MODERN OPERATING SYSTEMS

Fifth Edition

Andrew S. Tanenbaum
Herbert Bos

MODERN OPERATING SYSTEMS

FIFTH EDITION

ANDREW S. TANENBAUM
HERBERT BOS

Vrije Universiteit
Amsterdam, The Netherlands

1

INTRODUCTION

A modern computer consists of one or more processors, some amount of main memory, hard disks or Flash drives, printers, a keyboard, a mouse, a display, network interfaces, and various other input/output devices. All in all, a complex system. If every application programmer had to understand how all these things work in detail, no code would ever get written. Furthermore, managing all these components and using them optimally is an exceedingly challenging job. For this reason, computers are equipped with a layer of software called the **operating system**, whose job is to provide user programs with a better, simpler, cleaner, model of the computer and to handle managing all the resources just mentioned. Operating systems are the subject of this book.

It is important to realize that smart phones and tablets (like the Apple iPad) are just computers in a smaller package with a touch screen. They all have operating systems. In fact, Apple's iOS is fairly similar to macOS, which runs on Apple's desktop and MacBook systems. The smaller form factor and touch screen really doesn't change that much about what the operating system does. Android smartphones and tablets all run Linux as the true operating system on the bare hardware. What users perceive as "Android" is simply a layer of software running on top of Linux. Since macOS (and thus iOS) is derived from Berkeley UNIX and Linux is a clone of UNIX, by far the most popular operating system in the world is UNIX and its variants. For this reason, we will pay a lot of attention in this book to UNIX.

Most readers probably have had some experience with an operating system such as Windows, Linux, FreeBSD, or macOS, but appearances can be deceiving.

The program that users interact with, usually called the **shell** when it is text based and the **GUI** (**Graphical User Interface**) (which is pronounced "gooey") when it uses icons, is actually not part of the operating system, although it uses the operating system to get its work done.

A simple overview of the main components under discussion here is given in Fig. 1-1. Here we see the hardware at the bottom. The hardware consists of chips, boards, Flash drives, disks, a keyboard, a monitor, and similar physical objects. On top of the hardware is the software. Most computers have two modes of operation: kernel mode and user mode. The operating system, the most fundamental piece of software, runs in **kernel mode** (also called **supervisor mode**) for at least some of its functionality. In this mode, it has complete access to all the hardware and can execute any instruction the machine is capable of executing. The rest of the software runs in **user mode**, in which only a subset of the machine instructions is available. In particular, those instructions that affect control of the machine, determine the security boundaries, or do **I/O** (**Input/Output**) are forbidden to user-mode programs. We will come back to the difference between kernel mode and user mode repeatedly throughout this book. It plays a crucial role in how operating systems work.

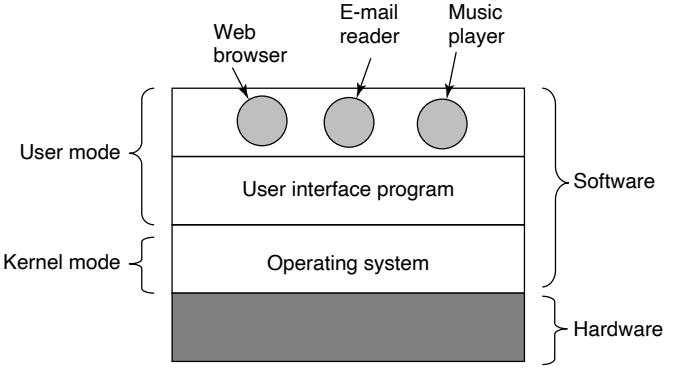

Figure 1-1. Where the operating system fits in.

The user interface program, shell or GUI, is the lowest level of user-mode software, and allows the user to start other programs, such as a Web browser, email reader, or music player. These programs, too, make heavy use of the operating system.

The placement of the operating system is shown in Fig. 1-1. It runs on the bare hardware and provides the base for all the other software.

An important distinction between the operating system and normal (user-mode) software is that if a user does not like a particular email reader, she is free to get a different one or write her own if she so chooses; she is typically not free to write her own clock interrupt handler, which is part of the operating system and is

protected by hardware against attempts by users to modify it. This distinction, however, is sometimes blurred, for instance in embedded systems (which may not have kernel mode) or interpreted systems (such as Java-based systems that use interpretation, not hardware, to separate the components).

Also, in many systems there are programs that run in user mode but help the operating system or perform privileged functions. For example, there is often a program that allows users to change their passwords. It is not part of the operating system and does not run in kernel mode, but it clearly carries out a sensitive function and has to be protected in a special way. In some systems, this idea is carried to an extreme, and pieces of what is traditionally considered to be the operating system (such as the file system) run in user mode. In such systems, it is difficult to draw a clear boundary. Everything running in kernel mode is clearly part of the operating system, but some programs running outside it are arguably also part of it, or at least closely associated with it.

Operating systems differ from user (i.e., application) programs in ways other than where they reside. In particular, they are huge, complex, and very long-lived. The source code for Windows is over 50 million lines of code. The source code for Linux is over 20 million lines of code. Both are still growing. To conceive of what this means, think of printing out 50 million lines in book form, with 50 lines per page and 1000 pages per volume (about the size of this book). Each book would contain 50,000 lines of code. It would take 1000 volumes to list an operating system of this size. Now imagine a bookcase with 20 books per shelf and seven shelves or 140 books in all. It would take a bit over seven bookcases to hold the full code of Windows 10. Can you imagine getting a job maintaining an operating system and on the first day having your boss bring you to a room with these seven bookcases of code and say: "Go learn that." And this is only for the part that runs in the kernel. No one at Microsoft understands all of Windows and probably most programmers there, even kernel programmers, understand only a small part of it. When essential shared libraries are included, the source code base gets much bigger. And this excludes basic application software (things like the browser, the media player, and so on).

It should be clear now why operating systems live a long time—they are very hard to write, and having written one, the owner is loath to throw it out and start again. Instead, such systems evolve over long periods of time. Windows 95/98/Me was basically one operating system and Windows NT/2000/XP/Vista/Windows 7/8/10 is a different one. They look similar to the users because Microsoft made very sure that the user interface of Windows 2000/XP/Vista/Windows 7 was quite similar to that of the system it was replacing, mostly Windows 98. This was not necessarily the case for Windows 8 and 8.1 which introduced a variety of changes in the GUI and promptly drew criticism from users who liked to keep things the same. Windows 10 reverted some of these changes and introduced a number of improvements. Windows 11 is built upon the framework of Windows 10. We will study Windows in detail in Chap. 11.

Besides Windows, the other main example we will use throughout this book is UNIX and its variants and clones. It, too, has evolved over the years, with versions like FreeBSD (and essentially, macOS) being derived from the original system, whereas Linux is a fresh code base, although very closely modeled on UNIX and highly compatible with it. The huge investment needed to develop a mature and reliable operating system from scratch led Google to adopt an existing one, Linux, as the basis of its Android operating system. We will use examples from UNIX throughout this book and look at Linux in detail in Chap. 10.

In this chapter, we will briefly touch on a number of key aspects of operating systems, including what they are, their history, what kinds are around, some of the basic concepts, and their structure. We will come back to many of these important topics in later chapters in more detail.

1.1 WHAT IS AN OPERATING SYSTEM?

It is hard to pin down what an operating system is other than saying it is the software that runs in kernel mode—and even that is not always true. Part of the problem is that operating systems perform two essentially unrelated functions: providing application programmers (and application programs, naturally) a clean abstract set of resources instead of the messy hardware ones and managing these hardware resources. Depending on who is doing the talking, you might hear mostly about one function or the other. Let us now look at both.

1.1.1 The Operating System as an Extended Machine

The **architecture** (instruction set, memory organization, I/O, and bus structure) of most computers at the machine-language level is primitive and awkward to program, especially for input/output. To make this point more concrete, consider modern **SATA** (**Serial ATA**) hard disks used on most computers. A book (Deming, 2014) describing an early version of the interface to the disk—what a programmer would have to know to use the disk—ran over 450 pages. Since then, the interface has been revised multiple times and is even more complicated than it was in 2014. Clearly, no sane programmer would want to deal with this disk at the hardware level. Instead, a piece of software, called a **disk driver**, deals with the hardware and provides an interface to read and write disk blocks, without getting into the details. Operating systems contain many drivers for controlling I/O devices.

But even this level is much too low for most applications. For this reason, all operating systems provide yet another layer of abstraction for using disks: files. Using this abstraction, programs can create, write, and read files, without having to deal with the messy details of how the hardware actually works.

This abstraction is the key to managing all this complexity. Good abstractions turn a nearly impossible task into two manageable ones. The first is defining and implementing the abstractions. The second is using these abstractions to solve the problem at hand. One abstraction that almost every computer user understands is the file, as mentioned above. It is a useful piece of information, such as a digital photo, saved email message, song, or Web page. It is much easier to deal with photos, emails, songs, and Web pages than with the details of SATA (or other) disks. The job of the operating system is to create good abstractions and then implement and manage the abstract objects thus created. In this book, we will talk a lot about abstractions. They are one of the keys to understanding operating systems.

This point is so important that it is worth repeating but in different words. With all due respect to the industrial engineers who so very carefully designed the Apple Macintosh computers (now known simply as "Macs"), hardware is grotesque. Real processors, memories, Flash drives, disks, and other devices are very complicated and present difficult, awkward, idiosyncratic, and inconsistent interfaces to the people who have to write software to use them. Sometimes this is due to the need for backward compatibility with older hardware. Other times it is an attempt to save money. Often, however, the hardware designers do not realize (or care) how much trouble they are causing for the software. One of the major tasks of the operating system is to hide the hardware and present programs (and their programmers) with nice, clean, elegant, consistent, abstractions to work with instead. Operating systems turn the awful into the beautiful, as shown in Fig. 1-2.

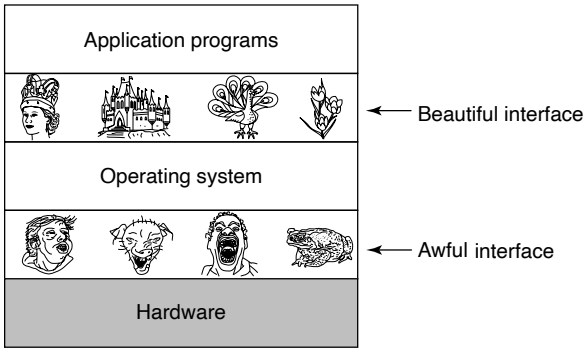

Figure 1-2. Operating systems turn awful hardware into beautiful abstractions.

It should be noted that the operating system's real customers are the application programs (via the application programmers, of course). They are the ones who deal directly with the operating system and its abstractions. In contrast, end users deal with the abstractions provided by the user interface, either a command-line shell or a graphical interface. While the abstractions at the user interface may be similar to the ones provided by the operating system, this is not always the case. To make this point clearer, consider the normal Windows desktop and the

line-oriented command prompt. Both are programs running on the Windows operating system and use the abstractions Windows provides, but they offer very different user interfaces. Similarly, a Linux user running Gnome or KDE sees a very different interface than a Linux user working directly on top of the underlying X Window System, but the underlying operating system abstractions are the same in both cases.

In this book, we will study the abstractions provided to application programs in great detail, but say rather little about user interfaces. That is a large and important subject, but one only peripherally related to operating systems.

1.1.2 The Operating System as a Resource Manager

The concept of an operating system as primarily providing abstractions to application programs is a top-down view. An alternative, bottom-up, view holds that the operating system is there to manage all the pieces of a complex system. Modern computers consist of processors, memories, timers, disks, mice, network interfaces, printers, touch screens, touch pad, and a wide variety of other devices. In the bottom-up view, the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs wanting them.

Modern operating systems allow multiple programs to be in memory and run at the same time. Imagine what would happen if three programs running on some computer all tried to print their output simultaneously on the same printer. The first few lines of printout might be from program 1, the next few from program 2, then some from program 3, and so forth. The result would be utter chaos. The operating system can bring order to the potential chaos by buffering all the output destined for the printer on the disk or Flash drive. When one program is finished, the operating system can then copy its output from the disk file where it has been stored for the printer, while at the same time the other program can continue generating more output, oblivious to the fact that the output is not really going to the printer (yet).

When a computer (or network) has more than one user, managing and protecting the memory, I/O devices, and other resources is even more important since the users might otherwise interfere with one another. In addition, users often need to share not only hardware, but information (files, databases, etc.) as well. In short, this view of the operating system holds that its primary task is to keep track of which programs are using which resource, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

Resource management includes **multiplexing** (sharing) resources in two different ways: in time and in space. When a resource is time multiplexed, different programs or users take turns using it. First one of them gets to use the resource, then another, and so on. For example, with only one CPU and multiple programs that want to run on it, the operating system first allocates the CPU to one program, then, after it has run long enough, another program gets to use the CPU, then

another, and then eventually the first one again. Determining how the resource is time multiplexed—who goes next and for how long—is the task of the operating system. Another example of time multiplexing is sharing the printer. When multiple print jobs are queued up for printing on a single printer, a decision has to be made about which one is to be printed next.

The other kind of multiplexing is space multiplexing. Instead of the customers taking turns, each one gets part of the resource. For example, main memory is normally divided up among several running programs, so each one can be resident at the same time (for example, in order to take turns using the CPU). Assuming there is enough memory to hold multiple programs, it is more efficient to hold several programs in memory at once rather than give one of them the entire mem, especially if it only needs a small fraction of the total. Of course, this raises issues of fairness, protection, and so forth, and it is up to the operating system to solve them. Other resource that are space multiplexed are disks and Flash drives. In many systems, a single disk can hold files from many users at the same time. Allocating disk space and keeping track of who is using which disk blocks is a typical operating system task. By the way, people commonly refer to all nonvolatile memory as "disks," but in this book we try to explicitly distinguish between disks, which have spinning magnetic platters, and **SSDs** (**Solid State Drives**), which are based on Flash memory and electronic rather than mechanical. Still, from a software point of view, SSDs are similar to disks in many (but not all) ways.

1.2 HISTORY OF OPERATING SYSTEMS

Operating systems have been evolving through the years. In the following sections, we will briefly look at a few of the highlights. Since operating systems have historically been closely tied to the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like. This mapping of operating system generations to computer generations is crude, but it does provide some structure where there would otherwise be none. The progression given below is largely chronological, but it has been a bumpy ride. Each development did not wait until the previous one nicely finished before getting started. There was a lot of overlap, not to mention many false starts and dead ends. Take this as a guide, not as the last word.

The first true digital computer was designed by the English mathematician Charles Babbage (1792–1871). Although Babbage spent most of his life and fortune trying to build his "analytical engine," he never got it working properly because it was purely mechanical, and the technology of his day could not produce the required wheels, gears, and cogs to the high precision that he needed. Needless to say, the analytical engine did not have an operating system.

As an interesting historical aside, Babbage realized that he would need software for his analytical engine, so he hired a young woman named Ada Lovelace,

who was the daughter of the famed British poet Lord Byron, as the world's first programmer. The programming language Ada® is named after her.

1.2.1 The First Generation (1945–1955): Vacuum Tubes

After Babbage's unsuccessful efforts, little progress was made in constructing digital computers until the World War II period, which stimulated an explosion of activity. Professor John Atanasoff and his graduate student Clifford Berry built what is now regarded as the first functioning digital computer at Iowa State University. It used 300 vacuum tubes. At roughly the same time, Konrad Zuse in Berlin built the Z3 computer out of electromechanical relays. In 1944, the Colossus was built and programmed by a group of scientists (including Alan Turing) at Bletchley Park, England, the Mark I was built by Howard Aiken at Harvard, and the ENIAC was built by William Mauchley and his graduate student J. Presper Eckert at the University of Pennsylvania. Some were binary, some used vacuum tubes, some were programmable, but all were very primitive and took seconds to perform even the simplest calculation.

In these early days, a single group of people (usually engineers) designed, built, programmed, operated, and maintained each machine. All programming was done in absolute machine language, or even worse yet, by wiring up electrical circuits by connecting thousands of cables to plugboards to control the machine's basic functions. Programming languages were unknown (even assembly language was unknown). Operating systems were unheard of. The usual mode of operation was for the programmer to sign up for a block of time using the signup sheet on the wall, then come down to the machine room, insert his or her plugboard into the computer, and spend the next few hours hoping that none of the 20,000 or so vacuum tubes would burn out during the run. Virtually all the problems were simple straightforward mathematical and numerical calculations, such as grinding out tables of sines, cosines, and logarithms, or computing artillery trajectories.

By the early 1950s, the routine had improved somewhat with the introduction of punched cards. It was now possible to write programs on cards and read them in instead of using plugboards; otherwise, the procedure was the same.

1.2.2 The Second Generation (1955–1965): Transistors and Batch Systems

The introduction of the transistor in the mid-1950s changed the picture radically. Computers became reliable enough that they could be manufactured and sold to paying customers with the expectation that they would continue to function long enough to get useful work done. For the first time, there was a clear separation between designers, builders, operators, programmers, and maintenance personnel.

These machines, which are now called **mainframes**, were locked away in large, specially air-conditioned computer rooms, with staffs of professional operators to run them. Only large corporations or government agencies or universities

could afford the multimillion-dollar price tag. To run a **job** (i.e., a program or set of programs), a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. The programmer would then bring the card deck down to the input room, hand it to one of the operators, and go drink coffee until the output was ready.

When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and carry it over to the output room, so that the programmer could collect it later. Then the operator would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the **batch system**. The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401, which was quite good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations. Other, much more expensive machines, such as the IBM 7094, were used for the real computing. This situation is shown in Fig. 1-3.

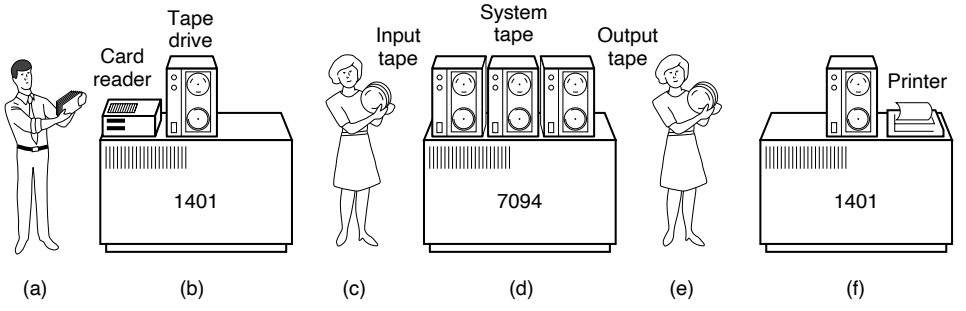

Figure 1-3. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

After about an hour of collecting a batch of jobs, the cards were read onto a magnetic tape, which was carried into the machine room, where it was mounted on a tape drive. The operator then loaded a special program (the ancestor of today's operating system), which read the first job from tape and ran it. The output was written onto a second tape, instead of being printed. After each job finished, the operating system automatically read the next job from the tape and began running it. When the whole batch was done, the operator removed the input and output

tapes, replaced the input tape with the next batch, and brought the output tape to a 1401 for printing **off line** (i.e., not connected to the main computer).

The structure of a typical input job is shown in Fig. 1-4. It started out with a $JOB card, specifying the maximum run time in minutes, the account number to be charged, and the programmer's name. Then came a $FORTRAN card, telling the operating system to load the FORTRAN compiler from the system tape. It was directly followed by the program to be compiled, and then a $LOAD card, directing the operating system to load the object program just compiled. (Compiled programs were often written on scratch tapes and had to be loaded explicitly.) Next came the $RUN card, telling the operating system to run the program with the data following it. Finally, the $END card marked the end of the job. These control cards were the forerunners of modern shells and command-line interpreters.

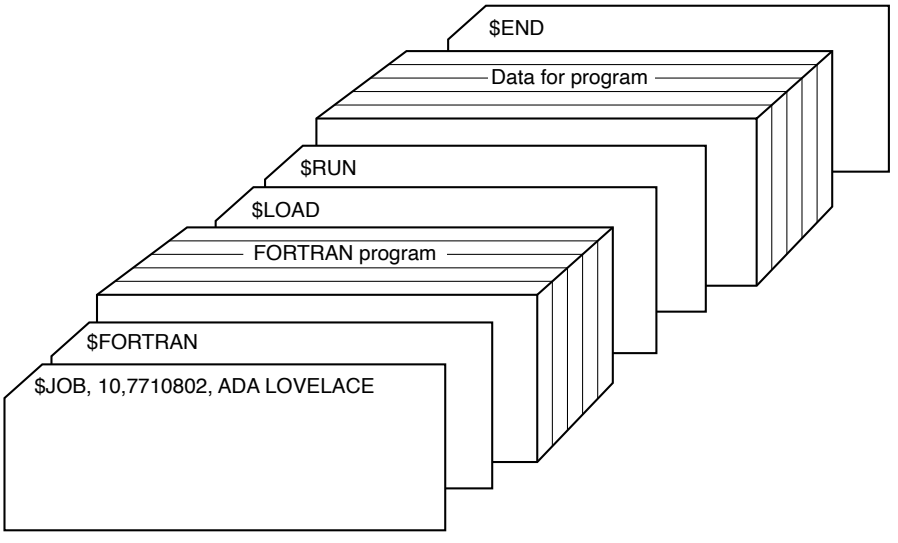

Figure 1-4. Structure of a typical FMS job.

Large, second-generation computers were used mostly for scientific and engineering calculations, such as solving the partial differential equations that often occur in physics and engineering. They were largely programmed in FORTRAN and assembly language. Typical operating systems were FMS (the Fortran Monitor System) and IBSYS, IBM's operating system for the 7094.

1.2.3 The Third Generation (1965–1980): ICs and Multiprogramming

By the early 1960s, most computer manufacturers had two distinct and incompatible, product lines. On the one hand, there were the word-oriented, large-scale scientific computers, such as the 7094, which were used for industrial-strength

numerical calculations in science and engineering. On the other hand, there were the character-oriented, commercial computers, such as the 1401, which were widely used for tape sorting and printing by banks and insurance companies.

Developing and maintaining two completely different product lines was an expensive proposition for the manufacturers. In addition, many new computer customers initially needed a small machine but later outgrew it and wanted a bigger machine that would run all their old programs, but faster.

IBM attempted to solve both of these problems at a single stroke by introducing the System/360. The 360 was a series of software-compatible machines ranging from 1401-sized models to much larger ones, more powerful than the mighty 7094. The machines differed only in price and performance (maximum memory, processor speed, number of I/O devices permitted, and so forth). Since they all had the same architecture and instruction set, programs written for one machine could run on all the others—at least in theory. (But as Yogi Berra reputedly said: "In theory, theory and practice are the same; in practice, they are not.") Since the 360 was designed to handle both scientific (i.e., numerical) and commercial computing, a single family of machines could satisfy the needs of all customers. In subsequent years, IBM came out with backward compatible successors to the 360 line, using more modern technology, known as the 370, 4300, 3080, and 3090. The zSeries is the most recent descendant of this line, although it has diverged considerably from the original.

