of S/390," <www-ti.informatik.uni-tuebingen.de/os390/arch/history.pdf>.
14. Amdahl, G.; G. Blaauw; and F. Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development,* Vol. 8, No. 2, April 1964, pp. 87–101.
15. Weizer, N., "A History of Operating Systems," *Datamation,* January 1981, pp. 119–126.
16. Evans, B., "System/360: A Retrospective View," *Annals of the History of Computing,* Vol. 8, No. 2, April 1986, pp. 155–179.
17. Mealy, G.; B. Witt; and W. Clark, "The Functional Structure of OS/360," *IBM Systems Journal,* Vol. 5, No. 1, 1966, pp. 3–51.
18. Case, R., and A. Padeges, "Architecture of the IBM System/370," *Communications of the ACM,* Vol. 21, No. 1, January 1978, pp. 73–96.
19. Gifford, D., and A. Spector, "Case Study: IBM's System/360–370 Architecture," *Communications of the ACM,* Vol. 30, No. 4, April 1987, pp. 291–307.
20. "The Evolution of S/390," <www-ti.informatik.uni-tuebingen.de/os390/arch/history.pdf>.
21. Berlind, D., "Mainframe Linux Advocates Explain It All," *ZDNet,* April 12, 2002, <techupdate.zdnet.com/techupdate/stories/main/0,14179,2860720,00.html>.
22. Frenkel, K., "Allan L. Scherr: Big Blue's Time-Sharing Pioneer," *Communications of the ACM,* Vol. 30, No. 10, October 1987, pp. 824–829.
23. Harrison, T., et al, "Evolution of Small Real-Time IBM Computer Systems," *IBM Journal of Research and Development,* Vol. 25, No. 5, September 1981, pp. 441–451.
24. Corbato, F., et al., *The Compatible Time-Sharing System, A Programmer's Guide*, Cambridge: MIT Press, 1964.
25. Crisman, P., et al., eds., *The Compatible Time-Sharing System,* Cambridge: MIT Press, 1964.
26. Lett, A., and W. Konigsford, "TSS/360: A Time-Shared Operating System," *Proceedings of the Fall Joint Computer Conference, AFIPS,* Vol. 33, Part 1, 1968, pp. 15-28.
27. Bensoussan, A.; C. Clingen; and R. Daley, "The Multics Virtual Memory: Concepts and Designs," *Communications of the ACM,* Vol. 15, No. 5, May 1972, pp. 308–318.
28. Creasy, R., "The Origins of the VM/370 Time-Sharing System," *IBM Journal of Research and Development,* Vol. 25, No. 5, pp. 483–490.
29. Conrow, K., "The CMS Cookbook," *Computing and Networking Services*, Kansas State University, June 29, 1994, <www.ksu.edu/cns/pubs/cms/cms-cook/cms-cook.pdf>.
30. Denning, P., "Virtual Memory," *ACM Computing Surveys,* Vol. 2, No. 3, September 1970, pp. 153–189.
31. Parmelee, R., et al., "Virtual Storage and Virtual Machine Concepts," *IBM Systems Journal,* Vol. 11, No. 2, 1972.
32. Ritchie, D., "Dennis M. Ritchie," <cm.bell-labs.com/cm/cs/who/dmr/bigbio1st.html>.
33. Bell Labs Lucent Technologies, "Ken Thompson," 2002, <www.bell-labs.com/history/unix/thompsonbio.html>.
34. Mackenzie, R., "A Brief History of UNIX," <www.stanford.edu/~rachelm/cs1u-197/unix.html>.
35. Cook, D.; J. Urban, and S. Hamilton, "UNIX and Beyond: An Interview with Ken Thompson," *Computer,* May 1999, pp. 58–64.
36. Cook, D.; J. Urban, and S. Hamilton, "UNIX and Beyond: An Interview with Ken Thompson," *Computer,* May 1999, pp. 58–64.
37. Bell Labs Lucent Technologies, "Ken Thompson," 2002, <www.bell-labs.com/history/unix/thompsonbio.html>.
38. Bell Labs Lucent Technologies, "Dennis Ritchie," 2002, <www.bell-labs.com/about/history/unix/ritchie-bio.html>.
39. Bell Labs Lucent Technologies, "Dennis Ritchie," 2002, <www.bell-labs.com/about/history/unix/ritchie-bio.html>.
40. Ritchie, D., "Dennis M. Ritchie," <cm.bell-labs.com/cm/cs/who/dmr/bigbio1st.html>.

