"Now! Now!" cried the Queen. "Faster! Faster!"

—Lewis Carroll—

To conquer without risk is to triumph without glory.

—Pierre Corneille—

Our life is frittered away by detail ...Simplify, simplify.

—Henry Thoreau—

O holy simplicity!

—John Huss—
(Last words, at the stake)

Chapter 2

Hardware and Software Concepts

Objectives

After reading this chapter, you should understand:

- *hardware components that must be managed by an operating system.*
- *how hardware has evolved to support operating system functions.*
- *how to optimize performance of various hardware devices.*
- *the notion of an application programming interface (API).*
- *the process of compilation, linking and loading.*

Chapter 2

Chapter Outline

2.1 Introduction

Today's computers allow users to access the Internet, browse Web pages, display graphics and video, play music and games—and more. Personal and office computers increase productivity by managing large amounts of data, providing application-development tools and presenting an intuitive interface for authoring content. Networks of computers coordinate to perform vast numbers of calculations and transactions per second. In the mobile computing market, cell phones store phone numbers, send and receive text messages and even capture photos and video. All of these computers contain various types of hardware and software, and they are all managed by operating systems.

Because the operating system is primarily a resource manager, its design must be intimately tied to the hardware and software resources that it manages. These resources include processors, memory, secondary storage (such as hard disks), other I/O devices, processes, threads, files, databases and so on. As computers evolve, operating systems must adapt to emerging hardware and software technologies and maintain compatibility with an installed base of older hardware and software. In this chapter, we introduce hardware and software concepts.

Self Review

1. List some common hardware and software resources managed by operating systems.
2. List the types of data referenced in the preceding introduction.

Ans: **1)** Processors, memory, secondary storage and other devices, processes, threads, files and databases. **2)** Web pages, graphics, video, music, game data, office data, content, transaction data, cell phone numbers, text messages, photos, data in memory, data in secondary storage, data input or output by I/O devices and data processed by processors.

2.2 Evolution of Hardware Devices

Every time technological development has allowed for increased computing speeds, the new capabilities have immediately been absorbed by demands placed on computing resources by more ambitious applications. Computing appears to be an inexhaustible resource. Ever more interesting problems await the availability of increasingly powerful computing systems, as predicted by Moore's law (see the Biographical Note, Gordon Moore and Moore's Law). We have a "chicken or the egg" situation. Is it increasing applications demands that force computing technology to evolve, or is it improvements in technology that tempt us to think about new and innovative applications?

Initially, **systems programming**, which entailed writing code to perform hardware management and provide services to programs, was relatively straightforward because the operating system managed a small number of programs and hardware resources. Operating systems facilitate **applications programming**, because developers can write software that requests services and resources from the operating system to perform tasks (e.g., text editing, loading Web pages or payroll processing)

without needing to write code to perform device management. As the number of hardware manufacturers and devices proliferated, operating systems became more complex. To facilitate systems programming and improve extensibility, most operating systems are written to be independent of a system's particular hardware configuration. Operating systems use device drivers, often provided by hardware manufacturers, to perform device-specific I/O operations. This enables the operating system to support a new device simply by using the appropriate device driver. In fact, device drivers are such an integral part of today's systems that they comprise approximately 60 percent of the source code for the Linux kernel.[1]

Many hardware components have been designed to interact with the operating system in a way that facilitates operating system extensibility. For example, **plug-and-play** devices identify themselves to the operating system when they are

Biographical Note

Gordon Moore and Moore's law

Dr. Gordon E. Moore earned his B.S. in Chemistry from the University of California at Berkeley and Ph.D. in Chemistry and Physics from the California Institute of Technology.[2] He co-founded the Intel Corporation, the largest processor manufacturer in the computing industry. Moore is currently a Chairman Emeritus of Intel Corporation.[3] He is also known for his prediction regarding the progress of computing power that has been named **Moore's law**. Contrary to its name, Moore's law is not a provable fact. In Moore's 1965 paper, "Cramming More Components onto Integrated Circuits," he observed that the number of transistors in processors had doubled roughly every year.[4] **Transistors** are miniature switches that control electric current (just as a light switch is turned on or off). The faster the switch can be flipped, the faster the processor can execute; the more transistors, the more tasks a processor can do at once. Moore predicted that the increase in transistor count would continue for about a decade. By 1975, Moore adjusted his "law" to predict that transistor counts would double every 24 months.

Currently, processor performance is doubling roughly every 18 months and transistor count is doubling every 24 months (Fig. 2.1). A key factor that enables this is that the cost per transistor in processors is decreasing exponentially. There are other trends related to Moore's law. For one, the size of transistors is becoming exponentially smaller. The reduction in transistor size has outpaced the growth of the number of transistors on the die (i.e., the chip containing the processor), providing increased computational power from smaller processors. Smaller transistors also operate faster than large ones.

Recent advances in nanotechnology (technology at the scale of molecules) have enabled semiconductor manufacturers to create transistors consisting of a handful of atoms. Soon, however, researchers will be limited by the size of an atom when designing a transistor. To continue to extend Moore's law, companies such as Intel are investigating new techniques to modify transistor construction and create high-performance alternatives to transistor technology.[5]

connected to the computer (see Section 2.4.4, Plug and Play). This enables the operating system to select and use an appropriate device driver with little or no user interaction, simplifying the installation of a new device. From the user perspective, devices that are added to the system are ready to use almost immediately.

The hardware discussions in the next several sections focus on general-purpose computers (e.g., personal computers and servers)—special-purpose computers, such as those in cell phones or cars, are beyond the scope of this book. We discuss the common hardware components found in typical computer systems, then focus on hardware components specifically designed to support operating system functionality.

Self Review

1. Why are operating systems more difficult to design today than 50 years ago?
2. How do drivers and interfaces such as plug-and-play facilitate operating system extensibility?

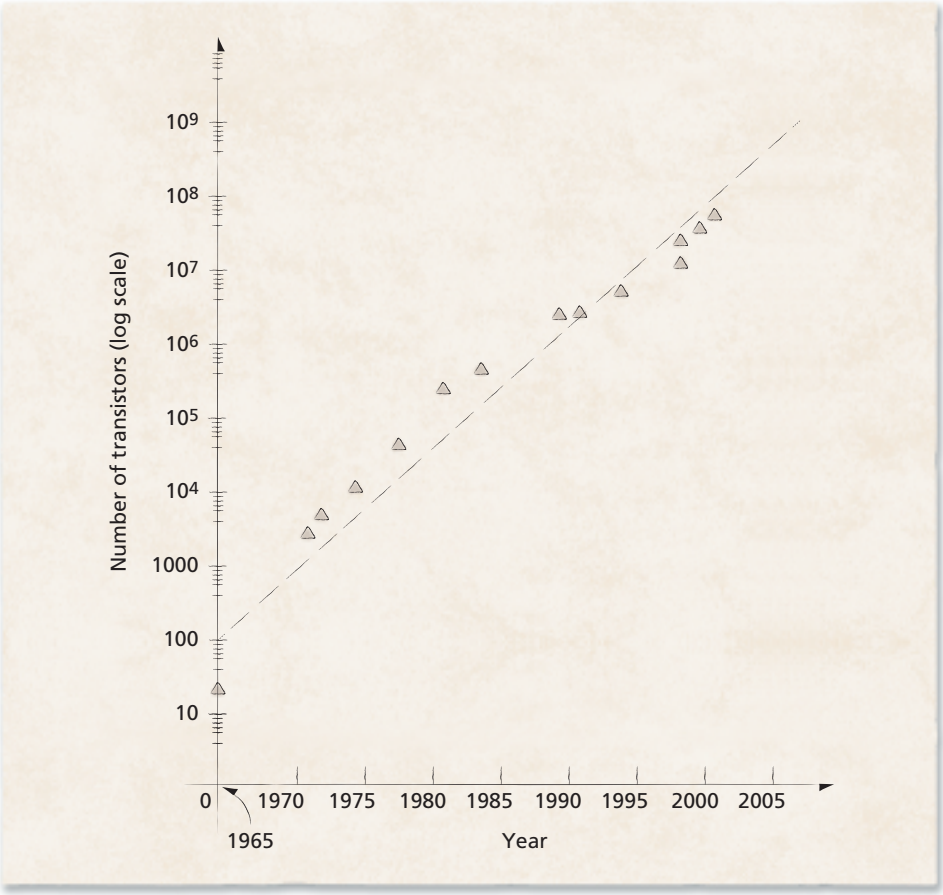

Figure 2.1 | Transistor count plotted against time for Intel processors.[6]

Ans: **1)** The operating systems of 50 years ago managed a small number of programs and hardware devices. Today's operating systems typically manage a large number of programs and a set of hardware devices that vary from one computer to another. 2) Drivers free the operating system designer from the details of interacting with hardware devices. Operating systems can support new hardware simply by using the appropriate device driver. Plug-and-play devices enable the operating system to easily identify a computer's hardware resources, which facilitates installation of devices and their corresponding drivers. From the user perspective, a device is ready to use almost immediately after it is installed.

2.3 Hardware Components

A computer's hardware consists of its physical devices—processor(s), main memory and input/output devices. The following subsections describe hardware components that an operating system manages to meet its users' computing needs.

2.3.1 Mainboards

Computers rely on interactions between many hardware devices to satisfy the requirements of the system. To enable communication among independent devices, computers are equipped with one or more **printed circuit boards (PCBs)**. A PCB is a hardware component that provides electrical connections between devices at various locations on the board.

The **mainboard** (also called the **motherboard**), the central PCB in a system, can be thought of as the backbone of a computer. The mainboard provides slots into which other components—such as the processor, main memory and other hardware devices—are inserted. These slots provide access to the electrical connections between the various hardware components and enable users to customize their computers' hardware configuration by adding devices to, and removing them from, the slots. The mainboard is one of four hardware components required to execute instructions in a general-purpose computer. The other three are the processor (Section 2.3.2, Processors), main memory (Section 2.3.5, Main Memory) and secondary storage (Section 2.3.6, Secondary Storage).

Traditional metal wires are too wide for establishing the large number of electrical connections between components in today's systems. Thus, mainboards typically consist of several extremely thin layers of silicon containing microscopic electrical connections called **traces** that serve as communication channels and provide connectivity on the board. A large set of traces forms a high-speed communication channel known as a **bus**.

Most mainboards include several computer chips to perform low-level operations. For example, mainboards typically contain a **basic input/output system (BIOS)** chip that stores instructions for basic hardware initialization and management. The BIOS is also responsible for loading the initial portion of the operating system into memory, a process called **bootstrapping** (see Section 2.4.3, Bootstrapping). After the operating system has been loaded, it can use the BIOS to communicate with a system's hardware to perform low-level (i.e., basic) I/O operations. Mainboards also

contain chips called **controllers** that manage data transfer on the board's buses. A mainboard's **chipset** is the collection of controllers, coprocessors, buses and other hardware integrated onto the mainboard that determine the system's hardware capabilities (e.g., which types of processors and memory are supported).

A recent trend in mainboard design is to integrate powerful hardware components onto the PCB. Traditionally, many of these were inserted into slots as **add-on cards**. Many of today's mainboards include chips that perform graphics processing, networking and RAID (Redundant Array of Independent Disks) operations. These **on-board devices** reduce the overall system cost and have contributed significantly to the continuing sharp decline in computer prices. A disadvantage is that they are permanently attached to the mainboard and cannot be replaced easily.

Self Review

1. What is the primary function of the mainboard?
2. Why is the BIOS crucial to computer systems?

Ans: **1)** The mainboard serves as the backbone for communication between hardware components, allowing them to communicate via the electrical connections on the board. **2)** The BIOS performs basic hardware initialization and management and loads the initial component of the operating system into memory. The BIOS also provides instructions that enable the operating system to communicate with system hardware.

2.3.2 Processors

A **processor** is a hardware component that executes a stream of machine-language instructions. Processors can take many forms in computers, such as a **central processing unit (CPU)**, a graphics **coprocessor** or a digital signal processor (DSP). A CPU is a processor that executes the instructions of a program; a coprocessor, such as a graphics or digital signal processor, is designed to efficiently execute a limited set of special-purpose instructions (such as 3D transformations). In embedded systems, processors might perform specific tasks, such as converting a digital signal to an analog audio signal in a cell phone—an example of a DSP. As a primary processor in the system, a CPU executes the bulk of the instructions, but might increase efficiency by sending computationally intensive tasks to a coprocessor specifically designed to handle them. Throughout the rest of this book, we use the term "processor" or "general-purpose processor" when referring to a CPU.

The instructions a processor can execute are defined by its instruction set. The size of each instruction, or the **instruction length**, might differ among architectures and within each architecture—some processors support multiple instruction sizes. The processor architecture also determines the amount of data that can be operated on at once. For instance, a 32-bit processor manipulates data in discrete units of 32 bits.

Modern processors perform many resource management operations in hardware to boost performance. Such features include support for virtual memory and hardware interrupts—two important concepts discussed later in this book.

Despite the variety of processor architectures, several components are present in almost all contemporary processors. Such components include the instruction fetch unit, branch predictor, execution unit, registers, caches and a bus interface (Fig. 2.2). The **instruction fetch unit** loads instructions into high-speed memory called instruction registers so that the processor can execute the instruction quickly. The **instruction decode unit** interprets the instruction and passes the corresponding input for the execution unit to perform the instruction. The main portion of the execution unit is the **arithmetic and logic unit (ALU)**, which performs basic arithmetic and logical operations, such as addition, multiplication and logical comparisons (note that the "V" shape of the ALU is common in architecture diagrams).

The bus interface allows the processor to interact with memory and other devices in the system. Because processors typically operate at much higher speeds than main memory, they contain high-speed memory called cache that stores copies of data in main memory. Caches increase processor efficiency by enabling fast access to data and instructions. Because high-speed caches are significantly more expensive than main memory, they tend to be relatively small. The caches are classified in levels—Level 1 (L1) is the fastest and most expensive cache and is located on the processor; the Level 2 (L2) cache, which is larger and slower than the L1 cache,

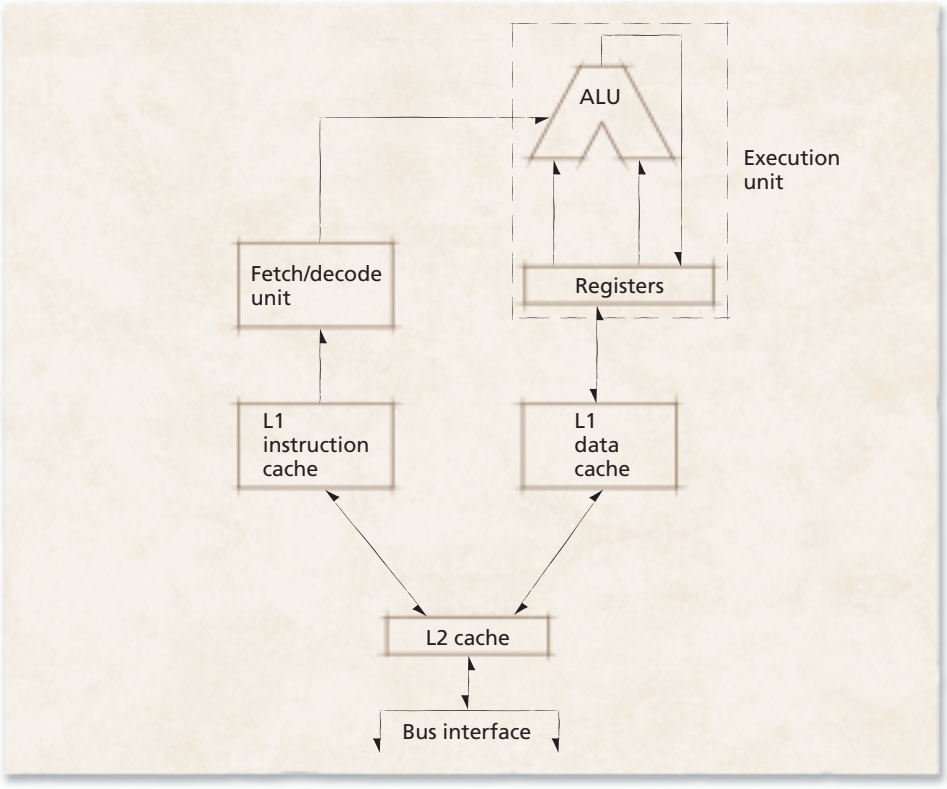

Figure 2.2 | *Processor components.*

is often located on the mainboard, but is increasingly being integrated onto the processor to improve performance.[7]

Registers are high-speed memories located on a processor that hold data for immediate use by the processor. Before a processor can operate on data, the data must be placed in registers. Storing processor instructions in any other slower type of memory would be inefficient, because the processor would idle while waiting for data access. Registers are hard-wired to the processor circuitry and physically located near the execution units, making access to registers faster than access to the L1 cache. The size of the registers is determined by the number of bits the processor can operate on at once. For example, a 32-bit processor can store 32 bits of data in each register. The majority of processors in personal computers today are 32-bit processors; 64-bit processors are becoming increasingly popular.[8]

Each processor architecture provides a different number of registers, and each register serves a particular purpose. For example, the Intel Pentium 4 processor provides 16 program execution registers. Typically, half of these registers are reserved for use by applications for quick access to data values and pointers during execution. Such registers are called **general-purpose registers**. IBM's PowerPC 970 processor (used in Apple's G5 computers) contains 32 general-purpose registers. The other registers (often called control registers) store system-specific information, such as the program counter, which the processor uses to determine the next instruction to execute.[9]

Self Review

1. Differentiate between a CPU and a coprocessor. How might a system benefit from multiple CPUs? How might a system benefit from multiple coprocessors?
2. What aspects of a system does a processor architecture specify?
3. Why is access to register memory faster than access to any other type of memory, including L1 cache?