The IBM 360 was the first major computer line to use (small-scale) **ICs** (**Integrated Circuits**), thus providing a major price/performance advantage over the second-generation machines, which were built up from individual transistors. It was an immediate and massive success, and the idea of a family of compatible computers was soon adopted by all the other major manufacturers. The descendants of these machines are still in use at computer centers today. Nowadays they are often used for managing huge databases (e.g., for airline reservation systems) or as servers for World Wide Web sites that must process thousands of requests per second.

The greatest strength of the "single-family" idea was simultaneously its greatest weakness. The original intention was that all software, including the operating system, **OS/360**, had to work on all models. It had to run on small systems, which often just replaced 1401s for copying cards to tape, and on very large systems, which often replaced 7094s for doing weather forecasting and other heavy computing. It had to be good on systems with few peripherals and on systems with many peripherals. It had to work in commercial environments and in scientific environments. Above all, it had to be efficient for all of these different uses.

There was no way that IBM (or anybody else for that matter) could write a piece of software to meet all those conflicting requirements. The result was an enormous and extraordinarily complex operating system, probably two to three orders of magnitude larger than FMS. It consisted of millions of lines of assembly language written by thousands of programmers, and contained thousands upon

thousands of bugs, which necessitated a continuous stream of new releases in an attempt to correct them. Each new release fixed some bugs and introduced new ones, so the number of bugs probably remained constant over time.

One of the designers of OS/360, Fred Brooks, subsequently wrote a now-classic, witty, and incisive book (Brooks, 1995) describing his experiences with OS/360. While it would be impossible to summarize the book here, suffice it to say that the cover shows a herd of prehistoric beasts stuck in a tar pit. The cover of Silberschatz et al. (2012) makes a similar point about operating systems being dinosaurs. He also made the comment that adding programmers to a late software project makes it even later as well saying that it takes 9 months to produce a child, no matter how many women you assign to the project.

Despite its enormous size and problems, OS/360 and the similar third-generation operating systems produced by other computer manufacturers actually satisfied most of their customers reasonably well. They also popularized several key techniques absent in second-generation operating systems. Probably the most important of these was **multiprogramming**. On the 7094, when the current job paused to wait for a tape or other I/O operation to complete, the CPU simply sat idle until the I/O finished. With heavily CPU-bound scientific calculations, I/O is infrequent, so this wasted time is not significant. With commercial data processing, the I/O wait time can often be 80% or 90% of the total time, so something had to be done to avoid having the (expensive) CPU be idle so much.

The solution that evolved was to partition memory into several pieces, with a different job in each partition, as shown in Fig. 1-5. While one job was waiting for I/O to complete, another job could be using the CPU. If enough jobs could be held in main memory at once, the CPU could be kept busy nearly 100% of the time. Having multiple jobs safely in memory at once requires special hardware to protect each job against snooping and mischief by the other ones, but the 360 and other third-generation systems were equipped with this hardware.

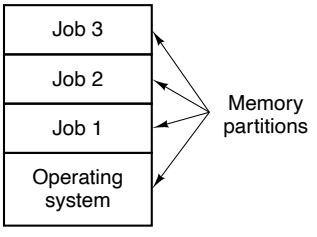

Figure 1-5. A multiprogramming system with three jobs in memory.

Another major feature present in third-generation operating systems was the ability to read jobs from cards onto the disk as soon as they were brought to the computer room. Then, whenever a running job finished, the operating system could load a new job from the disk into the now-empty partition and run it. This ability is

called **spooling** (from **Simultaneous Peripheral Operation On Line**) and was also used for output. With spooling, the 1401s were no longer needed, and much carrying of tapes disappeared.

Although third-generation operating systems were well suited for big scientific calculations and massive commercial data-processing runs, they were still basically batch systems. Many programmers pined for the first-generation days when they had the machine all to themselves for a few hours, so they could debug their programs quickly. With third-generation systems, the time between submitting a job and getting back the output was often several hours, so a single misplaced comma could cause a compilation to fail, and the programmer to waste half a day. Programmers did not like that very much.

This desire for quick response time paved the way for **timesharing**, a variant of multiprogramming, in which each user has an online terminal. In a timesharing system, if 20 users are logged in and 17 of them are thinking or talking or drinking coffee, the CPU can be allocated in turn to the three jobs that want service. Since people debugging programs usually issue short commands (e.g., compile a five-page procedure rather than long ones (e.g., sort a million-record file), the computer can provide fast, interactive service to a number of users and perhaps also work on big batch jobs in the background when the CPU is otherwise idle. The first general-purpose timesharing system, **CTSS** (**Compatible Time Sharing System**), was developed at M.I.T. on a specially modified 7094 (Corbató et al., 1962). However, timesharing did not really become popular until the necessary protection hardware became widespread during the third generation.

After the success of the CTSS system, M.I.T., Bell Labs, and General Electric (at that time a major computer manufacturer) decided to embark on the development of a "computer utility," that is, a machine that would support some hundreds of simultaneous timesharing users. Their model was the electricity system—when you need electric power, you just stick a plug in the wall, and within reason, as much power as you need will be there. The designers of this system, known as **MULTICS** (**MULTiplexed Information and Computing Service**), envisioned one huge machine providing computing power for everyone in the Boston area. The idea that machines 10,000 times faster than their GE-645 mainframe would be sold (for well under $1000) by the millions only 40 years later was pure science fiction. Sort of like the idea of supersonic trans-Atlantic undersea trains now.

MULTICS was a mixed success. It was designed to support hundreds of users on a machine 1000× slower than a modern smartphone and with a million times less memory. This is not quite as crazy as it sounds, since in those days people knew how to write small, efficient programs, a skill that has subsequently been completely lost. There were many reasons that MULTICS did not take over the world, not the least of which is that it was written in the PL/I programming language, and the PL/I compiler was years late and barely worked at all when it finally arrived. In addition, MULTICS was enormously ambitious for its time, much like Charles Babbage's analytical engine in the 19th century.

To make a long story short, MULTICS introduced many seminal ideas into the computer literature, but turning it into a serious product and a major commercial success was a lot harder than anyone had expected. Bell Labs dropped out of the project, and General Electric quit the computer business altogether. However, M.I.T. persisted and eventually got MULTICS working. It was ultimately sold as a commercial product by the company (Honeywell) that bought GE's computer business when GE got tired of it and was installed by about 80 major companies and universities worldwide. While their numbers were small, MULTICS users were fiercely loyal. General Motors, Ford, and the U.S. National Security Agency, for example, shut down their MULTICS systems only in the late 1990s, 30 years after MULTICS was released, after years of begging Honeywell to update the hardware. The last MULTICS system was shut down amid a lot of tears in October 2000. Can you imagine hanging onto your PC for 30 years because you think it is so much better than everything else out there? That's the kind of loyalty MULTICS inspired—and for good reason. It was hugely important.

By the end of the 20th century, the concept of a computer utility had fizzled out, but it came back in the form of **cloud computing**, in which relatively small computers (including smartphones, tablets, and the like) are connected to servers in vast and distant data centers where all the computing is done, with the local computer mostly handling the user interface. The motivation here is that most people do not want to administrate an increasingly complex and evolving computer system and would prefer to have that work done by a team of professionals, for example, people working for the company running the data center.

Despite its lack of commercial success, MULTICS had a huge influence on subsequent operating systems (especially UNIX and its derivatives, Linux, macOS, iOS, and FreeBSD). It is described in several papers and a book (Corbató and Vyssotsky, 1965; Daley and Dennis, 1968; Organick, 1972; Corbató et al., 1972; and Saltzer, 1974). It also has an active Website, located at *www.multicians.org*, with much information about the system, its designers, and its users.

Another major development during the third generation was the phenomenal growth of minicomputers, starting with the DEC PDP-1 in 1961. The PDP-1 had only 4K of 18-bit words, but at $120,000 per machine (less than 5% of the price of a 7094), it sold like hotcakes. For certain kinds of nonnumerical work, it was almost as fast as the 7094 and gave birth to a whole new industry. It was quickly followed by a series of other PDPs (unlike IBM's family, all incompatible) culminating in the PDP-11.

One of the computer scientists at Bell Labs who had worked on the MULTICS project, Ken Thompson, subsequently found a small PDP-7 minicomputer that no one was using and set out to write a stripped-down, one-user version of MULTICS. This work later developed into the **UNIX** operating system, which became popular in the academic world, with government agencies, and with many companies.

The history of UNIX has been told elsewhere (e.g., Salus, 1994). Part of that story will be given in Chap. 10. For now, suffice it to say that because the source

code was widely available, various organizations developed their own (incompatible) versions, which led to chaos. Two major versions developed, **System V**, from AT&T, and **BSD** (**Berkeley Software Distribution**) from the University of California at Berkeley. These had minor variants as well. To make it possible to write programs that could run on any UNIX system, IEEE developed a standard for UNIX, called **POSIX**, that most versions of UNIX now support. POSIX defines a minimal system-call interface that conformant UNIX systems must support. In fact, some other operating systems now also support the POSIX interface.

As an aside, it is worth mentioning that in 1987, one of the authors (Tanenbaum) released a small clone of UNIX, called **MINIX**, primarily for educational purposes. Functionally, MINIX is very similar to UNIX, including POSIX support. Since that time, the original version of MINIX has evolved into MINIX 3, which is highly modular and focused on very high reliability and available for free from the Website *www.minix3.org*. MINIX 3 has the ability to detect and replace faulty or even crashed modules (such as I/O device drivers) on the fly without a reboot and without disturbing running programs. A book describing its internal operation and listing the source code in an appendix is also available (Tanenbaum and Woodhull, 2006).

The desire for a free production (as opposed to educational) version of MINIX led a Finnish student, Linus Torvalds, to write **Linux**. This system was directly inspired by and developed on MINIX and originally supported various MINIX features (e.g., the MINIX file system). It has since been extended in many ways by many people but still retains some underlying structure common to MINIX and to UNIX. Readers interested in a detailed history of Linux and the open source movement might want to read Glyn Moody's (2001) book. Most of what will be said about UNIX in this book thus applies to System V, MINIX, Linux, and other versions and clones of UNIX as well.

Interestingly, both Linux and MINIX have become widely used. Linux powers a huge share of the servers in data centers and forms the basis of **Android** which dominates the smartphone market. MINIX was adapted by Intel for a separate and somewhat secret "management" processor embedded in virtually all its chipsets since 2008. In other words, if you have an Intel CPU, you also run MINIX deep in your processor, even if your main operating system is, say, Windows or Linux.

1.2.4 The Fourth Generation (1980–Present): Personal Computers

With the development of **LSI** (**Large Scale Integration**) circuits—chips containing thousands of transistors on a square centimeter of silicon—the age of the personal computer dawned. In terms of architecture, personal computers (initially called **microcomputers**) were not all that different from minicomputers of the PDP-11 class, but in terms of price they certainly were different. Where the minicomputer made it possible for a department in a company or university to have its

own computer, the microprocessor chip made it possible for a single individual to have his or her own personal computer.

In 1974, when Intel came out with the 8080, the first general-purpose 8-bit CPU, it wanted an operating system for it, in part to be able to test it. Intel asked one of its consultants, Gary Kildall, to write one. Kildall and a friend first built a controller for the newly released Shugart Associates 8-inch floppy disk and hooked the floppy disk up to the 8080, thus producing the first microcomputer with a disk. Kildall then wrote a disk-based operating system called **CP/M** (**Control Program for Microcomputers**) for it. Since Intel did not think that disk-based microcomputers had much of a future, when Kildall asked for the rights to CP/M, Intel granted his request. Kildall then formed a company, Digital Research, to further develop and sell CP/M.

In 1977, Digital Research rewrote CP/M to make it suitable for running on the many microcomputers using the 8080, Zilog Z80, and other CPU chips. Many application programs were written to run on CP/M, allowing it to completely dominate the world of microcomputing for about 5 years.

In the early 1980s, IBM designed the IBM PC and looked around for software to run on it. People from IBM contacted Bill Gates to license his BASIC interpreter. They also asked him if he knew of an operating system to run on the PC. Gates suggested that IBM contact Digital Research, then the world's dominant operating systems company. Making what was without a doubt the worst business decision in recorded history, Kildall refused to meet with IBM, sending a subordinate instead. To make matters even worse, his lawyer even refused to sign IBM's nondisclosure agreement covering the not-yet-announced PC. Consequently, IBM went back to Gates asking if he could provide them with an operating system.

When IBM came back to him, Gates quickly realized that a local computer manufacturer, Seattle Computer Products, had a suitable operating system, **DOS** (**Disk Operating System**). He approached them and asked to buy it (allegedly for $75,000), which they readily accepted. Gates then offered IBM a DOS/BASIC package, which IBM accepted. IBM wanted certain modifications, so Gates hired the person who wrote DOS, Tim Paterson, as an employee of Gates' fledgling company, Microsoft, to make them. The revised system was renamed **MS-DOS** (**MicroSoft Disk Operating System**) and quickly came to dominate the IBM PC market. A key factor here was Gates' (in retrospect, extremely wise) decision to sell MS-DOS to computer companies for bundling with their hardware, compared to Kildall's attempt to sell CP/M to end users one at a time (at least initially). After all this transpired, Kildall died suddenly and unexpectedly from causes that have not been fully disclosed.

By the time the successor to the IBM PC, the IBM PC/AT, came out in 1983 with the Intel 80286 CPU, MS-DOS was firmly entrenched and CP/M was on its last legs. MS-DOS was later widely used on the 80386 and 80486. Although the initial version of MS-DOS was fairly primitive, subsequent versions included more advanced features, including many taken from UNIX. (Microsoft was well aware

of UNIX, even selling a microcomputer version of it called XENIX during the company's early years.)

CP/M, MS-DOS, and other operating systems for early microcomputers were all based on users typing in commands from the keyboard. That eventually changed due to research done by Doug Engelbart at Stanford Research Institute in the 1960s. Engelbart invented the Graphical User Interface, complete with windows, icons, menus, and mouse. These ideas were adopted by researchers at Xerox PARC and incorporated into machines they built.

One fine day, Steve Jobs, who co-invented the Apple computer in his garage, visited PARC, saw a GUI, and instantly realized its potential value, something Xerox management famously did not. This strategic blunder of incredibly gargantuan proportions led to a book entitled *Fumbling the Future* (Smith and Alexander, 1988). Jobs then embarked on building an Apple with a GUI. This project led to the Lisa, which was too expensive and failed commercially. Jobs' second attempt, the Apple Macintosh, was an huge success, not only because it was much cheaper than the Lisa, but also because it was **user friendly**, meaning that it was intended for users who not only knew nothing about computers but furthermore had absolutely no intention whatsoever of learning. In the creative world of graphic design, professional digital photography, and professional digital video production, Macintoshes became widely used and their users loved them. In 1999, Apple adopted a kernel derived from Carnegie Mellon University's Mach microkernel which was originally developed to replace the kernel of BSD UNIX. Thus, Apple's **macOS** is a UNIX-based operating system, albeit with a distinctive interface.

When Microsoft decided to build a successor to MS-DOS, it was strongly influenced by the success of the Macintosh. It produced a GUI-based system called Windows, which originally ran on top of MS-DOS (i.e., it was more like a shell than a true operating system). For about 10 years, from 1985 to 1995, Windows was just a graphical environment on top of MS-DOS. However, starting in 1995 a freestanding version, Windows 95, was released that incorporated many operating system features into it, using the underlying MS-DOS system only for booting and running old MS-DOS programs. Microsoft rewrote much of the operating system from scratch for **Windows NT**, a full 32-bit system. The lead designer for Windows NT was David Cutler, who was also one of the designers of the VAX VMS operating system, so some ideas from VMS are present in NT. In fact, so many ideas from VMS were present in it that the owner of VMS, DEC, sued Microsoft. The case was settled out of court for an amount of money requiring many digits to express. Version 5 of Windows NT was renamed Windows 2000 in early 1999, and 2 years later Microsoft released a slightly upgraded version called Windows XP which had a longer run than other versions (6 years).

After Windows 2000, Microsoft broke up the Windows family into a client and a server line. The client line was based on XP and its successors, while the server line produced Windows Server 2003–2019 and now Windows Server vNext. Microsoft later also introduced a third line, for the embedded world. All of these

families of Windows forked off their own variations in the form of **service packs** in a dizzying proliferation of versions. It was enough to drive some administrators (and writers of operating systems textbooks) balmy.

When in January 2007 Microsoft finally released the successor to Windows XP, which it called Vista, it came with a new graphical interface, improved security, and many new or upgraded user programs. It bombed. Users complained about high system requirements and restrictive licensing terms. Its successor, Windows 7, a much less resource hungry version of the operating system, quickly overtook it. In 2012, Windows 8 came out. It had a completely new look and feel, geared largely for touch screens. The company hoped that the new design would become the dominant operating system on a wide variety of devices: desktops, notebooks, tablets, phones, and home theater PCs. It did not. While Windows 8 (and especially Windows 8.1) were successful, their popularity was mostly limited to PCs. In fact, many people did not like the new design much and Microsoft reverted it in 2015 in Windows 10. A few years later, Windows 10 overtook Windows 7 as the most popular Windows version. Windows 11 was released in 2021.

The other major contender in the personal computer world comprises the UNIX family. UNIX, and especially **Linux**, is strongest on network and enterprise servers but also popular on desktop computers, notebooks, tablets, embedded systems, and smartphones. **FreeBSD** is also a popular UNIX derivative, originating from the BSD project at Berkeley. Every modern Mac runs a modified version of FreeBSD (macOS). UNIX derivatives are widely used on mobile devices, such as those running iOS 7 or Android.

Many UNIX users, especially experienced programmers, prefer a command-based interface to a GUI, so nearly all UNIX systems support a windowing system called the **X Window System** (also known as **X11**) produced at M.I.T. This system handles the basic window management, allowing users to create, delete, move, and resize windows using a mouse. Often a complete GUI-based desktop environment, such as **Gnome** or **KDE**, is available to run on top of X11, giving UNIX a look and feel something like the Macintosh or Microsoft Windows, for those UNIX users who want such a thing.

An interesting development that began during the mid-1980s was the development of **network operating systems** and **distributed operating systems** to manage a collection of computers (Van Steen and Tanenbaum, 2017). In a network operating system, the users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own local user (or users). Such systems are not fundamentally different from single-processor operating systems. They obviously need a network interface and some low-level software to drive it, as well as programs to achieve remote login and remote file access, but these additions do not change the essential structure of the operating system.

A distributed operating system, in contrast, is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple

processors. The users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uniprocessor operating system, because distributed and centralized systems differ in certain critical ways. Distributed systems, for example, often allow applications to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism. Moreover, communication delays within the network often mean that these (and other) algorithms must run with incomplete, outdated, or even incorrect information. This situation differs radically from that in a single-processor system in which the operating system has complete information about the system state.

1.2.5 The Fifth Generation (1990–Present): Mobile Computers

Ever since detective Dick Tracy started talking to his "two-way radio wrist watch" in the 1940s comic strip, people have craved a communication device they could carry around wherever they went. The first real mobile phone appeared in 1946 and weighed some 40 kilos. You could take it wherever you went as long as you had a car in which to carry it.

The first true handheld phone appeared in the 1970s and, at roughly one kilogram, was positively featherweight. It was affectionately known as "the brick." Pretty soon everybody wanted one. Today, mobile phone penetration in developed countries is close to 90% of the global population. We can make calls not just with our portable phones and wrist watches, but even with eyeglasses and other wearable items. Moreover, the phone part is no longer central. We receive email, surf the Web, text our friends, play games, navigate around heavy traffic—and do not even think twice about it.

While the idea of combining telephony and computing in a phone-like device has been around since the 1970s also, the first real smartphone did not appear until the mid-1990s when Nokia released the N9000, which literally combined two, mostly separate devices: a phone and a Personal Digital Assistant. In 1997, Ericsson coined the term *smartphone* for its GS88 "Penelope."

Now that smartphones have become ubiquitous, the competition between the operating systems is as fierce as in the PC world. At the time of writing, Google's Android is the dominant operating system with Apple's iOS a clear second, but this was not always the case and all may be different again in just a few years. If anything is clear in the world of smartphones, it is that it is not easy to stay king of the mountain for long.

After all, most smartphones in the first decade after their inception were running **Symbian** OS. It was the operating system of choice for popular brands like Samsung, Sony Ericsson, Motorola, and especially Nokia. However, other operating systems like **RIM's** Blackberry OS (introduced for smartphones in 2002) and

Apple's iOS (released for the first **iPhone** in 2007) started eating into Symbian's market share. Many expected that RIM would dominate the business market, while iOS would dominate on consumer devices. Symbian's market share plummeted. In 2011, Nokia ditched Symbian and announced it would focus on Windows Phone as its primary platform. For some time, Apple and RIM were the toast of the town (although not nearly as dominant as Symbian had been), but it did not take very long for Android, a Linux-based operating system released by Google in 2008, to overtake all its rivals.

For phone manufacturers, Android had the advantage that it was open source and available under a permissive license. As a result, they could tinker with it and adapt it to their own hardware with ease. Also, it has a huge community of developers writing apps, mostly in the familiar Java programming language. Even so, the past years have shown that the dominance may not last, and Android's competitors are eager to claw back some of its market share. We will look at Android in detail in Sec. 10.8.

1.3 COMPUTER HARDWARE REVIEW

An operating system is intimately tied to the hardware of the computer it runs on. It extends the computer's instruction set and manages its resources. To work, it must know a great deal about the hardware, at least about how the hardware appears to the programmer. For this reason, let us briefly review computer hardware as found in modern personal computers. After that, we can start getting into the details of what operating systems do and how they work.

Conceptually, a simple personal computer can be abstracted to a model resembling that of Fig. 1-6. The CPU, memory, and I/O devices are all connected by a system bus and communicate with one another over it. Modern personal computers have a more complicated structure, involving multiple buses, which we will look at later. For the time being, this model will be sufficient. In the following sections, we will briefly review these components and examine some of the hardware issues that are of concern to operating system designers. Needless to say, this will be a very compact summary. Many books have been written on the subject of computer hardware and computer organization. Two well-known ones are by Tanenbaum and Austin (2012) and Patterson and Hennessy (2018).

1.3.1 Processors

The "brain" of the computer is the CPU. It fetches instructions from memory and executes them. The basic cycle of every CPU is to fetch the first instruction from memory, decode it to determine its type and operands, execute it, and then fetch, decode, and execute subsequent instructions. The cycle is repeated until the program finishes. In this way, programs are carried out.

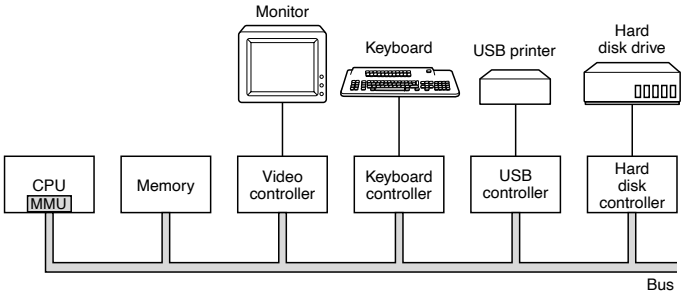

Figure 1-6. Some of the components of a simple personal computer.

