Chapter

6

An Introduction to System Software and Virtual Machines

Chapter Introduction
6.1 Introduction
6.2 System Software
6.2.1 The Virtual Machine
6.2.2 Types of System Software
6.3 Assemblers and Assembly Language
6.3.1 Assembly Language
6.3.2 Examples of Assembly Language Code
6.3.3 Translation and Loading
6.4 Operating Systems
6.4.1 Functions of an Operating System
6.4.2 Historical Overview of Operating Systems Development
6.4.3 The Future
Change font sizeMain content

Chapter Contents

Chapter Introduction

r studying this chapter, you will be able to:

Compare the virtual machine created for the user by system software with the naked machine
Describe the different types of system software
Explain the benefits of writing programs in assembly language rather than machine language
Describe how an assembler translates assembly language programs into machine instructions

List five key tasks of an operating system, and explain what each is and why it is critical to modern systems

Describe the different generations of operating systems, their features, and how each generation solved a drawback of the previous generation

Chapter Contents

6.1 Introduction

Chapters 4 and 5 described a computer model, the *Von Neumann architecture*, that is capable of executing programs written in machine language. This computer has all the hardware needed to solve important real-world problems, but it has no "support tools" to make that problem-solving task easy. The computer described in Chapter 5 is informally known as a **naked machine**: hardware bereft of any helpful user-oriented features.

Imagine what it would be like to work on a naked machine. To solve a problem, you would have to create hundreds or thousands of cryptic machine language instructions that look like this:

```
10110100110100011100111100001000
```

and you would have to do that without making a single mistake because, to execute properly, a program must be completely error free. Imagine the likelihood of writing a correct program containing thousands of instructions like the one shown above. Even worse, imagine trying to locate an error buried deep inside that incomprehensible mass of 0s and 1s!

On a naked machine, the data as well as the instructions must be represented in binary. For example, a program cannot refer to the decimal integer +9 directly but must express it as

```
0000000000001001  (the binary representation of 19 using 16 bits)
```

You cannot use the symbol *A* to refer to the first letter of the alphabet but must represent it using its 8-bit ASCII code value, which is decimal 65:

```
01000001    (the 8-bit ASCII code for A; see Figure 4.3)
```

As you can imagine, writing programs for a naked machine is very difficult.

Even if you write the program correctly, your work is still not done. A program for a Von Neumann computer must be stored in memory prior to execution. Therefore, you must now take the program and store its instructions into sequential cells in memory. On a naked machine, the programmer must perform this task, one instruction at a time. Assuming that each instruction occupies one memory cell, the programmer loads the first instruction into address 0, the second instruction into address 1, the third instruction into address 2, and so on, until all have been stored.

Finally, what starts the program running? A naked machine does not do this automatically. (As you are probably coming to realize, a naked machine does not do *anything* automatically, except fetch, decode, and execute machine language

instructions.) The programmer must initiate execution by storing a 0, the address of the first instruction of the program, into the program counter (PC) and pressing the START button. This begins the fetch/decode/execute cycle described in Chapter 5. The control unit fetches from memory the contents of the address in the PC, currently 0, and executes that instruction. The program continues sequentially from that point while the user prays that everything works because he or she cannot bear to face a naked machine again!

As this portrayal demonstrates, working directly with the underlying hardware is practically impossible for a human being. The functional units described in Chapter 5 are built according to what is easy for hardware to do, not what is easy for people to do.

To make a Von Neumann computer usable, we must create an *interface* between the user and the hardware. This interface does the following things:

- Hides from the user the messy details of the underlying hardware
- Presents information about what is happening in a way that does not require in-depth knowledge of the internal structure of the system
- Allows easy user access to the resources available on this computer
- Prevents accidental or intentional damage to hardware, programs, and data

By way of analogy, let's look at how people use another common tool—an automobile. The internal combustion engine is a highly complex piece of technology. For most of us, the functions of fuel-injection systems, distributors, and camshafts are a total mystery. However, most people find driving a car quite easy. This is because the driver does not have to lift the hood and interact directly with the hardware; that is, he or she does not have to drive a "naked automobile." Instead, there is an interface, the *dashboard*, which simplifies things considerably. The dashboard hides the details of engine operation that a driver does not need to know. The important things—such as oil pressure, fuel level, and vehicle speed—are presented in a simple, "people-oriented" way: oil indicator warning light, fuel gauge, and speed in miles or kilometers per hour. Access to the engine and transmission is achieved via a few easy-to-understand devices: a key to start and stop, pedals to speed up or slow down, a shift lever to go forward or backward, and a steering wheel to direct movement.

We need a similar interface for our Von Neumann machine. This "computer dashboard" would eliminate most of the hassles of working on a naked machine and let us view the hardware resources of Chapter 5 in a much friendlier way. Such an interface does exist, and it is called system software.

Chapter Contents

6.2 System Software

6.2.1 The Virtual Machine

System software is a collection of computer programs that manage the resources of a computer and facilitates access to those resources. This contrasts with application software that allows a user to address some specialized task of interest to that user, for example, write a document, create an image, browse the web, or solve a system of equations. (Application software is addressed in Level 5 of this text.) It is important to

remember that we are describing software, not hardware. There are no black boxes wired to a computer labeled “system software.” Software consists of sequences of instructions—namely, programs—that solve a problem. But again, instead of solving *user* problems, system software solves the problem of making a computer and its resources easier to access and use.

System software acts as an *intermediary* between the users and the hardware, as shown in Figure 6.1. System software presents the user with a set of services and resources across the interface labeled A in Figure 6.1. These resources may actually exist, or they may be simulated by the software to give the user the illusion that they exist. The set of services and resources created by the software and seen by the user is called a **virtual machine** or a **virtual environment**. The system software, not the user, interacts with the actual hardware (that is, the naked machine) across the interface labeled B in Figure 6.1.

Figure 6.1 The role of system software

The system software has the following responsibilities, analogous to those of the automobile dashboard:

- Hides the complex and unimportant (to the user) details of the internal structure of the Von Neumann architecture
- Presents important information to the user in a way that is easy to understand
- Allows the user to access machine resources in a simple and efficient way
- Provides a secure and safe environment in which to operate

For example, to add two numbers, it is much easier to use notation such as than to worry about

- (1)

  loading ALU registers from memory cells *b* and *c*,

- (2)

  activating the ALU,

- (3)

  selecting the output of the addition circuit, and

- (4)

  sending the result to memory cell *a*.

The programmer should not have to know about registers, addition circuits, and memory addresses but instead should see a virtual machine that “understands” the mathematical symbols + and =.

After a program has been written (or purchased), it should automatically be loaded into memory without the programmer having to specify where in memory it should be placed or having to set the program counter. Instead, he or she should be able to issue one simple command (or mouse click or finger tap) to the virtual machine that says, “Run this application.” Finally, when the program is running and generating results, the

programmer should be able to instruct the virtual machine to send the program's output to the printer in Room 105, without reference to the details related to I/O controllers, interrupt signals, and code sets.

All the useful services just described are provided by the system software available on any modern computer system. The following sections show how this friendly, user-oriented environment is created.

6.2.2 Types of System Software

System software is not a single monolithic entity but a collection of many different programs. The types found on a typical computer are shown in Figure 6.2.

Figure 6.2 Types of system software

The program that controls the overall operation of the computer is the **operating system**, and it is the single most important piece of system software on a computer. It is the operating system that communicates with users, determines what they want, and activates other system programs, applications packages, or user programs to carry out their requests. The software packages that handle these requests include the following:

- *User interface*—All modern operating systems provide a powerful **graphical user interface (GUI)** that gives the user an intuitive visual overview as well as graphical control of the capabilities and services of the computer. Control of the GUI is typically done with keystrokes, mouse clicks, finger taps, voice activation, or biometric scans such as fingerprints.
- *Language services*—These programs, called *assemblers, compilers*, and *interpreters*, allow you to write programs in a high-level, user-oriented language rather than the machine language of Chapter 5 and to execute these programs easily and efficiently. They often include components such as text editors and debuggers.
- *Memory managers*—These programs allocate memory space for programs and data and retrieve this memory space when it is no longer needed.
- *Information managers*—These programs handle the organization, storage, and retrieval of information on storage devices such as the hard drives, DVDs, flash drives, and tapes described in Section 5.2.2, as well as information stored remotely in data centers (*cloud storage*). They allow you to organize your information in an efficient hierarchical manner, using directories, folders, and files and to keep your personal data safe from accidental or intentional misuse.
- *I/O* systems—These software packages allow you to easily and efficiently use the many different types of input and output devices that exist on a modern computer system.
- *Scheduler*—This system program keeps a list of programs ready to run on the processor, and it selects the one that will execute next. The scheduler allows you to have several different programs active at a single time, for instance, surfing the web while you are waiting for a file to finish printing.