Ans: **1)** A CPU executes machine-language instructions; a coprocessor is optimized to perform special-purpose instructions. Multiple CPUs would allow a system to execute more than one program at once; multiple coprocessors could improve performance by performing processing in parallel with a CPU. **2)** A CPU's architecture specifies the computer's instruction set, virtual memory support and interrupt structure. **3)** Registers are hard-wired to the processor circuitry and physically located near the execution units.

2.3.3 Clocks

Computer time is often measured in **cycles**, also called a **clocktick**. The term cycle refers to one complete oscillation of an electrical signal provided by the system clock generator. The clock generator sets the cadence for a computer system, much like the conductor of an orchestra. Specifically, the clock generator determines the frequency at which buses transfer data, typically measured in cycles per second, or hertz (Hz). For example, the **frontside bus (FSB)**, which connects processors to memory modules, typically operates at several hundred megahertz (MHz; one megahertz is one million hertz).

Most modern desktop processors execute at top speeds of hundreds of megahertz (MHz) or even several billion hertz, or gigahertz (GHz), which is often faster than the frontside bus. Processors and other devices generate **derived speeds** by multiplying or dividing the speed of the frontside bus.[10] For example, a 2GHz processor with a 200MHz frontside bus uses a multiplier of 10 to generate its cycles; a 66MHz sound card uses a divider of 2.5 to generate its cycles.

Self Review

1. (T/F) All components of a system operate at the same clock speed.
2. What problems might arise if one component on a bus has an extremely high multiplier and another component on the same bus has an extremely high divider?

Ans: **1)** False. Devices usually use a multiplier or a divider that defines the device's speed relative to the speed of the frontside bus. **2)** Bottlenecks could occur, because a component with a high divider will operate at a much slower speed than a device with a high multiplier. A high-multiplier device that relies on information from a high-divider device will be made to wait.

2.3.4 Memory Hierarchy

The size and the speed of memory are limited by the laws of physics and economics. Almost all electronic devices transfer data using electrons passing through traces on PCBs. There is a limit to the speed at which electrons can travel; the longer the wire between two terminals, the longer the transfer will take. Further, it is prohibitively expensive to equip processors with large amounts of memory that can respond to requests for data at (or near) processor speeds.

The cost/performance trade-off characterizes the **memory hierarchy** (Fig. 2.3). The fastest and most expensive memory is at the top and typically has a small capacity. The slowest and least expensive memory is at the bottom and typically has a large capacity. Note that the size of each block represents how capacity increases for slower memories, but the figure is not drawn to scale.

Registers are the fastest and most expensive memory on a system—they operate at the same speed as processors. Cache memory speeds are measured according to their latency—the time required to transfer data. Latencies are typically measured in nanoseconds or processor cycles. For example, the L1 cache for an Intel Pentium 4 processor operates at a latency of two processor cycles.[11] Its L2 cache operates with a latency of approximately 10 cycles. In many of today's processors, the L1 and L2 cache are integrated onto the processor so that they can exploit the processor's high-speed interconnections. L1 caches typically store tens of kilobytes of data while L2 caches typically store hundreds of kilobytes or several megabytes. High-end processors might contain a third level of processor cache (called the L3 cache) that is slower than the L2 cache but is faster than main memory.

Next in the hierarchy is **main memory**—also called **real memory** or **physical memory**. Main memory introduces additional latency because data must pass through the frontside bus, which typically operates at a fraction of processor speeds. Main memory in today's architectures exhibits latencies of tens or hundreds of pro-

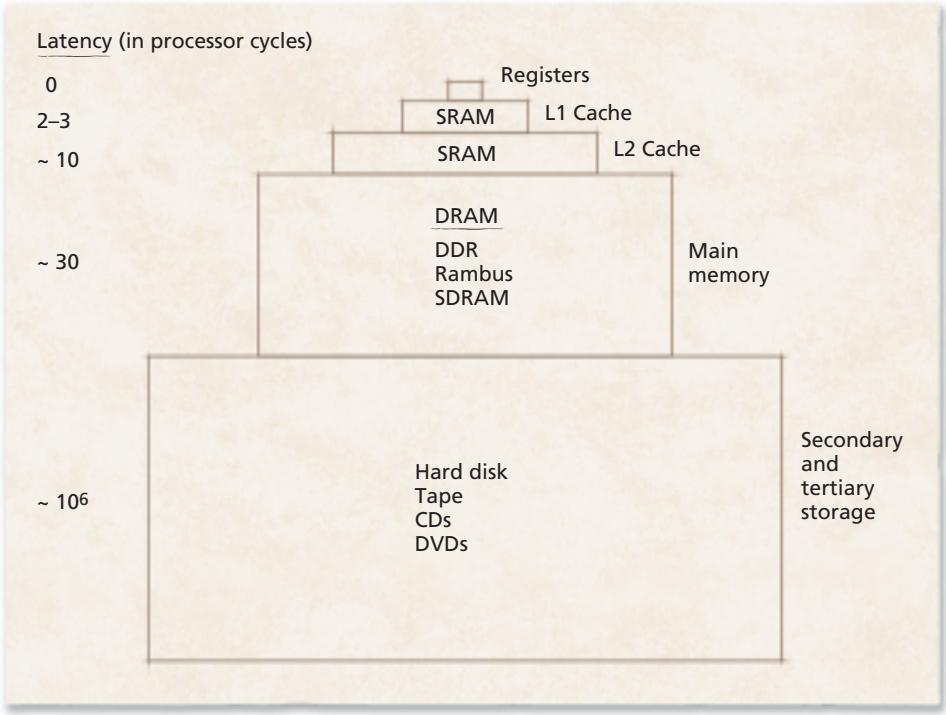

Figure 2.3 | *Memory hierarchy.*

cessor cycles.[12] Current general-purpose main memory sizes range from hundreds of megabytes (PCs) to tens or hundreds of gigabytes (high-end servers). Main memory is discussed in Section 2.3.5, Main Memory, and in Chapter 9, Real Memory Organization and Management. Registers, caches and main memory are typically **volatile** media, so their data vanishes when they lose power.

The hard disk and other storage devices such as CDs, DVDs and tapes are among the least expensive and slowest data storage units in a computer system. Disk storage device latencies are typically measured in milliseconds, typically a million times slower than processor cache latencies. Rather than allow a processor to idle while a process waits for data from secondary storage, the operating system typically executes another process to improve processor utilization. A primary advantage to secondary storage devices such as hard disks is that they have large capacities, often hundreds of gigabytes. Another advantage to secondary storage is that data is stored on a persistent medium, so data is preserved when power is removed from the device. Systems designers must balance the cost and the performance of various storage devices to meet the needs of users (see the Operating Systems Thinking feature, Caching).

Self Review

1. What is the difference between persistent and volatile storage media?
2. Why does the memory hierarchy assume a pyramidal shape?

Ans: **1)** Volatile media lose their data when the computer is turned off, whereas persistent media retain the data. In general, volatile storage is faster and more expensive than persistent storage. **2)** If a storage medium is less expensive, users can afford to buy more of it; thus, storage space increases.

2.3.5 Main Memory

Main memory consists of volatile **random access memory (RAM)**, "random" in the sense that processes can access data locations in any order. In contrast, data locations on a sequential storage medium (e.g., tape) must be read sequentially. Unlike tapes and hard disks, memory latencies for each main memory address are essentially equal.

The most common form of RAM is **dynamic RAM (DRAM)**, which requires that a refresh circuit periodically (a few times every millisecond) read the contents or the data will be lost. This differs from **static RAM (SRAM)**, which does not need to be refreshed to maintain the data it stores. SRAM, which is commonly employed in processor caches, is typically faster and more expensive than DRAM.

An important goal for DRAM manufacturers is to narrow the gap between processor speed and memory-transfer speed. Memory modules are designed to minimize data access latency within the module and maximize the number of times data is transferred per second. These techniques reduce overall latency and increase

Operating Systems Thinking

Caching

We all use caching in our everyday lives. Generally speaking, a cache is a place for storing provisions that can be accessed quickly. Squirrels stashing acorns as they prepare for the winter is a form of caching. We keep pencils, pens, staples, tape and paper clips in our desk drawers so that we can access them quickly when we need them (rather than having to walk down the hall to the supply closet). Operating systems employ many caching techniques, such as caching a process's data and instructions for rapid access in high-speed cache memories and caching data from disk in main memory for rapid access as a program runs.

Operating systems designers must be cautious when using caching because in computer systems, cached data is a copy of the data whose original is being maintained at a higher level in the memory hierarchy. The cached copy is usually the one to which changes are made first, so it can quickly become out of sync with the original data, causing inconsistency. If a system were to fail when the cache contains updated data and the original does not, then the modified data could be lost. So operating systems frequently copy the cached data to the original—this process is called flushing the cache. Distributed file systems often place cache on both the server and the client, which makes it even more complex to keep the cache consistent.

bandwidth—the amount of data that can be transferred per unit of time. As manufacturers develop new memory technologies, the memory speed and capacity tend to increase and the cost per unit of storage tends to decrease, in accordance with Moore's law.

Self Review

1. Compare main memory to disk in terms of access time, capacity and volatility.
2. Why is main memory called random access memory?

Ans: **1)** Access times for main memory are much smaller than those for disk. Disks typically have a larger capacity than main memory, because the cost per unit storage for disks is less than for main memory. Main memory is typically volatile, whereas disks store data persistently. **2)** Processes can access main memory locations in any order and at about the same speed, regardless of location.

2.3.6 Secondary Storage

Due to its limited capacity and volatility, main memory is unsuitable for storing data in large amounts or data that must persist after a power loss. To permanently store large quantities of data, such as data files and applications software, computers use **secondary storage** (also called **persistent** or **auxiliary storage**) that maintains its data after the computer's power is turned off. Most computers use hard disks for secondary storage.

Although hard disk drives store more and cost less than RAM, they are not practical as a primary memory store because access to hard disk drives is much slower than access to main memory. Accessing data stored on a hard disk requires mechanical movement of the read/write head, rotational latency as the data spins to the head, and transfer time as the data passes by the head. This mechanical movement is much slower than the speed of electrical signals between main memory and a processor. Also, data must be loaded from the disk into main memory before it can be accessed by a processor.[13] A hard disk is an example of a **block device**, because it transmits data in fixed-size blocks of bytes (normally hundreds of bytes to tends of kilobytes).

Some secondary storage devices record data on lower-capacity media that can be removed from the computer, facilitating data backup and data transfer between computers. However, this type of secondary storage typically exhibits higher latency than other devices such as hard disks. A popular storage device is the **compact disk (CD)**, which can store up to 700MB per side. Data on CDs is encoded in digital form and "burned" onto the CD as a series of pits on an otherwise flat surface that represent ones and zeroes. **Write-once, read-many (WORM)** disks, such as write-once compact disks (CD-R) and write-once digital versatile disks (DVD-R) are removable. Other types of persistent storage include Zip disks, floppy disks, Flash memory cards and tapes.

Data recorded on a CD-RW (rewritable CD) is stored in metallic material inside the plastic disk. Laser light changes the reflective property of the recording

medium, creating two states representing one and zero. CD-Rs and CD-ROMs consist of a dye between plastic layers that cannot be altered, once it has been burned by the laser.

Recently, digital versatile disk (DVD; also called digital video disk) technology, which was originally intended to record movies, has become an affordable data storage medium. DVDs are the same size as CDs, but store data in thinner tracks on up to two layers per side and can store up to 5.6 GB of data per layer.

Some systems contain levels of memory beyond secondary storage. For example, large data-processing systems often have tape libraries that are accessed by a robotic arm. Such storage systems, often classified as tertiary storage, are characterized by larger capacity and slower access times than secondary storage.

Self Review

1. Why is accessing data stored on disk slower than accessing data in main memory?
2. Compare and contrast CDs and DVDs.

Ans: **1)** Main memory can be accessed by electrical signals alone, but disks require mechanical movements to move the read/write head, rotational latency as the disk spins to move the requested data to the head and transfer time as the data passes by the head. **2)** CDs and DVDs are the same size and are accessed by laser light, but DVDs store data in multiple layers using thinner tracks and thus have a higher capacity.

2.3.7 Buses

A bus is a collection of traces (or other electrical connections) that transport information between hardware devices. Devices send electrical signals over the bus to communicate with other devices. Most buses consist of a **data bus**, which transports data, and an **address bus**, which determines the recipient or sources of that data.[14] A **port** is a bus that connects exactly two devices. A bus that several devices share to perform I/O operations is also called an **I/O channel**.[15]

Access to main memory is a point of contention for channels and processors. Typically, only one access to a particular memory module may occur at any given time; however, the I/O channels and the processor may attempt to access main memory simultaneously. To prevent the two signals from colliding on the bus, the memory accesses are prioritized by a hardware device called a controller, and channels are typically given priority over processors. This is called **cycle stealing**, because the I/O channel effectively steals cycles from the processor. I/O channels consume a small fraction of total processor cycles, which is typically offset by the enhanced I/O device utilization.

Recall that the frontside bus (FSB) connects a processor to main memory. As the FSB speed increases, the amount of data transferred between main memory and a processor increases, which tends to increase performance. Bus speeds are measured in MHz (e.g., 133MHz and 200MHz). Some chipsets implement an FSB of 200MHz but effectively operate at 400MHz, because they perform two memory transfers per clock cycle. This feature, which must be supported by both the chipset

and the RAM, is called **double data rate (DDR)**. Another implementation, called **quad pumping**, allows up to four data transfers per cycle, effectively quadrupling the system's memory bandwidth.

The **Peripheral Component Interconnect (PCI) bus** connects peripheral devices, such as sound cards and network cards, to the rest of the system. The first version of the PCI specification required that the PCI bus operate at 33MHz and be 32 bits wide, which considerably limited the speed with which data was transferred to and from peripheral devices. PCI Express is a recent standard that provides for variable-width buses. With PCI Express, each device is connected to the system by up to 32 lanes, each of which can transfer 250MB per second in each direction—a total of up to 16GB per second of bandwidth per link.[16]

The **Accelerated Graphics Port (AGP)** is primarily used with graphics cards, which typically require tens or hundreds of megabytes of RAM to perform 3D graphics manipulations in real time. The original AGP specification called for a 32-bit 66MHz bus, which provided approximately 260MB per second of bandwidth. Manufacturers have increased the speed of this bus from its original specification—denoting an increase in speed by a factor of 2 as 2x, by a factor of 4 as 4x, and so on. Current specifications allow for 2x, 4x and 8x versions of this protocol, permitting up to 2GB per second of bandwidth.

Self Review

1. How does FSB speed affect system performance?
2. How do controllers simplify access to shared buses?

Ans: **1)** The FSB determines how much data can be transferred between processors and main memory per cycle. If a processor generates requests for more data than can be transferred per cycle, system performance will decrease, because that processor may need to wait until its requested transfers complete. **2)** Controllers prioritize multiple simultaneous requests to access a bus so that devices do not interfere with one another.