Each CPU has a specific set of instructions that it can execute. Thus an x86 processor cannot execute ARM programs and an ARM processor cannot execute x86 programs. Please note that we will use the term **x86** to refer to all the Intel processors descended from the 8088, which was used on the original IBM PC. These include the 286, 386, and Pentium series, as well as the modern Intel core i3, i5, and i7 CPUs (and their clones).

Because accessing memory to get an instruction or data word takes much longer than executing an instruction, all CPUs contain registers inside to hold key variables and temporary results. Instruction sets often contains instructions to load a word from memory into a register, and store a word from a register into memory. Other instructions combine two operands from registers and/or memory, into a result, such as adding two words and storing the result in a register or in memory.

In addition to the general registers used to hold variables and temporary results, most computers have several special registers that are visible to the programmer. One of these is the **program counter**, which contains the memory address of the next instruction to be fetched. After that instruction has been fetched, the program counter is updated to point to its successor.

Another register is the **stack pointer**, which points to the top of the current stack in memory. The stack contains one frame for each procedure that has been entered but not yet exited. A procedure's stack frame holds those input parameters, local variables, and temporary variables that are not kept in registers.

Yet another register is the **PSW** (**Program Status Word**). This register contains the condition code bits, which are set by comparison instructions, the CPU priority, the mode (user or kernel), and various other control bits. User programs may normally read the entire PSW but typically may write only some of its fields. The PSW plays an important role in system calls and I/O.

The operating system must be fully aware of all the registers. When time multiplexing the CPU, the operating system will often stop the running program to (re)start another one. Every time it stops a running program, the operating system must save all the registers so they can be restored when the program runs later.

In fact, we distinguish between the **architecture** and the **micro-architecture**. The architecture consists of everything that is visible to the software such as the instructions and the registers. The micro-architecture comprises the implementation of the architecture. Here we find data and instruction caches, translation lookaside buffers, branch predictors, the pipelined datapath, and many other elements that should not normally be visible to the operating system or any other software.

To improve performance, CPU designers have long abandoned the simple model of fetching, decoding, and executing one instruction at a time. Many modern CPUs have facilities for executing more than one instruction at the same time. For example, a CPU might have separate fetch, decode, and execute units, so that while it is executing instruction n, it could also be decoding instruction $n + 1$ and fetching instruction $n + 2$. Such an organization is called a **pipeline** and is illustrated in Fig. 1-7(a) for a pipeline with three stages. Longer pipelines are common. In most pipeline designs, once an instruction has been fetched into the pipeline, it must be executed, even if the preceding instruction was a conditional branch that was taken. Pipelines cause compiler writers and operating system writers great headaches because they expose the complexities of the underlying machine to them and they have to deal with them.

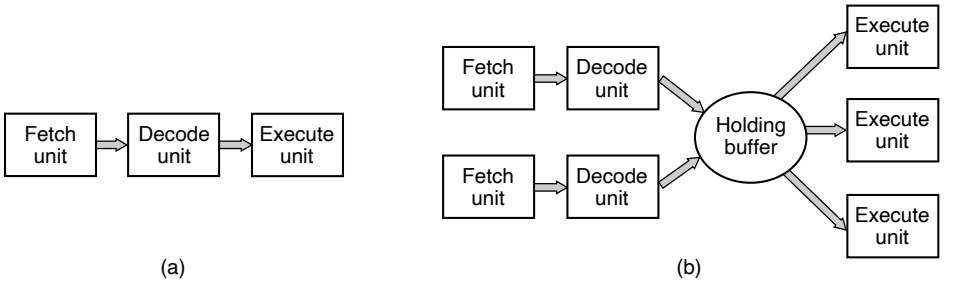

Figure 1-7. (a) A three-stage pipeline. (b) A superscalar CPU.

Even more advanced than a pipeline design is a **superscalar** CPU, shown in Fig. 1-7(b). In this design, multiple execution units are present, for example, one for integer arithmetic, one for floating-point arithmetic, and one for Boolean operations. Two or more instructions are fetched at once, decoded, and dumped into a holding buffer until they can be executed. As soon as an execution unit becomes available, it looks in the holding buffer to see if there is an instruction it can handle, and if so, it removes the instruction from the buffer and executes it. An implication of this design is that program instructions are often executed out of order. For the most part, it is up to the hardware to make sure the result produced is the same one a sequential implementation would have produced, but an annoying amount of the complexity is foisted onto the operating system, as we shall see.

Most CPUs, except very simple ones used in embedded systems, have (at least) two modes, kernel mode and user mode, as mentioned earlier. Usually, a bit in the PSW controls the mode. When running in kernel mode, the CPU can execute every instruction in its instruction set and use every feature of the hardware. On desktop, notebook, and server machines, the operating system normally runs in kernel mode, giving it access to the complete hardware. On most embedded systems, a small piece runs in kernel mode, with the rest of the operating system running in user mode.

User programs always run in user mode, which permits only a subset of the instructions to be executed and a subset of the features to be accessed. Generally, all instructions involving I/O and memory protection are disallowed in user mode. Setting the PSW mode bit to enter kernel mode is also forbidden, of course.

To obtain services from the operating system, a user program must make a **system call**, which traps into the kernel and invokes the operating system. The trap instruction (e.g., syscall on x86-64 processors) switches from user mode to kernel mode and starts the operating system. When the operating system is done, it returns control to the user program at the instruction following the system call. We will explain the details of the system call mechanism later in this chapter. For the time being, think of it as a special kind of procedure call that has the additional property of switching from user mode to kernel mode. As a note on typography, we will use the lower-case Helvetica font to indicate system calls in running text, like this: read.

It is worth noting that computers have traps other than the instruction for executing a system call. Most of the other traps are caused by the hardware to warn of an exceptional situation such as an attempt to divide by 0 or a floating-point underflow. In all cases, the operating system gets control and must decide what to do. Sometimes the program must be terminated with an error. Other times the error can be ignored (an underflowed number can be set to 0). Finally, when the program has announced in advance that it wants to handle certain kinds of conditions, control can be passed back to the program to let it deal with the problem.

Multithreaded and Multicore Chips

Moore's law states that the number of transistors on a chip doubles every 18 months. This "law" is not some kind of law of physics, like conservation of momentum, but is an observation by Intel cofounder Gordon Moore of how fast process engineers at the semiconductor companies are able to shrink their transistors. Without wanting to enter the debate about when it will end and whether or not the exponential is already slowing down some, we simply observe that Moore's law has held for half a century already and is expected to hold for at least a few years more. After that, the number of atoms per transistor will become too small and quantum mechanics will start to play a big role, preventing further shrinkage of transistor sizes. Outwitting quantum mechanics will be quite a challenge.

The abundance of transistors is leading to a problem: what to do with all of them? We saw one approach above: superscalar architectures, with multiple functional units. But as the number of transistors increases, even more is possible. One obvious thing to do is put bigger caches on the CPU chip. That is definitely happening, but eventually the point of diminishing returns will be reached.

The obvious next step is to replicate not only the functional units, but also some of the control logic. The Intel Pentium 4 introduced this property, called **multithreading** or **hyperthreading** (Intel's name for it), to the x86 processor, and several other CPU chips also have it—including the SPARC, the Power5, and some ARM processors. To a first approximation, what it does is allow the CPU to hold the state of two different threads and then switch back and forth on a nanosecond time scale. (A thread is a kind of lightweight process, which, in turn, is a running program; we will get into the details in Chap. 2.) For example, if one of the processes needs to read a word from memory (which takes many clock cycles), a multithreaded CPU can just switch to another thread. Multithreading does not offer true parallelism. Only one process at a time is running, but thread-switching time is reduced to the order of a nanosecond.

Multithreading has implications for the operating system because each thread appears to the operating system as a separate CPU. Consider a system with two actual CPUs, each with two threads. The operating system will see this as four CPUs. If there is only enough work to keep two CPUs busy at a certain point in time, it may inadvertently schedule two threads on the same CPU, with the other CPU completely idle. This is far less efficient than using one thread on each CPU.

Beyond multithreading, many CPU chips now have four, eight, or more complete processors or **cores** on them. The multicore chips of Fig. 1-8 effectively carry four minichips on them, each with its own independent CPU. (The caches will be explained later in the book.) Some models of popular processors like Intel Xeon and AMD Ryzen come with more than 50 cores, but there are also CPUs with core counts in the hundreds. Making use of such a multicore chip will definitely require a multiprocessor operating system.

Incidentally, in terms of sheer numbers, nothing beats a modern **GPU** (**Graphics Processing Unit**). A GPU is a processor with, literally, thousands of tiny cores. They are very good for many small computations done in parallel, like rendering polygons in graphics applications. They are not so good at serial tasks. They are also hard to program. While GPUs can be useful for operating systems (e.g., for encryption or processing of network traffic), it is not likely that much of the operating system itself will run on the GPUs.

1.3.2 Memory

The second major component in any computer is the memory. Ideally, memory should be extremely fast (faster than executing an instruction so that the CPU is not held up by the memory), abundantly large, and dirt cheap. No current technology

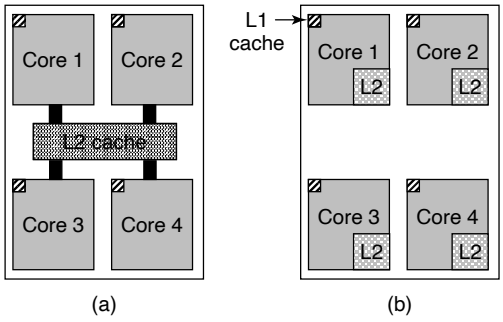

Figure 1-8. (a) A quad-core chip with a shared L2 cache. (b) A quad-core chip with separate L2 caches.

satisfies all of these goals, so a different approach is taken. The memory system is constructed as a hierarchy of layers, as shown in Fig. 1-9, which would be typical for a desktop computer or a server (notebooks use SSDs). The top layers have higher speed, smaller capacity, and greater cost per bit than the lower ones, often by factors of a billion or more.

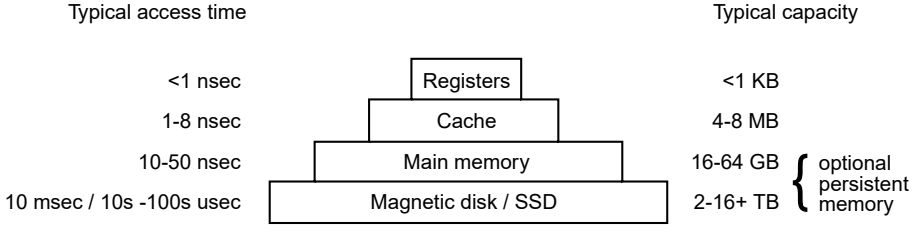

Figure 1-9. A typical memory hierarchy. The numbers are very rough approximations.

The top layer consists of the registers internal to the CPU. They are made of the same material as the CPU and are thus just as fast as the CPU. Consequently, there is no delay in accessing them. The storage capacity available in them is on the order of 32 × 32 bits on a 32-bit CPU and 64 × 64 bits on a 64-bit CPU. Less than 1 KB in both cases. Programs must manage the registers (i.e., decide what to keep in them) themselves, in software.

Next comes the cache memory, which is mostly controlled by the hardware. Main memory is divided up into **cache lines**, typically 64 bytes, with addresses 0 to 63 in cache line 0, 64 to 127 in cache line 1, and so on. The most heavily used cache lines are kept in a high-speed cache located inside or very close to the CPU. When the program needs to read a memory word, the cache hardware checks to see if the line needed is in the cache. If it is, called a **cache hit**, the request is satisfied from the cache and no memory request is sent over the bus to the main memory.

Cache hits normally take only a few clock cycles. Cache misses have to go to memory, with a substantial time penalty of tens to hundreds of cycles. Cache memory is limited in size due to its high cost. Some machines have two or even three levels of cache, each one slower and bigger than the one before it.

Caching plays a major role in many areas of computer science, not just caching lines of RAM. Whenever a resource can be divided into pieces, some of which are used much more heavily than others, caching is often used to improve performance. Operating systems use it all the time. For example, most operating systems keep (pieces of) heavily used files in main memory to avoid having to fetch them from stable storage repeatedly. Similarly, the results of converting long path names like

/home/ast/projects/minix3/src/kernel/clock.c

into a "disk address" for the SSD or disk where the file is located can be cached to avoid repeated lookups. Finally, when the address of a Web page (URL) is converted to a network address (IP address), the result can be cached for future use. Many other uses exist.

In any caching system, several questions come up fairly soon, including:

1. When to put a new item into the cache.
2. Which cache line to put the new item in.
3. Which item to remove from the cache when a slot is needed.
4. Where to put a newly evicted item in the larger memory.

Not every question is relevant to every caching situation. For caching lines of main memory in the CPU cache, a new item will generally be entered on every cache miss. The cache line to use is generally computed by using some of the high-order bits of the memory address referenced. For example, with 4096 cache lines of 64 bytes and 32 bit addresses, bits 6 through 17 might be used to specify the cache line, with bits 0 to 5 the byte within the cache line. In this case, the item to remove is the same one as the new data goes into, but in other systems it might not be. Finally, when a cache line is rewritten to main memory (if it has been modified since it was cached), the place in memory to rewrite it to is uniquely determined by the address in question.

Caches are such a good idea that modern CPUs have two or more of them. The first level or **L1 cache** is always inside the CPU and usually feeds decoded instructions into the CPU's execution engine. Most chips have a second L1 cache for very heavily used data words. The L1 caches are typically 32 KB each. In addition, there is often a second cache, called the **L2 cache**, that holds several megabytes of recently used memory words. The difference between the L1 and L2 caches lies in the timing. Access to the L1 cache is done without any delay, whereas access to the L2 cache involves a delay of several clock cycles.

On multicore chips, the designers have to decide where to place the caches. In Fig. 1-8(a), a single L2 cache is shared by all the cores. In contrast, in Fig. 1-8(b), each core has its own L2 cache. Each strategy has its pros and cons. For example, the shared L2 cache requires a more complicated cache controller but the per-core L2 caches makes keeping the caches consistent more difficult.

Main memory comes next in the hierarchy of Fig. 1-9. This is the workhorse of the memory system. Main memory is usually called **RAM** (**Random Access Memory**). Old-timers sometimes call it **core memory**, because computers in the 1950s and 1960s used tiny magnetizable ferrite cores for main memory. They have been gone for decades but the name persists. Currently, memories are often tens of gigabytes on desktop or server machines. All CPU requests that cannot be satisfied out of the cache go to main memory.

In addition to the main memory, many computers have different kinds of nonvolatile random-access memory. Unlike RAM, nonvolatile memory does not lose its contents when the power is switched off. **ROM** (**Read Only Memory**) is programmed at the factory and cannot be changed afterward. It is fast and inexpensive. On some computers, the bootstrap loader used to start the computer is contained in ROM. **EEPROM** (**Electrically Erasable PROM**) is also nonvolatile, but in contrast to ROM can be erased and rewritten. However, writing it takes orders of magnitude more time than writing RAM, so it is used in the same way ROM is, except that it is now possible to correct bugs in programs by rewriting them in the field. Boot strapping code may also be stored in **Flash memory**, which is similarly nonvolatile, but in contrast to ROM can be erased and rewritten. The boot strapping code is commonly referred to as **BIOS** (Basic Input/Output System). Flash memory is also commonly used as the storage medium in portable electronic devices such as smartphones and in SSDs to serve as a faster alternative to hard disks. Flash memory is intermediate in speed between RAM and disk. Also, unlike disk memory, if it is erased too many times, it wears out. Firmware inside the device tries to mitigate this through load balancing.

Yet another kind of memory is CMOS, which is volatile. Many computers use CMOS memory to hold the current time and date. The CMOS memory and the clock circuit that increments the time in it are powered by a small battery, so the time is correctly updated, even when the computer is unplugged. The CMOS memory can also hold the configuration parameters, such as which drive to boot from. CMOS is used because it draws so little power that the original factory-installed battery often lasts for several years. However, when it begins to fail, the computer can appear to be losing its marbles, forgetting things that it has known for years, like how to boot.

Incidentally, many computers today support a scheme known as **virtual memory**, which we will discuss at some length in Chap. 3. It makes it possible to run programs larger than physical memory by placing them on nonvolatile storage (SSD or disk) and using main memory as a kind of cache for the most heavily executed parts. From time to time, the program will need data that are currently not

in memory. It frees up some memory (e.g., by writing some data that have not been used recently back to SSD or disk) and then loads the new data at this location. Because the physical address for the data and code is now no longer fixed, the scheme remaps memory addresses on the fly to convert the address the program generated to the physical address in RAM where the data are currently located. This mapping is done by a part of the CPU called the **MMU** (**Memory Management Unit**), as shown in Fig. 1-6.

The MMU can have a major impact on performance as every memory access by the program must be remapped using special data structures that are also in memory. In a multiprogramming system, when switching from one program to another, sometimes called a **context switch**, these data structures must change as the mappings differ from process to process. Both the on-the-fly address translation and the context switch can be expensive operations.

1.3.3 Nonvolatile Storage

Next in the hierarchy are magnetic disks (hard disks), solid state drives (SSDs), and persistent memory. Starting with the oldest and slowest, hard disk storage is two orders of magnitude cheaper than RAM per bit and often two orders of magnitude larger as well. The only problem is that the time to randomly access data on it is close to three orders of magnitude slower. The reason is that a disk is a mechanical device, as shown in Fig. 1-10.

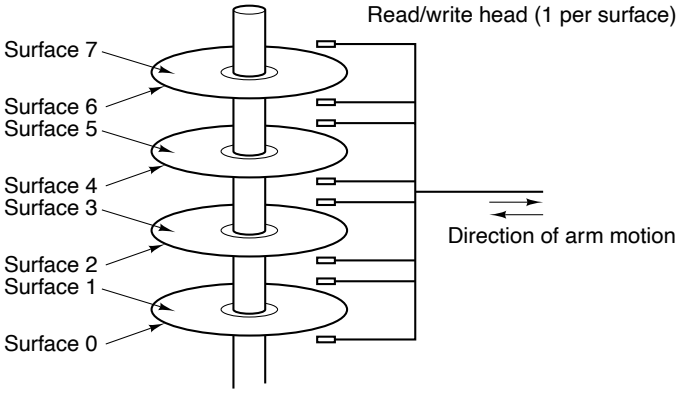

Figure 1-10. Structure of a disk drive.

A disk consists of one or more metal platters that rotate at 5400, 7200, 10,800, 15,000 RPM or more. A mechanical arm pivots over the platters from the corner, similar to the pickup arm on an old 33-RPM phonograph for playing vinyl records. Information is written onto the disk in a series of concentric circles. At any given

arm position, each of the heads can read an annular region known as a **track**. Together, all the tracks for a given arm position form a **cylinder**.

Each track is divided into some number of sectors, typically 512 bytes per sector. On modern disks, the outer cylinders contain more sectors than the inner ones. Moving the arm from one cylinder to the next takes about 1 msec. Moving it to a random cylinder typically takes 5–10 msec, depending on the drive. Once the arm is on the correct track, the drive must wait for the needed sector to rotate under the head, an additional delay of 5–10 msec, depending on the drive's RPM. Once the sector is under the head, reading or writing occurs at a rate of 50 MB/sec on low-end disks to 160–200 MB/sec on faster ones.

Many people also refer to SSDs as disks, even though they are physically not disks at all and do not have platters or moving arms. They store data in electronic (Flash) memory. The only way in which they resemble disks in terms of hardware is that they also store a lot of data which is not lost when the power is off. But from the operating system's point of view, they are somewhat disk-like. SSDs are much more expensive than rotating disks in terms of cost per byte stored, which is why they are not so much used in data centers for bulk storage. However, they are much faster than magnetic disks and since they have no mechanical arm to move, they are better at accessing data at random locations. Reading data from an SSD takes tens of microseconds instead of milliseconds as with hard disks. Writes are more complicated as they require a full data block to be erased first and take more time. But even if a write takes a few hundred microseconds, this is still better than a hard disk's performance.

The youngest and fastest member of the stable storage family is known as **persistent memory**. The best known example is Intel Optane which became available in 2016. In many ways, persistent memory can be seen as an additional layer between SSDs (or hard disks) and memory: it is both fast, only slightly slower than regular RAM, and it holds its content across power cycles. While it can be used to implement really fast SSDs, manufacturers may also attach it directly to the memory bus. In fact, it can be used like normal memory to store an application's data structures, except that the data will still be there when the power goes off. In that case, accessing it requires no special driver and may happen at byte granularity, obviating the need to transfer data in large blocks as in hard disks and SSDs.

1.3.4 I/O Devices

It should now be clear that CPU and memory are not the only resources that the operating system must manage. There are many other ones. Besides disks there many other I/O devices that interact heavily with the operating system. As we saw in Fig. 1-6, I/O devices generally consist of two parts: a controller and the device itself. The controller is a chip (or a set of chips) that physically controls the device. It accepts commands from the operating system, for example, to read data from the device, and carries them out.

In many cases, the actual control of the device is complicated and detailed, so it is the job of the controller to present a simpler (but still very complex) interface to the operating system. For example, a hard disk controller might accept a command to read sector 11,206 from disk 2. The controller then has to convert this linear sector number to a cylinder, sector, and head. This conversion may be complicated by the fact that outer cylinders have more sectors than inner ones and that some bad sectors have been remapped onto other ones. Then the controller has to determine which cylinder the disk arm is on and give it a command to move in or out the requisite number of cylinders. It has to wait until the proper sector has rotated under the head and then start reading and storing the bits as they come off the drive, removing the preamble and computing the checksum. Finally, it has to assemble the incoming bits into words and store them in memory. To do all this work, controllers often contain small embedded computers that are programmed to do their work.

The other piece is the actual device itself. Devices have fairly simple interfaces, both because they cannot do much and to make them standard. The latter is needed so that any SATA disk controller can handle any SATA disk, for example. **SATA** stands for **Serial ATA** and **ATA** in turn stands for **AT Attachment**. In case you are curious what AT stands for, this was IBM's second generation "Personal Computer Advanced Technology" built around the then-extremely-potent 6-MHz 80286 processor that the company introduced in 1984. What we learn from this is that the computer industry has a habit of continuously enhancing existing acronyms with new prefixes and suffixes. We also learned that an adjective like "advanced" should be used with great care, or you will look silly 40 years down the line.

SATA is currently the standard type of hard disk on many computers. Since the actual device interface is hidden behind the controller, all that the operating system sees is the interface to the controller, which may be quite different from the interface to the device.

Because each type of controller is different, different software is needed to control each one. The software that talks to a controller, giving it commands and accepting responses, is called a **device driver**. Each controller manufacturer has to supply a driver for each operating system it supports. Thus a scanner may come with drivers for macOS, Windows 11, and Linux, for example.

To be used, the driver has to be put into the operating system so it can run in kernel mode. Drivers can actually run outside the kernel, and operating systems like Linux and Windows nowadays do offer some support for doing so, but the vast majority of the drivers still run below the kernel boundary. Only very few current systems, such as MINIX 3 run all drivers in user space. Drivers in user space must be allowed to access the device in a controlled way, which is not straightforward without some hardware support.