- *Utilities*—These collections of library routines provide a wide range of useful services either to a user or to other system routines. Text editors, online help routines, image and sound applications, and control panels are examples of utility routines. Sometimes these utilities are organized into collections called **program libraries**.

These system routines are used during every phase of problem solving on a computer, and it would be virtually impossible to get anything done without them. Let's go back to the problem described earlier—the job of writing a program, loading it into memory, running it, and printing the results. On a naked machine, this job would be formidable. On the virtual machine created by system software, it is much simpler:

Step Task

1 Use a text editor to create program P written in a high-level, English-like notation rather than binary.

2 Use an information manager to store program P in a directory on your cloud storage account.

3 Use a language translator to translate program P from a high-level language into an equivalent machine language program M.

4 Use a scheduler to load, schedule, and run program M. The scheduler itself uses the memory manager to obtain memory space for program M.

5 Use the I/O system to display the output of your program on your screen.

6 If the program did not complete successfully, use a debugger to help you locate the error. Use a text editor to correct the program and the information manager to store the newly modified program.

Furthermore, most of these operations are easily invoked via mouse clicks or finger taps using the graphical interface provided by the operating system.

On a virtual machine, the low-level details of machine operation are no longer visible, and a user can concentrate on higher-level issues: writing the program, executing the program, and saving and analyzing results.

There are many types of system software, and it is impossible to cover them all in this section of the text. Instead, we will investigate two types of system software and use these as representatives of the entire group. Section 6.3 examines assemblers, and Section 6.4 looks at the design and construction of operating systems. These two packages create a friendly and usable virtual machine. In Chapter 7, we will extend that virtual environment from a single computer to a collection of computers by looking at the system software required to create one of the most important and widely used virtual environments—a computer network. Finally, in Chapter 8 we will investigate one of the most important services provided by the operating system—system security.

6.3Assemblers and Assembly Language

6.3.1 Assembly Language

One of the first places where we need a friendlier virtual environment is in our choice of programming language. Machine language, which is designed from a machine's point of view, not a person's, is complicated and virtually impossible to understand. What specifically are the problems with machine language?

- It uses binary. There are no natural language words, mathematical symbols, or other convenient mnemonics to make the language more readable.
- It allows only numeric memory addresses (in binary). A programmer cannot name an instruction or a piece of data and refer to it by name.
- It is difficult to change. If we insert or delete an instruction, all memory addresses following that instruction will change. For example, if we place a new instruction into memory location 503, then the instruction previously in location 503 is now in 504. All references to address 503 must be updated to point to 504. There may be hundreds of such references.
- It is difficult to create data. If a user wants to store a piece of data in memory, he or she must compute the internal binary representation for that data item. These conversion algorithms are complicated and time consuming.

Programmers working on early first-generation computers quickly realized the shortcomings of machine language. They developed a new language, called **assembly language**, designed for people as well as computers. Assembly languages created a more productive, user-oriented environment, and assemblers were one of the first pieces of system software to be widely used. When assembly languages first appeared in the early 1950s, they were one of the most important new developments in programming—so important, in fact, that they were considered an entirely new generation of language, analogous to the new generations of hardware described in Section 1.4.3. Assembly languages were termed *second-generation languages* to distinguish them from machine languages, which were viewed as *first-generation languages*.

Today, assembly languages are more properly called **low-level programming languages**, which means they are closely related to the machine language of Chapter 5. Each symbolic assembly language instruction is translated into exactly *one* binary machine language instruction.

This contrasts with languages like C++, Java, and Python, which are **high-level programming languages**. High-level languages are more user oriented, they are not machine specific, and they use both natural language and mathematical notation in their design. A single high-level language instruction is typically translated into *many* machine language instructions, and the virtual environment created by a high-level language is far more powerful than the one produced by an assembly language. We discuss high-level programming languages in detail in Chapters 9 and 10.

Figure 6.3 shows a "continuum of languages," from the lowest level (closest to the hardware) to the highest level (most abstract, farthest from the hardware).

Figure 6.3 The continuum of programming languages

The machine language of Chapter 5 is the most primitive; it is the language of the hardware itself. Assembly language, the topic of this section, represents the first step along the continuum from machine language. Highlevel programming languages like C++, Java, and Python are closer in style and structure to natural languages and are quite distinct from assembly language. Natural languages, such as English, Spanish, and Japanese, are the highest level; they are totally unrelated to hardware design.

Although it is rare today to write a program in assembly language, it is important to understand the philosophy behind its design. It was the very first time that computer scientists asked the question, "What can I do to make these machines easier to use and easier to understand?" This same question is still being asked and answered today.

A program written in assembly language is called the **source program**; it uses the features and services provided by the language. However, the processor does not "understand" assembly language instructions, in the sense of being able to fetch, decode, and execute them as described in Chapter 5. The source program must be translated into a corresponding machine language program, called the **object program**. This translation is carried out by a piece of system software called an **assembler**. (Translators for high-level languages are called **compilers**. They are discussed in Chapter 11.) The assembler goes through the entire program, carrying out a translation of one instruction at a time. Once the complete object program has been produced, its instructions can be loaded into memory and executed by the processor exactly as described in Section 5.3. The complete translation/loading/execution process is diagrammed in Figure 6.4.

Figure 6.4 The translation/loading/execution process

There are three major advantages to writing programs in assembly language rather than machine language:

- Use of symbolic operation codes rather than numeric (binary) ones
- Use of symbolic memory addresses rather than numeric (binary) ones
- Pseudo-operations that provide useful user-oriented services such as data generation

This section describes a simple, but realistic, assembly language that demonstrates these three advantages.

Our hypothetical assembly language is composed of instructions in the following format:

```
label:   op code mnemonic      address field      --comment
```

The *comment field*, preceded in our notation by a double dash (--), is not really part of the instruction. It is a helpful explanation added to the instruction by a programmer and intended for readers of the program. It is ignored by the machine during translation and execution.

Assembly languages allow the programmer to refer to op codes using a symbolic name, called the *op code mnemonic*, rather than by a number. We can write op codes using meaningful words like LOAD, ADD, and STORE rather than obscure binary codes like 0000, 0011, and 0001. Figure 6.5 shows an assembly language instruction set for a Von

Neumann machine that has a single ALU register R and three condition codes GT, EQ, and LT. Each numeric op code, its assembly language mnemonic, and its meaning are listed. This table is identical to Figure 5.25, which summarizes the language used in Chapter 5 to introduce the Von Neumann architecture and explains how instructions are executed. (However, Chapter 5 describes binary machine language and uses symbolic names only for convenience. In this chapter, we are describing assembly language, where symbolic names such as LOAD and ADD are actually part of the language.)

Figure 6.5

Typical assembly language instruction set

Binary Op Code	Operation	Meaning
0000	LOAD X	CON(X) → R
0001	STORE X	R → CON(X)
0010	CLEAR X	0 → CON(X)
0011	ADD X	R + CON(X) → R
0100	INCREMENT X	CON(X) + 1 → CON(X)
0101	SUBTRACT X	R – CON(X) → R
0110	DECREMENT X	CON(X) – 1 → CON(X)
0111	COMPARE X	if CON(X) > R then else 0 if then else 0 if CON(X) < R then else 0
1000	JUMP X	Get the next instruction from memory location X.
1001	JUMPGT X	Get the next instruction from memory location X if .
1010	JUMPEQ X	Get the next instruction from memory location X if .
1011	JUMPLT X	Get the next instruction from memory location X if .
1100	JUMPNEQ X	Get the next instruction from memory location X if .
1101	IN X	Input an integer value from the standard input device and store into memory cell X.
1110	OUT X	Output, in decimal notation, the value stored in memory cell X.
1111	HALT	Stop program execution.