2.3.8 Direct Memory Access (DMA)

Most I/O operations transfer data between main memory and an I/O device. In early computers, this was accomplished using **programmed I/O (PIO)**, which specifies a byte or word to be transferred between main memory and an I/O device, then waits idly for the operation to complete. This led to wasting a significant number of processor cycles while waiting for PIO operations to complete. Designers later implemented interrupt-driven I/O, which enabled a processor to issue an I/O request and immediately continue to execute software instructions. The I/O device notified the processor when the operation was complete by generating an interrupt.[17]

Direct memory access (DMA) improves upon these techniques by enabling devices and controllers to transfer blocks of data to and from main memory directly, which frees the processor to execute software instructions (Fig. 2.4). A direct memory access (DMA) channel uses an I/O controller to manage data transfer between I/O devices and main memory. To notify the processor, the I/O control-

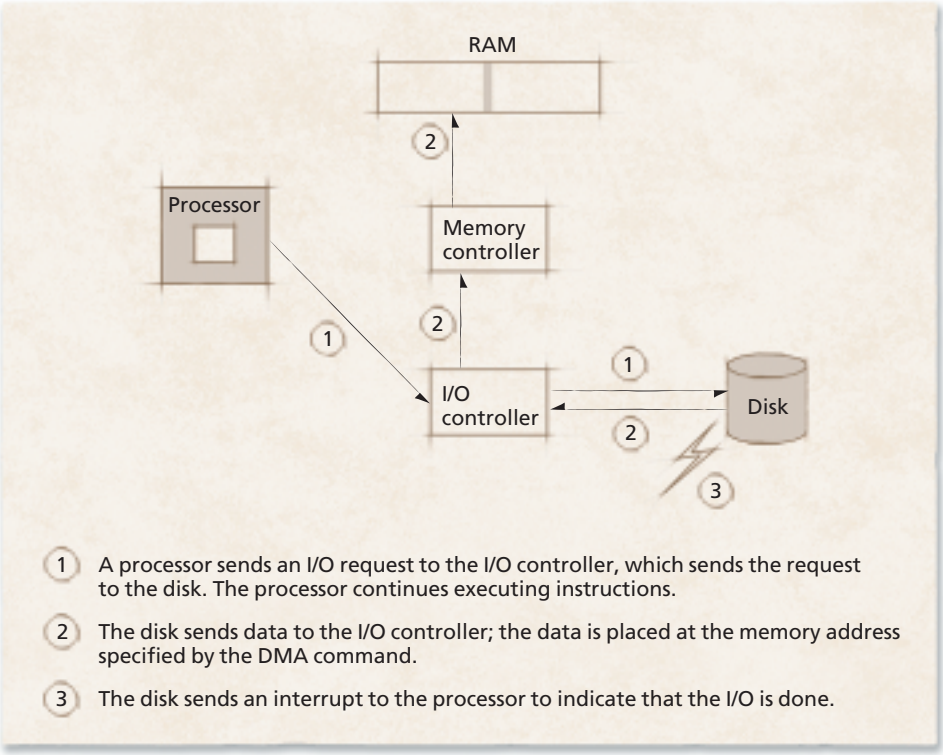

Figure 2.4 | Direct memory access (DMA).

ler generates an interrupt when the operation is complete. DMA improves performance significantly in systems that perform large numbers of I/O operations (e.g., mainframes and servers).[18]

DMA is compatible with several bus architectures. On legacy architectures (i.e., architectures that are still in use but are no longer actively produced), such as the Industry Standard Architecture (ISA), extended ISA (EISA) or Micro Channel Architecture (MCA) buses, a DMA controller (also called a "third-party device") manages transfers between main memory and I/O devices (see the Operating Systems Thinking feature, Legacy Hardware and Software). PCI buses employ "first-party" DMA using **bus mastering**—a PCI device takes control of the bus to perform the operation. In general, first-party DMA transfer is more efficient than third-party transfer and has been implemented by most modern bus architectures.[19]

Self Review

1. Why is DMA more efficient than PIO?
2. How does first-party DMA differ from third-party DMA?

Ans: **1)** In a system that uses PIO, a processor waits idly for each memory transfer to complete. DMA frees processors from performing the work necessary to transfer information

between main memory and I/O devices, which enables the processor to execute instructions instead. **2)** Third-party DMA requires a controller to manage access to the bus. First-party DMA enables devices to take control of the bus without additional hardware.

2.3.9 Peripheral Devices

A peripheral device is any hardware device that is not required for a computer to execute software instructions. Peripheral devices include many types of I/O devices (e.g., printers, scanners and mice), network devices (e.g., network interface cards and modems) and storage devices (e.g., CD, DVD and disk drives). Devices such as the processor, mainboard and main memory are not considered peripheral devices. Internal peripheral devices (i.e., those that are located inside the computer case) are often referred to as integrated peripheral devices; these include modems, sound cards and internal CD-ROM drives. Perhaps the most common peripheral device is a hard disk. Figure 2.5 lists several peripheral devices.[20] Keyboards and mice are example of **character devices**—ones that transfer data one character at a time. Peripheral devices can be attached to computers via ports and other buses.[21] **Serial ports** transfer data one bit at a time, typically connecting devices such as keyboards and mice; **parallel ports** transfer data several bits at a time, typically connecting printers.[22] **Universal Serial Bus (USB)** and **IEEE 1394 ports** are popular high-speed serial interfaces. The **small computer systems interface (SCSI)** is a popular parallel interface.

USB ports transfer data from and provide power to devices such as external disk drives, digital cameras and printers. USB devices can be attached to, recognized by and removed from the computer while the computer is on without damaging the system's hardware (a technique called "hot swapping"). USB 1.1 allows data transfer at speeds of 1.5Mbit (megabits, or 1 million bits; 8 bits = 1 byte) per second and 12Mbit per second. Because computers required fast access to large quantities of data on USB devices such as disk drives, USB 2.0 was developed to provide data transfers at speeds up to 480Mbit per second.[23]

Operating Systems Thinking

Legacy Hardware and Software

The latest versions of operating systems are designed to support the latest available hardware and software functionality. However, the vast majority of hardware and software that is "out there" is often older equipment and applications that individuals and organizations have invested in and want to keep using, even when a new operating system is installed. The older items are called legacy hardware and legacy software. An enormous challenge for OS designers is to provide support for such legacy systems, one that real-world operating systems must meet.

Device	Description
CD-RW drive	Reads data from, and writes data to, optical disks.
Zip drive	Transfers data to and from a removable, durable magnetic disk.
Floppy drive	Reads data from, and writes data to, removable magnetic disks.
Mouse	Transmits the change in location of a pointer or cursor in a graphical user interface (GUI).
Keyboard	Transmits characters or commands that a user types.
Multifunction printer	Can print, copy, fax and scan documents.
Sound card	Converts digital signals to audio signals for speakers. Also can receive audio signals via a microphone and produce a digital signal.
Video accelerator	Displays graphics on the screen; accelerates two- and three-dimensional graphics.
Network card	Sends data to and receives data from other computers.
Digital camera	Records, and often displays, digital images.
Biometric device	Scans human characteristics, such as fingerprints and retinas, typically for identification and authentication purposes.
Infrared device	Communicates data between devices via a line-of-sight wireless connection.
Wireless device	Communicates data between devices via an omnidirectional wireless connection.

Figure 2.5 | *Peripheral devices.*

The IEEE 1394 standard, branded as "iLink" by Sony and "FireWire" by Apple, is commonly found in digital video cameras and mass storage devices (e.g., disk drives). FireWire can transfer data at speeds up to 800Mbit per second; future revisions are expected to scale to up to 2Gbit (gigabits, or 1 billion bits) per second. Similar to USB, FireWire allows devices to be "hot swappable" and can provide power to devices. Further, the FireWire specification allows multiple devices to communicate without being attached to a computer.[24] For example, a user can directly connect two FireWire hard disks to copy the contents of one to the other.

Other interfaces used for connecting peripheral devices to the system include the small computer systems interface (SCSI) and the Advanced Technology Attachment (ATA), which implements the Integrated Drive Electronics (IDE) interface. These interfaces transfer data from a device such as a hard drive or a DVD drive to a mainboard controller, where it can be routed to the appropriate bus.[25] More recent interfaces include Serial ATA (SATA), which permits higher transfer rates than ATA, and several wireless interfaces including Bluetooth (for short-range wireless connections) and IEEE 802.11g (for medium-range, high-speed wireless connections).

SCSI (pronounced "scuh-zee") was developed in the early 1980s as a high-speed connection for mass storage devices. It is primarily used in high-performance

environments with many large-bandwidth devices.[26] The original SCSI specification allowed a maximum data transfer rate of 5MB per second and supported eight devices on an 8-bit bus. Current specifications, such as Ultra320 SCSI, permit data transfer at up to 320MB per second for 16 devices on a 16-bit bus.[27]

Self Review

1. What is the main difference between a peripheral device, such as a printer, and a device such as a processor?
2. Compare and contrast USB and FireWire.

Ans: **1)** Peripheral devices are not required for a computer to execute software instructions. By contrast, all computers need at least one processor to run. **2)** Both USB and FireWire provide large bandwidths and powered connections to devices. FireWire has a greater capacity than USB and enables devices to communicate without being attached to a computer.

2.4 Hardware Support for Operating Systems

Computer architectures contain features that perform operating system functions quickly in hardware to improve performance. They also provide features that enable the operating system to rigidly enforce protection, which improves the security and integrity of the system.

2.4.1 Processor

Most operating systems rely on processors to implement their protection mechanisms by preventing processes from accessing privileged instructions or accessing memory that has not been allocated to them. If processes attempt to violate a system's protection mechanisms, the processor notifies the operating system so that it can respond. The processor also invokes the operating system to respond to signals from hardware devices.

User Mode, Kernel Mode and Privileged Instructions

Computer systems generally have several different **execution modes**.[28] Varying the mode of a machine makes it possible to build more robust, fault-tolerant and secure systems. Normally, when the machine is operating in a particular mode, applications have access to only a subset of the machine's instructions. For user applications, the subset of instructions the user may execute in **user mode** (also called the **user state** or **problem state**) precludes, for example, the direct execution of input/output instructions; a user application allowed to perform arbitrary input/output could, for example, dump the system's master list of passwords, print the information of any other user or destroy the operating system. The operating system ordinarily executes with **most trusted user status** in **kernel mode** (also called the **supervisor state**); it has access to all the instructions in the machine's instruction set. In kernel mode, a processor may execute privileged instructions and access resources to perform tasks on behalf

of processes. Such a user mode/kernel mode dichotomy has been adequate for most modern computing systems. In highly secure systems, however, it is desirable to have more than two states to allow finer-grained protection. Multiple states allow access to be granted by the **principle of least privilege**—any particular user should be granted the least amount of privilege and access required to accomplish its designated tasks (see the Operating Systems Thinking feature, Principle of Least Privilege).

It is interesting that as computer architectures have evolved, the number of **privileged instructions** (i.e., those instructions not accessible in user mode) has tended to increase. This indicates a trend toward incorporating more operating systems functions in hardware.

Memory Protection and Management

Most processors provide mechanisms for memory protection and memory management. **Memory protection**, which prevents processes from accessing memory that has not been assigned to them (such as other users' memory and the operating system's memory), is implemented using processor registers that can be modified only by privileged instructions (see the Operating Systems Thinking feature, Protection). The processor checks the values of these registers to ensure that processes cannot access memory that has not been allocated to them. For example, in systems that do not use virtual memory, processes are allocated only a contiguous block of memory addresses. The system can prevent such processes from accessing memory locations that have not been allocated to them by providing **bounds registers** that specify the addresses of the beginning and end of a process's allocated memory. Protection is enforced by determining whether a given address is within the allocated block. Most hardware protection operations are performed in parallel with the execution of program instructions, so they do not degrade performance.

Most processors also contain hardware that translates virtual addresses referenced by processes to corresponding addresses in main memory. Virtual memory systems allow programs to reference addresses that need not correspond to the lim-

Operating Systems Thinking

Principle of Least Privilege

Generally speaking, the principle of least privilege says that in any system, the various entities should be given only the capabilities that they need to accomplish their jobs but no more. The government employs this principle in awarding security clearances. You employ it when deciding who gets the extra keys to your home. Businesses employ it when giving employees access to critical and confidential information. Operating systems employ it in many areas.

ited set of **real** (or **physical**) **addresses** available in main memory.[29] Using hardware, the operating system dynamically translates a process's virtual addresses into physical addresses at runtime. Virtual memory systems allow processes to reference address spaces much larger than the number of addresses available in main memory, which allows programmers to create applications that are independent (for the most part) of the constraints of physical memory. Virtual memory also facilitates programming for timesharing systems, because processes need not be aware of the actual location of their data in main memory. Memory management and protection are discussed in detail in Chapters 9–11.

Interrupts and Exceptions

Processors inform the operating system of events such as program execution errors and changes in device status (e.g., a network packet has arrived or a disk I/O has completed). A processor can do so by repeatedly requesting the status of each device, a technique called **polling**. However, this can lead to significant execution overhead when polled devices have not changed status.

Instead, most devices send a signal called an **interrupt** to the processor when an event occurs. The operating system can respond to a change in device status by notifying processes that are waiting on such events. **Exceptions** are interrupts generated in response to errors, such as hardware failures, logic errors and protection violations (see the Anecdote, Origins of the Term "Glitch"). Instead of causing the

Operating Systems Thinking

Protection

The earliest computers had primitive operating systems capable of running only one job at a time. That changed rapidly as parallel processing capabilities were added to local systems and as distributed systems were developed in which parallel activities occur across networks of computers like the Internet. Operating systems must be concerned with various kinds of protection, especially when connected to the Internet. The operating system and its data must be protected from being clobbered by errant user programs, either accidentally or maliciously. User programs must be protected from clobbering one another. Such protection must be enforced on the local machine and it must be enforced among users and operating system components spread across computer networks. We study protection in many chapters of this book, especially in Chapter 9, Real Memory Organization and Management and Chapter 10, Virtual Memory Organization. We consider protection in the form of file access controls in Chapter 13, File and Database Systems. We discuss protection in general throughout the main portion of the book and then discuss it in the context of the Linux and Windows XP case studies in Chapters 20 and 21, respectively.

system to fail, a processor will typically invoke the operating system to determine how to respond. For example, the operating system may determine that the process causing the error should be terminated or that the system must be restarted. If the system must fail, the operating system can do so gracefully, reducing the amount of lost work. Processes can also register exception handlers with the operating system. When the operating system receives an exception of the corresponding type, it calls the process's exception handler to respond. Interrupt mechanisms and exception handling are discussed in Section 3.4, Interrupts.

Self Review

1. What is the rationale for implementing multiple execution states?
2. How do exceptions differ from other types of interrupts?

Ans: **1)** Multiple execution states provide protection by preventing most software from maliciously or accidentally damaging the system and accessing resources without authorization. These operations are restricted to kernel mode, which enables the operating system to execute privileged instructions. **2)** Exceptions indicate that an error has occurred (e.g., division by zero or a protection violation) and invoke the operating system to determine how to respond. The operating system may then decide to do nothing or to terminate a process. If the operating system encounters a serious error that prevents it from executing properly, it may restart the computer.

Anecdote

Origins of the Term "Glitch"

There are a number of theories on the etymology of the computer term "glitch" which is typically used as a synonym for "bug." Many suggest that it is derived from the Yiddish word "glitshen," meaning "to slip." Here is another take on this. In the mid 1960s during the height of the space program, one of the top computer vendors built the first on-board computer system. The morning of the launch, the computer vendor took out full-page ads in major publications around the world, proclaiming that its computer was safely guiding the astronauts on their mission. That day, the computer failed, causing the space capsule to spin wildly out of control, putting the astronauts' lives at risk. The next morning, one of the major newspapers referred to this as the "Greatest Lemon in the Company's History!"

Lesson to operating systems designers: Always keep Murphy's Law in mind, "If something can go wrong, it will." And don't forget the common addendum, "and at the most inopportune time."