There are three ways the driver can be put into the kernel. The first way is to relink the kernel with the new driver and then reboot the system. Many older UNIX

systems work like this. The second way is to make an entry in an operating system file telling it that it needs the driver and then reboot the system. At boot time, the operating system goes and finds the drivers it needs and loads them. Older versions of Windows work this way. The third way is for the operating system to be able to accept new drivers while running and install them on the fly without the need to reboot. This way used to be rare but is becoming much more common now. Hot-pluggable devices, such as USB and Thunderbolt devices (discussed below), always need dynamically loaded drivers.

Every controller has a small number of registers that are used to communicate with it. For example, a minimal disk controller might have registers for specifying the disk address, memory address, sector count, and direction (read or write). To activate the controller, the driver gets a command from the operating system, then translates it into the appropriate values to write into the device registers. The collection of all the device registers forms the **I/O port space**, a subject we will come back to in Chap. 5.

On some computers, the device registers are mapped into the operating system's address space (the addresses it can use), so they can be read and written like ordinary memory words. On such computers, no special I/O instructions are required and user programs can be kept away from the hardware by not putting these memory addresses within their reach (e.g., by using base and limit registers). On other computers, the device registers are put in a special I/O port space, with each register having a port address. On these machines, special IN and OUT instructions are available in kernel mode to allow drivers to read and write the registers. The former scheme eliminates the need for special I/O instructions but uses up some of the address space. The latter uses no address space but requires special instructions. Both systems are widely used.

Input and output can be done in three different ways. In the simplest method, a user program issues a system call, which the kernel then translates into a procedure call to the appropriate driver. The driver then starts the I/O and sits in a tight loop continuously polling the device to see if it is done (usually there is some bit that indicates that the device is still busy). When the I/O has completed, the driver puts the data (if any) where they are needed and returns. The operating system then returns control to the caller. This method is called **busy waiting** and has the disadvantage of tying up the CPU polling the device until it is finished.

The second method is for the driver to start the device and ask it to give an interrupt when it is finished. At that point, the driver returns. The operating system then blocks the caller if need be and looks for other work to do. When the controller detects the end of the transfer, it generates an **interrupt** to signal completion.

Interrupts are very important in operating systems, so let us examine the idea more closely. In Fig. 1-11(a) we see a three-step process for I/O. In step 1, the driver tells the controller what to do by writing into its device registers. The controller then starts the device. When the controller has finished reading or writing

the number of bytes it has been instructed to transfer, it signals the interrupt controller chip using certain bus lines in step 2. If the interrupt controller is ready to accept the interrupt (which it may not be if it is busy handling a higher-priority interrupt), it asserts a pin on the CPU chip telling it, in step 3. In step 4, the interrupt controller puts the number of the device on the bus so the CPU can read it and know which device has just finished (many devices may be running at the same time).

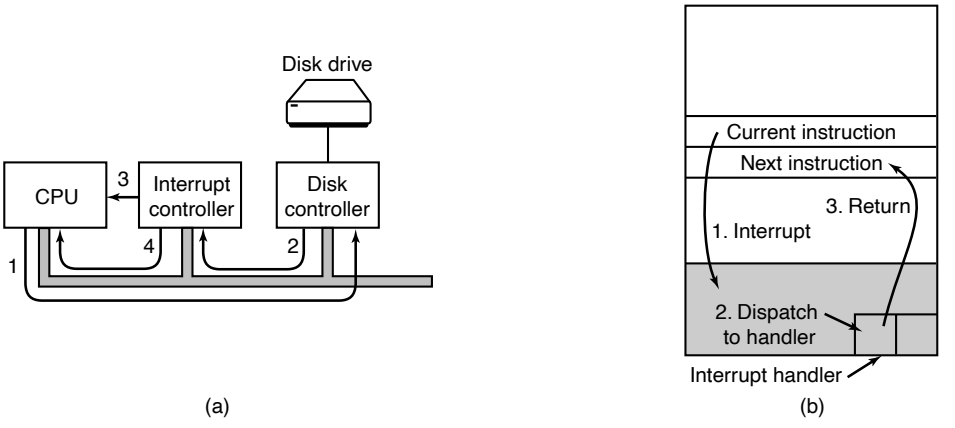

Figure 1-11. (a) The steps in starting an I/O device and getting an interrupt. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

Once the CPU has decided to take the interrupt, the program counter and PSW are typically then pushed onto the current stack and the CPU switched into kernel mode. The device number may be used as an index into part of memory to find the address of the interrupt handler for this device. This part of memory is called the **interrupt vector table**. Once the interrupt handler (part of the driver for the interrupting device) has started, it saves the stacked program counter, PSW, and other registers (typically in the process table). Then it queries the device to learn its status. When the handler is all finished, it restores the context and returns to the previously running user program to the first instruction that was not yet executed. These steps are shown in Fig. 1-11(b). We will discuss interrupt vectors further in in the next chapter.

The third method for doing I/O makes use of special hardware: a **DMA** (**Direct Memory Access**) chip that can control the flow of bits between memory and some controller without constant CPU intervention. The CPU sets up the DMA chip, telling it how many bytes to transfer, the device and memory addresses involved, and the direction, and lets it go. When the DMA chip is done, it causes an interrupt, which is handled as described above. DMA and I/O hardware in general will be discussed in more detail in Chap. 5.

1.3.5 Buses

The organization of Fig. 1-6 was used on minicomputers for years and also on the original IBM PC. However, as processors and memories got faster, the ability of a single bus (and certainly the IBM PC bus) to handle all the traffic was strained to the breaking point. Something had to give. As a result, additional buses were added, both for faster I/O devices and for CPU-to-memory traffic. As a consequence of this evolution, a large x86 system currently looks something like Fig. 1-12.

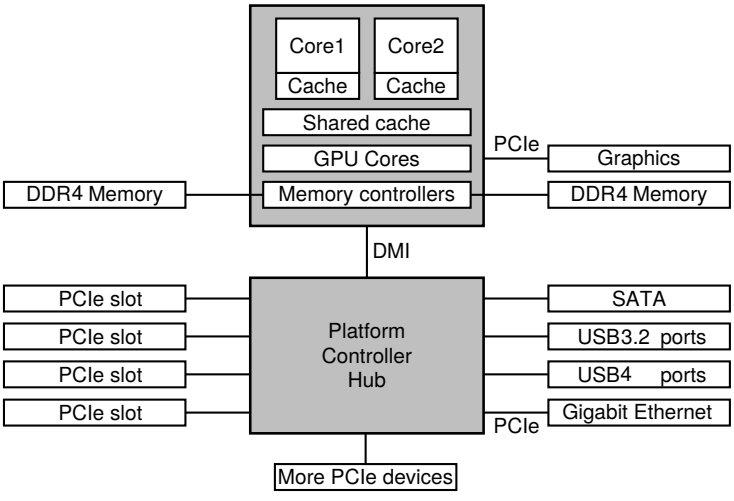

Figure 1-12. The structure of a large x86 system.

This system has many buses (e.g., cache, memory, PCIe, PCI, USB, SATA, and DMI), each with a different transfer rate and function. The operating system must be aware of all of them for configuration and management. The main bus is the **PCIe** (**Peripheral Component Interconnect Express**) bus.

The PCIe bus was invented by Intel as a successor to the older **PCI** bus, which in turn was a replacement for the original **ISA** (**Industry Standard Architecture**) bus. Capable of transferring tens of gigabits per second, PCIe is much faster than its predecessors. It is also very different in nature. Up to its creation in 2004, most buses were parallel and shared. A **shared bus architecture** means that multiple devices use the same wires to transfer data. Thus, when multiple devices have data to send, you need an arbiter to determine who can use the bus. In contrast, PCIe makes use of dedicated, point-to-point connections. A **parallel bus architecture** as used in traditional PCI means that you send each word of data over multiple wires. For instance, in regular PCI buses, a single 32-bit number is sent over 32 parallel wires. In contrast to this, PCIe uses a **serial bus architecture** and sends all bits in

a message through a single connection, known as a **lane**, much like a network packet. This is much simpler because you do not have to ensure that all 32 bits arrive at the destination at exactly the same time. Parallelism is still used, because you can have multiple lanes in parallel. For instance, we may use 32 lanes to carry 32 messages in parallel. As the speed of peripheral devices like network cards and graphics adapters increases rapidly, the PCIe standard is upgraded every 3–5 years. For instance, 16 lanes of PCIe 4.0 offer 256 gigabits per second. Upgrading to PCIe 5.0 will give you twice that speed and PCIe 6.0 will double that again. Meanwhile, we still have legacy devices for the older PCI standard. These devices can be hooked up to a separate hub processor.

In this configuration, the CPU talks to memory over a fast DDR4 bus, to an external graphics device over PCIe and to all other devices via a hub over a **DMI** (**Direct Media Interface**) bus. The hub in turn connects all the other devices, using the Universal Serial Bus to talk to USB devices, the SATA bus to interact with hard disks and DVD drives, and PCIe to transfer Ethernet frames. We have already mentioned the older PCI devices that use a traditional PCI bus.

Moreover, each of the cores has a dedicated cache and a much larger cache that is shared between them. Each of these caches introduces yet another bus.

The **USB** (**Universal Serial Bus**) was invented to attach all the slow I/O devices, such as the keyboard and mouse, to the computer. However, calling a modern USB4 device humming along at 40 Gbps "slow" may not come naturally for the generation that grew up with 8-Mbps ISA as the main bus in the first IBM PCs. USB uses a small connector with 4–11 wires (depending on the version), some of which supply electrical power to the USB devices or connect to ground. USB is a centralized bus in which a root device polls all the I/O devices every 1 msec to see if they have any traffic. USB 1.0 could handle an aggregate load of 12 Mbps, USB 2.0 increased the speed to 480 Mbps, USB 3.0 to 5 Gbps, USB 3.2 to 20 Gbps and USB 4 will double that. Any USB device can be connected to a computer and it will function immediately, without requiring a reboot, something pre-USB devices required, much to the consternation of a generation of frustrated users.

1.3.6 Booting the Computer

Very briefly, the boot process is as follows. Every PC contains a motherboard, which contains the CPU, slots for memory chips, and sockets for PCIe (or other) plug-in cards. On the motherboard a small amount of flash holds a program called the system firmware, which we commonly still refer to as the **BIOS** (**Basic Input Output System**), even though strictly speaking the name BIOS applies only to the firmware in somewhat older IBM PC compatible machines. Booting using the original BIOS was slow, architecture-dependent, and limited to smaller SSDs and disks (up to 2 TB). It was also very easy to understand. When Intel proposed what would become **UEFI** (**Unified Extensible Firmware Interface**) as a replacement,

it remedied all these issues: UEFI allows for fast booting, different architectures, and storage sizes up to 8 ZiB, or 8×2^{70} bytes. It is also so complex that trying to understand it fully has sucked the happiness out of many a life. In this chapter, we will cover both old and new style BIOS firmware, but only the essentials.

After we press the power button, the motherboard waits for the signal that the power supply has stabilized. When the CPU starts executing, it fetches code from a hard-coded physical address (known as the reset vector) that is mapped to the flash memory. In other words, it executes code from the BIOS which detects and initializes various resources, such as RAM, the Platform Controller Hub (see Fig. Fig. 1-12), and interrupt controllers. In addition, it scans the PCI and/or PCIe buses to detect and initialize all devices attached to them. If the devices present are different from when the system was last booted, it also configures the new devices. Finally, it sets up the runtime firmware which offers critical services (including low-level I/O) that can be used by the system after booting.

Next, it is time to move to the next stage of the booting process. In systems using the old-style BIOS, this was all very straightforward. The BIOS would determine the boot device by trying a list of devices stored in the CMOS memory. The user can change this list by entering a BIOS configuration program just after booting. For instance, you may ask the system to attempt to boot from a USB drive, if one is present. If that fails, the system boots from the hard disk or SSD. The first sector from the boot device is read into memory and executed. This sector, known as the **MBR** (**Master Boot Record**), contains a program that normally examines the partition table at the end of the boot sector to determine which partition is active. A partition is a distinct region on the storage device that may for instance contain its own file systems. Then a secondary boot loader is read in from that partition. This loader reads in the operating system from the active partition and starts it. The operating system then queries the BIOS to get the configuration information. For each device, it checks to see if it has the device driver. If not, it asks the user to install it, for instance by downloading it from the Internet. Once it has all the device drivers, the operating system loads them into the kernel. Then it initializes its tables, creates whatever background processes are needed, and starts up a login program or GUI.

With UEFI, things are different. First, it no longer relies on a Master Boot Record residing in the first sector of the boot device, but it looks for the location of the partition table in the second sector of the device. This **GPT** (**GUID Partition Table**) contains information about the location of the various partitions on the SSD or disk. Second, the BIOS itself has enough functionality to read file systems of specific types. According to the UEFI standard, it should support at least the FAT-12, FAT-16, and FAT-32 types. One such file system is placed in a special partition, known as the EFI system partition (ESP). Rather than a single magic boot sector, the boot process can now use a proper file system containing programs, configuration files, and anything else that may be useful during boot. Moreover, UEFI expects the firmware to be able to execute programs in a specific format,

called **PE** (**Portable Executable**). As you can see, the BIOS under UEFI very much looks like a little operating system itself which understands partitions, file systems, executables, etc. It even has a shell with some standard commands.

The boot code still needs to pick one of the bootloader programs to load Linux or Windows, or whatever operating system, but there may be many partitions with operating systems and given so much choice, which one should it pick? This is decided by the UEFI boot manager, which you can think of as a boot menu with different entries and a configurable order in which to try the different boot options. Changing the menu and the default bootloader is very easy and can be done from within the currently executing operating system. As before, the bootloader will continue loading the operating system of choice.

This is by no means the full story. UEFI is very flexible and highly standardized, and contains many advanced features. However, this is enough for now. In Chap. 9, we will pick up UEFI again when we discuss an interesting feature known as **Secure Boot**, which allows a user to be sure that the operating system is booted as intended and with the correct software.

1.4 THE OPERATING SYSTEM ZOO

Operating systems have been around now for over half a century. During this time, quite a variety of them have been developed, not all of them widely known. In this section, we will briefly touch upon nine of them. We will come back to some of these different kinds of systems later in the book.

1.4.1 Mainframe Operating Systems

At the high end are the operating systems for mainframes, those room-sized computers still found in major corporate data centers. These computers differ from personal computers in terms of their I/O capacity. A mainframe with 1000 hard disks and many terabytes of data are not unusual; a personal computer with these specifications would be the envy of its friends. Mainframes are also making something of a comeback as high-end servers for large-scale electronic commerce sites, banking, airline reservations, and servers for business-to-business transactions.

The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O. They typically offer three kinds of services: batch, transaction processing, and timesharing. A batch system is one that processes routine jobs without any interactive user present. Claims processing in an insurance company or sales reporting for a chain of stores is typically done in batch mode. Transaction-processing systems handle large numbers of small requests, for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second. Timesharing systems allow multiple remote users to run jobs on the computer at once, such as querying a big database. These functions are closely

related; mainframe operating systems often perform all of them. An example mainframe operating system is Z/OS, the successor of OS/390, which in turn was a direct descendant of OS/360. However, mainframe operating systems are gradually being replaced by UNIX variants such as Linux.

1.4.2 Server Operating Systems

One level down are the server operating systems. They run on servers, which are either very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, database service, or Web service. Internet providers run many server machines to support their customers and Websites use servers to store the Web pages and handle the incoming requests. Typical server operating systems are Linux, FreeBSD, Solaris, and the Windows Server family.

1.4.3 Personal Computer Operating Systems

The next category is the personal computer operating system. Modern ones all support multiprogramming, often with dozens of programs started up at boot time, and multiprocessor architectures. Their job is to provide good support to a single user. They are widely used for word processing, spreadsheets, games, and Internet access. Common examples are Windows 11, macOS, Linux, and FreeBSD. Personal computer operating systems are so widely known that probably little introduction is needed. In fact, many people are not even aware that other kinds exist.

1.4.4 Smartphone and Handheld Computer Operating Systems

Continuing on down to smaller and smaller systems, we come to tablets (like Apple's iPad), smartphones and other handheld computers. A handheld computer, originally known as a **PDA** (**Personal Digital Assistant**), is a small computer that can be held in your hand during operation. Smartphones and tablets are the best-known examples. As we have already seen, this market is currently dominated by Google's Android and Apple's iOS. Most of these devices boast multicore CPUs, GPS, cameras and other sensors, copious amounts of memory, and sophisticated operating systems. Moreover, all of them have more third-party applications (**apps**) than you can shake a (USB) stick at. Google has over 3 million Android apps in the Play Store and Apple has over 2 million in the App Store.

1.4.5 The Internet of Things and Embedded Operating Systems

The **IOT** (**Internet of Things**) comprises all the billions of physical objects with sensors and actuators that are increasingly connected to the network, such as fridges, thermostats, security camera's motion sensors, and so on. All of these

devices contain small computers and most of them run small operating systems. In addition, we may have even more embedded systems controlling devices that are not connected to a network at all. Examples include traditional microwave ovens and washing machines. Such systems do not accept user-installed software, so the main property which distinguishes such embedded systems from the computers we discussed earlier is the certainty that no untrusted software will ever run on it. Few microwave ovens allow you to download and run new applications—all the software is in ROM. However, even that is changing. Some high-end cameras have their own app stores that allow users to install custom apps for in-camera editing, multiple exposures, different focus algorithms, different image compression algorithms, and more.

Nevertheless, for most embedded systems there is no need for protection between applications, leading to design simplification. Conversely, such operating systems may provide more support for functionality such as real-time scheduling or low-power networking which are important in many embedded systems. Systems such as Embedded Linux, QNX, and VxWorks are popular in this domain. For severely resource-constrained devices, the operating system should be able to run in just a few kilobytes. For instance, RIOT, an open source operating system for IoT devices, can run in less 10 KB and support systems that range from 8-bit microcontrollers to general-purpose 32-bit CPUs. The TinyOS operating similarly offers a very small footprint, making it popular in sensor nodes.

1.4.6 Real-Time Operating Systems

Real-time systems are characterized by having time as a key parameter. For example, in industrial process-control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. Often there are hard deadlines that must be met. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time. If, for example, a welding robot welds too early or too late, the car will be ruined. If the action absolutely *must* occur at a certain moment (or within a certain range), we have a **hard real-time system**. Many of these are found in industrial process control, avionics, military, and similar application areas. These systems must provide absolute guarantees that a certain action will occur by a certain time.

A **soft real-time system** is one where missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage. Digital audio or multimedia systems fall in this category. Smartphones are also soft real-time systems.

Since meeting deadlines is crucial in (hard) real-time systems, sometimes the operating system is simply a library linked in with the application programs, with everything tightly coupled and no protection between parts of the system. An example of this type of real-time system is eCos.

We should emphasize that the categories of IoT, embedded, real-time and even handheld systems overlap considerably. Many of them have at least some soft real-time aspects. The embedded and real-time systems run only software put in by the system designers; users cannot add their own software, which makes protection easier.

1.4.7 Smart Card Operating Systems

The smallest operating systems run on smart cards, which are credit-card-sized devices containing a CPU. They have very severe processing power and memory constraints. Some are powered by contacts in the reader into which they are inserted, while contactless smart cards are inductively powered (which greatly limits what they can do.) Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions. Often these are proprietary systems.

Some smart cards are Java oriented. This means that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

1.5 OPERATING SYSTEM CONCEPTS

Most operating systems provide certain basic concepts and abstractions such as processes, address spaces, and files that are central to understanding them. In the following sections, we will look at some of these basic concepts ever so briefly, as an introduction. We will come back to each of them in great detail later in this book. To illustrate these concepts we will, from time to time, use examples, generally drawn from UNIX. Similar examples typically exist in other systems as well, however, and we will study some of them later.

1.5.1 Processes

A key concept in all operating systems is the **process**. A process is basically a program in execution. Associated with each process is its **address space**, a list of memory locations from 0 to some maximum, which the process can read and write. The address space contains the executable program, the program's data, and its stack. Also associated with each process is a set of resources, commonly including registers (including the program counter and stack pointer), a list of open files,

outstanding alarms, lists of related processes, and all the other information needed to run the program. A process is fundamentally a container that holds all the information needed to run a program.

We will come back to the process concept in much more detail in Chap. 2. For the time being, the easiest way to get a good intuitive feel for a process is to think about a multiprogramming system. The user may have started a video editing program and instructed it to convert a 2-hour video to a certain format (something that can take hours) and then gone off to surf the Web. Meanwhile, a background process that wakes up periodically to check for incoming email may have started running. Thus, we have (at least) three active processes: the video editor, the Web browser, and the email receiver. Periodically, the operating system decides to stop running one process and start running another, perhaps because the first one has used up more than its share of CPU time in the past second or two.

When a process is suspended temporarily like this, it must later be restarted in exactly the same state it had when it was stopped. This means that all information about the process must be explicitly saved somewhere during the suspension. For example, the process may have several files open for reading at once. Associated with each of these files is a pointer giving the current position (i.e., the number of the byte or record to be read next). When a process is temporarily suspended, all these pointers must be saved so that a read call executed after the process is restarted will read the proper data. In many operating systems, all the information about each process, other than the contents of its own address space, is stored in an operating system table called the **process table**, which is an array of structures, one for each process currently in existence.

Thus, a (suspended) process consists of its address space, usually called the **core image** (in honor of the magnetic core memories used in days of yore), and its process table entry, which contains the contents of its registers and many other items needed to restart the process later.

The key process-management system calls are those dealing with the creation and termination of processes. Consider a typical example. A process called the **command interpreter** or (i.e., shell) reads commands from a terminal. The user has just typed a command requesting that a program be compiled. The shell must now create a new process that will run the compiler. When that process has finished the compilation, it executes a system call to terminate itself.

If a process can create one or more other processes (referred to as **child processes**) and these processes in turn can create child processes, we quickly arrive at the process tree structure of Fig. 1-13. Related processes that are cooperating to get some job done often need to communicate with one another and synchronize their activities. This communication is called **interprocess communication**, and will be addressed in detail in Chap. 2.

Other process system calls are available to request more memory (or release unused memory that is not needed anymore), wait for a child process to terminate, and overlay its program with a different one.

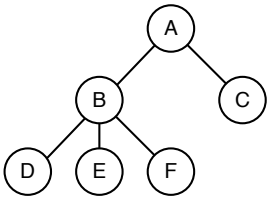

Figure 1-13. A process tree. Process *A* created two child processes, *B* and *C*. Process *B* created three child processes, *D*, *E*, and *F*.

Occasionally, there is a need to convey information to a running process that is not sitting around waiting for this information. For example, a process that is communicating with another process on a different computer does so by sending messages to the remote process over a computer network. To guard against the possibility that a message or its reply is lost, the sender may request that its own operating system notify it after a specified number of seconds, so that it can retransmit the message if no acknowledgement has been received yet. After setting this timer, the program may continue doing other work.