Another advantage of assembly language is that it lets programmers use *symbolic addresses* instead of numeric addresses. In machine language, to jump to the instruction stored in memory location 17, you must refer directly to address 17; that is, you must write JUMP 17 (in binary, of course). This is cumbersome, because if a new instruction is inserted anywhere within the first 17 lines of the program, the jump location changes to 18. The old reference to 17 is incorrect, and the address field must be changed. This makes modifying programs very difficult, and even small changes become big efforts. It is not unlike identifying yourself in a waiting line by position—as, say, the 10th person

in line. As soon as someone in front of you leaves (or someone cuts in line ahead of you), that number changes. It is far better to identify yourself using a characteristic that does not change as people enter or exit the line. For example, you are the person wearing the green jacket and the orange shirt. Those characteristics won't change (though maybe they should).

In assembly language, we can attach a *symbolic label* to any instruction or piece of data in the program. The label then becomes a permanent identification for this instruction or data, regardless of where it appears in the program or where it may be moved in memory. A label is a name (followed by a colon to identify it as a label) placed at the beginning of an instruction.

```
LOOPSTART:    LOAD X
```

The label LOOPSTART has been attached to the instruction LOAD X. This means that the name LOOPSTART is *equivalent* to the address of the memory cell that holds the instruction LOAD X. If, for example, the LOAD X instruction is stored in memory cell 62, then the name LOOPSTART is equivalent to address 62. Any use of the name LOOPSTART in the address field of an instruction is treated as though the user had written the numeric address 62. For example, to jump to the load instruction shown above, we do not need to know that it is stored in location 62. Instead, we need only write the instruction

```
JUMP    LOOPSTART
```

Symbolic labels have two advantages over numeric addresses. The first is *program clarity*. As with the use of mnemonics for op codes, the use of meaningful symbolic names can make a program much more readable. Names like LOOPSTART, COUNT, and ERROR carry a good deal of meaning and help people to understand what the code is doing. Memory addresses such as 73, 147, and 2001 do not. A second advantage of symbolic labels is *maintainability*. When we refer to an instruction via a symbolic label rather than an address, we no longer need to modify the address field when instructions are added to or removed from the program. Consider the following example:

```
        JUMP    LOOP
        ⋮       ← point A
LOOP:   LOAD    X
```

Say a new instruction is added to the program at point A. When the modified program is translated by the assembler into machine language, all instructions following point A are placed in a memory cell whose address is 1 higher than it was before (assuming that each instruction occupies one memory cell). However, the JUMP refers to the LOAD instruction only by the name LOOP, not by the address where it is stored. Therefore, neither the JUMP nor the LOAD instruction needs to be changed. We need only retranslate the modified program. The assembler determines the new address of the LOAD X instruction, makes the label LOOP equivalent to this new address, and places this new address into the address field of the JUMP LOOP instruction. The assembler does the messy bookkeeping previously done by the machine language programmer.

The final advantage of assembly language programming is *data generation*. In Section 4.2.1 we showed how to represent data types such as unsigned and signed integers, floating-point values, and characters in binary. When writing in machine language, the programmer must do these conversions. In assembly language, however, the programmer can ask the assembler to do them by using a special type of assembly language op code called a **pseudo-op**.

A pseudo-op (preceded in our notation by a period to indicate its type) does not generate a machine language instruction like other operation codes. Instead, it invokes a useful service of the assembler. One of these useful services is generating data in the proper binary representation for this system. There are typically assembly language pseudo-ops to generate integer, character, and (if the hardware supports it) real data values. In our sample language, we will limit ourselves to one data generation pseudo-op called .DATA that builds signed integers. This pseudo-op converts the signed decimal integer in the address field to the proper binary representation. For example, the pseudo-op

```
FIVE:    .DATA    +5
```

tells the assembler to generate the binary representation for the signed integer +5, put it into memory, and make the label "FIVE" equivalent to the address of that cell. If a memory cell contains 16 bits, and the next available memory cell is address 53, then this pseudo-op produces

address	*contents*
53	0000000000000101

and the name FIVE is equivalent to memory address 53. Similarly, the pseudo-op

```
NEGSEVEN: .DATA   −7
```

might produce the following 16-bit quantity, assuming sign/magnitude representation:

address	*contents*
54	1000000000000111

and the symbol NEGSEVEN is equivalent to memory address 54.

We can now refer to these data items by their attached label. For example, to load the value +5 into register R, we can say

```
LOAD     FIVE
```

This is equivalent to writing LOAD 53, which loads register R with the contents of memory cell 53—that is, the integer +5. Note that if we had incorrectly said

```
LOAD     5
```

the *contents* of memory cell 5 would be loaded into register R. This is not what we intended, and the program would be wrong. This is a good example of why it is so important to distinguish between the address of a cell and its contents.

To add the value 27 to the current contents of register R, we write

```
ADD     NEGSEVEN
```

The contents of R (currently +5) and the contents of address NEGSEVEN (address 54, whose contents are −7) are added together, producing −2. This becomes the new contents of register R.

When generating and storing data values, we must be careful not to place them in memory locations where they can be misinterpreted as instructions. In Chapter 4, we said that the only way a computer can tell that the binary value 01000001 represents the letter A rather than the decimal value 65 is by the context in which it appears. The same is true for instructions and data. They are indistinguishable from each other, and the only way a Von Neumann machine can determine whether a sequence of 0s and 1s is an instruction or a piece of data is by how we use it. If we attempt to execute a value stored in memory, then that value *becomes* an instruction whether we meant it to be or not.

For example, if we incorrectly have the following sequence in our program:

```
LOAD        X
.DATA       +1
```

then, after completing the execution of the LOAD X command, the processor will come to the memory location where the .DATA pseudo-op has stored the value +1. The processor will fetch, decode, and attempt to execute the data value +1 as if it were an "instruction." This might sound meaningless, but to a processor, it is not. The representation of +1, using 16 bits, is

```
0000000000000001
```

Because this value is being used as an instruction, some of the bits will be interpreted as the op code and some as the address field. If we assume a 16-bit, one-address instruction format, with the first 4 bits being the op code and the last 12 bits being the address field, then these 16 bits will be interpreted as follows:

```
0000            000000000001
```

op code *address*

The "op code" is 0, which is a LOAD on our hypothetical machine (see Figure 6.5), and the "address field" contains a 1. Thus, the data value 11 has accidentally turned into the following instruction:

```
LOAD    1       --Load the contents of memory cell 1 into register R
```

This is obviously incorrect, but how is the problem solved? The easiest way is to remember to place all data created by the program using the .DATA pseudo-op in memory locations where they cannot possibly be misinterpreted as instructions and accidently executed. One convenient place that meets this criterion is after a HALT instruction because the HALT prevents any further execution. The data values can be referenced, but they cannot be executed.

A second service provided by pseudo-ops is *program construction*. Pseudo-ops that mark the beginning (.BEGIN) and end (.END) of the assembly language program specify where to start and stop the translation process, and they do not generate any instructions or data. Remember that it is the HALT instruction, not the .END pseudo-op, that terminates execution of the program. The .END pseudo-op ends the translation process. Figure 6.6, which shows the organization of a typical assembly language program, helps explain this distinction.

Figure 6.6 Structure of a typical assembly language program

Practice Problems

Assume that register R and memory cells 80 and 81 contain the following values:

R: 20 memory cell 80: 43 memory cell 81: 97

Using the instruction set shown in Figure 6.5, determine what value ends up in register R and memory cells 80 and 81 after each of the following instructions is executed. Assume that each question begins with the values shown above.

```
LOAD 80
STORE 81
COMPARE 80
ADD 81
IN 80
OUT 81
```

Answer

Assume that memory cell 50 contains a 4 and label L is equivalent to memory location 50. What value does each of the following LOAD instructions load into register R?

```
LOAD 50
LOAD 4
LOAD L
LOAD L+1 (Assume that this is legal.)
```

Answer

Explain why both the HALT operation code described in Figure 6.5 and the .END pseudo-op mentioned at the end of this section are needed in an assembly language program and what might happen if one or both were omitted.