2.4.2 Timers and Clocks

An **interval timer** periodically generates an interrupt that causes a processor to invoke the operating system. Operating systems often use interval timers to prevent processes from monopolizing the processor. For example, the operating system may respond to the timer interrupt by removing the current process from the processor so that another can run. A **time-of-day clock** enables the computer to keep track of "wall clock time," typically accurate to thousandths or millionths of a second. Some time-of-day clocks are battery powered, allowing them to tick even when there is no external power supplied to the computer. Such clocks provide a measure of continuity in a system; for example, when the operating system loads, it may read the time-of-day clock to determine the current time and date.

Self Review

1. How does an interval timer prevent one process from monopolizing a processor?
2. Processors often contain a counter that is incremented after each processor cycle, providing a measure of time accurate to nanoseconds. Compare and contrast this measure of time to that provided by the time-of-day clock.

Ans: **1)** The interval timer generates interrupts periodically. The processor responds to each interrupt by invoking the operating system, which can then assign a different process to a processor. 2) A processor counter enables the system to determine with high precision how much time has passed between events, but does not maintain its information when the system is powered down. Because a time-of-day clock is battery powered, it is more appropriate for determining wall clock time. However, it measures time with coarser granularity than a processor counter.

2.4.3 Bootstrapping

Before an operating system can begin to manage resources, it must be loaded into memory. When a computer system is powered up, the BIOS initializes the system hardware, then attempts to load instructions into main memory from a region of secondary storage (e.g., a floppy disk, hard disk or CD) called the **boot sector**, a technique called bootstrapping (Fig. 2.6). The processor is made to execute these instructions, which typically load operating system components into memory, initialize processor registers and prepare the system to run user applications.

In many systems, the BIOS can load an operating system from a predefined location on a limited number of devices (e.g., the boot sector of a hard disk or a compact disk). If the boot sector is not found on a supported device, the system will not load and the user will be unable to access any of the computer's hardware. To enable greater functionality at boot time, the Intel Corporation has developed the **Extensible Firmware Interface (EFI)** as a replacement for the BIOS. EFI supports a shell through which users can directly access computer devices, and it incorporates device drivers to support access to hard drives and networks immediately after powering up the system.[30]

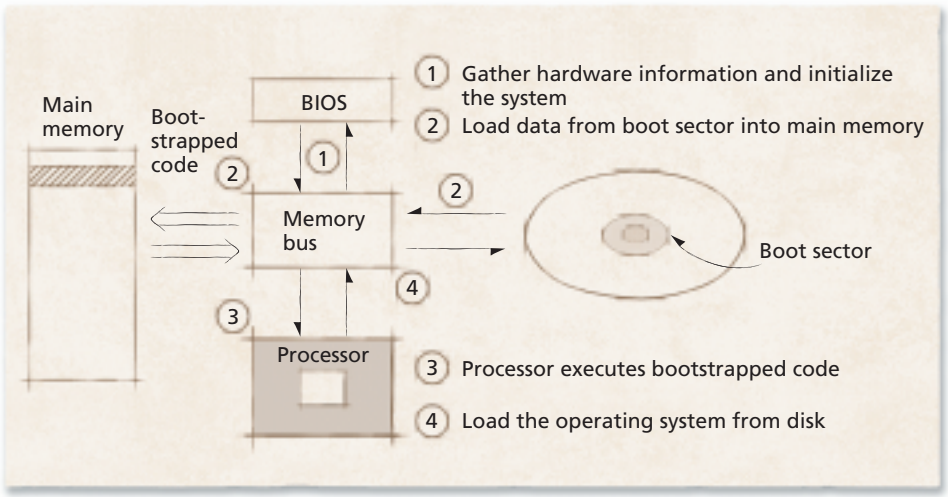

Figure 2.6 | Bootstrapping.

Self Review

1. How does EFI address the limitations of BIOS?
2. Why should the operating system prevent users from accessing the boot sector?

Ans: **1)** A typical BIOS contains low-level instructions that provide limited functionality and restrict how software is initially loaded. EFI supports drivers and provides a shell, enabling a user to interact with a system and customize the way that the operating system is loaded. **2)** If users could access the boot sector, they could accidentally or maliciously modify operating system code, making the system unusable or enabling an attacker to gain control of the system.

2.4.4 Plug and Play

Plug-and-play technology allows operating systems to configure and use newly installed hardware without user interaction. A plug-and-play hardware device

1. uniquely identifies itself to the operating system,
2. communicates with the operating system to indicate the resources and services it requires to function properly, and
3. identifies its corresponding driver and allows the operating system to use it to configure the device (e.g., assign the device to a DMA channel and allocate to the device a region of main memory).[31]

These features enable users to add hardware to a system and use the hardware immediately with proper operating system support.

As mobile computing devices become more popular, an increasing number of systems rely on batteries for power. Consequently, plug-and-play has evolved to include power management features that enable a system to dynamically adjust its power consumption to increase battery life. The **Advanced Configuration and Power Interface (ACPI)** defines a standard interface for operating systems to con-

figure devices and manage their power consumption. All recent Windows operating systems support plug-and-play; Linux version 2.6 is compatible with many plug-and-play devices.[32]

Self Review

1. Why, do you suppose, is it necessary for a plug-and-play device to uniquely identify itself to the operating system?
2. Why is power management particularly important for mobile devices?

Ans: **1)** Before an operating system can configure and make a device available to users, it must determine the resource needs that are unique to the device. **2)** Mobile devices rely on battery power; managing a device's power consumption can improve battery life.

2.5 Caching and Buffering

In Section 2.3.4, we discussed how computers contain a hierarchy of storage devices that operate at different speeds. To improve performance, most systems perform caching by placing copies of information that processes reference in faster storage. Due to the high cost of fast storage, caches can contain only a small portion of the information contained in slower storage. As a result, cache entries (also called **cache lines**) must be managed appropriately to minimize the number of times referenced information is not present in cache, an event called a **cache miss**. When a cache miss occurs, the system must retrieve the referenced information from slower storage. When a referenced item is present in the cache, a **cache hit** occurs, enabling the system to access data at relatively high speed.[33]

To realize increased performance from caching, systems must ensure that a significant number of memory references result in cache hits. As we discuss in Section 11.3, Demand Paging, it is difficult to predict with high accuracy the information that processes will soon reference. Therefore, most caches are managed using **heuristics**—rules of thumb and other approximations—that yield good results with relatively low execution overhead (see the Operating Systems Thinking Feature, Heuristics).

Examples of caches include the L1 and L2 processor caches, which store recently used data to minimize the number of cycles during which the processor is idle. Many operating systems allocate a portion of main memory to cache data from secondary storage such as disks, which typically exhibit latencies several orders of magnitude larger than main memory.

A **buffer** is storage area that temporarily holds data during transfers between devices or processes that operate at different speeds.[34] Buffers improve system performance by allowing software and hardware devices to transmit data and requests **asynchronously** (i.e., independently of one another). Examples of buffers include hard disk buffers, the keyboard buffer and the printer buffer.[35, 36] Because hard disks operate at much slower speeds than main memory, operating systems typically buffer data corresponding to write requests. The buffer holds the data until the hard disk has completed the write operation, enabling the operating system to execute

other processes while waiting for the I/O to complete. A keyboard buffer is often used to hold characters typed by users until a process can acknowledge and respond to the corresponding keyboard interrupts.

Spooling (simultaneous peripheral operations online) is a technique in which an intermediate device, such as a disk, is interposed between a process and a low-speed or buffer-limited I/O device. For example, if a process attempts to print a document but the printer is busy printing another document, the process, instead of waiting for the printer to become available, writes its output to disk. When the printer becomes available, the data on disk is printed. Spooling allows processes to request operations from a peripheral device without requiring that the device be ready to service the request.[37] The term "spooling" comes from the notion of winding thread onto a spool from which it can be unwound as needed.

Self Review

1. How does caching improve system performance?
2. Why do buffers generally not improve performance if one device or process produces data significantly faster than it is consumed?

Ans: **1)** Caches improve performance by placing in fast storage information that a process is likely to reference soon; processes can reference data and instructions from a cache much faster than from main memory. **2)** If the producing entity is much faster than the consuming entity, the buffer would quickly fill, then the relationship would be limited by the relatively

Operating Systems Thinking

Heuristics

A heuristic is a "rule of thumb"—a strategy that sounds reasonable and when employed, typically yields good results. It often does not have a basis in mathematics because the system to which it applies is sufficiently complex to defy easy mathematical analysis. As you leave your home each morning, you may use the heuristic, "If it looks like rain, take my umbrella." You do this because from your experience, "looks like rain" is a reasonable (although not perfect) indicator that it will rain. By applying this heuristic in the past, you avoided a few soakings, so you tend to rely on it. As you look at the pile of paperwork on your desk and schedule your work for the day, you may use another heuristic, "Do the shortest tasks first." This one has the satisfying result that you get a bunch of tasks done quickly; on the downside, it has the unfortunate side effect of postponing (possibly important) lengthier tasks. Worse yet, if a steady stream of new short tasks arrives for you to do, you could indefinitely postpone important longer tasks. We will see operating systems heuristics in many chapters of the book, especially in the chapters that discuss resource management strategies, such as Chapter 8, Processor Scheduling and Chapter 12, Disk Performance Optimization.

slow speed of the consuming entity—the producing entity would have to slow down because it would repeatedly find the buffer full and would have to wait (rather than execute at its normally faster speed) until the consumer eventually freed space in the buffer. Similarly, if the consuming entity were faster, it would repeatedly find the buffer empty and would have to slow down to about the speed of the producing entity.

2.6 Software Overview

In this section we review basic concepts of computer programming and software. Programmers write instructions in various programming languages; some are directly understandable by computers, while others require translation. Programming languages can be classified generally as either machine, assembly or high-level languages.

2.6.1 Machine Language and Assembly Language

A computer can understand only its own **machine language**. As the "natural language" of a particular computer, machine language is defined by the computer's hardware design. Machine languages generally consist of streams of numbers (ultimately reduced to 1s and 0s) that instruct computers how to perform their most elementary operations. Machine languages are machine dependent—a particular machine language can be used on only one type of computer. The following section of an early machine-language program, which adds *overtime pay* to *base pay* and stores the result in *gross pay*, demonstrates the incomprehensibility of machine language to humans:

```
1300042774
1400593419
1200274027
```

As the popularity of computers increased, machine-language programming proved to be slow and error prone. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent the computer's basic operations. These abbreviations formed the basis of **assembly languages**. Translator programs called **assemblers** convert assembly-language programs to machine-language. The following section of a simplified assembly-language program also adds *overtime pay* to *base pay* and stores the result in *gross pay*, but it presents the steps somewhat more clearly to human readers:

```
LOAD    BASEPAY
ADD     OVERPAY
STORE   GROSSPAY
```

This assembly-language code is clearer to humans, but computers cannot understand it until it is translated into machine language by an assembler program.

Self Review

1. (T/F) Computers typically execute assembly code directly.
2. Is software written in machine language portable?

Ans: **1)** False. Assemblers translate assembly code into machine-language code before the code can execute. **2)** No; machine languages are machine dependent, so software written in machine language executes only on machines of the same type.

2.6.2 Interpreters and Compilers

Although programming is faster in assembly languages than in machine language, assembly languages still require many instructions to accomplish even the simplest tasks. To increase programmer efficiency, **high-level languages** were developed. High-level languages accomplish more substantial tasks with fewer statements, but require translator programs called **compilers** to convert high-level language programs into machine language. High-level languages enable programmers to write instructions that look similar to everyday English and that contain common mathematical notations. For example, a payroll application written in a high-level language might contain a statement such as

```
grossPay = basePay + overTimePay
```

This statement produces the same result as the machine-language and assembly-language instructions in the prior sections.

Whereas compilers convert high-level language programs to machine language programs, **interpreters** are programs which directly execute source code or code has been reduced to a low-level language that is not machine code. Programming languages such as Java compile to a format called **bytecode** (although Java also can be compiled to machine language), which acts as machine code for a so-called virtual machine. Thus, bytecode is not dependent on the physical machine on which it executes, which promotes application portability. A Java interpreter analyzes each statement and executes the bytecode on the physical machine. Due to the execution-time overhead incurred by translation, programs executed via interpreters tend to execute slower than those that have been compiled to machine code.[38, 39]

Self Review

1. Discuss the benefits of high-level languages over assembly languages.
2. Why are programs compiled to bytecode more portable than those compiled to machine code?

Ans: **1)** High-level language programs require many fewer instructions than assembly-language programs; also, programming in high-level languages is easier than in assembly language because high-level languages more closely mirror everyday English and common mathematical notations. **2)** Bytecode is compiled to execute on a virtual machine that can be installed on many different platforms. By contrast, programs compiled to machine language can execute only on the type of machine for which the program was compiled.

2.6.3 High-Level Languages

Although hundreds of high-level languages have been developed, relatively few have achieved broad acceptance. Today, programming languages tend to be either structured or object oriented. In this section we enumerate some of the more popular languages, then discuss how they relate to each programming model.

IBM developed **Fortran** in the mid-1950s to create scientific and engineering applications that require complex mathematical computations. Fortran is still widely used, mainly in high-performance environments such as mainframes and supercomputers.

COmmon Business Oriented Language (COBOL) was developed in the late 1950s by a group of computer manufacturers, government agencies and industrial computer users. COBOL is designed for business applications that manipulate large volumes of data. A considerable portion of today's business software is still programmed in COBOL.

The **C** language, which Dennis Ritchie developed at Bell Laboratories in the early 1970s, gained widespread recognition as the development language of the UNIX operating system. In the early 1980s at Bell Laboratories, Bjarne Stroustrup developed **C++**, an extension of C. C++ provides capabilities for **object-oriented programming (OOP)**. **Objects** are reusable software components that model items in the real world. Object-oriented programs are often easier to understand, debug and modify than programs developed with previous techniques. Many of today's popular operating systems are written in C or C++.

When the World Wide Web exploded in popularity in 1993, Sun Microsystems saw immediate potential for using its new object-oriented **Java** programming language to create applications that could be downloaded over the Web and executed in Web browsers. Sun announced Java to the public in 1995, gaining the attention of the business community because of the widespread interest in the Web. Java has become a widely used software development language; it is used to generate Web pages dynamically, build large-scale enterprise applications, enhance the functionality of Web servers, provide applications for consumer devices (for example, cell phones, pagers and PDAs) and for many other purposes.

In 2000, Microsoft announced **C#** (pronounced "C-Sharp") and its .NET strategy. The C# programming language was designed specifically to be the key development language for the .NET platform; it has roots in C, C++ and Java. C# is object-oriented and has access to .NET's powerful library of prebuilt components, enabling programmers to develop applications quickly.

Self Review

1. Classify each of the following programming languages as structured or object oriented: **a)** C#; **b)** C; **c)** Java; **d)** C++.
2. What are some benefits of OOP?

Ans: **1) a)** object oriented; **b)** structured; **c)** object oriented; **d)** object-oriented. **2)** Object-oriented programs are often easier to understand, debug and modify than programs developed with previous techniques. Also OOP focuses on creating reusable software components.

2.6.4 Structured Programming

During the 1960s, software development efforts often ran behind schedule, costs greatly exceeded budgets and finished products were unreliable. People began to realize that software development was a far more complex activity than they had imagined. Research activity addressing these issues resulted in the evolution of **structured programming**—a disciplined approach to creating programs that are clear, correct and easy to modify.

This research led to the development of the **Pascal** programming language by Professor Nicklaus Wirth in 1971. Pascal was named after the 17th-century mathematician and philosopher Blaise Pascal. Designed for teaching structured programming, it rapidly became the preferred introductory programming language at most colleges. The language lacked many features needed to make it useful in commercial, industrial and government applications.

The **Ada** programming language was developed under the sponsorship of the U.S. Department of Defense (DoD) during the 1970s and early 1980s. It was named after Lady Ada Lovelace, daughter of the poet Lord Byron. Lady Lovelace is generally credited as being the world's first computer programmer, having written an application in the middle 1800s for the Analytical Engine mechanical computing device designed by Charles Babbage. Ada was one of the first languages designed to facilitate concurrent programming, which is discussed with examples in pseudocode and Java in Chapter 5, Asynchronous Concurrent Execution, and Chapter 6, Concurrent Programming.

Self Review

1. What problems in early software development did structured programming languages address?
2. How did the Ada programming language differ from other structured programming languages such as Pascal and C?

Ans: **1)** In the early days of programming, developers did not have a systematic approach to constructing complex programs, resulting in unnecessarily high costs, missed deadlines and unreliable products. Structured programming filled the need for a disciplined approach to software development. **2)** Ada was designed to facilitate concurrent programming.