When the specified number of seconds has elapsed, the operating system sends an **alarm signal** to the process. The signal causes the process to temporarily suspend whatever it was doing, save its registers on the stack, and start running a special signal-handling procedure, for example, to retransmit a presumably lost message. When the signal handler is done, the running process is restarted in the state it was in just before the signal. Signals are the software analog of hardware interrupts and can be generated by a variety of causes in addition to timers expiring. Many traps detected by hardware, such as executing an illegal instruction or using an invalid address, are also converted into signals to the guilty process.

Each person authorized to use a system is assigned a **UID** (**User IDentification**) by the system administrator. Every process started has the UID of the person who started it. On UNIX, a child process has the same UID as its parent. Users can be members of groups, each of which has a **GID** (**Group IDentification**).

One UID, called the **superuser** or **root** (in UNIX), or **Administrator** (in Windows), has special power and may override many of the protection rules. In large installations, only the system administrator knows the superuser password, but many of the ordinary users (especially students) devote considerable effort seeking flaws in the system that allow them to become superuser without the password.

We will study processes and interprocess communication in Chap. 2.

1.5.2 Address Spaces

Every computer has some main memory that it uses to hold executing programs. In a very simple operating system, only one program at a time is in memory. To run a second program, the first one has to be removed and the second one placed in memory. This is known as swapping.

More sophisticated operating systems allow multiple programs to be in memory at the same time. To keep them from interfering with one another (and with the operating system), some kind of protection mechanism is needed. While the hardware must provide this mechanism, it is the operating system that controls it.

The above viewpoint is concerned with managing and protecting the computer's main memory. A different, but equally important, memory-related issue is managing the address space of the processes. Normally, each process has some set of addresses it can use, typically running from 0 up to some maximum. In the simplest case, the maximum amount of address space a process has is less than the main memory. In this way, a process can fill up its address space and there will be enough room in main memory to hold it all.

However, on many computers addresses are 32 or 64 bits, giving an address space of 2^{32} or 2^{64} bytes, respectively. What happens if a process has more address space than the computer has main memory and the process wants to use it all? In the first computers, such a process was just out of luck. Nowadays, a technique called virtual memory exists, as mentioned earlier, in which the operating system keeps part of the address space in main memory and part on SSD or disk and shuttles pieces back and forth between them as needed. In essence, the operating system creates the abstraction of an address space as the set of addresses a process may reference. The address space is decoupled from the machine's physical memory and may be either larger or smaller than the physical memory. Management of address spaces and physical memory forms an important part of what an operating system does, so all of Chap. 3 is devoted to this topic.

1.5.3 Files

Another key concept supported by virtually all operating systems is the file system. As noted before, a major function of the operating system is to hide the peculiarities of the SSDs, disks, and other I/O devices and present the programmer with a nice, clean abstract model of device-independent files. System calls are obviously needed to create files, remove files, read files, and write files. Before a file can be read, it must be located on the storage device and opened, and after being read it should be closed, so calls are provided to do these things.

To provide a place to keep files, most PC operating systems have the concept of a **directory**, sometimes called a **folder** or a **map**, as a way of grouping files together. A student, for example, might have one directory for each course she is taking (for the programs needed for that course), another directory her email, and still another directory for her home page on the Web. System calls are then needed to create and remove directories. Calls are also provided to put an existing file in a directory and to remove a file from a directory. Directory entries may be either files or other directories, giving rise to a hierarchy—the file system—as shown in Fig. 1-14. Just like many other innovations in operating systems, hierarchical file systems were pioneered by Multics.

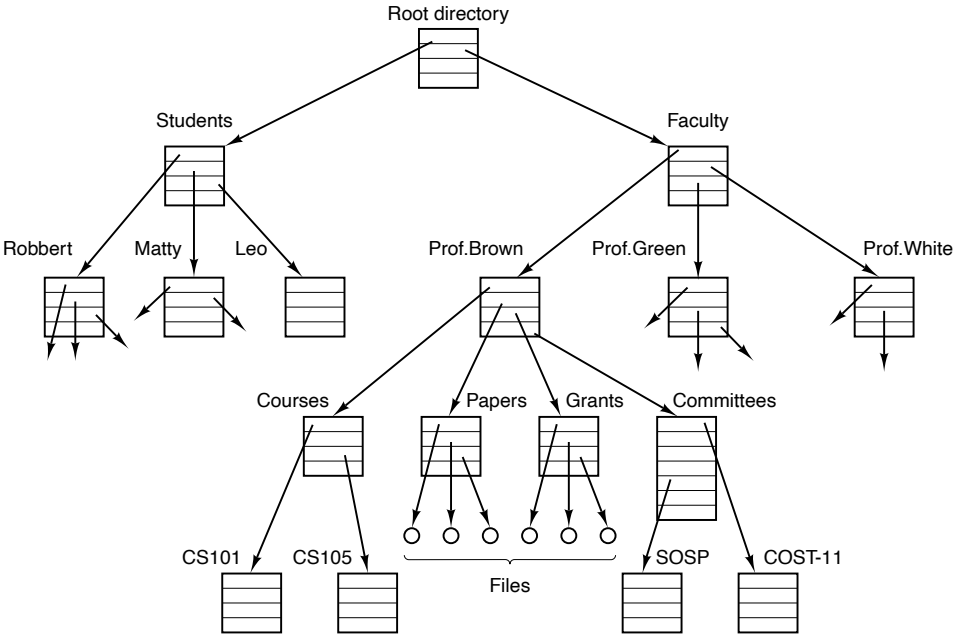

Figure 1-14. A file system for a university department.

The process and file hierarchies both are organized as trees, but the similarity stops there. Process hierarchies usually are not very deep (more than five levels is unusual), whereas file hierarchies are commonly six, seven, or even more levels deep. Process hierarchies are typically more short-lived than directory hierarchies which may exist for years. Ownership and protection also differ for processes and files. Typically, only a parent process may control or even access a child process, but mechanisms nearly always exist to allow files and directories to be read by a wider group than just the owner.

Every file within the directory hierarchy can be specified by giving its **path name** from the top of the directory hierarchy, the **root directory**. Such absolute path names consist of the list of directories that must be traversed from the root directory to get to the file, with slashes separating the components. In Fig. 1-14, the path for file *CS101* is */Faculty/Prof.Brown/Courses/CS101*. The leading slash indicates that the path is absolute, that is, starting at the root directory. As an aside, in Windows, the backslash (\) character is used as the separator instead of the slash (/) character (for historical reasons), so the file path given above would be written as *\Faculty\Prof.Brown\Courses\CS101*. Throughout this book, we will generally use the UNIX convention for paths.

At every instant, each process has a current **working directory**, in which path names not beginning with a slash are looked for. For example, in Fig. 1-14, if

/Faculty/Prof.Brown were the working directory, use of the path *Courses/CS101* would yield the same file as the absolute path name given above. Processes can change their working directory by issuing a system call specifying the new working directory.

Before a file can be read or written, it must be opened, at which time the permissions are checked. If the access is permitted, the system returns a small integer called a **file descriptor** to use in subsequent operations. If the access is prohibited, an error code is returned.

Another important concept in UNIX is the mounted file system. Most desktop computers and notebooks have one or more regular USB ports into which USB memory sticks (really, flash drives) can be plugged, or USB-C ports which can be used to connect external SSDs and hard disks. Some computers have optical drives for Blu-ray disks and you can ask old-timers for war stories about DVDs, CD-ROMs, and floppy disks. To provide an elegant way to deal with these removable media UNIX allows the file system on these separate storage devices to be attached to the main tree. Consider the situation of Fig. 1-15(a). Before the mount call, the **root file system**, on the hard disk, and a second file system, on USB drive, are separate and unrelated. The USB drive may even be formatted with, say, FAT-32 and the hard disk with, say, ext4.

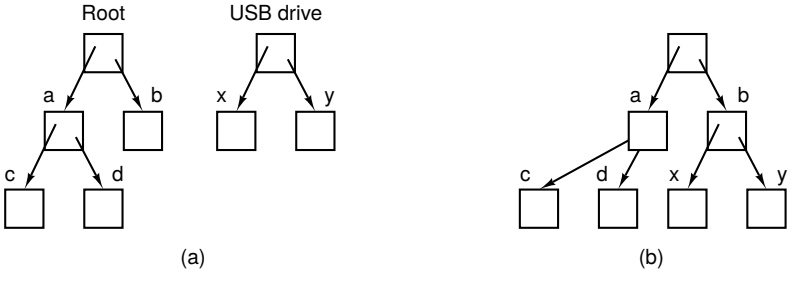

Figure 1-15. (a) Before mounting, the files on the USB drive are not accessible. (b) After mounting, they are part of the file hierarchy.

Unfortunately, if the current directory of the shell is in the hard disk file system, the file system on the USB drive cannot be used, because there is no way to specify path names on it. UNIX does not allow path names to be prefixed by a drive name or number; that would be precisely the kind of device dependence that operating systems ought to eliminate. Instead, the mount system call allows the file system on the USB drive to be attached to the root file system wherever the program wants it to be. In Fig. 1-15(b) the file system on the USB drive has been mounted on directory *b*, thus allowing access to files */b/x* and */b/y*. If directory *b* had contained any files, they would not be accessible while the USB drive was mounted, since */b* would now refer to the root directory of the USB drive. (Not being able to access these files is not as serious as it at first seems: file systems are

nearly always mounted on empty directories.) If a system contains multiple hard disks, they can all be mounted into a single tree as well.

Another important concept in UNIX is the **special file**. Special files are provided in order to make I/O devices look like files. That way, they can be read and written using the same system calls as are used for reading and writing files. Two kinds of special files exist: **block special files** and **character special files**. Block special files are used to model devices that consist of a collection of randomly addressable blocks, such as SSDs and disks. By opening a block special file and reading, say, block 4, a program can directly access the fourth block on the device, without regard to the structure of the file system contained on it. Similarly, character special files are used to model printers, keyboards, mice, and other devices that accept or output a character stream. By convention, the special files are kept in the */dev* directory. For example, */dev/lp* might be the printer (once called the line printer).

The last feature we will discuss in this overview relates to both processes and files: pipes. A **pipe** is a sort of pseudofile that can be used to connect two processes, as shown in Fig. 1-16. If processes *A* and *B* wish to talk using a pipe, they must set it up in advance. When process *A* wants to send data to process *B*, it writes on the pipe as though it were an output file. In fact, the implementation of a pipe is very much like that of a file. Process *B* can read the data by reading from the pipe as though it were an input file. Thus, communication between processes in UNIX looks very much like ordinary file reads and writes. Stronger yet, the only way a process can discover that the output file it is writing on is not really a file, but a pipe, is by making a special system call. File systems are very important. We will have much more to say about them in Chap. 4 and also in Chaps. 10 and 11.

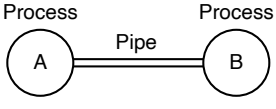

Figure 1-16. Two processes connected by a pipe.

1.5.4 Input/Output

All computers have physical devices for acquiring input and producing output. After all, what good would a computer be if the users could not tell it what to do and could not get the results after it did the work requested? Many kinds of input and output devices exist, including keyboards, monitors, printers, and so on. It is up to the operating system to manage these devices.

Consequently, every operating system has an I/O subsystem for managing its I/O devices. Some of the I/O software is device independent, that is, it applies to many or all I/O devices equally well. Other parts of it, such as device drivers, are specific to particular I/O devices. In Chap. 5, we will have a look at I/O software.

1.5.5 Protection

Computers contain large amounts of information that users often want to protect and keep confidential. This information may include email, business plans, tax returns, and much more. It is up to the operating system to manage the system security so that files, for example, are accessible only to authorized users.

As a simple example, just to get an idea of how security can work, consider UNIX. Files in UNIX are protected by assigning each one a 9-bit binary protection code. The protection code consists of three 3-bit fields, one for the owner, one for other members of the owner's group (users are divided into groups by the system administrator), and one for everyone else. Each field has a bit for read access, a bit for write access, and a bit for execute access. These 3 bits are known as the **rwx bits**. For example, the protection code *rwxr-x--x* means that the owner can **r**ead, **w**rite, or e**x**ecute the file, other group members can read or execute (but not write) the file, and everyone else can execute (but not read or write) the file. For a directory, *x* indicates search permission. A dash means that the corresponding permission is absent.

In addition to file protection, there are many other security issues. Protecting the system from unwanted intruders, both human and nonhuman (e.g., viruses), is one of them. We will look at various security issues in Chap. 9.

1.5.6 The Shell

The operating system is the code that carries out the system calls. Editors, compilers, assemblers, linkers, utility programs, and command interpreters definitely are not part of the operating system, even though they are important and useful. At the risk of confusing things somewhat, in this section we will look briefly at the UNIX command interpreter, the shell. Although it is not part of the operating system, it makes heavy use of many operating system features and thus serves as a good example of how the system calls are used. It is also the main interface between a user sitting at his terminal and the operating system, unless the user is using a graphical user interface. Many shells exist, including *sh*, *csh*, *ksh*, *bash*, and *zsh*. All of them support the functionality described below, which derives from the original shell (*sh*).

When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the **prompt**, a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types

```
date
```

for example, the shell creates a child process and runs the *date* program as the child. While the child process is running, the shell waits for it to finish. When the child finishes, the shell types the prompt again and tries to read the next input line.

The user can specify that standard output be redirected to a file, for example,

```
date >file
```

Similarly, standard input can be redirected, as in

```
sort <file1 >file2
```

which invokes the sort program with input taken from *file1* and output sent to *file2*.

The output of one program can be used as the input for another program by connecting them with a pipe. Thus

```
cat file1 file2 file3 | sort >/dev/lp
```

invokes the *cat* program to con*cat*enate three files and send the output to *sort* to arrange all the lines in alphabetical order. The output of *sort* is redirected to the file */dev/lp*, typically the printer.

If a user puts an ampersand after a command, the shell does not wait for it to complete. Instead it just gives a prompt immediately. Consequently,

```
cat file1 file2 file3 | sort >/dev/lp &
```

starts up the sort as a background job, allowing the user to continue working normally while the sort is going on. The shell has a number of other interesting features, which we do not have space to discuss here. Most books on UNIX discuss the shell at some length (e.g., Kochan and Wood, 2016; and Shotts, 2019).

Most personal computers these days use a GUI. In fact, the GUI is just a program running on top of the operating system, like a shell. In Linux systems, this fact is made obvious because the user has a choice of multiple GUIs: Gnome, KDE or even none at all (using a terminal window on X11). In Windows, replacing the standard GUI desktop is not typically done.

1.5.7 Ontogeny Recapitulates Phylogeny

After Charles Darwin's book *On the Origin of the Species* was published, the German zoologist Ernst Haeckel stated that "ontogeny recapitulates phylogeny." By this he meant that the development of an embryo (ontogeny) repeats (i.e., recapitulates) the evolution of the species (phylogeny). In other words, after fertilization, a human egg goes through stages of being a fish, a pig, and so on before turning into a human baby. Modern biologists regard this as a gross simplification, but it still has a kernel of truth in it.

Something vaguely analogous has happened in the computer industry. Each new species (mainframe, minicomputer, personal computer, handheld, embedded computer, smart card, etc.) seems to go through the development that its ancestors did, both in hardware and in software. We often forget that much of what happens in the computer business and a lot of other fields is technology driven. The reason the ancient Romans lacked automobiles is not that they enjoyed walking so much.

It is because they did not know how to build them. Personal computers exist *not* because millions of people have a centuries-old pent-up desire to own a computer, but because it is now possible to manufacture them cheaply. We often forget how much technology affects our view of systems and it is worth reflecting on this point from time to time.

In particular, it frequently happens that a change in technology renders some idea obsolete and it quickly vanishes. However, another change in technology could revive it again. This is especially true when the change has to do with the relative performance of different parts of the system. For instance, when CPUs became much faster than memories, caches became important to speed up the "slow" memory. If new memory technology someday makes memories much faster than CPUs, caches will vanish. And if a new CPU technology makes them faster than memories again, caches will reappear. In biology, extinction is forever, but in computer science, it is sometimes only for a few years.

As a consequence of this impermanence, in this book we will from time to time look at "obsolete" concepts, that is, ideas that are not optimal with current technology. However, changes in the technology may bring back some of the so-called "obsolete concepts." For this reason, it is important to understand why a concept is obsolete and what changes in the environment might bring it back again.

To make this point clearer, let us consider a simple example. Early computers had hardwired instruction sets. The instructions were executed directly by hardware and could not be changed. Then came microprogramming (first introduced on a large scale with the IBM 360), in which an underlying interpreter carried out the "hardware instructions" in software. Hardwired execution became obsolete. It was not flexible enough. Then RISC computers were invented, and microprogramming (i.e., interpreted execution) became obsolete because direct execution was faster. Now we are seeing the resurgence of microprogramming because it allows CPUs to be updated in the field (for instance in response to dangerous CPU vulnerabilities such as Spectre, Meltdown, and RIDL). Thus the pendulum has already swung several cycles between direct execution and interpretation and may yet swing again in the future.

Large Memories

Let us now examine some historical developments in hardware and how they have affected software repeatedly. The first mainframes had limited memory. A fully loaded IBM 7090 or 7094, which played king of the mountain from late 1959 until 1964, had just over 128 KB of memory. It was mostly programmed in assembly language and its operating system was written in assembly language to save precious memory.

As time went on, compilers for languages like FORTRAN and COBOL got good enough that assembly language was pronounced dead. But when the first commercial minicomputer (the PDP-1) was released, it had only 4096 18-bit words

of memory, and assembly language made a surprise comeback. Eventually, minicomputers acquired more memory and high-level languages became prevalent on them.

When microcomputers hit in the early 1980s, the first ones had 4-KB memories and assembly-language programming rose from the dead. Embedded computers often used the same CPU chips as the microcomputers (8080s, Z80s, and later 8086s) and were also programmed in assembler initially. Now their descendants, the personal computers, have lots of memory and are programmed in C, C++, Python, Java, and other high-level languages. Smart cards are undergoing a similar development, although beyond a certain size, the smart cards often have a Java interpreter and execute Java programs interpretively, rather than having Java being compiled to the smart card's machine language.

Protection Hardware

Early mainframes, like the IBM 7090/7094, had no protection hardware, so they just ran one program at a time. A buggy program could wipe out the operating system and easily crash the machine. With the introduction of the IBM 360, a primitive form of hardware protection became available. These machines could then hold several programs in memory at the same time and have them take turns running (multiprogramming). Monoprogramming was declared obsolete.

At least until the first minicomputer showed up—without protection hardware—so multiprogramming was not possible. Although the PDP-1 and PDP-8 had no protection hardware, eventually the PDP-11 did, and this feature led to multiprogramming and eventually to UNIX.

When the first microcomputers were built, they used the Intel 8080 CPU chip, which had no hardware protection, so we were back to monoprogramming—one program in memory at a time. It was not until the Intel 80286 chip that protection hardware was added and multiprogramming became possible. Until this day, many embedded systems have no protection hardware and run just a single program. That works because the system designers have total control over all the software.

Disks

Early mainframes were largely magnetic-tape based. They would read in a program from tape, compile it, run it, and write the results back to another tape. There were no disks and no concept of a file system. That began to change when IBM introduced the first hard disk—the RAMAC (RAndoM ACcess) in 1956. It occupied about 4 square meters of floor space and could store 5 million 7-bit characters, enough for a single medium-resolution digital photo. But with an annual rental fee of about $35,000, assembling enough of them to store the equivalent of a roll of film got pricey quite fast. But eventually prices came down and primitive file systems were developed for the successors of these unwieldy devices.

Typical of these new developments was the CDC 6600, introduced in 1964 and for years by far the fastest computer in the world. Users could create so-called "permanent files" by giving them names and hoping that no other user had also decided that, say, "data" was a suitable name for a file. This was a single-level directory. Eventually, mainframes developed complex hierarchical file systems, perhaps culminating in the MULTICS file system.

As minicomputers came into use, they eventually also had hard disks. The standard disk on the PDP-11 when it was introduced in 1970 was the RK05 disk, with a capacity of 2.5 MB, about half of the IBM RAMAC, but it was only about 40 cm in diameter and 5 cm high. But it, too, had a single-level directory initially. When microcomputers came out, CP/M was initially the dominant operating system, and it, too, supported just one directory on the (floppy) disk. Later minicomputers and microcomputers got hierarchical file systems also.

Virtual Memory

Virtual memory (discussed in Chap. 3) gives the ability to run programs larger than the machine's physical memory by rapidly moving pieces back and forth between RAM and stable storage (SSD or disk). It underwent a similar development, first appearing on mainframes, then moving to the minis and the micros. Virtual memory also allowed having a program dynamically link in a library at run time instead of having it compiled in. MULTICS was again the first system to allow this. Eventually, the idea propagated down the line and is now widely used on most UNIX and Windows systems.

In all these developments, we see ideas invented in one context and later thrown out when the context changes (assembly-language programming, monoprogramming, single-level directories, etc.) only to reappear in a different context often a decade later. For this reason, in this book we will sometimes look at ideas and algorithms that may seem dated on today's high-end PCs, but which may soon come back on embedded computers, smart watches, or smart cards.

1.6 SYSTEM CALLS

We have seen that operating systems have two main functions: providing abstractions to user programs and managing the computer's resources. For the most part, the interaction between user programs and the operating system deals with the former; for example, creating, writing, reading, and deleting files. The resource--management part is largely transparent to the users and done automatically. Thus, the interface between user programs and the operating system is primarily about dealing with the abstractions. To really understand what operating systems do, we must examine this interface closely. The system calls available in the interface vary from one operating system to another, but the underlying concepts are similar.

We are thus forced to make a choice between (1) vague generalities ("operating systems have system calls for reading files") and (2) some specific system ("UNIX has a read system call with three parameters: one to specify the file, one to tell where the data are to be put, and one to tell how many bytes to read").

We have chosen the latter approach. It's more work that way, but it gives more insight into what operating systems really do. Although this discussion specifically refers to POSIX (International Standard 9945-1), hence also to UNIX, System V, BSD, Linux, MINIX 3, and so on, most other modern operating systems have system calls that perform the same functions, even if the details differ. Since the actual mechanics of issuing a system call are highly machine dependent and often must be expressed in assembly code, a procedure library is provided to make it possible to make system calls from C programs and often from other languages as well.

It is useful to keep the following in mind. Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap instruction to transfer control to the operating system. The operating system then figures out what the calling process wants by inspecting the parameters. Then it carries out the system call and returns control to the instruction following the system call. In a sense, making a system call is like making a special kind of procedure call—only system calls enter the kernel and procedure calls do not.

To make the system-call mechanism clearer, let us take a quick look at the read system call. As mentioned above, it has three parameters: the first one specifying the file, the second one pointing to the buffer, and the third one giving the number of bytes to read. Like nearly all system calls, it is invoked from C programs by calling a library procedure with the same name as the system call: *read*. A call from a C program might look like this:

```
count = read(fd, buffer, nbytes);
```

The system call (and the library procedure) returns the number of bytes actually read in *count*. This value is normally the same as *nbytes*, but may be smaller, if, for example, end-of-file is encountered while reading.