Answer

The HALT operation tells the CPU to stop program execution. If the program is organized as in Figure 6.6, then without the HALT instruction the CPU will fetch the data value stored in the next memory location after the last instruction and attempt to execute it. The .END pseudo-op tells the assembler to stop the translation process. The assembler is a piece of software that is acting on the source code, loaded into memory, as its “data”; without the .END pseudo-op, the assembler will try to translate whatever might be stored in memory after the last legitimate source code statement.

Explain exactly what would occur if a processor tried to execute the following pair of instructions:

```
        LOAD        L
L:      .DATA       1
```

6. Answer
7. The first instruction will cause the memory location labeled L to be loaded into register R. Because L contains the data value +1, this will go into R, overwriting whatever was there previously. After completing one instruction, a processor will go on to the next one unless told to do otherwise— that is the essence of the Fetch/Decode/Execute cycle. Thus, the processor will next try to execute the "instruction" +1. As we explained in the text, this will be incorrectly interpreted as the op code 0 and address field of 1, which is a LOAD 1. Thus, the value +1 in register R will be overwritten with the contents of memory location 1.

Assume machine language instructions occupy 16 bits, with the first four bits holding the op code (as given in Figure 6.5) and the final 12 bits holding the address of the operand. Also, assume that *X* corresponds to memory address 20 and *Y* corresponds to memory address 31. Show the internal binary representation of the following assembly language instructions:

SUBTRACT X

Answer

The SUBTRACT op code is 0101. The binary representation for the unsigned integer 20 using 12 bits is 000000010100. Putting this together results in:

LOAD Y

Answer

The LOAD op code is 0000. The binary representation for the unsigned integer 31 using 12 bits is 000000011111. Putting this together results in:

HALT

Answer

The HALT op code is 1111. HALT does not require an address but something has to be put into those 12 bits. Typically, an assembler will set all the bits to 0, although it does not really matter. Assuming we put zeroes into the address field we will end up with:

Change font sizeMain content

Chapter Contents

6.3.2 Examples of Assembly Language Code

This section describes how to use assembly language to translate algorithms into programs that can be executed on a Von Neumann computer. Today, software development is rarely performed in assembly language except for very special-purpose tasks; most programmers use higher-level languages such as those mentioned in Figure 6.3 and described in Chapters 9 and 10. Our purpose in offering these examples is to demonstrate how system software, in this case an assembler, can create a user-oriented virtual environment that supports effective and productive problem solving.

One of the most common operations in any algorithm is the evaluation of arithmetic expressions. For example, the sequential search algorithm of Figure 2.13 contains the following arithmetic operations:

Set the value of *i* to 1 (Line 2).

⋮

Add 1 to the value of *i* (Line 7).

These algorithmic operations can be translated quite easily into assembly language as follows:

```
        LOAD    ONE --Put a 1 into register R
        STORE   I   --Store the constant 1 into I
        ⋮
        INCREMENT I --Add 1 to the contents of memory location I
        ⋮           --The following .DATA statements should be placed after the HALT
                    instruction
I:      .DATA   0   --The memory location holding the variable I, initially set to 0
ONE:    .DATA   1   --The integer constant 1
```

Note how readable this code is, compared with machine language, because of such op code mnemonics as LOAD and STORE and the use of descriptive labels such as I and ONE.

Another example is the following assembly language translation of the arithmetic expression . (Assume that *B* and *C* have already been assigned values.)

```
        LOAD    B       --Put the value B into register R
        ADD     C       --R now holds the sum (B + C)
        SUBTRACT SEVEN  --R now holds (B + C − 7)
        STORE   A       --Store the result into A
        ⋮               --The following .DATA statements should be placed after the
                        HALT instruction
A:      .DATA   0
B:      .DATA   0
C:      .DATA   0
SEVEN:  .DATA   7       --The integer constant 7
```

Another important algorithmic operation involves testing and comparing values. The comparison of values and the subsequent use of the outcome to decide what to do next are termed a conditional operation, which we first saw in Section 2.2.3. Here is

a *conditional* operation that outputs the larger of two values *x* and *y*. Algorithmically, it is expressed as follows:

In assembly language, this conditional operation can be translated as follows:

```
        IN      X       --Read the first data value
        IN      Y       --and now the second
        LOAD    X       --Load the value of X into register R
        COMPARE Y       --Compare Y to X and set the condition codes according to the
                        outcome of the comparison
        JUMPGT  DONE    --If Y is greater than X, go to DONE
        OUT     X       --We get here only if X ≥ Y, so print X
        JUMP    NEXT    --Skip over the next instruction and continue
DONE:   OUT     Y       --We get here if Y > X, so print Y
        ⋮
NEXT:   ⋮               --The program continues here
                        --The following .DATA statements go after the HALT instruction
X:      .DATA   0       --Space for the two data values
Y:      .DATA   0
        ⋮
```

Another important algorithmic primitive is *looping*, which was also introduced in Section 2.2.3. The following algorithmic example contains a while loop that executes 10,000 times.

This looping construct is easily translated into assembly language.

```
        LOAD    ZERO    --Initialize the loop counter to 0
        STORE   I       --This is Step 1 of the algorithm
LOOP:   LOAD    MAXVALUE--Put the integer value 10,000 into
                        register R
        COMPARE I       --Compare I against 10,000
        JUMPEQ  DONE    --If  we are done (Step 2)
        ⋮               --Here is the loop body (Steps 3–8)
        INCREMENT I     --Add 1 to I (Step 9)
```

```
            JUMP    LOOP    --End of the loop body (Step 10)
DONE:       HALT            --Stop execution (Step 11)
ZERO:       .DATA   0       --This is the constant 0
I:          .DATA   0       --The loop counter; it goes from
                            --0 to 10,000
MAXVALUE:   .DATA   10000   --Maximum number of executions
            ⋮
```

As a final example, we will show a complete assembly language program (including all necessary pseudo-ops) to solve the following problem:

Read in a sequence of nonnegative numbers, one number at a time, and compute a running sum. When you encounter a negative number, print out the sum of the nonnegative values and stop.

Thus, if the input is

```
8
31
7
5
−1
```

then the program should output the value 51, which is the sum (8 + 31 + 7 + 5). An algorithm to solve this problem is shown in Figure 6.7, using the pseudocode notation of Chapter 2.

Figure 6.7 Algorithm to compute the sum of nonnegative numbers

Our next task is to convert the algorithmic primitives of Figure 6.7 into assembly language instructions. A program that does this is shown in Figure 6.8.

Figure 6.8

Assembly language program to compute the sum of nonnegative numbers

```
            .BEGIN          --This marks the start of the program

            CLEAR   SUM     --Set the running sum to 0 (line 1)

            IN      N       --Input the first number N (line 2)

--The next three instructions test whether N is a negative number (line
3)
```

```
AGAIN:  LOAD      ZERO    --Put 0 into register R
        COMPARE   N       --Compare N and 0
        JUMPLT    NEG     --Go to NEG if N < 0
--We get here if N ≥ 0. We add N to the running sum (line 4)
        LOAD      SUM     --Put SUM into R
        ADD       N       --Add N. R now holds (N + SUM)
        STORE     SUM     --Put the result back into SUM
--Get the next input value (line 5)
        IN        N
--Now go back and repeat the loop (line 6)
        JUMP      AGAIN
--We get to this section of the program only when we encounter a
negative value
NEG:    OUT       SUM     --Print the sum (line 7)
        HALT              --and stop (line 8)
--Here are the data generation pseudo-ops
SUM:    .DATA     0       --The running sum goes here
N:      .DATA     0       --The input data are placed here
ZERO:   .DATA     0       --The constant 0
--Now we mark the end of the entire program
        .END
```

Of all the examples in this chapter, the program in Figure 6.8 demonstrates best what is meant by the phrase *user-oriented virtual environment*. Although it is not as clear as natural language or the pseudocode of Figure 6.7, this program can be read and understood by humans as well as computers. Tasks such as modifying the program and

locating an error would be significantly easier using the assembly language code of Figure 6.8 than on its machine language equivalent.