2.6.5 Object-Oriented Programming

As the benefits of structured programming were realized in the 1970s, improved software technology began to appear. However, not until object-oriented programming became widely established in the 1980s and 1990s did software developers

finally feel they had the necessary tools to improve the software development process dramatically.

Object technology is a packaging scheme for creating meaningful software units. Almost any noun can be represented as a software object. Objects have **properties** (also called **attributes**), such as color, size and weight; and they perform **actions** (also called **behaviors** or **methods**), such as moving, sleeping or drawing. **Classes** are types of related objects. For example, all cars belong to the "car" class, even though individual cars vary in make, model, color and options packages. A class specifies the general format of its objects, and the properties and actions available to an object depend on its class. An object is related to its class in much the same way as a building is related to its blueprint.

Before object-oriented languages appeared, **procedural programming languages** (such as Fortran, Pascal, BASIC and C) focused on actions (verbs) rather than objects (nouns). This made programming a bit awkward. However, using today's popular object-oriented languages, such as C++, Java and C#, programmers can program in an object-oriented manner that more naturally reflects the way in which people perceive the world, resulting in significant gains in programmer productivity.

Object technology permits properly designed classes to be reused in multiple projects. Using libraries of classes can greatly reduce the effort required to implement new systems. However, some organizations report that the key benefit from object-oriented programming is not software reusability, but rather the production of software that is more understandable because it is better organized and easier to maintain.

Object-oriented programming allows programmers to focus on the "big picture." Instead of worrying about the minute details of how reusable objects are implemented, they can focus on the behaviors and interactions of objects. Programmers can also focus on modifying one object without worrying about the effect on another object. A road map that shows every tree, house and driveway would be difficult, if not impossible, to read. When such details are removed and only the essential information (roads) remains, the map becomes easier to understand. In the same way, an application that is divided into objects is easier to understand, modify and update because it hides much of the detail.

Self Review

1. How is the central focus of object-oriented programming different from that of structured programming?
2. How do objects facilitate modifications to existing software?

Ans: **1)** Object-oriented programming focuses on manipulating objects (nouns), whereas procedural programming focuses on actions (verbs). **2)** Objects hide much of the detail of an overall application, allowing programmers to focus on the big picture. Programmers can focusing on modifying one object without worrying about the effect on another object.

2.7 Application Programming Interfaces (APIs)

Today's applications require access to many resources that are managed by the operating system, such as files on disk and data from remote computers. Because the operating system must act as a resource manager, it typically will not allow processes to acquire these resources without first explicitly requesting them.

Application programming interfaces (APIs) provide a set of routines that programmers can use to request services from the operating system (Fig. 2.7). In most of today's operating systems, communication between software and the operating system is performed exclusively through APIs. Examples of APIs include the Portable Operating System Interface (POSIX) standards and the **Windows API** for developing Microsoft Windows applications. POSIX recommends standard APIs that are based on early UNIX systems and are widely used in UNIX-based operating systems. The Win32 API is Microsoft's interface for applications that execute in a Windows environment.

Processes execute function calls defined by the API to access services provided by a lower layer of the system. These function calls may issue **system calls** to request services from the operating system. System calls are analogous to interrupts for hardware devices—when a system call occurs, the system switches to kernel mode and the operating system executes to service the system call.

Self Review

1. Why must processes issue system calls to request operating system services?
2. How does the POSIX attempt to improve application portability?

Ans: **1)** To protect the system, the operating system cannot allow processes to access operating system services or privileged instructions directly. Instead, the services that an operating system can provide to processes are packaged into APIs. Processes can access these

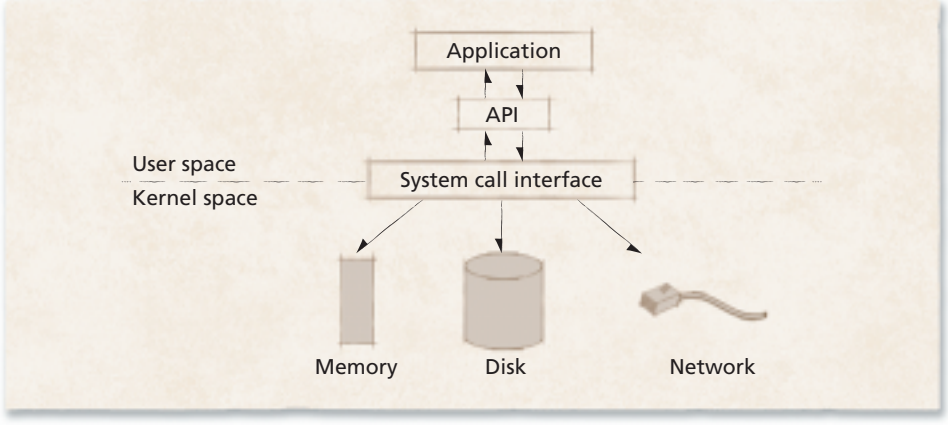

Figure 2.7 | *Application programming interface (API).*

services only through the system call interface, which essentially puts the operating system in control. **2)** Software written using a particular API can be run only on systems that implement the same API. POSIX attempts to address this problem by specifying a standard API for UNIX-based systems. Even many non-UNIX systems now support POSIX.

2.8 Compiling, Linking and Loading

Before a program written in a high-level language can execute, it must be translated into machine language, linked with various other machine-language programs on which it depends and loaded into memory. In this section we consider how programs written in high-level languages are **compiled** into machine-language code and we describe how linkers and loaders prepare compiled code for execution.[40]

2.8.1 Compiling

Although each type of computer can understand only its own machine language, nearly all programs are written in high-level languages. The first stage in the process of creating executable programs is compiling the high-level programming language code to machine language. A compiler accepts **source code**, which is written in a high-level language, as input and returns **object code**, containing the machine-language instructions to execute, as output. Nearly all commercially available programs are delivered as object code, and some distributions (i.e., open-source software) also include the source code.[41]

The compiling process can be divided into several phases; one view of compiling is presented in Fig. 2.8. Each phase modifies the program so that it can be interpreted by the next phase, until the program has been translated into machine code. First, the source code is passed to the **lexer** (also called **lexical analyzer** or **scanner**), which separates the characters of a program's source into **tokens**. Examples of tokens include keywords (e.g., `if`, `else` and `int`), identifiers (e.g., named variables and constants), operators (e.g., `-`, `+`, `*` and `/`) and punctuation (e.g., semicolons).

The lexer passes this stream of tokens to the **parser** (also called the **syntax analyzer**), which groups the tokens into syntactically correct statements. The **inter-**

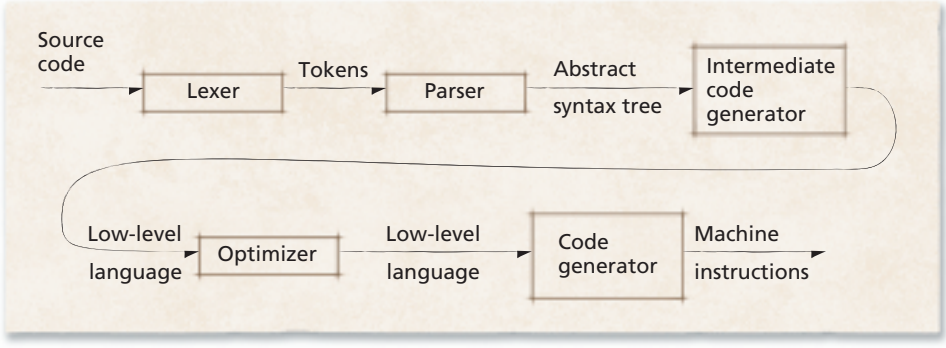

Figure 2.8 | *Compiler phases.*

mediate code generator converts this syntactic structure into a stream of simple instructions that resemble assembly language (although it does not specify the registers used for each operation). The **optimizer** attempts to improve the code's execution efficiency and reduce the program's memory requirements. In the final phase, the **code generator** produces the object file containing the machine-language instructions.[42, 43]

Self Review

1. What is the difference between compiling and assembling?
2. Could a Java program run directly on a physical machine instead of on a virtual machine?

Ans: **1)** The assembly process simply translates assembly-language instructions into machine language. A compiler translates high-level language code into machine-language code and may also optimize the code. **2)** A Java program could run on a physical machine by using a compiler that translates Java source code or bytecode into the corresponding machine language.

2.8.2 Linking

Often, programs consist of several independently developed subprograms, called **modules**. Functions to perform common computer routines such as I/O manipulations or random number generation are packaged into precompiled modules called **libraries**. **Linking** is the process of integrating the various modules referenced by a program into a single executable unit.

When a program is compiled, its corresponding object module contains program data and instructions retrieved from the program's source file. If the program referenced functions or data from another module, the compiler translates these into **external references**. Also, if the program makes functions or data available to other programs, each of these is represented as an **external name**. Object modules store these external references and names in a data structure called a **symbol table** (Fig. 2.9). The integrated module produced by the linker is called a **load module**.

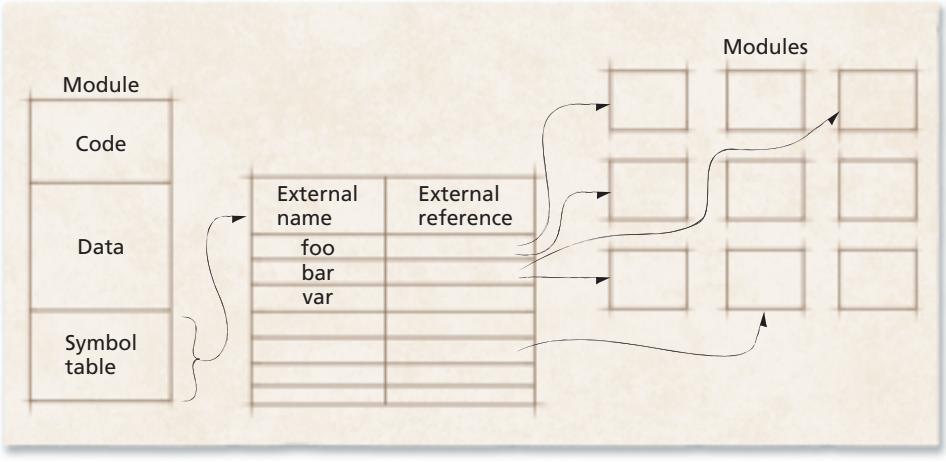

Figure 2.9 | *Object module.*

Input to the linker can include object modules, load modules and control statements, such as the location of referenced library files.[44]

The linker often is provided with several object files that form a single program. These object files typically specify the locations of data and instructions using addresses that are relative to the beginning of each file, called **relative addresses**.

In Fig. 2.10, symbol `X` in object module `A` and symbol `Y` in object module `B` have the same relative address in their respective modules. The linker must modify these addresses so that they do not reference invalid data or instructions when the modules are combined to form a linked program. **Relocating** addresses ensures that each statement is uniquely identified by an address within a file. When an address is modified, all references to it must be updated with the new location. In the resulting load module, `X` and `Y` have been relocated to new relative addresses that are unique within the load module. Often, linkers also provide relative addressing in the load module; however, the addresses are assigned such that they are all relative to the beginning of the entire load module.

Linkers also perform **symbol resolution**, which converts external references in one module to their corresponding external names in another module.[45, 46] In

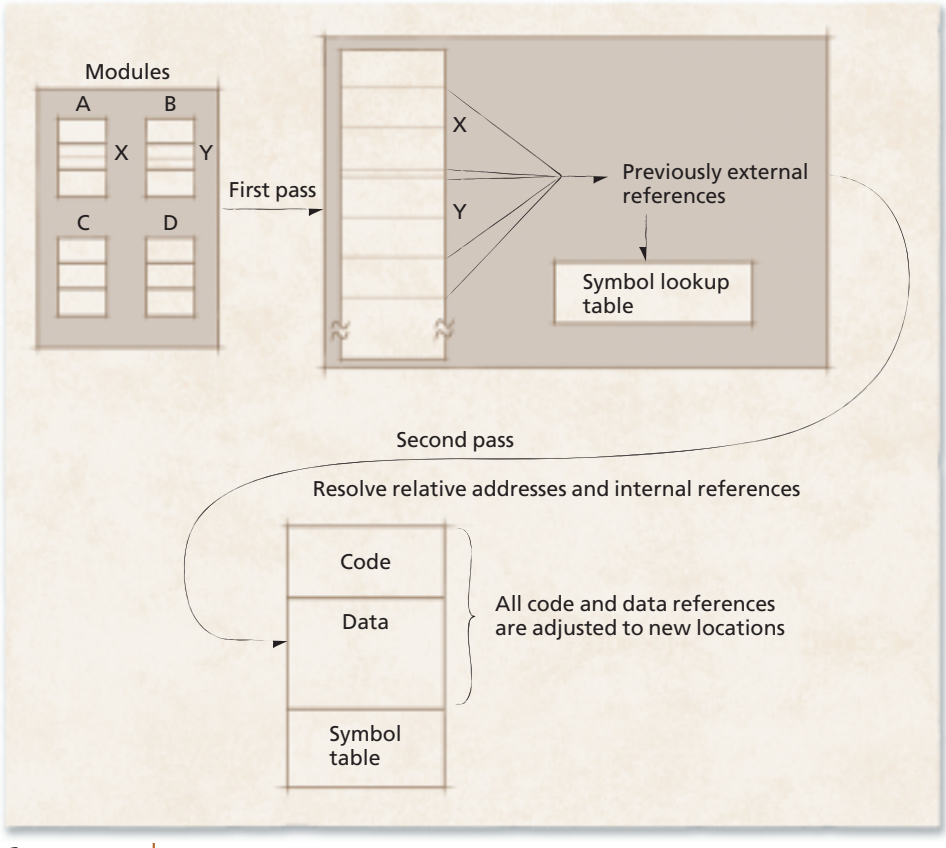

Figure 2.10 | *Linking process.*

Fig. 2.11, the external reference to symbol C in object module 2 is resolved with the external name C from object module 1. Once an external reference is paired with the corresponding name in a separate module, the address of the external reference must be modified to reflect this integration.

Often, linkage occurs in two passes. The first pass determines the size of each module and constructs a symbol table. The symbol table associates each symbol (such as a variable name) with an address, so that the linker can locate the reference. On the second pass, the linker assigns addresses to different instruction and data units

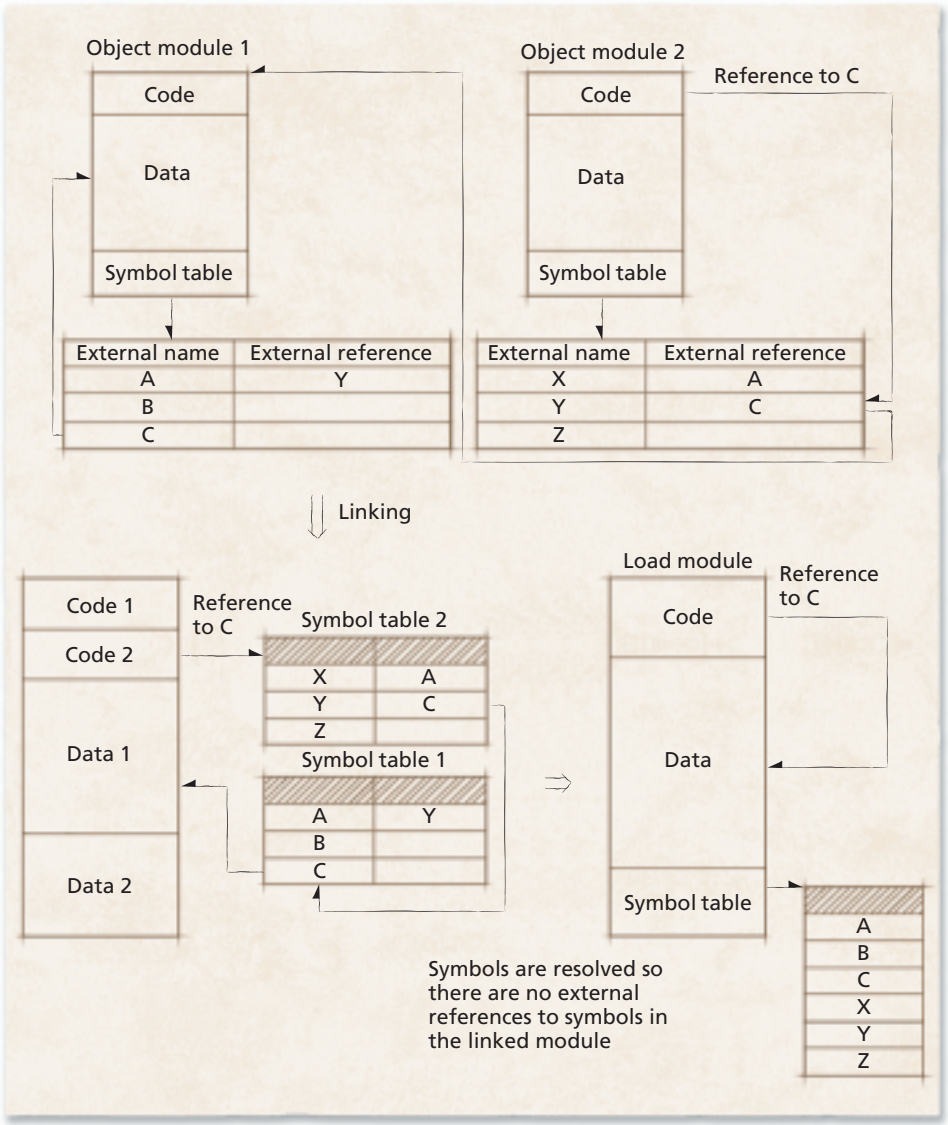

Figure 2.11 | *Symbol resolution.*

and resolves external symbol references.[47] Because a load module can become the input of another linking pass, the load module contains a symbol table, in which all symbols are external names. Notice that, in Fig. 2.11, the external reference to symbol Y is not listed in the load module's symbol table because it has been resolved.