If the system call cannot be carried out owing to an invalid parameter or a disk error, *count* is set to −1, and the error number is put in a global variable, *errno*. Programs should always check the results of a system call to see if an error occurred.

System calls are performed in a series of steps. To make this concept clearer, let us examine the read call discussed above. In preparation for calling the *read* library procedure, which actually makes the read system call, the calling program first prepares the parameters, for instance by storing them in a set of registers that by convention are used for parameters. For instance, on x86-64 CPUs, Linux, FreeBSD, Solaris, and macOS use the System V AMD64 ABI **calling convention**, which means that the first six parameters are passed in registers RDI, RSI, RDX,

RCX, R8, and R9. If there are more than six arguments, the remainder will be pushed onto the stack. As we have only three arguments for *read* library procedure, this is shown as steps 1–3 in Fig. 1-17.

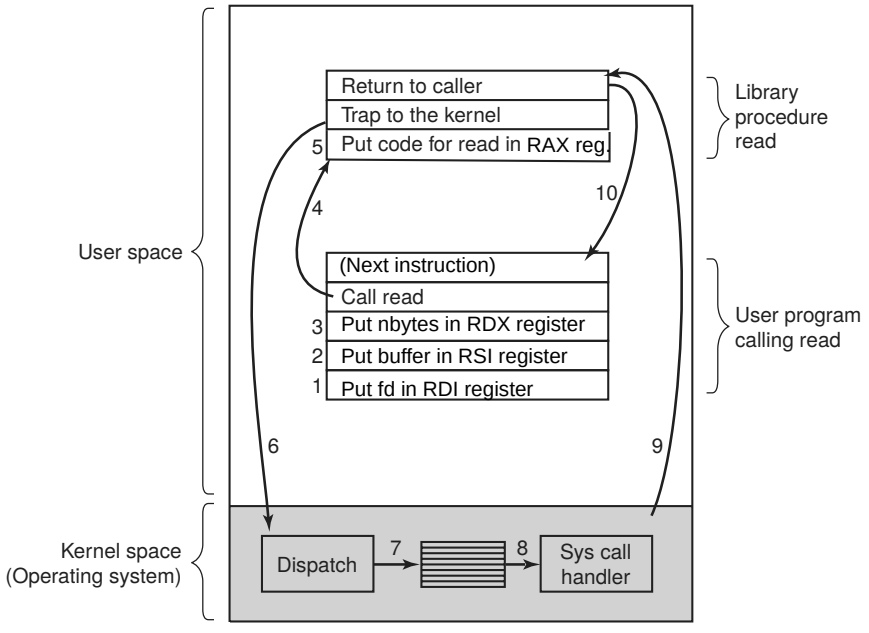

Figure 1-17. The 10 steps in making the system call read(fd, buffer, nbytes).

The first and third parameters are passed by value, but the second parameter is a reference, meaning that the address of the buffer is passed, not the contents of the buffer. Then comes the actual call to the library procedure (step 4). This instruction is the normal procedure-call instruction used to call all procedures.

The library procedure, written in assembly language, typically puts the system-call number in a place where the operating system expects it, such as the RAX register (step 5). Then it executes a **trap** instruction (such the X86-64 SYSCALL instruction) to switch from user mode to kernel mode and start execution at a fixed address within the kernel (step 6). The trap instruction is actually fairly similar to the procedure-call instruction in the sense that the instruction following it is taken from a distant location and the return address is saved on the stack for use later.

Nevertheless, the trap instruction also differs from the procedure-call instruction in two fundamental ways. First, as a side effect, it switches into kernel mode. The procedure call instruction does not change the mode. Second, rather than giving a relative or absolute address where the procedure is located, the trap instruction cannot jump to an arbitrary address. Depending on the architecture, either it

jumps to a single fixed location (this is the case for the x86-4 SYSCALL instruction) or there is an 8-bit field in the instruction giving the index into a table in memory containing jump addresses, or equivalent.

The kernel code that starts following the trap examines the system-call number in the RAX register and then dispatches to the correct system-call handler, usually via a table of pointers to system-call handlers indexed on system-call number (step 7). At that point, the system-call handler runs (step 8). Once it has completed its work, control may be returned to the user-space library procedure at the instruction following the trap instruction (step 9). This procedure then returns to the user program in the usual way procedure calls return (step 10), which then continues with the next instruction in the program (step 11).

In step 9 above, we said "may be returned to the user-space library procedure" for good reason. The system call may block the caller, preventing it from continuing. For example, if it is trying to read from the keyboard and nothing has been typed yet, the caller has to be blocked. In this case, the operating system will look around to see if some other process can be run next. Later, when the desired input is available, this process will get the attention of the system and run steps 9 and 10.

In the following sections, we will examine some of the most heavily used POSIX system calls, or more specifically, the library procedures that make those system calls. POSIX has about 100 procedure calls. Some of the most important ones are listed in Fig. 1-18, grouped for convenience in four categories. In the text, we will briefly examine each call to see what it does.

To a large extent, the services offered by these calls determine most of what the operating system has to do, since the resource management on personal computers is minimal (at least compared to big machines with multiple users). The services include things like creating and terminating processes, creating, deleting, reading, and writing files, managing directories, and performing input and output.

As an aside, it is worth pointing out that the mapping of POSIX procedure calls onto system calls is not one-to-one. The POSIX standard specifies a number of procedures that a conformant system must supply, but it does not specify whether they are system calls, library calls, or something else. If a procedure can be carried out without invoking a system call (i.e., without trapping to the kernel), it will usually be done in user space for reasons of performance. However, most of the POSIX procedures do invoke system calls, usually with one procedure mapping directly onto one system call. In a few cases, especially where several required procedures are only minor variations of one another, one system call handles more than one library call.

1.6.1 System Calls for Process Management

The first group of calls in Fig. 1-18 deals with process management. Fork is a good place to start the discussion. Fork is the only way to create a new process in POSIX. It creates an exact duplicate of the original process, including all the file

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory- and file-system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure 1-18. Some of the major POSIX system calls. The return code *s* is –1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time. The parameters are explained in the text.

descriptors, registers—everything. After the fork, the original process and the copy (the parent and child) go their separate ways. All the variables have identical values at the time of the fork, but since the parent's data are copied to create the child, subsequent changes in one of them do not affect the other one. In fact, the memory of the child may be shared **copy-on-write** with the parent. This means that parent

and child share a single physical copy of the memory until one of the two modifies a value at a location in memory—in which case the operating system makes a copy of the small chunk of memory containing that location. Doing so minimizes the amount of memory that needs to be copied a priori, as much can remain shared. Moreover, part of the memory, for instance, the program text does not change at, so it can always be shared between parent and child. The fork call returns a value, which is zero in the child and equal to the child's **PID** (**Process IDentifier**) in the parent. Using the returned PID, the two processes can see which one is the parent process and which one is the child process.

In most cases, after a fork, the child will need to execute different code from the parent. Consider the case of the shell. It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent executes a waitpid system call, which just waits until the child terminates (any child if more than one exists). Waitpid can wait for a specific child, or for any old child by setting the first parameter to −1. When waitpid completes, the address pointed to by the second parameter, *statloc*, will be set to the child process' exit status (normal or abnormal termination and exit value). Various options are also provided, specified by the third parameter. For example, returning immediately if no child has already exited.

Now consider how fork is used by the shell. When a command is typed, the shell forks off a new process. This child process must execute the user command. It does this by using the execve system call, which causes its entire core image to be replaced by the file named in its first parameter. A highly simplified shell illustrating the use of fork, waitpid, and execve is shown in Fig. 1-19.

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
     type_prompt( );                            /* display prompt on the screen */
     read_command(command, parameters);         /* read input from terminal */

     if (fork( ) != 0) {                        /* fork off child process */
          /* Parent code. */
          waitpid(-1, &status, 0);              /* wait for child to exit */
     } else {
          /* Child code. */
          execve(command, parameters, 0);       /* execute command */
     }
}
```

Figure 1-19. A stripped-down shell. Throughout this book, *TRUE* is assumed to be defined as 1.

In the most general case, execve has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment

array. These will be described shortly. Various library routines, including *execl*, *execv*, *execle*, and *execve*, are provided to allow the parameters to be omitted or specified in various ways. Throughout this book we will use the name exec to represent the system call invoked by all of these.

Let us consider the case of a command such as

```
cp file1 file2
```

used to copy *file1* to *file2*. After the shell has forked, the child process locates and executes the file *cp* and passes to it the names of the source and target files.

The main program of *cp* (and main program of most other C programs) contains the declaration

```
main(argc, argv, envp)
```

where *argc* is a count of the number of items on the command line, including the program name. For the example above, *argc* is 3.

The second parameter, *argv*, is a pointer to an array. Element *i* of that array is a pointer to the *i*th string on the command line. In our example, *argv*[0] would point to the string "cp", *argv*[1] would point to the string "file1", and *argv*[2] would point to the string "file2".

The third parameter of *main*, *envp*, is a pointer to the environment, an array of strings containing assignments of the form *name = value* used to pass information such as the terminal type and home directory name to programs. There are library procedures that programs can call to get the environment variables, which are often used to customize how a user wants to perform certain tasks (e.g., the default printer to use). In Fig. 1-19, no environment is passed to the child, so the third parameter of *execve* is a zero.

If exec seems complicated, do not despair; it is (semantically) the most complex of all the POSIX system calls. All the other ones are much simpler. As an example of a simple one, consider exit, which processes should use when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent via *statloc* in the waitpid system call.

Processes in UNIX have their memory divided up into three segments: the **text segment** (i.e., the program code), the **data segment** (i.e., the variables), and the **stack segment**. The data segment grows upward and the stack grows downward, as shown in Fig. 1-20. Between them is a gap of unused address space. The stack grows into the gap automatically, as needed, but expansion of the data segment is done explicitly by using a system call, brk, which specifies the new address where the data segment is to end. This call, however, is not defined by the POSIX standard, since programmers are encouraged to use the *malloc* library procedure for dynamically allocating storage, and the underlying implementation of *malloc* was not thought to be a suitable subject for standardization since few programmers use it directly and it is doubtful that anyone even notices that brk is not in POSIX. (In

most systems, there are other memory areas also, for instance those create with the mmap system call, which creates a new virtual memory areas, but we will get to those later.)

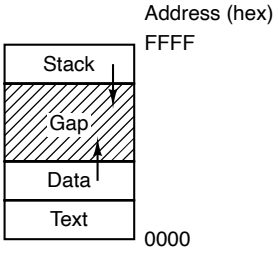

Figure 1-20. Processes have three segments: text, data, and stack.

1.6.2 System Calls for File Management

Many system calls relate to the file system. In this section, we will look at calls that operate on individual files; in the next one we will examine those that involve directories or the file system as a whole.

To read or write a file, it must first be opened. This call specifies the file name to be opened, either as an absolute path name or relative to the working directory, as well as a code of *O_RDONLY*, *O_WRONLY*, or *O_RDWR*, meaning open for reading, writing, or both. To create a new file, the *O_CREAT* parameter is used. The file descriptor returned can then be used for reading or writing. Afterward, the file can be closed by close, which makes the file descriptor available for reuse on a subsequent open.

The most heavily used calls are undoubtedly read and write. We saw read earlier. Write has the same parameters.

Although most programs read and write files sequentially, for some applications programs need to be able to access any part of a file at random. Associated with each file is a pointer that indicates the current position in the file. When reading (writing) sequentially, it normally points to the next byte to be read (written). The lseek call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file.

Lseek has three parameters: the first is the file descriptor for the file, the second is a file position, and the third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by lseek is the absolute position in the file (in bytes) after changing the pointer.

For each file, UNIX keeps track of the file mode (regular file, special file, directory, and so on), size, time of last modification, and other information. Programs can ask to see this information via the stat system call. The first parameter

specifies the file to be inspected; the second one is a pointer to a structure where the information is to be put. The fstat calls does the same thing for an open file.

1.6.3 System Calls for Directory Management

In this section, we will look at some system calls that relate more to directories or the file system as a whole, rather than just to one specific file as in the previous section. The first two calls, mkdir and rmdir, create and remove empty directories, respectively. The next call is link. Its purpose is to allow the same file to appear under two or more names, often in different directories. A typical use is to allow several members of the same programming team to share a common file, with each of them having the file appear in his own directory, possibly under different names. Sharing a file is not the same as giving every team member a private copy; having a shared file means that changes that any member of the team makes are instantly visible to the other members—there is only one file. When copies are made of a file, subsequent changes made to one copy do not affect the others.

To see how link works, consider the situation of Fig. 1-21(a). Here are two users, *ast* and *jim*, each having his own directory with some files. If *ast* now executes a program containing the system call

```
link("/usr/jim/memo", "/usr/ast/note");
```

the file *memo* in *jim*'s directory is now entered into *ast*'s directory under the name *note*. Thereafter, */usr/jim/memo* and */usr/ast/note* refer to the same file. As an aside, whether user directories are kept in */usr*, */user*, */home*, or somewhere else is simply a decision made by the local system administrator.

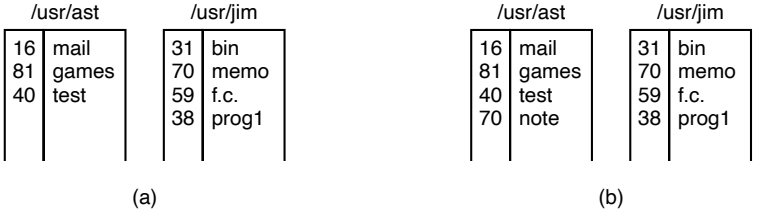

Figure 1-21. (a) Two directories before linking */usr/jim/memo* to *ast*'s directory. (b) The same directories after linking.

Understanding *how* link works will probably make clearer *what* it does. Every file in UNIX has a unique number, its i-number, that identifies it. This i-number is an index into a table of **i-nodes**, one per file, telling who owns the file, where its disk blocks are, and so on[†]. A directory is simply a file containing a set of (i-number, ASCII name) pairs. In the first versions of UNIX, each directory entry was 16

[†]Most people still call them disk blocks, even if they reside on SSD.

bytes—2 bytes for the i-number and 14 bytes for the name. Now a more complicated structure is needed to support long file names, but conceptually a directory is still a set of (i-number, ASCII name) pairs. In Fig. 1-21, *mail* has i-number 16, and so on. What link does is simply create a brand new directory entry with a (possibly new) name, using the i-number of an existing file. In Fig. 1-21(b), two entries have the same i-number (70) and thus refer to the same file. If either one is later removed, using the unlink system call, the other one remains. If both are removed, UNIX sees that no entries to the file exist (a field in the i-node keeps track of the number of directory entries pointing to the file), so the file is removed from the SSD or disk and its blocks are returned to the free block pool.

As we have mentioned earlier, the mount system call allows two file systems to be merged into one. A common situation is to have the root file system, containing the binary (executable) versions of the common commands and other heavily used files, on an SSD / hard disk (sub)partition and user files on another (sub)partition. Further, the user can then insert a USB disk with files to be read.

By executing the mount system call, the USB file system can be attached to the root file system, as shown in Fig. 1-22. A typical statement in C to mount is

```
mount("/dev/sdb0", "/mnt", 0);
```

where the first parameter is the name of a block special file for USB drive 0, the second parameter is the place in the tree where it is to be mounted, and the third parameter tells whether the file system is to be mounted read-write or read-only.

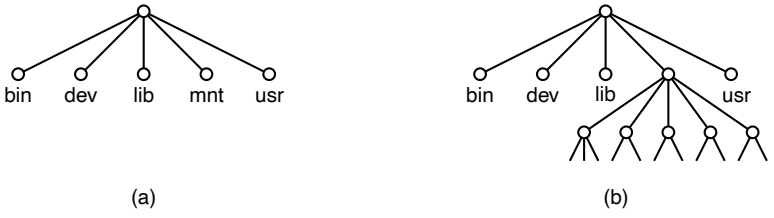

Figure 1-22. (a) File system before the mount. (b) File system after the mount.

After the mount call, a file on drive 0 can be accessed by just using its path from the root directory or the working directory, without regard to which drive it is on. In fact, second, third, and fourth drives can also be mounted anywhere in the tree. The mount call makes it possible to integrate removable media into a single integrated file hierarchy, without having to worry about which device a file is on. Although this example involves USB drives, portions of hard disks (often called **partitions** or **minor devices**) can also be mounted this way, as well as external hard disks and SSDs. When a file system is no longer needed, it can be unmounted with the umount system call. After that, it is no longer accessible. Of course, if it is needed later on, it can be mounted again.

1.6.4 Miscellaneous System Calls

A variety of other system calls exist as well. We will look at just four of them here. The chdir call changes the current working directory. After the call

```
chdir("/usr/ast/test");
```

an open on the file *xyz* will open */usr/ast/test/xyz*. The concept of a working directory eliminates the need for typing (long) absolute path names all the time.

In UNIX every file has a mode used for protection. The mode includes the read-write-execute bits for the owner, group, and others. The chmod system call makes it possible to change the mode of a file. For example, to make a file read-only by everyone except the owner, one could execute