The program in Figure 6.8 is an important milestone in our discussion of computer science in that it represents a culmination of the algorithmic problem-solving process. Earlier chapters introduced algorithms and problem solving (Chapters 1, 2, 3), discussed how to build computers to execute algorithms (Chapters 4, 5), and introduced system software that enables us to code algorithms into a language that computers can translate and execute (Chapter 6). The program in Figure 6.8 is the end product of this discussion: This program can be input to an assembler, translated into machine language, loaded into a Von Neumann computer, and executed to produce answers to our problem. This **algorithmic problem-solving cycle** is one of the central themes of computer science.

Practice Problems

Using the instruction set in Figure 6.5, translate the following algorithmic operations into assembly code. Show all necessary .DATA pseudo-ops.

Add 1 to the value of *x*

Answer

Another way to do the same thing is

However, the first way is much more efficient. It takes two fewer instructions and one fewer DATA pseudo-op.

Add 50 to the value of *x*

Answer

Set *x* to the value $y + z - 2$

Answer

If $x > 50$ then output the value of x, otherwise input a new value of *x*

Answer

Answer

Using the instruction set in Figure 6.5, write a complete assembly language program (including all necessary pseudo-ops) that reads in numbers and counts how many nonnegative inputs it reads in until it encounters the first negative value. It then outputs that count and stops. For example, if the input data is 42, 108, 99, 60, 1, 42, 3, −27, then your program outputs the value 7 because there are seven nonnegative values before the appearance of the negative value −27.

Answer

Now modify your program from Practice Problem 2 so that if you have not encountered a negative value after 100 inputs, your program stops and outputs the value 100.

Answer

Discuss how the modifications you had to make in Practice Problem 3 were made easier by the use of assembly language in place of binary machine language.

Answer

The most important thing was that you were looking at a program that contained English words and familiar mathematical terminology. That made it much easier to understand what was going on and where the modifications were to be made. Also, the use of labels allowed you to make changes in the program without having to make changes to binary memory addresses.

Change font sizeMain content

Chapter Contents

6.3.3 Translation and Loading

What must happen in order for the assembly language program in Figure 6.8 to be executed on a processor? Figure 6.4 shows that before our source program can be run, we must invoke two system software packages—an *assembler* and a *loader*.

An assembler translates a symbolic assembly language program, such as the one in Figure 6.8, into machine language. We usually think of translation as an extremely difficult task. In fact, if two languages differ greatly in vocabulary, grammar, and syntax, it can be quite formidable. (This is why a translator for a high-level programming language is a very complex piece of software.) However, machine language and assembly language are very similar, and therefore an assembler is a relatively simple piece of system software. Understanding how an assembler works will give you a good appreciation for the tasks that system software must carry out in order to create a user-friendly virtual environment.

An assembler must perform the following four tasks, none of which is particularly difficult.

1. Convert symbolic op codes to binary.
2. Convert symbolic addresses to binary.
3. Perform the assembler services requested by the pseudo-ops.
4. Put the translated instructions into a file for future use.
   Let's see how these operations are carried out using the hypothetical assembly language of Figure 6.5.

The conversion of symbolic op codes such as LOAD, ADD, and OUT to binary makes use of a structure called the *op code table*. This is an alphabetized list of all legal assembly language op codes and their binary equivalents. Part of an op code table for the instruction set of Figure 6.5 is shown in Figure 6.9. (The table assumes that the op code field is 4 bits wide.)

Figure 6.9

Structure of the op code table

Operation	Binary Value
ADD	0011

Operation	Binary Value
CLEAR	0010
COMPARE	0111
DECREMENT	0110
HALT	1111
OUT	1110
⋮	
STORE	0001
SUBTRACT	0101

The assembler finds the operation code mnemonic in column 1 of the table and replaces the characters with the 4-bit binary value in column 2. (If the mnemonic is not found, then the user has written an illegal op code, which results in an error message.) Thus, for example, if we use the mnemonic OUT in our program, the assembler converts it to the binary value 1110.

To look up the code in the op code table, we could use the sequential search algorithm introduced in Chapter 2 and shown in Figure 2.13. However, using this algorithm could significantly slow down the translation of our program. The analysis of the sequential search algorithm in Chapter 3 showed that locating a single item in a list of *N* items takes, on the average, *N/2* comparisons if the item is in the table and *N* comparisons if it is not. In Chapter 5, we stated that modern computers may have as many as 300 machine language instructions in their instruction set, so the size of the op code table of Figure 6.9 could be as large as . This means that using sequential search, we must perform an average of *N*/2, about 150, comparisons for every legal op code in our program. If our assembly language program contains 500,000 instructions (not an unreasonably large number for a complex piece of system software), the op code translation task requires a total of . That is a lot of searching, even for a high-speed computer.
Because the op code table of Figure 6.9 is sorted alphabetically, we can instead use the more efficient binary search algorithm discussed in Section 3.4.2 and shown in Figure 3.18. On the average, the number of comparisons needed to find an element using binary search is not *N*/2 but lg *N*, the logarithm of *N* to the base 2. [*Note*: lg *N* is the value *k* such that .] For a table of size , *N*/2 is 150, whereas (lg *N*) is approximately 8 . This says that on the average, we find an op code in the table, assuming it is there, in about 8 comparisons rather than 150. If our assembly language program contains 500,000 instructions, then the op code translation task now requires only about comparisons rather than 75 million, a reduction of 71 million lookups. By selecting a better search

algorithm, we achieve an increase in speed of about 95%—quite a significant improvement!
This example demonstrates why algorithm analysis, introduced in Chapter 3, is such a critically important part of the design and implementation of software. Replacing a slow algorithm with a faster one can turn a practically-speaking unsolvable problem into a solvable one and a worthless solution into a highly worthwhile one. Remember that, in computer science, we are looking not just for correct solutions but for efficient ones as well.

After the op code has been converted into binary, the assembler must perform a similar task on the address field. It must convert the address from a symbolic value, such as X or LOOP, into the correct binary address. This task is a bit more difficult than converting the op code because the assembler itself must determine the correct numeric value of all symbols used in the label field. There is no "built-in" address conversion table equivalent to the op code table of Figure 6.9.

In assembly language, a symbol is defined when it appears in the label field of an instruction or data pseudo-op. Specifically, the symbol is given the value of the address of the instruction to which it is attached. Assemblers usually make two passes over the source code, where a **pass** is defined as the process of examining and processing every assembly language instruction in the program, one instruction at a time. During the *first pass* over the source code, the assembler looks at every instruction, keeping track of the memory address where this instruction will be stored when it is translated and loaded into memory. It does this by knowing where the program begins in memory and knowing how many memory cells are required to store each machine language instruction or piece of data. It also determines whether there is a symbol in the label field of the instruction. If there is, it enters the symbol and the address of this instruction into a special table that it is building called a **symbol table**.

We can see this process more clearly in Figure 6.10. The figure assumes that each instruction and data value occupies one memory cell and that the first instruction of the program will be placed into address 0.

Figure 6.10Generation of the symbol table

The assembler looks at the first instruction in the program, IN X, and determines that when this instruction is loaded into memory, it will go into memory cell 0. Because the label LOOP is attached to that instruction, the name LOOP is made equivalent to address 0. The assembler enters the (name, value) pair (LOOP, 0) into the symbol table. This process of associating a symbolic name with a physical memory address is called **binding**, and the two primary purposes of the first pass of an assembler are

- (1)

  to bind all symbolic names to address values and
- (2)

  to enter those bindings into the symbol table.

Now, any time the programmer uses the name LOOP in the address field, the assembler can look up that symbol in column 1 of the symbol table and replace it with the address

value in column 2, in this case address 0. (If it is not found, the programmer has used an undefined symbol, which produces an error message.)

The next six instructions of Figure 6.10(a), from IN Y to JUMP LOOP, do not contain labels, so they do not add new entries to the symbol table. However, the assembler must still update the counter it is using to determine the address where each instruction will ultimately be stored. The variable used to determine the address of a given instruction or piece of data is called the *location counter*. The location counter values are shown in the third column of Figure 6.10(a). Using the location counter, the assembler can determine that the address values of the labels DONE, X, and Y are 7, 9, and 10, respectively. It binds these symbolic names and addresses and enters them in the symbol table, as shown in Figure 6.10(b). When the first pass is done, the assembler has constructed a symbol table that it can use during pass 2. The algorithm for pass 1 of a typical assembler is shown (using an alternative form of algorithmic notation called a *flowchart*) in Figure 6.11.