41. Reagan, P., and D. Cunningham, "Bell Labs Unveils Open Source Release of Plan 9 Operating System," June 7, 2000, <www.bell-labs.com/news/2000/june/7/2.html>.

42. Bell Labs, "Vita Nuova Publishes Source Code for Inferno Operating System, Moving Network Computing into the 21st Century," <www.cs.bell-labs.com/inferno/>.

43. Kildall, G., "CP/M: A Family of 8- and 16-bit Operating Systems," *Byte,* Vol. 6, No. 6, June 1981, pp. 216–232.

44. Quarterman, J. S., and J. C. Hoskins, "Notable Computer Networks," *Communications of the ACM,* Vol. 29, No. 10, October 1986, pp. 932–971.

45. Stefik, M., "Strategic Computing at DARPA: Overview and Assessment," *Communications of the ACM,* Vol. 28, No. 7, July 1985, pp. 690–707.

46. Comer, D., *Internetworking with TCP/IP: Principles, Protocols, and Architecture*, Englewood Cliffs, NJ: Prentice Hall, 1988.

47. Martin, J., and K. K. Chapman, *Local Area Networks: Architectures and Implementations*, Englewood Cliffs, NJ: Prentice Hall, 1989.

48. Metcalfe, R., and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM,* Vol. 19, No. 7, July 1976.

49. Balkovich, E.; S. Lerman; and R. Parmelee, "Computing in Higher Education: The Athena Experience," *Computer,* Vol. 18, No. 11, November 1985, pp. 112–127.

50. Zmoelnig, Christine, "The Graphical User Interface. Time for a Paradigm Shift?" August 30, 2001, <www.sensomatic.com/chz/gui/history.html>.

51. Engelbart, D., "Who we are. How we think. What we do." June 24, 2003, <www.bootstrap.org/index.html>.

52. Martin, E., "The Context of STARS," *Computer,* Vol. 16, No. 11, November 1983, pp. 14–20.

53. Ecklund, J., "Interview with Douglas Engelbart," May 4, 1994, <americanhistory.si.edu/csr/comphist/englebar.htm>.

54. Stauth, D., "Prominent Oregon State Alum Receives Leading Prize for Inventors," April 9, 1997, <oregonstate.edu/dept/ncs/newsarch/1997/April97/engelbart.htm>.

55. "Douglas Engelbart Inventor Profile," 2002, <www.invent.org/hall_of_fame/53.html>.

56. "Engelbart's Unfinished Revolution," December 9, 1998, <stanford-online.stanford.edu/engelbart/>.

57. "Douglas Engelbart Inventor Profile," 2002, <www.invent.org/hall_of_fame/53.html>.

58. World Wide Web Consortium, "Longer Bio for Tim Berners-Lee," <www.w3.org/People/Berners-Lee/Longer.html>.

59. "Engelbart's Unfinished Revolution," December 9, 1998, <stanford-online.stanford.edu/engelbart/>.

60. "Douglas Engelbart Inventor Profile," 2002, <www.invent.org/hall_of_fame/53.html>.

61. World Wide Web Consortium, "Longer Bio for Tim Berners-Lee," <www.w3.org/People/Berners-Lee/Longer.html>.

62. Quittner, J., "Tim Berners-Lee," March 29, 1999, <www.time.com/time/time100/scientist/profile/bernerslee.html>.

63. Berners-Lee, T., et al, "The World-Wide Web," *Communications of the ACM*, Vol. 37, No. 8, August 1994, pp. 76-82.

64. World Wide Web Consortium, "Longer Bio for Tim Berners-Lee," <www.w3.org/People/Berners-Lee/Longer.html>.

65. Quittner, J., "Tim Berners-Lee," March 29, 1999, <www.time.com/time/time100/scientist/profile/bernerslee.html>.

66. Berners-Lee, T., et al, "The World-Wide Web," *Communications of the ACM*, Vol. 37, No. 8, August 1994, pp. 76-82.

67. World Wide Web Consortium, "Longer Bio for Tim Berners-Lee," <www.w3.org/People/Berners-Lee/Longer.html>.

68. Quittner, J., "Tim Berners-Lee," March 29, 1999, <www.time.com/time/time100/scientist/profile/bernerslee.html>.

69. Moore, G., "Cramming More Components onto Integrated Circuits," *Electronics*, Vol. 38, No. 8, April 19, 1965.

70. "One Trillion-Operations-Per-Second," *Intel Press Release,* December 17, 1996, <www.intel.com/pressroom/archive/releases/cn121796.htm>.