```
chmod("file", 0644);
```

The kill system call is the way users and user processes send signals. If a process is prepared to catch a particular signal, then when it arrives, a signal handler is run. If the process is not prepared to handle a signal, then its arrival kills the process (hence the name of the call).

POSIX defines a number of procedures for dealing with time. For example, time just returns the current time in seconds, with 0 corresponding to Jan. 1, 1970 at midnight (just as the day was starting, not ending). On computers using 32-bit words, the maximum value time can return is $2^{32} - 1$ seconds (assuming an unsigned integer is used). This value corresponds to a little over 136 years. Thus in the year 2106, 32-bit UNIX systems will go berserk, not unlike the famous Y2K problem that would have wreaked havoc with the world's computers in 2000, were it not for the massive effort the IT industry put into fixing the problem. If you currently have a 32-bit UNIX system, you are advised to trade it in for a 64-bit one sometime before the year 2106.

1.6.5 The Windows API

So far we have focused primarily on UNIX. Now it is time to look briefly at Windows. Windows and UNIX differ in a fundamental way in their respective programming models. A UNIX program consists of code that does something or other, making system calls to have certain services performed. In contrast, a Windows program is normally event driven. The main program waits for some event to happen, then calls a procedure to handle it. Typical events are keys being struck, the mouse being moved, a mouse button being pushed, or a USB drive inserted or removed from the computer. Handlers are then called to process the event, update the screen, and update the internal program state. All in all, this leads to a somewhat different style of programming than with UNIX, but since the focus of this book is on operating system function and structure, these different programming models will not concern us much more.

Of course, Windows also has system calls. With UNIX, there is almost a one-to-one relationship between the system calls (e.g., read) and the library procedures (e.g., *read*) used to invoke the system calls. In other words, for each system call, there is roughly one library procedure that is called to invoke it, as indicated in Fig. 1-17. Furthermore, POSIX has only on the order of 100 procedure calls.

With Windows, the situation is radically different. To start with, the library calls and the actual system calls are highly decoupled. Microsoft has defined a set of procedures called the **WinAPI**, **Win32 API**, or **Win64 API** (**Application Programming Interface**) that programmers are expected to use to get operating system services. This interface is (partially) supported on all versions of Windows since Windows 95. By decoupling the API interface that programmer's use from the actual system calls, Microsoft retains the ability to change the actual system calls in time (even from release to release) without invalidating existing programs. What actually constitutes Win32 is also slightly ambiguous because recent versions of Windows have many new calls that were not previously available. In this section, Win32 means the interface supported by all versions of Windows. Win32 provides compatibility among versions of Windows. Win64 is largely Win32 with bigger pointers so we will focus on Win32 here.

The number of Win32 API calls is extremely large, numbering in the thousands. Furthermore, while many of them do invoke system calls, a substantial number are carried out entirely in user space. As a consequence, with Windows it is impossible to see what is a system call (i.e., performed by the kernel) and what is simply a user-space library call. In fact, what is a system call in one version of Windows may be done in user space in a different version, and vice versa. When we discuss the Windows system calls in this book, we will use the Win32 procedures (where appropriate) since Microsoft guarantees that these will be stable over time. But it is worth remembering that not all of them are true system calls (i.e., traps to the kernel).

The Win32 API has a huge number of calls for managing windows, geometric figures, text, fonts, scrollbars, dialog boxes, menus, and other features of the GUI. To the extent that the graphics subsystem runs in the kernel (true on some versions of Windows but not on all), these are system calls; otherwise they are just library calls. Should we discuss these calls in this book or not? Since they are not really related to the function of an operating system, we have decided not to, even though they may be carried out by the kernel. Readers interested in the Win32 API should consult one of the many books on the subject (e.g., Yosifovich, 2020).

Even introducing all the Win32 API calls here is out of the question, so we will restrict ourselves to those calls that roughly correspond to the functionality of the UNIX calls listed in Fig. 1-18. These are listed in Fig. 1-23.

Let us now briefly go through the list of Fig. 1-23. CreateProcess creates a new process. It does the combined work of fork and execve in UNIX. It has many parameters specifying the properties of the newly created process. Windows does not have a process hierarchy like UNIX does, so there is no concept of a parent

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount, so no umount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Figure 1-23. The Win32 API calls that roughly correspond to the UNIX calls of Fig. 1-18. It is worth emphasizing that Windows has a very large number of other system calls, most of which do not correspond to anything in UNIX.

process and a child process. After a process is created, the creator and createe are equals. WaitForSingleObject is used to wait for an event. Many possible events can be waited for. If the parameter specifies a process, then the caller waits for the specified process to exit, which is done using ExitProcess.

The next six calls operate on files and are functionally similar to their UNIX counterparts although they differ in the parameters and details. Still, files can be opened, closed, read, and written pretty much as in UNIX. The SetFilePointer and GetFileAttributesEx calls set the file position and get some of the file attributes.

Windows has directories and they are created with CreateDirectory and RemoveDirectory API calls, respectively. There is also a notion of a current directory, set by SetCurrentDirectory. The current time of day is acquired using GetLocalTime.

The Win32 interface does not have links to files, mounted file systems, security, or signals, so the calls corresponding to the UNIX ones do not exist. Of course, Win32 has a huge number of other calls that UNIX does not have, especially for

managing the GUI. For instance, Windows 11 has an elaborate security system and also supports file links.

One last note about Win32 is perhaps worth making. Win32 is not a terribly uniform or consistent interface. The main culprit here was the need to be backward compatible with the previous 16-bit interface used in Windows 3.x.

1.7 OPERATING SYSTEM STRUCTURE

Now that we have seen what operating systems look like on the outside (i.e., the programmer's interface), it is time to take a look inside. In the following sections, we will examine six different structures that have been tried, in order to get some idea of the spectrum of possibilities. These are by no means exhaustive, but they give an idea of some designs that have been tried in practice. The six designs we will discuss here are monolithic systems, layered systems, microkernels, client-server systems, virtual machines, and exo- and unikernels.

1.7.1 Monolithic Systems

By far the most common organization, the monolithic approach is to run the entire operating system as a single program in kernel mode. The operating system is written as a collection of procedures, linked together into a single large executable binary program. When this technique is used, each procedure in the system is free to call any other one, if the latter provides some useful computation that the former needs. Being able to call any procedure you want is very efficient, but having thousands of procedures that can call each other without restriction may also lead to a system that is unwieldy and difficult to understand. Also, a crash in any of these procedures will take down the entire operating system.

To construct the actual object program of the operating system when this approach is used, one first compiles all the individual procedures (or the files containing the procedures) and then binds them all together into a single executable file using the system linker. In terms of information hiding, there is essentially none—every procedure is visible to every other procedure (as opposed to a structure containing modules or packages, in which much of the information is hidden away inside modules, and only the officially designated entry points can be called from outside the module).

Even in monolithic systems, however, it is possible to have some structure. The services (system calls) provided by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and then executing a trap instruction. This instruction switches the machine from user mode to kernel mode and transfers control to the operating system, shown as step 6 in Fig. 1-17. The operating system then fetches the parameters and determines which system call is to be carried out. After that, it indexes into a table that contains in slot *k* a pointer to the procedure that carries out system call *k* (step 7 in Fig. 1-17).

This organization suggests a basic structure for the operating system:

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

In this model, for each system call there is one service procedure that takes care of it and executes it. The utility procedures do things that are needed by several service procedures, such as fetching data from user programs. This division of the procedures into three layers is shown in Fig. 1-24.

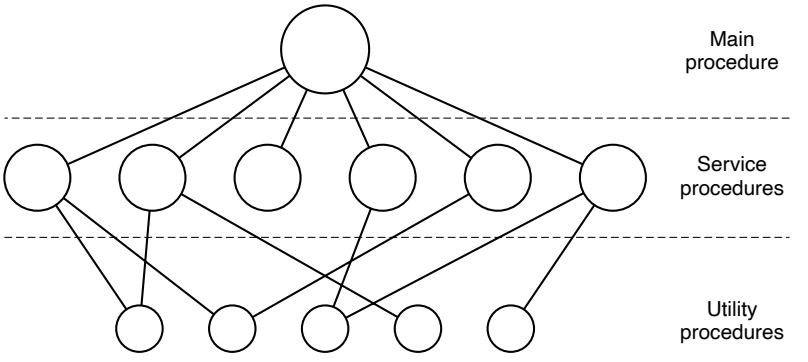

Figure 1-24. A simple structuring model for a monolithic system.

In addition to the core operating system that is loaded when the computer is booted, many operating systems support loadable extensions, such as I/O device drivers and file systems. These components are loaded on demand. In UNIX they are called **shared libraries**. In Windows they are called **DLLs** (**Dynamic-Link Libraries**). They have file extension *.dll* and the *C:\Windows\system32* directory on Windows systems has well over 1000 of them.

1.7.2 Layered Systems

A generalization of the approach of Fig. 1-24 is to organize the operating system as a hierarchy of layers, each one constructed upon the one below it. The first system constructed in this way was the THE system built at the Technische Hogeschool Eindhoven in the Netherlands by E. W. Dijkstra (1968) and his students. The THE system was a simple batch system for a Dutch computer, the Electrologica X8, which had 32K of 27-bit words (bits were expensive back then).

The system had six layers, as shown in Fig. 1-25. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of

which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 1-25. Structure of the THE operating system.

Layer 1 did the memory management. It allocated space for processes in main memory and on a 512K word drum used for holding parts of processes (pages) for which there was no room in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum; the layer 1 software took care of making sure pages were brought into memory at the moment they were needed and removed when they were not needed.

Layer 2 handled communication between each process and the operator console (that is, the user). On top of this layer, each process effectively had its own operator console. Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities. Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management. The system operator process was located in layer 5.

A further generalization of the layering concept was present in the MULTICS system. Instead of layers, MULTICS was described as having a series of concentric rings, with the inner ones being more privileged than the outer ones (which is effectively the same thing). When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to make the equivalent of a system call, that is, a TRAP instruction whose parameters were carefully checked for validity before the call was allowed to proceed. Although the entire operating system was part of the address space of each user process in MULTICS, the hardware made it possible to designate individual procedures (memory segments, actually) as protected against reading, writing, or executing.

Whereas the THE layering scheme was really only a design aid, because all the parts of the system were ultimately linked together into a single executable program, in MULTICS, the ring mechanism was very much present at run time and enforced by the hardware. The advantage of the ring mechanism is that it can easily be extended to structure user subsystems. For example, a professor could write a

program to test and grade student programs and run this program in ring n, with the student programs running in ring $n + 1$ so that they could not change their grades.

1.7.3 Microkernels

With the layered approach, the designers have a choice where to draw the kernel-user boundary. Traditionally, all the layers went in the kernel, but that is not necessary. In fact, a strong case can be made for putting as little as possible in kernel mode because bugs in the kernel can bring down the system instantly. In contrast, user processes have less power so that a bug there may not be fatal.

Various researchers have repeatedly studied the number of bugs per 1000 lines of code (e.g., Basilli and Perricone, 1984; and Ostrand and Weyuker, 2002). Bug density depends on module size, module age, and more, but a ballpark figure for serious industrial systems is between two and ten bugs per thousand lines of code. This means that a monolithic operating system of five million lines of code is likely to contain between 10,000 and 50,000 kernel bugs. Not all of these are fatal, of course, since some bugs may be things like a minor misspelling in an error message is rarely needed.

The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules, only one of which—the microkernel—runs in kernel mode and the rest run as relatively powerless ordinary user processes. In particular, by running each device driver and file system as a separate user process, a bug in one of these can crash that component, but cannot crash the entire system. Thus, a bug in the audio driver will cause the sound to be garbled or stop, but will not crash the computer. In contrast, in a monolithic system with all the drivers in the kernel, a buggy audio driver can easily reference an invalid memory address and bring the system to a grinding halt instantly.

Many microkernels have been implemented and deployed for decades (Haertig et al., 1997; Heiser et al., 2006; Herder et al., 2006; Hildebrand, 1992; Kirsch et al., 2005; Liedtke, 1993, 1995, 1996; Pike et al., 1992; and Zuberi et al., 1999). With the exception of macOS, which is based on the Mach microkernel (Accetta et al., 1986), common desktop operating systems do not use microkernels. However, they are dominant in real-time, industrial, avionics, and military applications that are mission critical and have very high reliability requirements. A few of the better-known microkernels include Integrity, K42, L4, PikeOS, QNX, Symbian, and MINIX 3. We now give a brief overview of MINIX 3, which has taken the idea of modularity to the limit, breaking most of the operating system up into a number of independent user-mode processes. MINIX 3 is a POSIX-conformant, open source system freely available at *www.minix3.org* (Giuffrida et al., 2012; Giuffrida et al., 2013; Herder et al., 2006; Herder et al., 2009; and Hruby et al., 2013). Intel adopted MINIX 3 for its management engine in virtually all its CPUs.

The MINIX 3 microkernel is only about 15,000 lines of C and some 1400 lines of assembler for very low-level functions such as catching interrupts and switching processes. The C code manages and schedules processes, handles interprocess communication (by passing messages between processes), and offers a set of about 40 kernel calls to allow the rest of the operating system to do its work. These calls perform functions like hooking handlers to interrupts, moving data between address spaces, and installing memory maps for new processes. The process structure of MINIX 3 is shown in Fig. 1-26, with the kernel call handlers labeled *Sys*. The device driver for the clock is also in the kernel because the scheduler interacts closely with it. The other device drivers run as separate user processes.

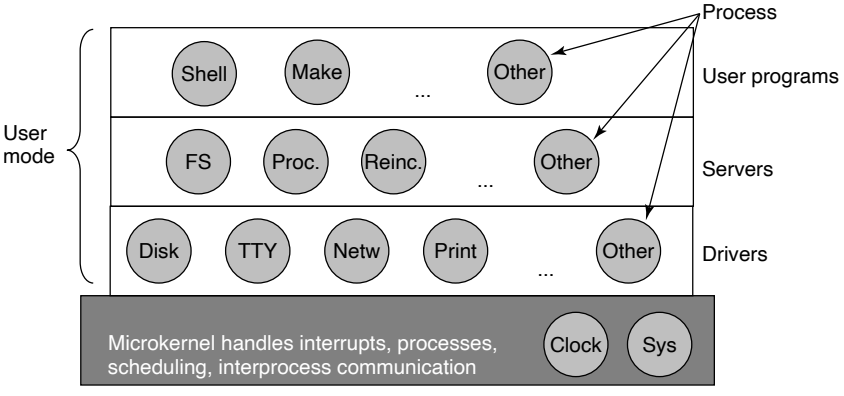

Figure 1-26. Simplified structure of the MINIX system.

Outside the kernel, the system is structured as three layers of processes all running in user mode. The lowest layer contains the device drivers. Since they run in user mode, they do not have physical access to the I/O port space and cannot issue I/O commands directly. Instead, to program an I/O device, the driver builds a structure telling which values to write to which I/O ports and makes a kernel call telling the kernel to do the write. This approach means that the kernel can check to see that the driver is writing (or reading) from I/O it is authorized to use. Consequently (and unlike a monolithic design), a buggy audio driver cannot accidentally write on the SSD or disk.

Above the drivers is another user-mode layer containing the servers, which do most of the work of the operating system. One or more file servers manage the file system(s), the process manager creates, destroys, and manages processes, and so on. User programs obtain operating system services by sending short messages to the servers asking for the POSIX system calls. For example, a process needing to do a read sends a message to one of the file servers telling it what to read.

One interesting server is the **reincarnation server**, whose job is to check if the other servers and drivers are functioning correctly. In the event that a faulty one is

detected, it is automatically replaced without any user intervention. In this way, the system is self-healing and can achieve high reliability.

The system has many restrictions limiting the power of each process. As mentioned, drivers can touch only authorized I/O ports, but access to kernel calls is also controlled on a per-process basis, as is the ability to send messages to other processes. Processes can also grant limited permission for other processes to have the kernel access their address spaces. As an example, a file system can grant permission for the disk driver to let the kernel put a newly read-in disk block at a specific address within the file system's address space. The sum total of all these restrictions is that each driver and server has exactly the power to do its work and nothing more, thus greatly limiting the damage a buggy component can do. Restricting what a component can do to exactly that what it needs to do its work is known as the **POLA** (**Principle of Least Authority**) an important design principle for building secure systems. We will discuss other such principles in Chap. 9.

An idea somewhat related to having a minimal kernel is to put the **mechanism** for doing something in the kernel but not the **policy**. To make this point better, consider the scheduling of processes. A relatively simple scheduling algorithm is to assign a numerical priority to every process and then have the kernel run the highest-priority process that is runnable. The mechanism—in the kernel—is to look for the highest-priority process and run it. The policy—assigning priorities to processes—can be done by user-mode processes. In this way, policy and mechanism can be decoupled and the kernel can be made smaller.

1.7.4 Client-Server Model

A slight variation of the microkernel idea is to distinguish two classes of processes, the **servers**, each of which provides some service, and the **clients**, which use these services. This model is known as the **client-server** model. The essence is the presence of client processes and server processes.

Communication between clients and servers is often by message passing. To obtain a service, a client process constructs a message saying what it wants and sends it to the appropriate service. The service then does the work and sends back the answer. If the client and server happen to run on the same machine, certain optimizations are possible, but conceptually, we are still talking about message passing here.

A generalization of this idea is to have the clients and servers run on different computers, connected by a local or wide-area network, as depicted in Fig. 1-27. Since clients communicate with servers by sending messages, the clients need not know whether the messages are handled locally on their own machines, or whether they are sent across a network to servers on a remote. machine. As far as the client is concerned, the same thing happens in both cases: requests are sent and replies come back. Thus, the client-server model is an abstraction that can be used for a single machine or for a network of machines.

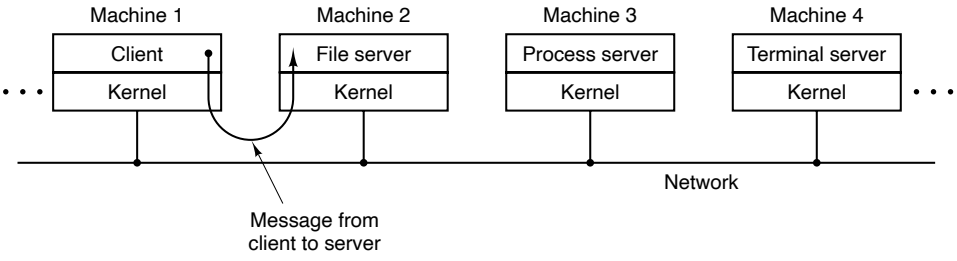

Figure 1-27. The client-server model over a network.

Increasingly many systems involve users at their home PCs as clients and large machines elsewhere running as servers. In fact, much of the Web operates this way. A PC sends a request for a Web page to the server and the Web page comes back. This is a typical use of the client-server model in a network.

1.7.5 Virtual Machines

The initial releases of OS/360 were strictly batch systems. Nevertheless, many 360 users wanted to be able to work interactively at a terminal, so various groups, both inside and outside IBM, decided to write timesharing systems for it. The official IBM timesharing system, TSS/360, was delivered late, and when it finally arrived it was so big and slow that few sites converted to it. It was eventually abandoned after its development had consumed some $50 million (Graham, 1970). But a group at IBM's Scientific Center in Cambridge, Massachusetts, produced a radically different system that IBM eventually accepted as a product. A linear descendant of it, called **z/VM**, is now widely used on IBM's current mainframes, the zSeries, which are heavily used in large corporate data centers, for example, as e-commerce servers that handle hundreds or thousands of transactions per second and use databases whose sizes run to millions of gigabytes.

VM/370

This system, originally called CP/CMS and later renamed VM/370 (Seawright and MacKinnon, 1979), was based on an astute observation: a timesharing system provides (1) multiprogramming and (2) an extended machine with a more convenient interface than the bare hardware. The essence of VM/370 is to completely separate these two functions.

The heart of the system, called a **virtual machine monitor**, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up, as shown in Fig. 1-28. However, unlike all other operating systems, these virtual machines are not extended machines, with files

and other nice features. Instead, they are *exact* copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.

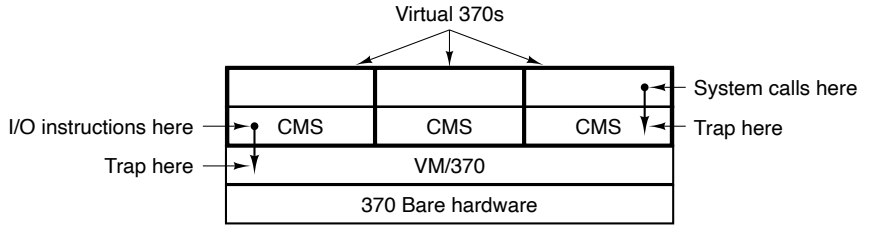

Figure 1-28. The structure of VM/370 with CMS.

Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware. Different virtual machines can, and frequently do, run different operating systems. On the original IBM VM/370 system, some ran OS/360 or one of the other large batch or transaction-processing operating systems, while others ran a single-user, interactive system called **CMS** (**Conversational Monitor System**) for interactive timesharing users. The latter was popular with programmers.

When a CMS program executed a system call, the call was trapped to the operating system in its own virtual machine, not to VM/370, just as it would be were it running on a real machine instead of a virtual one. CMS then issued the normal hardware I/O instructions for reading its virtual disk or whatever was needed to carry out the call. These I/O instructions were trapped by VM/370, which then performed them as part of its simulation of the real hardware. By completely separating the functions of multiprogramming and providing an extended machine, each of the pieces could be much simpler, more flexible, and much easier to maintain.

In its modern incarnation, z/VM is usually used to run multiple complete operating systems rather than stripped-down single-user systems like CMS. For example, the zSeries is capable of running one or more Linux virtual machines along with traditional IBM operating systems.

Virtual Machines Rediscovered

While IBM has had a virtual-machine product available for four decades, and a few other companies, including Oracle and Hewlett-Packard, have recently added virtual-machine support to their high-end enterprise servers, the idea of virtualization has largely been ignored in the PC world until recently. But in the past decades, a combination of new needs, new software, and new technologies have combined to make it a hot topic.

First the needs. Many companies have traditionally run their mail servers, Web servers, FTP servers, and other servers on separate computers, sometimes with

different operating systems. They see virtualization as a way to run them all on the same machine without having a crash of one server bring down the rest.

Virtualization is also popular in the Web hosting world. Without virtualization, Web hosting customers are forced to choose between **shared hosting** (which just gives them a login account on a Web server, but no control over the server software) and dedicated hosting (which gives them their own machine, which is very flexible but not cost effective for small to medium Websites). When a Web hosting company offers virtual machines for rent, a single physical machine can run many virtual machines, each of which appears to be a complete machine. Customers who rent a virtual machine can run whatever operating system and software they want to, but at a fraction of the cost of a dedicated server (because the same physical machine supports many virtual machines at the same time).

Another use of virtualization is for end users who want to be able to run two or more operating systems at the same time, say Windows and Linux, because some of their favorite application packages run on one and some run on the other. This situation is illustrated in Fig. 1-29(a), where the term "virtual machine monitor" has been renamed **type 1 hypervisor**, which is commonly used nowadays because "virtual machine monitor" requires more keystrokes than people are prepared to put up with now. Note that many authors use the terms interchangeably though.

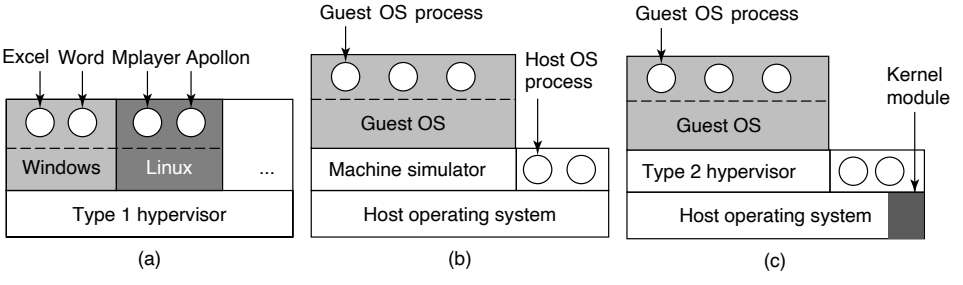

Figure 1-29. (a) A type 1 hypervisor. (b) A pure type 2 hypervisor. (c) A practical type 2 hypervisor.

While no one disputes the attractiveness of virtual machines today, the problem then was implementation. In order to run virtual machine software on a computer, its CPU must be virtualizable (Popek and Goldberg, 1974). In a nutshell, here is the problem. When an operating system running on a virtual machine (in user mode) executes a privileged instruction, such as modifying the PSW or doing I/O, it is essential that the hardware trap to the virtual-machine monitor so the instruction can be emulated in software. On some CPUs—notably the Pentium, its predecessors, and its clones—attempts to execute privileged instructions in user mode are just ignored. This property made it impossible to have virtual machines on this hardware, which explains the lack of interest in the x86 world. Of course, there

were interpreters for the Pentium, such as *Bochs*, that ran on the Pentium, but with a performance loss of one to two orders of magnitude, they were not useful for serious work.

This situation changed as a result of several academic research projects in the 1990s and early years of this millennium, notably Disco at Stanford (Bugnion et al., 1997) and Xen at Cambridge University (Barham et al., 2003). These research papers led to several commercial products (e.g., VMware Workstation and Xen) and a revival of interest in virtual machines. Besides VMware and Xen, popular hypervisors today include KVM (for the Linux kernel), VirtualBox (by Oracle), and Hyper-V (by Microsoft).

Some of the early research projects improved the performance over interpreters like *Bochs* by translating blocks of code on the fly, storing them in an internal cache, and then reusing them if they were executed again. This improved the performance considerably, and led to what we will call **machine simulators**, as shown in Fig. 1-29(b). However, although this technique, known as **binary translation**, helped improve matters, the resulting systems, while good enough to publish papers about in academic conferences, were still not fast enough to use in commercial environments where performance matters a lot.

The next step in improving performance was to add a kernel module to do some of the heavy lifting, as shown in Fig. 1-29(c). In practice now, all commercially available hypervisors, such as VMware Workstation, use this hybrid strategy (and have many other improvements as well). They are called **type 2 hypervisors** by everyone, so we will (somewhat grudgingly) go along and use this name in the rest of this book, even though we would prefer to called them type 1.7 hypervisors to reflect the fact that they are not entirely user-mode programs. In Chap. 7, we will describe in detail how VMware Workstation works and what the various pieces do.

In practice, the real distinction between a type 1 hypervisor and a type 2 hypervisor is that a type 2 makes uses of a **host operating system** and its file system to create processes, store files, and so on. A type 1 hypervisor has no underlying support and must perform all these functions itself.

After a type 2 hypervisor is started, it reads the installation image file for the chosen **guest operating system** and installs the guest OS on a virtual disk, which is just a big file in the host operating system's file system. Type 1 hypervisors cannot do this because there is no host operating system to store files on. They must manage their own storage on a raw disk partition.

When the guest operating system is booted, it does the same thing it does on the actual hardware, typically starting up some background processes and then a GUI. To the user, the guest operating system behaves the same way it does when running on the bare metal even though that is not the case here.

A different approach to handling control instructions is to modify the operating system to remove them. This approach is not true virtualization. Instead it is called **paravirtualization**. We will discuss virtualization in Chap. 7.

The Java Virtual Machine

Another area where virtual machines are used, but in a somewhat different way, is for running Java programs. When Sun Microsystems invented the Java programming language, it also invented a virtual machine (i.e., a computer architecture) called the **JVM** (**Java Virtual Machine**). Sun no longer exists today (because Oracle bought the company), but Java is still with us. The Java compiler produces code for JVM, which then typically is executed by a software JVM interpreter. The advantage of this approach is that the JVM code can be shipped over the Internet to any computer that has a JVM interpreter and run there. If the compiler had produced SPARC or x86 binary programs, for example, they could not have been shipped and run anywhere as easily. (Of course, Sun could have produced a compiler that produced SPARC binaries and then distributed a SPARC interpreter, but JVM is a much simpler architecture to interpret.) Another advantage of using JVM is that if the interpreter is implemented properly, which is not completely trivial, incoming JVM programs can be checked for safety and then executed in a protected environment so they cannot steal data or do any damage.

Containers

Besides full virtualization, we can also run multiple instances of an operating system on a single machine at the same time by having the operating system itself support different systems, or **containers**. Containers are provided by the host operating system such as Windows or Linux and mostly run just the user mode portion of an operating system. Each container shares the host operating system kernel and typically the binaries and libraries in a read-only fashion. This way, a Linux host can support many Linux containers. Since a container does not contain a full operating system, it can be extremely lightweight.

Of course, there are downsides to containers also. First, it is not possible to run a container with a completely different operating system from that of the host. Also, unlike virtual machines, there is no strict resource partitioning. The container may be restricted in what it may access on SSD or disk and how much CPU time it gets, but all containers still share the resources in the underlying host operating system. Phrased differently, containers are process-level isolated. This means that a container that messes with the stability of the underlying kernel will also affect other containers.

1.7.6 Exokernels and Unikernels

Rather than cloning the actual machine, as is done with virtual machines, another strategy is partitioning it, in other words, giving each user a subset of the resources. Thus one virtual machine might get disk blocks 0 to 1023, the next one might get blocks 1024 to 2047, and so on.

At the bottom layer, running in kernel mode, is a program called the **exokernel** (Engler et al., 1995). Its job is to allocate resources to virtual machines and then check attempts to use them to make sure no machine is trying to use somebody else's resources. Each user-level virtual machine can run its own operating system, as on VM/370 and the Pentium virtual 8086s, except that each one is restricted to using only the resources it has asked for and been allocated.

The advantage of the exokernel scheme is that it saves a layer of mapping. In the other designs, each virtual machine thinks it has its own disk or SSD, with blocks running from 0 to some maximum, so the virtual machine monitor must maintain tables to remap disk block addresses (and all other resources). With the exokernel, this remapping is not needed. The exokernel need only keep track of which virtual machine has been assigned which resource. This method still has the advantage of separating the multiprogramming (in the exokernel) from the user operating system code (in user space), but with less overhead, since all the exokernel has to do is keep the virtual machines out of each other's hair.

The operating system functions were linked with the applications in the virtual machine in the form of a **LibOS** (**Library Operating System**), that needed only the functionality for the application(s) running in the user-level virtual machine. This idea, like so many others, was forgotten for a few decades, only to be rediscovered in recent years, in the form of **Unikernels**, minimal LibOS-based systems that contain just enough functionality to support a single application (such as a Web server) on a virtual machine. Unikernels have the potential to be highly efficient as protection between the operating system (LibOS) and application is not needed: since there is only one application on the virtual machine, all code can run in kernel mode.

1.8 THE WORLD ACCORDING TO C

Operating systems are normally large C (or sometimes C++) programs consisting of many pieces written by many programmers. The environment used for developing operating systems is very different from what individuals (such as students) are used to when writing small Java programs. This section is an attempt to give a very brief introduction to the world of writing an operating system for small-time Java or Python programmers.

1.8.1 The C Language

This is not a guide to C, but a short summary of some of the key differences between C and languages like **Python** and especially Java. Java is based on C, so there are many similarities between the two. Python is somewhat different, but still fairly similar. For convenience, we focus on Java. Java, Python, and C are all imperative languages with data types, variables, and control statements, for example. The primitive data types in C are integers (including short and long ones),

characters, and floating-point numbers. Composite data types can be constructed using arrays, structures, and unions. The control statements in C are similar to those in Java, including if, switch, for, and while statements. Functions and parameters are roughly the same in both languages.

One feature C has that Java and Python do not is explicit pointers. A **pointer** is a variable that points to (i.e., contains the address of) a variable or data structure. Consider the statements