Figure 6.11 Outline of pass 1 of the assembler

During the *second pass*, the assembler translates the source program into machine language. It has the op code table to translate mnemonic op codes to binary, and it has the symbol table to translate symbolic addresses to binary. Therefore, the second pass is relatively simple, involving two table lookups and the generation of two binary fields. For example, if we assume that our instruction format is a 4-bit op code followed by a single 12-bit address, then given the instruction

```
OUT     X
```

the assembler

1. looks up OUT in the op code table of Figure 6.9 and places the 4-bit binary value 1110 in the op code field
2. looks up the symbol X in the symbol table of Figure 6.10(b) and places the binary address value 0000 0000 1001 (decimal 9) into the address field
   After these two steps, the assembler produces the 16-bit instruction

```
1110 0000 0000 1001
```

which is the correct machine language equivalent of the assembly language statement

```
OUT     X.
```

When it is done with one instruction, the assembler moves on to the next and translates it in the same fashion. This continues until it sees the pseudo-op .END, which terminates translation.

The other responsibilities of pass 2 are also relatively simple:

- Handling data generation pseudo-ops (only .DATA in our example)
- Producing the object file needed by the loader

The .DATA pseudo-op asks the assembler to build the proper binary representation for the signed decimal integer in the address field. To do this, the assembler must implement the sign/magnitude integer representation algorithms described in Section 4.2.

Finally, after all the fields of an instruction have been translated into binary, the newly built machine language instruction and the address of where it is to be loaded are written to a file called the **object file**. (On Windows machines, this is referred to as an .exe file, which stands for "executable program.") The algorithm for pass 2 of the assembler is shown in Figure 6.12.

Figure 6.12 Outline of pass 2 of the assembler

After completion of pass 1 and pass 2, the object file contains the translated machine language object program, referred to in Figure 6.4. One possible object program for the assembly language program of Figure 6.10(a) is shown in Figure 6.13. (Note that a real object file contains only the address and instruction fields. The meaning field is included here for clarity only.)

Figure 6.13 Example of an object program

The object program shown in Figure 6.13 becomes input to yet another piece of system software called a **loader**. It is the task of the loader to read instructions from the object file and store them into memory for execution. To do this, it reads an address value—column 1 of Figure 6.13—and a machine language instruction—column 2 of Figure 6.13—and stores that instruction into the specified memory address. This operation is repeated for every instruction in the object file. When loading is complete, the loader places the address of the first instruction (0 in this example) into the program counter (PC) to initiate execution. The hardware, as we learned in Chapter 5, then begins the fetch, decode, and execute cycle starting with the instruction whose address is located in the PC, namely, the beginning of this program.

Practice Problems

Translate the following algorithm into assembly language using the instructions in Figure 6.5.

Answer

What is the machine language representation of each of the following instructions? Assume the symbol table values are as shown in Figure 6.10(b) and the instruction format is that of Figure 6.13.

Answer

a.

b.

c.

What is wrong or inconsistent with the instruction that is shown in Practice Problem 2c?

Answer

LOOP is the address of an instruction (IN X), but decrement is treating this instruction as though it were a piece of data and subtracting 1 from it. Thus, what this instruction is doing is "computing" (IN X) - 1, which is meaningless. However, the computer will be very happy to carry out this meaningless operation.

Take the assembly language program that you developed in Practice Problem 1 and determine the physical memory address associated with each label in the symbol table. (Assume the first instruction is loaded into address 0 and that each instruction occupies one cell.)

Answer

The address values that you come up with will depend entirely on your solution. The symbol table for the program in Practice Problem 1 would look like the following example. (*Note*: The solution assumes that each instruction occupies one memory location.)

Is the following assembly language pseudo-op illegal? Explain why or why not.

```
TWO:    DATA    1
```

Answer

No, it is not illegal, just highly misleading. You can label a piece of data with any name you want, but you should pick something that is helpful to the people who have to read and modify the program at some future time. By labeling the constant +1 with the symbolic name TWO, you will confuse and mislead the people looking at your code.

Change font sizeMain content

Chapter Contents

6.4 Operating Systems

To carry out the services just described (translate a program, load a program, run a program), a user must issue **system commands**, which are commands sent to the operating system to perform a service on the user's behalf. In earlier times these commands were lines of text typed at a terminal, such as

```
>assemble MyProg     (Invoke the assembler to translate a program called MyProg.)

>run MyProg          (Load the translated MyProg and start execution.)
```

Today, however, these services are invoked using icons displayed on a screen and selected with a mouse, button, finger tap, or voice command.

Regardless of how the process is initiated, the important question is: Which program examines these commands? Which piece of system software waits for requests from a user and activates other system programs like a translator or information manager to service these requests? The answer is the operating system, and, as shown in Figure 6.2, it is the "top-level" system software component on a computer.

Some of the more well-known operating systems in widespread use today include Windows 10, macOS, and Linux for mainframes, desktops, and laptops, and Google Android and Apple iOS for mobile devices.

6.4.1 Functions of an Operating System

An operating system is an enormously large and complex piece of software that has many responsibilities within a computer system. This section examines five of the most important tasks that it performs.

The User Interface. The operating system is executing whenever no other piece of user or system software is using the processor. Its most important task is to wait for a user command delivered via a keypad, mouse, finger tap, voice command, or other input device. If the command is legal, the operating system activates and schedules the appropriate software package to process the request. In this sense, the operating system acts like the computer's *receptionist* and *dispatcher*.

Operating system commands usually request access to hardware resources (processor, output device, communication line, camera), software services (web browser, application program), or information (data files, contact lists). Examples of typical operating system commands are shown in Figure 6.14. Modern operating systems can typically recognize and execute hundreds of unique commands.

Figure 6.14

Some typical operating system commands

Load a program into memory
Run a program
Save information in a file or a directory
Retrieve a file previously stored on the local machine or in the cloud
List all the files for this user
Delete or rename a file
Display a file on a specified device
Copy a file from one device to another
Establish a network connection
Let the user set or change one of the current machine settings
Tell how much memory or data storage is currently in use
Put the device to sleep or turn it off

After a user enters a command, the operating system determines which software package needs to be loaded and put on the schedule for execution. When that package completes execution, control returns to the operating system, which waits for a user to enter the next command. This user interface algorithm is diagrammed in Figure 6.15.

Figure 6.15 User interface responsibility of the operating system

The user interfaces on the operating systems of the 1950s, 1960s, and 1970s were text oriented. The system displayed a *prompt character* on the screen to indicate that it was waiting for input, and then it waited for something to happen. The user entered commands in a special, and sometimes quite cryptic, **command language**. For example, on the UNIX operating system, widely used on personal computers and workstations, the following command asks the system to list the names and access privileges of the files contained in the home directory of a user called mike:

```
> ls -al/usr/mike/home        (">" is the prompt character)
```

As you can see, commands were not always easy to understand, and learning the command language of the operating system was a major stumbling block for new users. Unfortunately, without first learning some basic commands, no useful work could be done.

Because users found text-oriented command languages very cumbersome, all modern operating systems utilize a graphical user interface, (GUI). To communicate with a user, a GUI supports visual aids and point-and-click or touchscreen operations, rather than textual commands. These interfaces use *icons, pull-down menus, scrolling, resizable windows*, and other visual elements and graphical metaphors that make it much easier for a user to formulate requests. Operating systems for mobile devices such as tablets and smartphones allow users to employ finger taps and voice-activated commands to specify the operations they wish to perform.

Graphical, touchscreen, and voice-activated user interfaces are excellent examples of the high-level virtual machine created by the operating system. They hide the complexity of the underlying hardware and software and make your computer, tablet, or smartphone appear very easy to use. In reality, the processor inside is identical to the horribly complicated "naked" Von Neumann machine described in Chapters 4 and 5.