The time at which a program is linked depends on the environment. A program can be linked at compile time if the programmer includes all necessary code in the source file so that there are no references to external names. This is accomplished by searching for the source code for any externally referenced symbols and placing those symbols in the resulting object file. This method typically is not feasible because many programs rely on **shared libraries**, which are collections of functions that can be shared between different processes. Many programs can reference the same functions (such as library functions that manipulate input and output streams) without including them in their object code. This type of linking is typically performed after compilation but before loading. As discussed in the Mini Case Study, Mach, shared libraries enable the Mach microkernel to emulate multiple operating systems.

This same process can be performed at load time (see Section 2.8.3, Loading). Linking and loading are sometimes both performed by one application called a

Mini Case Study

Mach

The **Mach** system was developed at Carnegie-Mellon University from 1985–1994 and was based on CMU's earlier Accent research OS.[48] The project was directed by Richard Rashid, now the senior vice president of Microsoft Research.[49] Mach was one of the first and best-known microkernel operating systems (see Section 1.13.3, Microkernel Architecture). It has been incorporated into later systems, including Mac OS X, NeXT and OSF/1, and had a strong influence on Windows NT (and ultimately on Windows XP, which we discuss in Chapter 21).[50, 51, 52] An open-source implementation, GNU Mach, is used as the kernel for the GNU Hurd operating system, which is currently under development.[53]

A powerful capability of the Mach microkernel system is that it can emulate other operating systems. Mach achieves this using "transparent shared libraries."[54] A transparent shared library implements the actions for the system calls of the OS it is emulating, then intercepts system calls made by programs that are written to run on the emulated OS.[55, 56] The intercepted system calls can then be translated into Mach system calls, and any results are translated back into the emulated form.[57, 58] Thus the user's program does not have to be ported to run on a system running Mach. In addition, any number of these transparent libraries can be in memory, so Mach can emulate multiple operating systems simultaneously.[59]

linking loader. Linking can also occur at runtime, a process called **dynamic linking**. In this case, references to external functions are not resolved until the process is loaded into memory or issues a call to the function. This is useful for large programs that use programs controlled by another party, because a dynamically linked program does not have to be relinked when a library that it uses is modified.[60] Further, because dynamically linked programs are not linked until they are in main memory, shared library code can be stored separately from other program code. Thus, dynamic linking also saves space in secondary storage because only one copy of a shared library is stored for any number of programs that use it.

Self Review

1. How does linking facilitate the development of large programs built by many developers?
2. What is one possible drawback of using a dynamic linker? What is a benefit?

Ans: **1)** Linking permits programs to be written as many separate modules. The linker combines these modules into a final load module when all pieces of the program have been compiled. **2)** If a library cannot be found during execution, an executing program will be forced to terminate, possibly losing all of the work performed up to that point. A benefit is that programs that are dynamically linked do not have to be relinked when a library changes.

2.8.3 Loading

Once the linker has created the load module, it passes it to a **loader** program. The loader is responsible for placing each instruction and data unit at a particular memory address, a process called **address binding**. There are several techniques for loading programs into main memory, most of which are important only for systems that do not support virtual memory. If the load module already specifies physical addresses in memory, the loader simply places the instruction and data units at the addresses specified by the programmer or compiler (assuming the memory addresses are available), a technique called **absolute loading**. **Relocatable loading** is performed when the load module contains relative addresses that need to be converted to actual memory addresses. The loader is responsible for requesting a block of memory space in which to place the program, then relocating the program's addresses to correspond to its location in memory.

In Fig. 2.12, the operating system has allocated the block of memory beginning with memory address 10,000. As the program is loaded, the loader must add 10,000 to each address in the load module. The loader updates the memory address of the variable `Example` in the Fig. 2.12 to 10,450 from its original relative address of 450.

Dynamic loading is a technique that loads program modules upon first use.[61] In many virtual memory systems, each process is assigned its own set of virtual addresses starting at zero, so the loader is responsible for loading the program into a valid memory region.

We review the entire compiling, linking and loading process (using load-time address binding) from source code to execution in Fig. 2.13. The programmer begins by writing the source code in some high-level language—in this case, C.

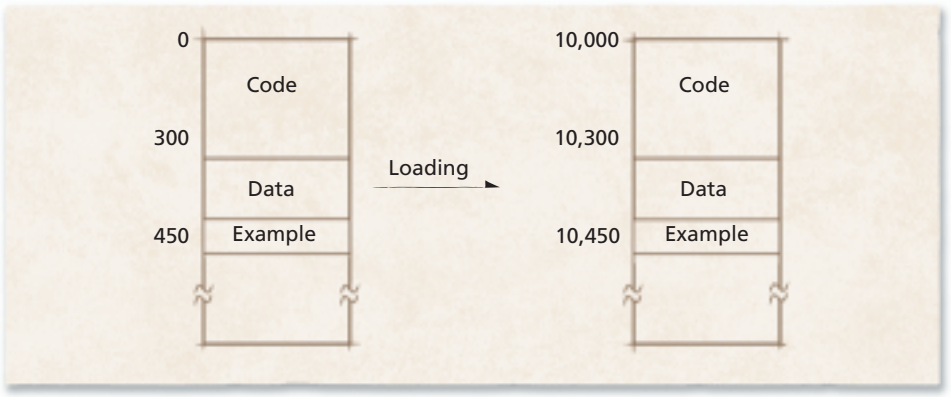

Figure 2.12 | *Loading.*

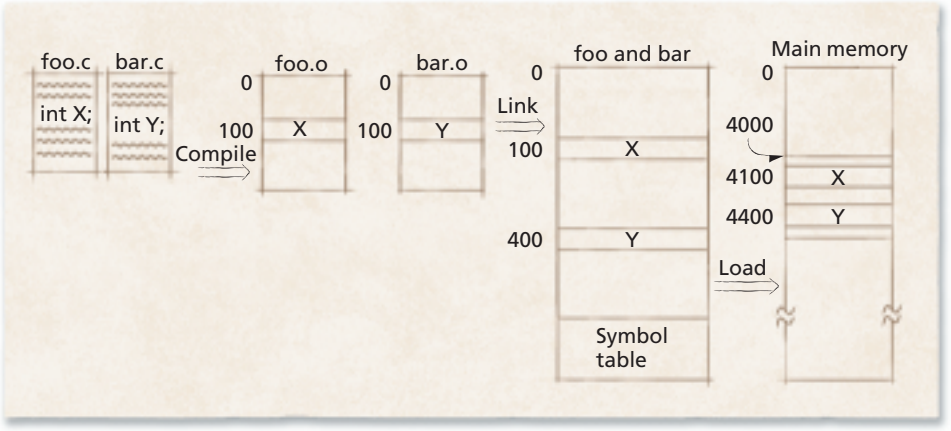

Figure 2.13 | *Compiling, linking and loading.*

Next, the compiler transforms the `foo.c` and `bar.c` source-code files into machine language, creating the object modules `foo.o` and `bar.o`. In the code, the programmer has defined variable `X` in `foo.c` and variable `Y` in `bar.c`; both are located at relative address 100 in their respective object modules. The object modules are placed in secondary storage until requested by the user or another process, at which point the modules must be linked.

In the next step, the linker integrates the two modules into a single load module. The linker accomplishes this task by collecting information about module sizes and external symbols in the first pass and linking the files together in the second pass. Notice that the linker relocates variable `Y` to relative address 400.

In the third step, the loader requests a block of memory for the program. The operating system provides the address range of 4000 to 5050, so the loader relocates variable `X` to the absolute address 4100 and variable `Y` to the absolute address 4400.

Self Review

1. How might absolute loading limit a system's degree of multiprogramming?
2. How does dynamic loading improve a system's degree of multiprogramming?

Ans: **1)** Two programs that specify overlapping addresses cannot execute at the same time, because only one can be resident at the same location in memory at once. **2)** Modules are loaded as needed, so memory contains only the modules that are used.

2.9 Firmware

In addition to hardware and software, most computers contain **firmware**, executable instructions stored in persistent, often read-only, memory attached to a device. Firmware is programmed with **microprogramming**, which is a layer of programming below a computer's machine language.

Microcode (i.e., microprogram instructions) typically includes simple, fundamental instructions necessary to implement all machine-language operations.[62] For example, a typical machine instruction might specify that the hardware perform an addition operation. The microcode for this instruction specifies the actual primitive operations that hardware must perform, such as incrementing the pointer that references to the current machine instruction, adding each bit of the numbers, storing the result in a new register and fetching the next instruction.[63, 64]

Professor Maurice Wilkes, the creator of the early EDSAC computer, first introduced the concepts of microprogramming in 1951.[65] However, not until the IBM System/360 appeared in 1964 was microcode used on a wide scale. Machine instruction sets implemented in microcode reached a peak with the VAX operating system, but have declined in recent years, because the execution of microcode instructions limits a processor's maximum speed. Thus, operations formerly performed by microcode instructions are now performed by processor hardware.[66] Today, many hardware devices, including hard drives and peripheral devices, contain miniature processors. The instructions for these processors are often implemented using microcode.[67]

Self Review

1. (T/F) There are no instructions smaller than machine-language instructions.
2. Describe the role of firmware in a computer system.

Ans: **1)** False. Microcode specifies a layer of programming below a processor's machine language. **2)** Firmware specifies simple, fundamental instructions necessary to implement machine-language instructions.

2.10 Middleware

Software plays an important role in distributed systems in which computers are connected across a network. Often, the computers that compose a distributed system are

heterogeneous—they use different hardware, run different operating systems and communicate across different network architectures using various network protocols. The nature of distributed systems requires middleware to enable interactions among multiple processes running on one or more computers across a network.

Middleware allows an application running on one computer to communicate with an application running on a remote computer, enabling communications between computers in distributed systems. Middleware also permits applications to run on heterogeneous computer platforms, as long as each computer has the middleware installed. Middleware simplifies application development, because developers do not need to know the details of how the middleware performs its tasks. Developers can concentrate on developing programs rather than developing communication protocols. **Open DataBase Connectivity (ODBC)** is an example of an API for database access that permits applications to access databases through middleware called an ODBC driver. When developing such applications, developers need to provide only the database to which the application should connect. The ODBC driver handles connecting to the database and retrieving the information required by the application. Section 17.3.1, Middleware, through Section 17.3.4, CORBA (Common Object Request Broker Architecture), discuss common middleware implementations and protocols that form the backbone of many distributed systems.

Self Review

1. What are the costs and benefits of using middleware?
2. How does middleware facilitate the construction of heterogeneous systems?

Ans: 1) Middleware promotes program modularity and facilitates application programming, because the developer does not need to write code to manage interactions between processes. However, communication between middleware and processes incurs overhead compared to direct communication. 2) Middleware facilitates communication between computers using different protocols by translating messages into different formats as they are passed between sender and receiver.

Web Resources

`www.pcguide.com`
Provides articles discussing various aspects of computer hardware and the motivation for creating various interfaces; covers a broad range of topics relating to computer architecture.

`www.tomshardware.com`
Tom's Hardware Guide is one of the most thorough hardware review sites on the Web.

`www.anandtech.com`
Reviews new and emerging hardware.

`developer.intel.com`
Provides technical documentation about Intel products, articles on current technologies and topics investigated by their research and development teams.

`www.ieee.org`
IEEE defines many standards in computer hardware design. Members may access its journals online.

`sourceforge.net`
World's most popular site for open-source software development; provides resources and utilities for software developers.

Summary

An operating system is primarily a resource manager, so the design of an operating system is intimately tied to the hardware and software resources the operating system must manage.

A PCB is a hardware component that provides electrical connections between devices at various locations on the PCB. The mainboard is the PCB to which devices such as processors and main memory are attached.

A processor is a hardware component that executes a stream of machine-language instructions. A CPU is a processor that executes the instructions of a program; a coprocessor executes special-purpose instructions (such as graphics or audio) efficiently. In this book, we use the term "processor" to refer to a CPU. Registers are high-speed memory located on a processor that hold data for immediate use by the processor. Before a processor can operate on

data, the data must be placed in registers. The instruction length is the size of a machine-language instruction; some processors support multiple instruction lengths.

Computer time is measured in cycles; each cycle is one complete oscillation of an electrical signal provided by the system clock generator. Processor speeds are often measured in GHz (billions of cycles per second).

The memory hierarchy is a scheme for categorizing memory, which places the fastest and most expensive memory at the top and the slowest and least expensive memory at the bottom. It has a steep, pyramidal shape in which register memory occupies the hierarchy's top level, followed by L1 cache memory, L2 cache memory, main memory, secondary storage and tertiary storage. A system's main memory is the lowest data store in the memory hierarchy that the processor can reference directly. Main memory is volatile, meaning it loses its contents when the system loses power. Secondary storage, such as hard disks, CDs, DVDs and floppy disks, persistently store large quantities of data at a low cost per unit storage.

A bus is a collection of thin electrical connections called traces that transport information between hardware devices. A port is a bus connecting two devices. I/O channels are special-purpose components devoted to handling I/O independently of the computer system's main processors.

A peripheral device is hardware that is not required for a computer to execute instructions. Printers, scanners and mice are peripheral devices; processors and main memory are not.

Some hardware exists specifically to improve performance and simplify the design of operating systems. Computer systems generally have several different execution states. For user applications, the subset of instructions the user may execute in user mode precludes, for example, the direct execution of input/output instructions. The operating system ordinarily executes with most trusted user status in kernel mode; in kernel mode, a processor may execute privileged instructions and access resources to perform tasks on behalf of processes. Memory protection, which prevents processes from accessing memory that has not been assigned to them (such as other users' memory and the operating system's memory), is implemented using processor registers that can be modified only by privileged instructions. Most devices send a signal called an interrupt to the processor when an event occurs. The operating system can respond to a change in device status by notifying processes that are waiting on such events.

Programmed I/O (PIO) is a technique whereby a processor idles while data is transferred between a device and main memory. By contrast, direct memory access (DMA) enables devices and controllers to transfer blocks of data to and from main memory directly, which frees the processor to execute software instructions.

Interrupts allow hardware to send signals to the processor, which notifies the operating system of the interrupt. The operating system decides what action to take in response to the interrupt.

A computer contains several types of clocks and timers. An interval timer is useful for preventing a process from monopolizing a processor. After a designated interval, the timer generates an interrupt to gain the attention of the processor; as a result of this interrupt, the processor might then be assigned to another application. The time-of-day clock keeps track of "wall clock time."

Bootstrapping is the process of loading initial operating system components into memory. This process is performed by a computer's BIOS.