71. Mukherjee, B.; K. Schwan; and P. Gopinath, "A Survey of Multiprocessor Operating Systems," Georgia Institute of Technology, November 5, 1993, p. 2.

72. "Microsoft Timeline," <www.microsoft.com/museum/mustimeline.mspx>.

73. Lea, R.; P. Armaral; and C. Jacquemot, "COOL-2: An Object-oriented Support Platform Built Above the CHORUS Micro-kernel," *Proceedings of the International Workshop on Object Orientation in Operating Systems 1991*, October, 1991.

74. Weiss, A., "The Politics of Free (Software)," *netWorker,* September 2001, p. 26.

75. "The GNU Manifesto," <www.delorie.com/gnu/docs/GNU/GNU>.

76. Weiss, A., "The Politics of Free (Software)," *netWorker,* September 2001, p. 27.

77. de Brouwer, C., eds, "Linus Torvalds," October 19, 2002, <www.thocp.net/biographies/torvalds_linus.html>.

78. Learmonth, M., "Giving It All Away," May 8, 1997, <www.metroactive.com/papers/metro/05.08.97/cover/linus-9719.html>.

79. Torvalds, L., "Linux History," July 31, 1992, <www.li.org/linuxhistory.php>.

80. Free Software Foundation, "GNU General Public License," May 26, 2003, <www.gnu.org/copyleft/gpl.html>.

81. Torvalds, L., "Linux History," July 31, 1992, <www.li.org/linuxhistory.php>.

82. Ghosh, R., "What Motivates Free Software Developers?" 1998, <www.firstmonday.dk/issues/issue3_3/torvalds/>.

83. Wirzenius, L., "Linux: The Big Picture," April 28, 2003, <liw.iki.fi/liw/texts/linux-the-big-picture.html>.

84. Learmonth, M., "Giving It All Away," May 8, 1997, <www.metroactive.com/papers/metro/05.08.97/cover/linus-9719.html>.

85. Wirzenius, L., "Linux: The Big Picture," April 28, 2003, <liw.iki.fi/liw/texts/linux-the-big-picture.html>.

86. "Linux Creator Linus Torvalds Joins OSDL," June 17, 2003, <www.osdl.org/newsroom/press_releases/2003/2003_06_17_beaverton.html>.

87. de Brouwer, C., eds, "Linus Torvalds," October 19, 2002, <www.thocp.net/biographies/torvalds_linus.html>.

88. Weiss, A., "The Politics of Free (Software)," *netWorker,* September 2001, pp. 27–28.

89. Stallman, R., "A Serious Bio," <www.stallman.org/#serious>.

90. Stallman, R., "Overview of the GNU Project," June 7, 2003, <www.gnu.org/gnu/gnu-history.html>.

91. DiBona, C.; S. Ockman; and M. Stone, eds., *Open Sources: Voices from the Open Source Revolution*, Boston, MA: O'Reilly, 1999.

92. R. Stallman, "Overview of the GNU Project," June 7, 2003, <www.gnu.org/gnu/gnu-history.html>.

93. DiBona, C.; S. Ockman; and M. Stone, eds., *Open Sources: Voices from the Open Source Revolution*, Boston, MA: O'Reilly, 1999.

94. Stallman, R., "A Serious Bio," <www.stallman.org/#serious>.

95. "Free Software Foundation," June 12, 2002, <www.gnu.org/fsf/fsf.html>.

96. DiBona, C.; S. Ockman; and M. Stone, eds., *Open Sources: Voices from the Open Source Revolution*, Boston, MA: O'Reilly, 1999.

97. Leon, M., "Richard Stallman, GNU/Linux," October 6, 2000, <archive.infoworld.com/articles/hn/xml/00/10/09/001009hnrs.xml>.

98. Stallman, R., "A Serious Bio," <www.stallman.org/#serious>.

99. Ricciuti, M., "New Windows Could Solve Age Old Format Puzzle—at a Price," *CNet,* March 13, 2002, <news.com.com/2009-1017-857509.html>.