A Machine for the Rest of Us

In January 1984, Apple Computer launched its new line of Macintosh computers with a great deal of showmanship: a TV commercial at the 1984 NFL Super Bowl. The company described the Macintosh as a computer that anyone could understand and use—"a machine for the rest of us." People who saw and used it quickly agreed, and in the early days, its major selling point was that "a Macintosh is much easier to use than an IBM PC." However, the Macintosh and IBM PC were extremely similar in terms of hardware. Both systems used Von Neumann–type processors, and these processors executed similar sets of machine language instructions exactly as described in Chapter 5. (In fact, in 2006 Apple began using the same type of Intel processors as in the IBM PC and its clones.) It certainly was not the underlying hardware that created these huge differences in ease of use.

What made the Macintosh easier to use was its radically new graphical user interface that most users found much easier to understand than the text-oriented interface of MS-DOS, the most popular PC-based operating system at that time. IBM users quickly realized the importance of having a powerful visual interface, and in the early- and mid-1990s began to switch to Microsoft Windows, which provided a windowing environment similar to the Macintosh. Newer versions of these systems, such as macOS and Windows 10, represent attempts at creating an even more powerful and easy-to-use virtual environment.

We can see now that it was wrong for Apple to say that "a Macintosh is easier to use than a PC." What it should have said is that "the virtual machine environment created by the Macintosh operating system is

easier to use than the virtual machine environment created by the MS-DOS operating system." However, maybe that was just a little too wordy!

System Security and Protection. In addition to being a receptionist, the operating system also has the responsibilities of a *security guard*—controlling access to the computer and its resources. It must prevent unauthorized users from accessing the system and prevent authorized users from doing unauthorized things.

At a minimum, the operating system must not allow people to access the computer if they have not been granted permission. In the "olden days" of computing (the 1950s and 1960s), security was implemented by physical means—walls and locked doors around the computer and security guards at the door to prevent unauthorized access. However, when telecommunications networks appeared on the scene in the late 1960s and 1970s (we will discuss them in detail in Chapter 7), access to computers over networks became possible from virtually anywhere in the world, and responsibility for access control migrated from the guard at the door to the operating system inside the machine.

In most operating systems, access control means requiring a user to enter a legal *username* and *password* before any other requests are accepted. If an incorrect username or password is entered, the operating system does not allow access to the computer.

It is also the operating system's responsibility to safeguard the *password file* that stores all valid username/password combinations. It must prevent this file from being accessed by any unauthorized users because that would compromise the security of the entire system. This is analogous to putting a lock on your door but also making sure that you don't lose the key. (Of course, some privileged users, called *superusers*—usually computer center employees or system administrators—must be able to access and maintain this file.) To provide this security, the operating system may choose to *encrypt* the password file using an encoding algorithm that is extremely difficult to crack. Operating systems use encryption algorithms whenever they must provide a high degree of security for sensitive information. We will learn more about these encryption algorithms in Chapter 8.

Even when valid users gain access to the system, there are operations they should not be allowed to do. For example, although users can delete any personal application programs on their smartphones, they are not allowed to delete system applications such as "Change Settings," "Notes," and "Calendar." On multiuser systems, users are allowed to access only their own personal information. They should not be able to look at the files of other users. Therefore, if the operating system gets a request such as

> open filename (Open a file and allow this user to access it.)

(Or click Open in the File menu.)

it must determine who is the owner of the file—that is, who created it. If the individual accessing the file is not the owner, then the operating system usually rejects the request.

Most modern operating systems not only determine whether you are allowed to access a file, they also check what operations you are permitted to perform on that file by keeping an *access control list*.

For example, the grade file GRADES of a student named Smith could have the access control list shown in Figure 6.16. This access control list says that Smith, the student whose grades are in the file, has the right to access his or her own file, but only to read the information. Jones, a clerk in the administration center, can read the file and can append new grades to the end of the file at the completion of the term. Adams, the school's registrar, can read and append information and also is allowed to change the student's grades if an error was made. Doe, the director of the computer center, can do all of these operations as well as delete the file and all its information.

Figure 6.16 Access control list for the file GRADES

Permission to look at information can be given to a number of people. However, changing information in a file is a sensitive operation (think about changing a payroll file), and permission to make changes must be limited. Deleting information (such as smartphone system programs like "Settings") is the most powerful and potentially damaging operation of all, and its use must be restricted to people at the highest level. It is the operating system's responsibility to help ensure that individuals are authorized to carry out the operation they request.

Computers today play such a critical role in the storage and management of military, medical, economic, social, and personal data that this security responsibility of the operating system has taken on an increasingly important role. We will investigate this topic in more detail in Chapter 8.

Efficient Management of Resources. Section 5.2.2 described the enormous difference in speed between a processor and an I/O unit: up to 5 orders of magnitude. A hardware device called an I/O controller (Figure 5.15) frees the processor to do useful work while the I/O operation is being completed. What useful work can a processor do in this free time? What ensures that this valuable resource is used efficiently? Again, it is the operating system's responsibility to see that the resources of a computer system are used efficiently as well as correctly.

To ensure that a processor does not sit idle if there is useful work to do, the operating system keeps a *queue* (a waiting line) of programs that are ready to run. Whenever the processor is idle, the operating system picks one of these jobs and assigns it to the processor. This guarantees that the processor always has something to do.

To see how this algorithm might work, let's define the following three classes of programs:

Running The one program currently executing on the processor (assume only a single processor on the computer)

Ready Programs that are loaded in memory and ready to run but are not yet executing

Waiting Programs that cannot run because they are waiting for an operation to complete or some special event, such as an incoming phone call, to occur

Here is how these three lists might look at some instant in time:

Waiting	*Ready*	*Running*
	B	A
	C	
	D	

There are four programs, called A, B, C, and D, in memory. Program A is executing on the processor; B, C, and D are ready to run and are in line waiting their turn. Assume that program A performs the I/O operation "read a sector from the disk." (Maybe it is a word processor, and it needs to get another piece of the document on which you are working.) We saw in Section 5.2.2 that, relative to processing speeds, this operation takes a long time, about 10 msec or so. While it is waiting for this disk I/O operation to finish, the processor has nothing to do, and system efficiency plummets.

To solve this problem, the operating system can do some shuffling. It first moves program A to the waiting list because it must wait for its I/O operation to finish before it can continue. It then selects one of the ready programs (say B) and assigns it to the processor, which starts executing it. This leads to the following situation:

Waiting	*Ready*	*Running*
A	C	B
	D	

Instead of sitting idle while A waits for I/O, the processor works on program B and gets something useful done. This is equivalent to working on another project while a large document is printing, instead of waiting and doing nothing.

Perhaps B also does an I/O operation, such as asking the user to input a piece of data. If so, then the operating system repeats the same steps. It moves B to the waiting list, picks any ready program (say C) and starts executing it, producing the following situation:

Waiting	*Ready*	*Running*
A	D	C
B		

As long as there is at least one program that is ready to run, the processor always has something useful to do.

At some point, the I/O operation that A started finishes, and the "I/O completed interrupt signal" described in Section 5.2.2 is generated. The appearance of that signal indicates that program A is now ready to run, but it cannot do so immediately because the processor is currently assigned to C. Instead, the operating system moves A to the ready list, producing the following situation:

Waiting	*Ready*	*Running*
B	D	C
	A	

Programs cycle from running to waiting to ready and back to running, each one using only a portion of the resources of the processor. (However, there are situations in which a program must be started immediately, ahead of other programs on the waiting list. For example, when a phone call arrives we immediately suspend whatever we are doing and execute the program that displays an "Incoming Call" message and allows us to accept the call. If we do not do that, the caller will most likely hang up.)

In Chapter 5, we stated that the execution of a program is an unbroken repetition of the fetch/decode/execute cycle from the first instruction of the program to the HALT. Now we see that this view may not be completely accurate. For reasons of efficiency or priority, the history of a program may be a sequence of starts and stops—a cycle of execution, waits for I/O operations, waits for the processor, followed again by execution. By having many programs loaded in memory and sharing the processor, the operating system can manage the processor to its fullest capability and run the overall system more efficiently.

The Safe Use of Resources. Not only must resources be used *efficiently*, they must also be used *safely*. That doesn't mean an operating system must prevent users from sticking their fingers in the power supply and getting electrocuted! The job of the operating system is to prevent programs or users from attempting operations that cause the computer system to enter a state in which it is incapable of doing any further work—a "frozen" state where all useful work comes to a grinding halt.