Plug-and-play technology allows operating systems to configure newly installed hardware without user interaction. To support plug-and-play, a hardware device must uniquely identify itself to the operating system, communicate with the operating system to indicate the resources and services the device requires to function properly, and identify the driver that supports the device and allows software to configure the device (e.g., assign the device to a DMA channel).

Caches are relatively fast memory designed to increase program execution speed by maintaining copies of data that will be accessed soon. Examples of caches are the L1 and L2 processor caches and the main memory cache for hard disks.

A buffer is a temporary storage area that holds data during I/O transfers. Buffer memory is used primarily to coordinate communication between devices operating at different speeds. Buffers can store data for asynchronous processing, coordinate input/output between devices operating at different speeds or allow signals to be delivered asynchronously.

Spooling is a buffering technique in which an intermediate device, such as a disk, is interposed between a process and a low-speed or buffer-limited I/O device. Spooling allows processes to request operations from a peripheral device without requiring that the device be ready to service the request.

Three types of programming languages are machine, assembly and high-level languages. Machine languages consist of streams of numbers (ultimately reduced to 1s and 0s) that instruct computers how to perform their most ele-

mentary operations. A computer can understand only its own machine language. Assembly languages represent machine-language instructions using English-like abbreviations. High-level languages enable programmers to write instructions that look similar to everyday English and that contain common mathematical notations. High-level languages accomplish more substantial tasks with fewer statements, but require translator programs called compilers to convert high-level language programs into machine language. C, C++, Java and C# are examples of high-level languages.

Today, high-level languages tend to fall into two types, structured programming languages and object-oriented programming languages. Structured programming is a disciplined approach to creating programs that are clear, correct and easy to modify. Pascal and Fortran are structured programming languages. Object-oriented programs focus on manipulating objects (nouns) to create reusable software that is easy to modify and understand. C++, C# and Java are object-oriented programming languages.

APIs allow a program to request services from the operating system. Programs call API functions, which may access the operating system by making system calls.

Before a high-level-language program can execute, it must be translated into machine code and loaded into memory. Compilers transform high-level-language code into machine code. Linkers assign relative addresses to different program or data units and resolve external references between subprograms. Loaders convert these addresses into physical addresses and place the program or data units into main memory.

Most computers contain firmware, which specifies software instructions but is physically part of a hardware device. Most firmware is programmed with microprogramming, which is a layer of programming below a computer's machine language.

Middleware enables communication among multiple processes running on one or more computers across a network. Middleware facilitates heterogeneous distributed systems and simplifies application programming.

Key Terms

absolute loading—Loading technique in which the loader places the program in memory at the address specified by the programmer or compiler.

Accelerated Graphics Port (AGP)—Popular bus architecture used for connecting graphics devices; AGPs typically provide 260MB/s of bandwidth.

Ada—Concurrent, procedural programming language developed by the DoD during the 1970s and 1980s.

address binding—Assignment of memory addresses to program data and instructions.

address bus—Part of a bus that specifies the memory location from or to which data is to be transferred.

add-on card—Device that extends the functionality of a computer (e.g., sound and video cards).

Advanced Configuration and Power Interface (ACPI)—Interface to which a plug-and-play device must conform so that Microsoft Windows operating systems can manage the device's power consumption.

application programming—Software development that entails writing code that requests services and resources from the operating system to perform tasks (e.g., text editing, loading Web pages or payroll processing).

application programming interface (API)—Set of functions that allows an application to request services from a lower level of the system (e.g., the operating system or a library module).

Arithmetic and Logic Unit (ALU)—Component of a processor that performs basic arithmetic and logic operations.

assembler—Translator program that converts assembly-language programs to machine language.

assembly language—Low-level language that represents basic computer operations as English-like abbreviations.

attribute (of an object)—See property.

asynchronous transmission—Transferring data from one device to another that operates independently via a buffer to eliminate the need for blocking; the sender can perform other work once the data arrives in the buffer, even if the receiver has not yet read the data.

auxiliary storage—See secondary storage.

bandwidth—Measure of the amount of data transferred over a unit of time.

behavior (of an object)—See method.

basic input/output system (BIOS)—Low-level software instructions that control basic hardware initialization and management.

block device—Device such as a disk that transfers data in fixed-size groups of bytes, as opposed to a character device, which transfers data one byte at a time.

boot sector—Specified location on a disk in which the initial operating system instructions are stored; the BIOS instructs the hardware to load these initial instructions when the computer is turned on.

bootstrapping—Process of loading initial operating system components into system memory so that they can load the rest of the operating system.

bounds register—Register that stores information regarding the range of memory addresses accessible to a process.

buffer—Temporary storage area that holds data during I/O between devices operating at different speeds. Buffers enable a faster device to produce data at its full speed (until the buffer fills) while waiting for the slower device to consume the data.

bus—Collection of traces that form a high-speed communication channel for transporting information between different devices on a mainboard.

bus mastering—DMA transfer in which a device assumes control of the bus (preventing others from accessing the bus simultaneously) to access memory.

bytecode—Intermediate code that is intended for virtual machines (e.g., Java bytecode runs on the Java Virtual Machine).

C—Procedural programming language developed by Dennis Ritchie that was used to create UNIX.

C++—Object-oriented extension of C developed by Bjarne Stroustup.

C#—Object-oriented programming language developed by Microsoft that provides access to .NET libraries.

cache hit—Request for data that is present in the cache.

cache line—Entry in a cache.

cache miss—Request for data that is not present in the cache.

central processing unit (CPU)—Processor responsible for the general computations in a computer.

character device—Device such as a keyboard or mouse that transfers data one byte at a time, as opposed to a block device, which transfers data in fixed-size groups of bytes.

chipset—Collection of controllers, coprocessors, buses and other hardware specific to the mainboard that determine the hardware capabilities of a system.

class—Type of an object. Determines an object's methods and attributes.

clocktick—See cycle.

COmmon Business Oriented Language (COBOL)—Procedural programming language developed in the late 1950s that was designed for writing business software that manipulates large volumes of data.

code generator—Part of a compiler responsible for producing object code from a higher-level language.

compact disk (CD)—Digital storage medium in which data is stored as a series of microscopic pits on a flat surface.

compile—Translate high-level-language source code into machine code.

compiler—Application that translates high-level-language source code into machine code.

controller—Hardware component that manages access to a bus by devices.

coprocessor—Processor, such as a graphics or digital signal processor, designed to efficiently execute a limited set of special-purpose instructions (e.g., 3D transformations).

cycle (clock)—One complete oscillation of an electrical signal. The number of cycles that occur per second determines a device's frequency (e.g., processors, memory and buses) and can be used by the system to measure time.

cycle stealing—Method that gives channels priority over a processor when accessing the bus to prevent signals from channels and processors from colliding.

data bus—Bus that transfers data to or from locations in memory that are specified by the address bus.

derived speed—Actual speed of a device as determined by the frontside bus speed and clock multipliers or dividers.

direct memory access (DMA)—Method of transferring data from a device to main memory via a controller that requires only interrupting the processor when the transfer completes. I/O transfer via DMA is more efficient than programmed I/O or interrupt-driven I/O because the processor does not need to supervise the transfer of each byte or word of data.

double data rate (DDR)—Chipset feature that enables a frontside bus to effectively operate at twice its clock speed by performing two memory transfers per clock cycle. This feature must be supported by the system's chipset and RAM.

dynamic linking—Linking mechanism that resolves references to external functions when the process first makes a call to the function. This can reduce linking overhead because external functions that are never called while the process executes are not linked.

dynamic loading—Method for loading that specifies memory addresses at runtime.

dynamic RAM (DRAM)—RAM that must be continuously read by a refresh circuit to keep the contents in memory.

execution mode—Operating system execution mode (e.g., user mode or kernel mode) that determines which instructions can be executed by a process.

exception—Error caused by a process. Processor exceptions invoke the operating system, which determines how to respond. Processes can register exception handlers that are executed when the operating system receives the corresponding exception.

Extensible Firmware Interface (EFI)—Interface designed by Intel that improves upon a traditional BIOS by supporting device drivers and providing a shell interface at boot time.

external name—Symbol defined in a module that can be referenced by other modules.

external reference—Reference from one module to an external name in a different module.

firmware—Microcode that specifies simple, fundamental instructions necessary to implement machine-language instructions.

Fortran—Procedural programming language developed by IBM in the mid-1950s for scientific applications that require complex mathematical computations.

frontside bus (FSB)—Bus that connects a processor to main memory.

general-purpose register—Register that can be used by processes to store data and pointer values. Special-purpose registers cannot be accessed by user processes.

heuristics—Technique that solves complex problems using rules of thumb or other approximations that incur low execution overhead and generally provide good results.

high-level language—Programming language that uses English-like identifiers and common mathematical notation to represent programs using fewer statements than assembly-language programming.

IEEE 1394 port—Commonly used serial port that provides transfer speeds of up to 800MB per second, sometimes supplies power to devices and allows devices to be hot swappable; these ports are commonly referred to as FireWire (from Apple) or iLink (from Sony).

instruction decode unit—Component of a processor that interprets instructions and generates appropriate control signals that cause the processor to perform each instruction.

instruction fetch unit—Component of a processor that loads instructions from the instruction cache so they can be decoded and executed.

instruction length—Number of bits that comprise an instruction in a given architecture. Some architectures support variable-length instructions; instruction lengths also vary among different architectures.

intermediate code generator—Stage of the compilation process that receives input from the parser and outputs a stream of instructions to the optimizer.

interpreter—Application that can execute code other than machine code (e.g., high-level-language instructions).

interrupt—Message that informs the system that another process or device has completed an operation or produced an error. An interrupt causes the processor to pause program execution and invoke the operating system so it can respond to the interrupt.

interval timer—Hardware that generates periodic interrupts that cause operating system code to execute, which can ensure that a processor will not be monopolized by a malicious or malfunctioning process.

I/O channel—Component responsible for handling device I/O independently of a main processor.

Java—Object-oriented programming language developed by Sun Microsystems that promotes portability by running on a virtual machine.

kernel mode—Execution mode of a processor that allows processes to execute privileged instructions.

lane—Route between two points in a PCI Express bus. PCI Express devices are connected by a link that may contain up to 32 lanes.

lexer—See lexical analyzer.

lexical analyzer—Part of a compiler that separates the source code into tokens.

library module—Precompiled module that performs common computer routines, such as I/O routines or mathematical functions.

linking—Process of integrating a program's object modules into a single executable file.

linking loader—Application that performs both linking and loading.

loader—Application that loads linked executable modules into memory.

load module—Integrated module produced by a linker that consists of object code and relative addresses.

Mach—Early microkernel operating system, designed at Carnegie-Mellon University by a team led by Richard Rashid. Mach has influenced the design of Windows NT and has been used to implement Mac OS X.

machine language—Language that is defined by a computer's hardware design and can be natively understood by the computer.

mainboard—Printed circuit board that provides electrical connections between computer components such as processor, memory and peripheral devices.

main memory—Volatile memory that stores instructions and data; it is the lowest level of the memory hierarchy that can be directly referenced by a processor.

memory hierarchy—Classification of memory from fastest, lowest-capacity, most expensive memory to slowest, highest-capacity, least expensive memory.

memory protection—Mechanism that prevents processes from accessing memory used by other processes or the operating system.

method (of an object)—Part of an object that manipulates object attributes or performs a service.

microcode—Microprogramming instructions.

microprogramming—Layer of programming below a computer's machine language that includes instructions necessary to implement machine-language operations. This enables processors to divide large, complex instructions into simpler ones that are performed by its execution unit.

module—Independently developed subprogram that can be combined with other subprograms to create a larger, more complex program; programmers often use precompiled library modules to perform common computer functions such as I/O manipulations or random number generation.

Moore's law—Prediction regarding the evolution of processor design that asserts the number of transistors in a processor will double approximately every 18 months.

motherboard—See mainboard.

most trusted user status—See kernel mode.

object—Reusable software component that can model real-world items through properties and actions.

object code—Code generated by a compiler that contains machine-language instructions that must be linked and loaded before execution.

object-oriented programming (OOP)—Style of programming that allows programmers to quickly build complex software systems by reusing components called objects, built from "blueprints" called classes.

on-board device—Device that is physically connected to a computer's mainboard.

Open DataBase Connectivity (ODBC)—Protocol for middleware that permits applications to access a variety of databases that use different interfaces. The ODBC driver handles connections to the database and retrieves information requested by applications. This frees the application programmer from writing code to specify database-specific commands.

optimizer—Part of the compiler that attempts to improve the execution efficiency and reduce the space requirement of a program.

parallel port—Interface to a parallel I/O device such as a printer.

parser—Part of the compiler that receives a stream of tokens from the lexical analyzer and groups the tokens so they can be processed by the intermediate code generator.

Pascal—Structured programming language developed in 1971 by Wirth that became popular for teaching introductory programming courses.

Peripheral Components Interconnect (PCI) bus—Popular bus used to connect peripheral devices, such as network and sound cards, to the rest of the system. PCI provides a 32-bit or 64-bit bus interface and supports transfer rates of up to 533MB per second.

persistent storage—See secondary storage.

physical address—See real address.

physical memory—See main memory.

plug-and-play—Technology that facilitates driver installation and hardware configuration performed by the operating system.

polling—Technique to discover hardware status by repeatedly testing each device. Polling can be implemented in lieu of interrupts but typically reduces performance due to increased overhead.

port—Bus that connects two devices.

principle of least privilege—Resource access policy that states that a user should only be granted the amount of privilege and access that the user needs to accomplish its designated task.

printed circuit board (PCB)—Piece of hardware that provides electrical connections to devices that can be placed at various locations throughout the board.

privileged instruction—Instruction that can be executed only from kernel mode. Privileged instructions perform operations that access protected hardware and software resources (e.g., switching the processor between processes or issuing a command to a hard disk).

problem state—See user mode.

procedural programming language—Programming language that is based on functions rather than objects.

processor—Hardware component that executes machine-language instructions and enforces protection for system resources such as main memory.

programmed I/O (PIO)—Implementation of I/O for devices that do not support interrupts in which the transfer of every word from memory must be supervised by the processor.

property (of an object)—Part of an object that stores data about the object.

quad pumping—Technique for increasing processor performance by performing four memory transfers per clock cycle.

random access memory (RAM)—Memory whose contents can be accessed in any order.

real address—Address in main memory.

real memory—See main memory.

register—High-speed memory located on a processor that holds data for immediate use by the processor.

relative address—Address that is specified based on its location in relation to the beginning of a module.

relocatable loading—Method of loading that translates relative addresses in a load module to absolute addresses based on the location of a requested block of memory.

relocating—Process of adjusting the addresses of program code and data.

scanner—See lexical analyzer.

secondary storage—Memory that typically stores large quantities of data persistently. Secondary storage is one level lower than main memory in the memory hierarchy. After a computer is powered on, information is shuttled between secondary storage and main memory so that program instructions and data can be accessed by a processor. Hard disks are the most common form of secondary storage.

serial port—Interface to a device that transfers one bit at a time (e.g, keyboards and mice).

small computer systems interface (SCSI)—Interface designed to support multiple devices and high-speed connections. The SCSI interface supports a large number of devices than the less inexpensive IDE interface and is popular in Apple systems and computers containing large numbers of peripheral devices.

shared library—Collection of functions shared between several programs.

source code—Program code typically written in a high-level language or assembly language that must be compiled or interpreted before it can be understood by a computer.

spool (simultaneous peripheral operations online)—Method of I/O in which processes write data to secondary storage where it is buffered before being transferred to a low-speed device.

static RAM (SRAM)—RAM that does not need to be refreshed and will hold data as long as it receives power.

structured programming—Disciplined approach to creating programs that are clear, correct and easy to modify.

supervisor state—See kernel mode.

symbol resolution—Procedure performed by a linker that matches external references in one module to external names in another.

symbol table—Part of an object module that lists an entry for each external name and each external reference found in the module.

syntax analyzer—See parser.

system call—Procedure call that requests a service from an operating system. When a process issues a system call, the processor execution mode changes from user mode to kernel mode to execute operating system instructions that respond to the call.

systems programming—Development of software to manage a system's devices and applications.

trace—Tiny electrically conducting line that forms part of a bus.

transistor—Miniature switch that either allows or prevents current from passing to enable processors to perform operations on bits.

time-of-day clock—Clock that measures time as perceived outside of a computer system, typically accurate to thousandths or millionths of a second.

token—Characters in a program, separated by the lexical analyzer, that generally represent keywords, identifiers, operators or punctuation.

universal serial bus (USB)—Serial bus interface that transfers data up to 480Mbits per second, can supply power to its devices and supports hot swappable devices.

user mode—Mode of operation that does not allow processes to directly access system resources.

user state—See user mode.

volatile storage—Storage medium that loses data in the absence of power.