```
char c1, c2, *p;
c1 = 'c';
p = &c1;
c2 = *p;
```

which declare *c1* and *c2* to be character variables and *p* to be a variable that points to (i.e., contains the address of) a character. The first assignment stores the ASCII code for the character "c" in the variable *c1*. The second one assigns the address of *c1* to the pointer variable *p*. The third one assigns the contents of the variable pointed to by *p* to the variable *c2*, so after these statements are executed, *c2* also contains the ASCII code for "c". In theory, pointers are typed, so you are not supposed to assign the address of a floating-point number to a character pointer, but in practice compilers accept such assignments, albeit sometimes with a warning. Pointers are a very powerful construct, but also a great source of errors when used carelessly.

Some things that C does not have include built-in strings, threads, packages, classes, objects, type safety, and garbage collection. The last one is a show stopper for operating systems. All storage in C is either static or explicitly allocated and released by the programmer, usually with the library functions *malloc* and *free*. It is the latter property—total programmer control over memory—along with explicit pointers that makes C attractive for writing operating systems. Operating systems are basically real-time systems to some extent, even general-purpose ones. When an interrupt occurs, the operating system may have only a few microseconds to perform some action or lose critical information. Having the garbage collector kick in at an arbitrary moment is intolerable.

1.8.2 Header Files

An operating system project generally consists of some number of directories, each containing many *.c* files containing the code for some part of the system, along with some *.h* header files that contain declarations and definitions used by one or more code files. Header files can also include simple **macros**, such as

```
#define BUFFER_SIZE 4096
```

which allow the programmer to name constants, so that when *BUFFER_SIZE* is used in the code, it is replaced during compilation by the number 4096. Good C

programming practice is to name every constant except 0, 1, and −1, and sometimes even them. Macros can have parameters, such as

```
#define max(a, b) (a > b ? a : b)
```

which allows the programmer to write

```
i = max(j, k+1)
```

and get

```
i = (j > k+1 ? j : k+1)
```

to store the larger of *j* and *k+1* in *i*. Headers can also contain conditional compilation, for example

```
#ifdef X86
intel_int_ack();
#endif
```

which compiles into a call to the function *intel_int_ack* if the macro *X86* is defined and nothing otherwise. Conditional compilation is heavily used to isolate architecture-dependent code so that certain code is inserted only when the system is compiled on the X86, other code is inserted only when the system is compiled on a SPARC, and so on. A *.c* file can bodily include zero or more header files using the *#include* directive. There are also many header files that are common to nearly every *.c* and are stored in a central directory.

1.8.3 Large Programming Projects

To build the operating system, each *.c* is compiled into an **object file** by the C compiler. Object files, which have the suffix *.o*, contain binary instructions for the target machine. They will later be directly executed by the CPU. There is nothing like Java byte code or Python byte code in the C world.

The first pass of the C compiler is called the **C preprocessor**. As it reads each *.c* file, every time it hits a *#include* directive, it goes and gets the header file named in it and processes it, expanding macros, handling conditional compilation (and certain other things) and passing the results to the next pass of the compiler as if they were physically included.

Since operating systems are very large (five million lines of code is not unusual), having to recompile the entire thing every time one file is changed would be unbearable. On the other hand, changing a key header file that is included in thousands of other files does require recompiling those files. Keeping track of which object files depend on which header files is completely unmanageable without help.

Fortunately, computers are very good at precisely this sort of thing. On UNIX systems, there is a program called *make* (with numerous variants such as *gmake*,

pmake, etc.) that reads the *Makefile*, which tells it which files are dependent on which other files. What *make* does is see which object files are needed to build the operating system binary and for each one, check to see if any of the files it depends on (the code and headers) have been modified subsequent to the last time the object file was created. If so, that object file has to be recompiled. When *make* has determined which *.c* files have to recompiled, it then invokes the C compiler to recompile them, thus reducing the number of compilations to the bare minimum. In large projects, creating the *Makefile* is error prone, so there are tools that do it automatically.

Once all the *.o* files are ready, they are passed to a program called the **linker** to combine all of them into a single executable binary file. Any library functions called are also included at this point, interfunction references are resolved, and machine addresses are relocated as need be. When the linker is finished, the result is an executable program, traditionally called *a.out* on UNIX systems. The various components of this process are illustrated in Fig. 1-30 for a program with three C files and two header files. Although we have been discussing operating system development here, all of this applies to developing any large program.

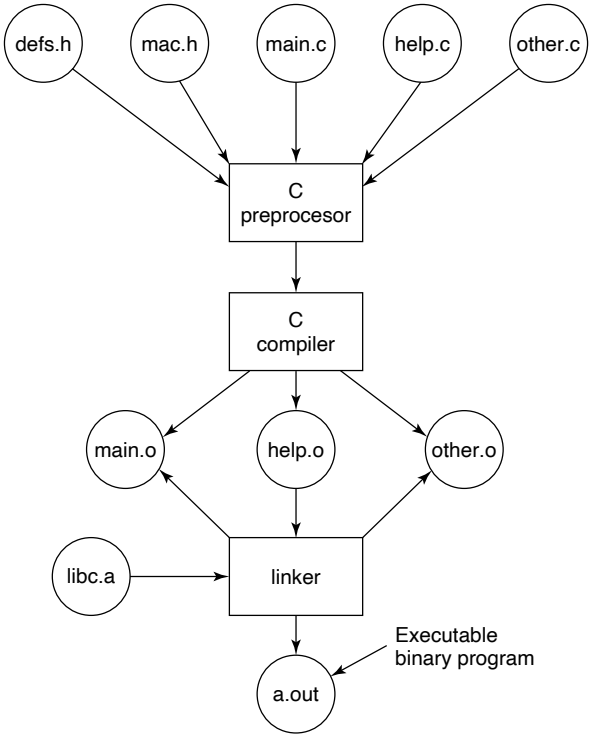

Figure 1-30. The process of compiling C and header files to make an executable.

1.8.4 The Model of Run Time

Once the operating system binary has been linked, the computer can be rebooted and the new operating system started. Once running, it may dynamically load pieces that were not statically included in the binary such as device drivers and file systems. At run time, the operating system may consist of multiple segments, for the text (the program code), the data, and the stack. The text segment is normally immutable, not changing during execution. The data segment starts out at a certain size and initialized with certain values, but it can change and grow as need be. The stack is initially empty but grows and shrinks as functions are called and returned from. Often the text segment is placed near the bottom of memory, the data segment just above it, with the ability to grow upward, and the stack segment at a high virtual address, with the ability to grow downward, but different systems work differently.

In all cases, the operating system code is directly executed by the hardware, with no interpreter and no just-in-time compilation, as is normal with Java.

1.9 RESEARCH ON OPERATING SYSTEMS

Computer science is a rapidly advancing field and it is hard to predict where it is going. Researchers at universities and industrial research labs are constantly thinking up new ideas, some of which go nowhere but some of which become the cornerstone of future products and have massive impact on the industry and users. Telling which is which turns out to be easier to do in hindsight than in real time. Separating the wheat from the chaff is especially difficult because it often takes 20 to 30 years from idea to impact.

For example, when President Dwight Eisenhower set up the Dept. of Defense's Advanced Research Projects Agency (ARPA) in 1958, he was trying to keep the Army from killing the Navy and the Air Force over the Pentagon's research budget. He was not trying to invent the Internet. But one of the things ARPA did was fund some university research on the then-obscure concept of packet switching, which led to the first experimental packet-switched network, the ARPANET. It went live in 1969. Before long, other ARPA-funded research networks were connected to the ARPANET, and the Internet was born. The Internet was then happily used by academic researchers for sending email to each other for 20 years. In the early 1990s, Tim Berners-Lee invented the World Wide Web at the CERN research lab in Geneva and Marc Andreesen wrote a graphical browser for it at the University of Illinois. All of a sudden, the Internet was full of twittering teenagers. President Eisenhower is probably rolling over in his grave.

Research in operating systems has also led to dramatic changes in practical systems. As we discussed earlier, the first commercial computer systems were all batch systems, up until M.I.T. invented general-purpose timesharing in the early

1960s. Computers were all text-based until Doug Engelbart invented the mouse and the graphical user interface at Stanford Research Institute in the late 1960s. Who knows what will come next?

In this section, and in comparable sections throughout the book, we will take a brief look at some of the research in operating systems that has taken place during the past 5–10 years, just to give a flavor of what might be on the horizon. This introduction is certainly not comprehensive. It is based largely on papers that have been published in the top research conferences because these ideas have at least survived a rigorous peer review process in order to get published. Note that in computer science—in contrast to other scientific fields—most research is published in conferences, not in journals. Most of the papers cited in the research sections were published by either ACM, the IEEE Computer Society, or USENIX and are available over the Internet to (student) members of these organizations. For more information about these organizations and their digital libraries, see

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

All operating systems researchers realize that current operating systems are massive, inflexible, unreliable, insecure, and loaded with bugs, certain ones more than others (*names withheld to protect the guilty*). Consequently, there is a lot of research on how to build better ones. Work has recently been published about bugs and debugging (Kasikci et al., 2017; Pina et al., 2019; Li et al., 2019), crashes and recovery (Chen et al., 2017; and Bhat et al., 2021), energy management (Petrucci and Loques, 2012; Shen et al., 2013; and Li et al., 2020), file and storage systems (Zhang et al., 2013a; Chen et al., 2017; Maneas et al., 2020; Ji et al., 2021; and Miller et al., 2021), high-performance I/O (Rizzo, 2012; Li et al., 2013a; and Li et al., 2017); hyperthreading and multithreading (Li et al., 2019), dynamic updates (Pina et al., 2019), managing GPUs (Volos et al., 2018), memory management (Jantz et al., 2013; and Jeong et al., 2013), embedded systems (Levy et al., 2017), operating system correctness and reliability (Klein et al., 2009; and Chen et al., 2017), operating system reliability (Chen et al., 2017; Chajed et al., 2019; and Zou et al., 2019), security (Oliverio et al., 2017; Konoth et al., 2018; Osterlund et al., 2019; Duta et al. 2021), virtualization and containers (Tack Lim et al., 2017; Manco et al., 2017; and Tarasov et al., 2013) among many other topics.

1.10 OUTLINE OF THE REST OF THIS BOOK

We have now completed our introduction and bird's-eye view of the operating system. It is time to get down to the details. As mentioned, from the programmer's point of view, the primary purpose of an operating system is to provide some

key abstractions, the most important of which are processes and threads, address spaces, and files. Accordingly the next three chapters are devoted to these topics.

Chapter 2 is about processes and threads. It discusses their properties and how they communicate with one another. It also gives a number of detailed examples of how interprocess communication works and how to avoid some of the pitfalls.

In Chap. 3 we study address spaces and their adjunct, memory management. The important topic of virtual memory will be examined, together with paging.

Then, in Chap. 4, we come to the topic of file systems. To a considerable extent, what the user sees is the file system. We will look at both the file-system interface and the file-system implementation.

Input/Output is covered in Chap. 5. We will cover the concept of device (in)dependence using examples such as storage devices, keyboards, and displays.

Chapter 6 is about deadlocks, including ways to prevent or avoid them.

At this point, we will have completed our study of the basic principles of single-CPU operating systems. However, there is more to say, especially about advanced topics. In Chap. 7, we examine virtualization. We discuss both the principles, and some of the existing virtualization solutions in detail. Another advanced topic is multiprocessor systems, including multicores, parallel computers, and distributed systems. These subjects are covered in Chap. 8. Another important subject is operating system security, which we cover in Chap 9.

Next we have some case studies of real operating systems. These are UNIX, Linux, and Android (Chap. 10), and Windows 11 (Chap. 11). The text concludes with some wisdom and thoughts about operating system design in Chap. 12.

1.11 METRIC UNITS

To avoid any confusion, it is worth stating explicitly that in this book, as in computer science in general, metric units are used instead of traditional English units (the furlong-stone-fortnight system). The principal metric prefixes are listed in Fig. 1-31. The prefixes are typically abbreviated by their first letters, with the units greater than 1 capitalized. Thus a 1-TB database occupies 10^{12} bytes of storage and a 100-psec (or 100-ps) clock ticks every 10^{-10} seconds. Since milli and micro both begin with the letter "m," a choice had to be made. Normally, "m" is for milli and "μ" (the Greek letter mu) is for micro.

It is also worth pointing out that, in common industry practice, the units for measuring memory sizes have slightly different meanings. There kilo means 2^{10} (1024) rather than 10^3 (1000) because memories are always a power of two. Thus a 1-KB memory contains 1024 bytes, not 1000 bytes. Similarly, a 1-MB memory contains 2^{20} (1,048,576) bytes and a 1-GB memory contains 2^{30} (1,073,741,824) bytes. However, a 1-Kbps communication line transmits 1000 bits per second and a 1-Gbps LAN runs at 1,000,000,000 bits/sec because these speeds are not powers of two. Unfortunately, many people tend to mix up these two systems, especially for

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

Figure 1-31. The principal metric prefixes.

SSD or disk sizes. To avoid ambiguity, in this book, we will use the symbols KB, MB, and GB for 2^{10}, 2^{20}, and 2^{30} bytes, respectively, and the symbols Kbps, Mbps, and Gbps for 10^{3}, 10^{6}, and 10^{9} bits/sec, respectively.

1.12 SUMMARY

Operating systems can be viewed from two viewpoints: resource managers and extended machines. In the resource-manager view, the operating system's job is to manage the different parts of the system efficiently. In the extended-machine view, the system provides the users with abstractions that are more convenient to use than the actual machine. These include processes, address spaces, and files.

Operating systems have a long history, starting from the days when they replaced the operator, to modern multiprogramming systems. Highlights include early batch systems, multiprogramming systems, and personal computer systems.

Since operating systems interact closely with the hardware, some knowledge of computer hardware is useful to understanding them. Computers are built up of processors, memories, and I/O devices. These parts are connected by buses.

Operating systems can be structured as monolithic, layered, microkernel/client-server, virtual machine, or exokernel/unikernel systems. Regardless, the basic concepts on which they are built are processes, memory management, I/O management, the file system, and security. The main interface of an operating system is the set of system calls that it can handle. These tell us what it really does.

PROBLEMS

1. What are the two main functions of an operating system?

2. What is multiprogramming?

3. In Sec. 1.4, nine different types of operating systems are described. Give a list of possible applications for each of these systems (at least one for each of the operating systems types).

4. To use cache memory, main memory is divided into cache lines, typically 32 or 64 bytes long. An entire cache line is cached at once. What is the advantage of caching an entire line instead of a single byte or word at a time?

5. What is spooling? Do you think that advanced personal computers will have spooling as a standard feature in the future?

6. On early computers, every byte of data read or written was handled by the CPU (i.e., there was no DMA). What implications does this have for multiprogramming?

7. Why was timesharing not widespread on second-generation computers?

8. Instructions related to accessing I/O devices are typically privileged instructions, that is, they can be executed in kernel mode but not in user mode. Give a reason why these instructions are privileged.

9. One reason GUIs were initially slow to be adopted was the cost of the hardware needed to support them. How much video RAM is needed to support a 25-line × 80-row character monochrome text screen? How much for a 1024 × 768-pixel 24-bit color bitmap? What was the cost of this RAM at 1980 prices ($5/KB)? How much is it now?

10. There are several design goals in building an operating system, for example, resource utilization, timeliness, robustness, and so on. Give an example of two design goals that may contradict one another.

11. What is the difference between kernel and user mode? Explain how having two distinct modes aids in designing an operating system.

12. A 255-GB disk has 65,536 cylinders with 255 sectors per track and 512 bytes per sector. How many platters and heads does this disk have? Assuming an average cylinder seek time of 11 msec, average rotational delay of 7 msec. and reading rate of 100 MB/sec, calculate the average time it will take to read 100 KB from one sector.

13. Consider a system that has two CPUs, each CPU having two threads (hyperthreading). Suppose three programs, *P0*, *P1*, and *P2*, are started with run times of 5, 10 and 20 msec, respectively. How long will it take to complete the execution of these programs? Assume that all three programs are 100% CPU bound, do not block during execution, and do not change CPUs once assigned.

14. List some differences between personal computer operating systems and mainframe operating systems.

15. A computer has a pipeline with four stages. Each stage takes the same time to do its work, namely, 1 nsec. How many instructions per second can this machine execute?

16. Consider a computer system that has cache memory, main memory (RAM) and disk, and an operating system that uses virtual memory. It takes 2 nsec to access a word from the cache, 10 nsec to access a word from the RAM, and 10 msec to access a word from the disk. If the cache hit rate is 95% and main memory hit rate (after a cache miss) is 99%, what is the average time to access a word?

17. When a user program makes a system call to read or write a disk file, it provides an indication of which file it wants, a pointer to the data buffer, and the count. Control is then transferred to the operating system, which calls the appropriate driver. Suppose that the driver starts the disk and terminates until an interrupt occurs. In the case of reading from the disk, obviously the caller will have to be blocked (because there are no data for it). What about the case of writing to the disk? Need the caller be blocked awaiting completion of the disk transfer?

18. What is the key difference between a trap and an interrupt?

19. Is there any reason why you might want to mount a file system on a nonempty directory? If so, what is it?

20. What is the purpose of a system call in an operating system?

21. Give one reason why mounting file systems is a better design option than prefixing path names with a drive name or number. Explain why file systems are almost always mounted on empty directories.

22. For each of the following system calls, give a condition that causes it to fail: `open`, `close`, and `lseek`.

23. What type of multiplexing (time, space, or both) can be used for sharing the following resources: CPU, memory, SSD/disk, network card, printer, keyboard, and display?

24. Can the

```
count = write(fd, buffer, nbytes);
```

call return any value in *count* other than *nbytes*? If so, why?

25. A file whose file descriptor is *fd* contains the following sequence of bytes: 2, 7, 1, 8, 2, 8, 1, 8, 2, 8, 4. The following system calls are made:

```
lseek(fd, 3, SEEK_SET);
read(fd, &buffer, 4);
```

where the `lseek` call makes a seek to byte 3 of the file. What does *buffer* contain after the read has completed?

26. Suppose that a 10-MB file is stored on a disk on the same track (track 50) in consecutive sectors. The disk arm is currently situated over track number 100. How long will it take to retrieve this file from the disk? Assume that it takes about 1 msec to move the arm from one cylinder to the next and about 5 msec for the sector where the beginning of the file is stored to rotate under the head. Also, assume that reading occurs at a rate of 100 MB/s.

27. What is the essential difference between a block special file and a character special file?

28. In the example given in Fig. 1-17, the library procedure is called *read* and the system call itself is called `read`. Is it essential that both of these have the same name? If not, which one is more important?

29. The client-server model is popular in distributed systems. Can it also be used in a single-computer system?

30. To a programmer, a system call looks like any other call to a library procedure. Is it important that a programmer know which library procedures result in system calls? Under what circumstances and why?

31. Figure 1-23 shows that a number of UNIX system calls have no Win32 API equivalents. For each of the calls listed as having no Win32 equivalent, what are the consequences for a programmer of converting a UNIX program to run under Windows?

32. A portable operating system is one that can be ported from one system architecture to another without any modification. Explain why it is infeasible to build an operating system that is completely portable. Describe two high-level layers that you will have in designing an operating system that is highly portable.

33. Explain how separation of policy and mechanism aids in building microkernel-based operating systems.

34. Virtual machines have become very popular for a variety of reasons. Nevertheless, they have some downsides. Name one.

35. Here are some questions for practicing unit conversions:

(a) How long is a microyear in seconds?
(b) Micrometers are often called microns. How long is a gigamicron?
(c) How many bytes are there in a 1-TB memory?
(d) The mass of the earth is 6000 yottagrams. What is that in grams?

36. Write a shell that is similar to Fig. 1-19 but contains enough code that it actually works so you can test it. You might also add some features such as redirection of input and output, pipes, and background jobs.

37. If you have a personal UNIX-like system (Linux, MINIX 3, FreeBSD, etc.) available that you can safely crash and reboot, write a shell script that attempts to create an unlimited number of child processes and observe what happens. Before running the experiment, type `sync` to the shell to flush the file system buffers to disk to avoid ruining the file system. You can also do the experiment safely in a virtual machine. **Note**: Do not try this on a shared system without first getting permission from the system administrator. The consequences will be instantly obvious so you are likely to be caught and sanctions may follow.

38. Examine and try to interpret the contents of a UNIX-like or Windows directory with a tool like the UNIX *od* program. (*Hint*: How you do this will depend upon what the OS allows. One trick that may work is to create a directory on a USB stick with one operating system and then read the raw device data using a different operating system that allows such access.)