100. Thurrott, P., "Windows 'Longhorn' FAQ," *Paul Thurrott's SuperSite for Windows*, modified October 6, 2003, <www.winsupersite.com/faq/longhorn.asp>.

101. Cannon, M. D., et al., "A Virtual Machine Emulator for Performance Evaluation," *Communications of the ACM*, Vol. 23, No. 2, February 1980, p. 72.

102. Cannon, M. D., et. al., "A Virtual Machine Emulator for Performance Evaluation," *Communications of the ACM*, Vol. 23, No. 2, February 1980, p. 72.

103. "VMware: Simplifying Computer Infrastructure and Expanding Possibilities", <www.vmware.com/company/>.

104. Cannon, M. D., et. al., "A Virtual Machine Emulator for Performance Evaluation," *Communications of the ACM*, Vol. 23, No. 2, February 1980, p. 73.

105. "Shell," *whatis.com,* <www.searchsolaris.techtarget.com/sDefinition/0,,sid12_gci212978,00.html>.

106. Mukherjee, B.; K. Schwan; and P. Gopinath, "A Survey of Multiprocessor Operating Systems," *Georgia Institute of Technology (GIT-CC-92/0)*, November 5, 1993, p. 4.

107. Dijkstra, E. W., "The Structure of the 'THE'-Multiprogramming System," *Communications of the ACM*, Vol. 11, No. 5, May 1968, pp. 341–346.

108. Karnik, N. M., and A. R. Tripathi, "Trends in Multiprocessor and Distributed Operating Systems," *Journal of Supercomputing*, Vol. 9, No. 1/2, 1995, pp. 4–5.

109. Mukherjee, B.; K. Schwan; and P. Gopinath, "A Survey of Multiprocessor Operating System Kernels," *Georgia Institute of Technology (GIT-CC-92/0)*, November 5, 1993, p. 10.

110. Miljocic, D. S.; F. Douglis; Y. Paindaveine; R. Wheeler; and S. Zhou, "Process Migration," *ACM Computing Surveys,* Vol. 32, No. 3, September, 2000, p. 263.

111. Liedtke, J., "Toward Real Microkernels," *Communications of the ACM*, Vol. 39, No. 9, September 1996, p. 75. Camp, T., and G. Oberhsause, "Microkernels: A Submodule for a Traditional Operating Systems Course," *Communications of the ACM*, 1995, p. 155.

112. Liedtke, J., "Toward Real Microkernels," *Communications of the ACM*, Vol. 39, No. 9, September 1996, p. 75. Camp, T., and G. Oberhsause, "Microkernels: A Submodule for a Traditional Operating Systems Course," *Communications of the ACM*, 1995, p. 155.

113. Miljocic, D. S.; F. Douglis; Y. Paindaveine; R. Wheeler; and S. Zhou, "Process Migration," *ACM Computing Surveys*, Vol. 32, No. 3, September 2000, p. 263.

114. Tanenbaum, A. S., and R. V. Renesse, "Distributed Operating Systems," *Computing Surveys,* Vol. 17, No. 4, December 1985, p. 424.

115. Tanenbaum, A. S., and R. V. Renesse, "Distributed Operating Systems," *Computing Surveys,* Vol. 17, No. 4, December 1985, p. 424.

116. Blair, G. S.; J. Malik; J. R. Nicol; and J. Walpole, "Design Issues for the COSMOS Distributed Operating System," *Proceedings from the 1988 ACM SIGOPS European Workshop*, 1988, pp. 1–2.

117. "MIT LCS Parallel and Distributed Operating Systems," June 2, 2003, <www.pdos.lcs.mit.edu>.

118. "Amoeba WWW Home Page," April 1998, <www.cs.vu.nl/pub/amoeba/>.

119. Engler, D. R.; M. F. Kaashoek; and J. O'Toole, Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *SIGOPS '95*, December 1995, p. 252.

120. Brooks, Jr., F. P., *The Mythical Man-Month: Essays on Software Engineering*, Anniversary edition, Reading, MA: Addison-Wesley, 1995.

121. Brinch Hansen, P., *Classic Operating Systems: From Batch to Distributed Systems*, Springer-Verlag, 2001.