Windows API—Microsoft's interface for applications that execute in a Windows environment. The API enables programmers to request operating system services, which free the application programmer from writing the code to perform these operations and enables the operating system to protect its resources.

Write-Once, Read-Many (WORM) medium—Storage medium that can be modified only once, but whose contents can be accessed repeatedly.

Exercises

2.1 Distinguish among hardware, software and firmware.

2.2 Some hardware devices follow.

i. mainboard
ii. processor
iii. bus
iv. memory
v. hard disk
vi. peripheral device
vii. tertiary storage device
viii. register
ix. cache

Indicate which of these devices is best defined by each of the following. (Some items can have more than one answer.)

a. executes program instructions
b. not required for a computer to execute program instructions.
c. volatile storage medium.
d. the PCB that connects a system's processors to memory, secondary storage and peripheral devices.
e. fastest memory in a computer system
f. set of traces that transmit data between hardware devices
g. fast memory that improves application performance.
h. lowest level of memory in the memory hierarchy that a processor can reference directly.

2.3 Processor speeds have doubled roughly every 18 months. Has overall computer performance doubled at the same rate? Why or why not?

2.4 Sort the following list from fastest and most expensive memory to cheapest and least expensive memory: secondary storage, registers, main memory, tertiary storage, L2 cache, L1 cache. Why do systems contain several data stores of different size and speed? What is the motivation behind caching?

2.5 What are some costs and benefits of using nonvolatile RAM in all caches and main memory?

2.6 Why is it important to support legacy architectures?

2.7 Relate the principle of least privilege to the concepts of user mode, kernel mode and privileged instructions.

2.8 Describe several techniques for implementing memory protection.

2.9 Double buffering is a technique that allows an I/O channel and a processor to operate in parallel. On double-buffered input, for example, while a processor consumes one set of data in one buffer, the channel reads the next set of data into the other buffer so that the data will (hopefully) be ready for the processor. Explain in detail how a triple-buffering scheme might operate. In what circumstances would triple buffering be effective?

2.10 Describe two different techniques for handling the communications between a processor and devices.

2.11 Explain how DMA improves system performance. and cycle stealing.

2.12 Why is it appropriate for channel controllers to steal cycles from processors when accessing memory?

2.13 Explain the notion of spooling and why it is useful. How, do you suppose, does an input spooling system designed to read punched cards from a card reader operate?

2.14 Consider the following types of programming languages:

i. machine language
ii. assembly language
iii. high-level language
iv. object-oriented programming language
v. structured programming language

Indicate which of these categories is best defined by each of the following. (Some items can have more than one answer.)

a. focuses on manipulating things (nouns)
b. requires a translator programmer to convert the code into something a specific processor can understand

c. written using 1's and 0's

d. defines a disciplined approach to software development and focuses on actions (verbs).

e. specifies basic computer operations using English-like abbreviations for instructions

f. Java and C++

g. Fortran and Pascal

h. enables programmers to write code using everyday English words and mathematical notation

2.15 Briefly describe how a program written in a high-level language is prepared for execution.

2.16 Compare and contrast absolute loaders with relocating loaders. Give several motivations for the concept of relocatable loading.

2.17 What is microprogramming? Why is the term "firmware" appropriate for describing microcode that is part of a hardware device?

Suggested Projects

2.18 Prepare a research paper on MRAM, a form of nonvolatile RAM (see `www.research.ibm.com/resources/news/20030610_mram.shtml`).

2.19 Prepare a research paper on MEMS (MicroElectroMechanical System) storage, a secondary storage device intended to improve access times over hard disks (see `www.pdl.cmu.edu/MEMS/`).

2.20 Research the difference between the SCSI interface and the IDE interface for secondary storage devices. Why has IDE become the more popular choice?

2.21 Prepare a research paper on the design and implementation of Microsoft's .NET framework.

Recommended Reading

Several textbooks describe computer organization and architecture. Hennessy and Patterson's text is an excellent summary of computer architecture.[68] Blaauw and Brooks, Jr. also discuss computer architecture, providing detailed coverage of low-level mechanisms in computers.[69] For discussion of the most recent hardware technologies, however, only online journals can truly keep pace—see the Web Resources section.

An excellent essay regarding software engineering in the development of the OS/360 is *The Mythical Man-Month*, by Frederick P. Brooks.[70] Steve Maguire's *Debugging the Development Process* is a discussion of how to manage project teams, drawing from his managerial experience at Microsoft.[71] *Code Complete* by Steve McConnell is a valuable resource on good programming practices and software design.[72]

A review of compiler design and techniques can be found in a text by Aho et. al.[73] For those interested in modern compiler techniques, see the text by Grune et al.[74] Levine's text is an excellent resource for students interested in concerns and techniques regarding linking and loading an application.[75] The bibliography for this chapter is located on our Web site at `www.deitel.com/books/os3e/Bibliography.pdf`.

Works Cited

1. Wheeler, D. A., "More Than a Gigabuck: Estimating GNU/Linux's Size," June 30, 2001, updated July 29, 2002, Ver. 1.07, <www.dwheeler.com/sloc/>.
2. "Intel Executive Bio—Gordon E. Moore," *Intel Corporation*, October 30, 2003, <www.intel.com/pressroom/kits/bios/moore.htm>.
3. "Intel Executive Bio—Gordon E. Moore," *Intel Corporation*, October 30, 2003, <www.intel.com/pressroom/kits/bios/moore.htm>.
4. Moore, G., "Cramming More Components onto Integrated Circuits," *Electronics*, Vol. 38, No. 8, April 19, 1965.
5. "Expanding Moore's Law: The Exponential Opportunity," *Intel Corporation*, updated Fall 2002, 2002.
6. Gilheany, S., "Evolution of Intel Microprocessors: 1971 to 2007," Berghell Associates, March 28, 2002, <www.berghell.com/whitepapers/Evolution%20of%20Intel%20Microprocessors%201971%20to%202007.pdf>.

7. "Processors," *PCTechGuide,* <www.pctechguide.com/02procs.htm>.
8. "Registers," *PCGuide,* <www.pcguide.com/ref/cpu/arch/int/compRegisters-c.html>.
9. *PowerPC Microprocessor Family: Programming Environments Manual for 64- and 32-Bit Microprocessors*, Ver. 2.0, *IBM*, June 10, 2003.
10. "Clock Signals, Cycle Time and Frequency," *PCGuide.com*, April 17, 2001, <www.pcguide.com/intro/fun/clock.htm>.
11. "IA-32 Intel Architecture Software Developer's Manual," *System Programmer's Guide,* Vol. 1, 2002, p. 41.
12. De Gelas, J., "Ace's Guide to Memory Technology," *Ace's Hardware*, July 13, 2000, <www.aceshardware.com/Spades/read.php?article_id=5000172>.
13. "Hard Disk Drives," *PCGuide,* April 17, 2001, <www.pcguide.com/ref/hdd/>.
14. "System Bus Functions and Features," *PCGuide,* <www.pcguide.com/ref/mbsys/buses/func.htm>.
15. 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.
16. "PCI Express," *PCI-SIG*, <www.pcisig.com/specifications/pciexpress/>.
17. Scott, T. A., "Illustrating Programmed and Interrupt Driven I/O," *Proceedings of the Seventh Annual CCSC Midwestern Conference on Small Colleges*, October 2000, pp. 230–238.
18. Hennessy, J., and D. Patterson, *Computer Organization and Design*, San Francisco: Morgan Kaufmann Publishers, 1998, pp. 680–681.
19. "DMA Channel Function and Operation," *PCGuide.com*, <www.pcguide.com/ref/mbsys/res/dma/func.htm>.
20. "Peripheral Device," *Webopedia,* December 14, 2001, <www.webopedia.com/TERM/P/peripheral_device.html>.
21. "Serial Port," *CNET Glossary*, <www.cnet.com/Resources/Info/Glossary/Terms/serialport.html>.
22. "Serial Port," *CNET Glossary*, <www.cnet.com/Resources/Info/Glossary/Terms/serialport.html>.
23. "USB," *Computer Peripherals,* <peripherals.about.com/library/glossary/bldefusb.htm>.
24. Liu, P., and D. Thompson, "IEEE 1394: Changing the Way We Do Multimedia Communications," *IEEE Multimedia*, April 2000, <www.computer.org/multimedia/articles/firewire.htm>.
25. "IDE/ATA vs. SCSI: Interface Drive Comparison," *PCGuide.com*, <www.pcguide.com/ref/hdd/if/comp.htm>.
26. "SCSI FAQ," <www.faqs.org/faqs/scsi-faq/part1/>.
27. <www.scsita.org/aboutscsi/termsTermin.html>.
28. 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.
29. Denning, P., "Virtual Memory," *ACM Computing Surveys*, Vol. 2, No. 3, September 1970, pp. 153–189.
30. "Intel Developer Forum Day 3—More from the Tech Showcase," *Anandtech.com,* February 20, 2003, <www.anandtech.com/showdoc.html?i=1791&p=2>.
31. "Plug and Play Technology", *Microsoft Windows Platform Development,* March 21, 2003, <www.microsoft.com/hwdev/tech/pnp/default.asp>.
32. "Plug and Play for Windows 2000 and Windows XP," *Microsoft Windows Platform Development,* March 21, 2003, <www.microsoft.com/hwdev/tech/PnP/PnPNT5_2.asp>.
33. Smith, A., "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, September 1982, pp. 473–530.
34. "Buffer," *Data Recovery Glossary*, <www.datarecoverygroup.com/glossary/buffer.html>.
35. "Buffer," *Webopedia,* September 1, 1997, <www.webopedia.com/TERM/B/buffer.html>.
36. "Definition: Buffer," *FS-1037,* August 23, 1996, <www.its.bldrdoc.gov/fs-1037/dir-005/_0739.htm>.
37. "Spooling," *Sun Product Documentation: Glossary*, *Solaris 2.4 System Administrator AnswerBook*, <docs.sun.com/db/doc/801-6628/6i108opae?a=view>.
38. Glass, R. L., "An Elementary Discussion of Compiler/Interpreter Writing," *ACM Computing Surveys (CSUR)*, Vol. 1, No. 1, January 1969.
39. "Interpreter (Computer Software)," *Wikipedia, The Free Encyclopedia*, modified February 19, 2003, <www.wikipedia.org/wiki/Interpreter_(computer_software)>.
40. Presser, L., and J. White, "Linkers and Loaders," *ACM Computer Surveys*, Vol. 4, No. 3, September 1972, pp. 149–151.
41. "Object Code," April 7, 2001, <whatis.techtarget.com/definition/0,,sid9_gci539287,00.html>.
42. Aho, A., and J. Ullman, *Principles of Compiler Design*, Reading, MA: Addison-Wesley, 1977, pp. 6–7.
43. "Compiler," *IBM Reference/Glossary,* <www-1.ibm.com/ibm/history/reference/glossary_c.html>.
44. Presser, L., and J. White, "Linkers and Loaders," *ACM Computer Surveys*, Vol. 4, No. 3, September 1972, p. 153.
45. Levine, J., *Linkers and Loaders*, San Francisco: Morgan Kaufman Publishers, 2000, p. 5.
46. Presser, L., and J. White, "Linkers and Loaders," *ACM Computer Surveys*, Vol. 4, No. 3, September 1972, p. 164.
47. Levine, J., *Linkers and Loaders*, San Francisco: Morgan Kaufman Publishers, 2000, p. 6.
48. Carnegie-Mellon University, "The Mach Project Home Page," February 21, 1997, <www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html>.
49. Microsoft Corporation, "Microsoft–PressPass Rick Rashid Biography," 2003, <www.microsoft.com/presspass/exec/rick/default.asp>.
50. Westmacott, I., "The UNIX vs. NT Myth," July 1997, <webserver.cpg.com/wa/2.6>.

51. Carnegie-Mellon University, "The Mach Project Home Page," February 21, 1997, <www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html>.

52. Apple Computer, Inc., "Mac OS X Technologies—Darwin," 2003, <www.apple.com/macosx/technologies/darwin.html>.

53. Free Software Foundation, "GNU Mach," May 26, 2003, <www.gnu.org/software/hurd/gnumach.html>.

54. Rashid, R., et al., "Mach: A System Software Kernel," *Proceedings of the 1989 IEEE International Conference, COMPCON 89*, February 1989, <ftp://ftp.cs.cmu.edu/project/mach/doc/published/syskernel.ps>.

55. Coulouris, G.; J. Dollimore; and T Kindberg, "UNIX Emulation in Mach and Chorus," *Distributed Systems: Concepts and Design*, Addison-Wesley, 1994, pp. 597-584, <www.cdk3.net/oss/Ed2/UNIXEmulation.pdf>.

56. Rashid, R., et al., "Mach: A System Software Kernel," *Proceedings of the 1989 IEEE International Conference, COMPCON 89*, February 1989, <ftp://ftp.cs.cmu.edu/project/mach/doc/published/syskernel.ps>.

57. Coulouris, G.; J. Dollimore; and T Kindberg, "UNIX Emulation in Mach and Chorus," *Distributed Systems: Concepts and Design*, Addison-Wesley, 1994, pp. 597-584, <www.cdk3.net/oss/Ed2/UNIXEmulation.pdf>.

58. Rashid, R., et al., "Mach: A System Software Kernel," *Proceedings of the 1989 IEEE International Conference, COMPCON 89*, February 1989, <ftp://ftp.cs.cmu.edu/project/mach/doc/published/syskernel.ps>.

59. Rashid, R., et al., "Mach: A System Software Kernel," *Proceedings of the 1989 IEEE International Conference, COMPCON 89*, February 1989, <ftp://ftp.cs.cmu.edu/project/mach/doc/published/syskernel.ps>.

60. Presser, L., and J. White, "Linkers and Loaders," *ACM Computer Surveys*, Vol. 4, No. 3, September 1972, p. 151.

61. Presser, L., and J. White, "Linkers and Loaders," *ACM Computer Surveys*, Vol. 4, No. 3, September 1972, p. 150.

62. Hennessy, J., and D. Patterson, *Computer Organization and Design*, San Francisco: Morgan Kaufmann Publishers, 1998, pp. 399–400.

63. Rauscher, T., and P. Adams, "Microprogramming: A Tutorial and Survey of Recent Developments," *IEEE Transactions on Computers*, Vol. C-29, No. 1, January 1980, pp. 2–20.

64. Hennessy, J., and D. Patterson, *Computer Organization and Design,* San Francisco: Morgan Kaufmann Publishers, 1998, pp. 424–425.

65. Wilkes, M., "The Best Way to Design an Automatic Calculating Machine," *Report of the Machine University Computer Inaugural Conference*, Electrical Engineering Department of Manchester University, Manchester, England, July 1951, pp. 16–18.

66. Hennessy, J., and D. Patterson, *Computer Organization and Design,* San Francisco: Morgan Kaufmann Publishers, 1998, pp. 424–425.

67. "Firmware," *PCGuide,* April 17, 2001, <www.pcguide.com/ref/hdd/op/logicFirmware-c.html>.

68. Hennessy, J., and D. Patterson, *Computer Organization and Design*, San Francisco: Morgan Kaufmann Publishers, 1998.

69. Blaauw, G., and F. Brooks, Jr., *Computer Architecture*, Reading, MA: Addison-Wesley, 1997.

70. Brooks, F. P., *The Mythical Man-Month*, Reading, MA: Addison-Wesley, 1995.

71. Maguire, S., *Debugging the Development Process: Practical Strategies for Staying Focused, Hitting Ship Dates, and Building Solid Teams*, Microsoft Press, 1994.

72. McConnell, S., *Code Complete*, Microsoft Press, 1993.

73. Aho, A.; R. Sethi; and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley, 1986.

74. Grune, D.; H. Bal; C. Jacobs; and K. Langendoen, *Modern Compiler Design*, New York: John Wiley, 2000.

75. Levine, J., *Linkers and Loaders*, San Francisco: Morgan Kaufman Publishers, 2000.