To see how this can happen, imagine a computer system that has one laser printer, one data file called D, and two programs A and B. Program A wants to load data file D and print it on the laser printer. Program B wants to do the same thing. Each of them makes the following requests to the operating system:

Program A	***Program B***
Get data file D.	Get the laser printer.
Get the laser printer.	Get data file D.
Print the file.	Print the file.

Practice Problems

Assume that programs spend about 25% of their time waiting for I/O operations to complete. If there are two programs loaded into memory, what is the likelihood that both programs will be blocked waiting for I/O and there will be nothing for the processor to do? What percentage of time will the processor be busy? (This value is called *processor utilization*.) By how much does processor utilization improve if we have four programs in memory instead of two?

Answer

If there is one chance in four that a program is blocked waiting for input/output, then there is a chance in 16 that both of the two programs in memory are simultaneously blocked waiting for I/O. Therefore, the processor will be busy 15/16, or about 94%, of the time. This is the processor utilization. If we increase the number of programs in memory to four, then the probability that all four of these programs are blocked at the same time waiting for I/O is chance in 256. Now the utilization of the processor is

255/256, or about 99.6%. We can see clearly now why it is helpful to have more programs in memory. It increases the likelihood that at least one program will always be ready to run.
Why are passwords extremely vulnerable to security breaches? Suggest ways to improve their use and reduce the risk associated with them.

Answer

Passwords are not secure because people often write them down on a piece of paper put in plain sight for all to see. Furthermore, when choosing passwords people will often pick something that can be easily guessed, like their child's name or birthday. You can reduce the risk of using passwords by requiring passwords to (1) be at least 8–10 characters in length, (2) include at least one special character like /$*&, and (3) not be something found in a dictionary. You can also use *personal information* rather than a memorized password. Personal information is a fact that you would know without having to write it down, but which an acquaintance or coworker would likely not know—maybe the name of your first pet or your maternal grandfather's first name.

If the operating system satisfies the first request of each program, then A "owns" data file D, and B has the laser printer. When A requests ownership of the laser printer, it is told that the printer is being used by B. Similarly, B is told that it must wait for the data file until A is finished with it. Each program is waiting for a resource to become available that will never become free. This situation is called a **deadlock**. Programs A and B are in a permanent waiting state, and if there is no other program ready to run, all useful work on the system ceases.

More formally, deadlock means that there is a set of programs, each of which is waiting for an event to occur before it may proceed, but that event can be caused only by another waiting program in the set. Another example is a telecommunications system in which program A sends messages to program B, which acknowledges their correct receipt. Program A cannot send another message to B until it knows that the last one has been correctly received.

Program A	***Program B***
Message →	
	← Acknowledge
Message →	
	← Acknowledge
Message →	

Suppose B now sends an acknowledgment, but it gets lost. (Perhaps there was static on the line, or a lightning bolt jumbled the signal.) What happens? Program A stops and waits for receipt of an acknowledgment from B. Program B stops and waits for the next message from A. Deadlock! Neither side can proceed, and unless something is done, all communication between the two will cease.

How does an operating system handle deadlock conditions? There are two basic approaches, called *deadlock prevention* and *deadlock recovery*. In deadlock prevention, the operating system uses resource allocation algorithms that prevent deadlock from occurring in the first place. In the example of the two programs simultaneously requesting the laser printer and the data file, the problem is caused by the fact that each

program has a portion of the resources needed to solve its problem, but neither has all that it requested. To prevent this, the operating system can use the following algorithm:

If a program cannot get all the resources that it needs to solve a problem, it must give up all the resources it currently owns and issue a completely new request.

Essentially, this resource allocation algorithm says, "If you cannot get everything you need, then you get nothing." If we had used this algorithm, then after program A acquired the laser printer but not the data file, it would have had to relinquish ownership of the printer. Now B could get everything it needed to execute, and no deadlock would occur. (It could also work in the reverse direction, with B relinquishing ownership of the data file and A getting the needed resources. Which scenario unfolds depends on the exact order in which requests are made.)

In the telecommunications example, one deadlock prevention algorithm is to require that messages and acknowledgments never get garbled or lost. Unfortunately, that is impossible. Real-world communication systems (telephone, WiFi, satellite) do make errors, so we are powerless to guarantee that deadlock conditions can never occur. Instead we must detect them and recover from them when they do occur. This is typical of the class of methods called *deadlock recovery algorithms*.

For example, here is a possible algorithmic solution to our telecommunications problem:

Sender: Number your messages with the nonnegative integers 0, 1, 2, … and send them in numerical order. If you send message number *i* and have not received an acknowledgment for 30 seconds, send message *i* again.

Receiver: When you send an acknowledgment, include the number of the message you received. If you get a duplicate copy of message *i*, send another acknowledgment and discard the duplicate.

Using this algorithm, here is what might happen:

Program A	***Program B***
Message (1) →	
	← Acknowledge (1)
Message (2) →	
	← Acknowledge (2)
	(Assume this acknowledgment is lost.)

At this point, we have exactly the same deadlock condition described earlier. However, this time we are able to recover in a relatively short period. For 30 seconds nothing happens. However, after 30 seconds A sends message (2) a second time. B acknowledges it and discards it (because it already has a copy), and communication continues:

Program A	***Program B***
(Wait 30 seconds.)	
Message (2) →	(Discard this duplicate copy but acknowledge it.)

We have successfully recovered from the error, and the system is again up and running.

Regardless of whether we prevent deadlocks from occurring or recover from those that do occur, it is the responsibility of the operating system to create a virtual machine in which the user never sees deadlocks and does not worry about them. The operating system should create the illusion of a smoothly functioning, highly efficient, error-free environment—even if, as we know from our glimpse behind the scenes, that is not always the case. (We all know how frustrating it can be when our computer or tablet freezes up, and we must restart the entire system. A well-designed operating system should make this an extremely rare event.)

Summary. In this section, we have highlighted some of the major responsibilities of the critically important software package called the operating system:

- User interface management (a receptionist)
- Control of access to the system and to data files (a security guard)
- Program scheduling and activation (a dispatcher)
- Efficient resource allocation (an efficiency expert)
- Deadlock detection and error detection (a traffic officer)

These are by no means the operating system's only responsibilities, which can also include such areas as input/output processing, allocating priorities to programs, swapping programs in and out of memory, recovering from power failures, managing the system clock, and dozens of other tasks, large and small, essential to keeping the computer system running smoothly.

As you can imagine, given all these responsibilities, an operating system is an extraordinarily complex piece of software. An operating system for a large network of computers can require millions of lines of code, take thousands of person-years to develop, and cost more to develop than the hardware on which it runs. Even operating systems for smartphones and tablets (such as Android and iOS) are huge programs developed over periods of years by teams of dozens of computer scientists. Designing and creating a high-level virtual environment is a difficult job, but without it, computers would not be so widely used nor anywhere near as important as they are today.

Main content

Chapter Contents

6.4.2 Historical Overview of Operating Systems Development

Like the hardware on which it runs, system software has gone through a number of changes since the earliest days of computing. The functions and capabilities of a modern operating system described in the previous section did not appear all at once but evolved over many years.

During the *first generation* of system software (roughly 1945–1955), there really were no operating systems and there was very little software support of any kind—typically

just the assemblers and loaders described in Section 6.3. All machine operation was "hands-on." Programmers would sign up for a block of time and, at the appointed time, show up in the machine room carrying their programs on punched cards or tapes. They had the entire computer to themselves, and they were responsible for all machine operation. They loaded their assembly language programs into memory along with the assembler and, by punching some buttons on the console, started the translation process. Then they loaded their program into memory and started it running. Working with first-generation software was a lot like working on the naked machine described at the beginning of the chapter. It was attempted only by highly trained professionals intimately familiar with the computer and its operation.

System administrators quickly realized that this was a horribly inefficient way to use an expensive piece of equipment. (Remember that these early computers cost millions of dollars.) A programmer would sign up for an hour of computer time, but the majority of that time was spent analyzing results and trying to figure out what to do next. During this "thinking time," the system was idle and doing nothing of value. Eventually, the need to keep machines busy led to the development of a *second generation* of system software called **batch operating systems** (1955–1965).

In second-generation batch operating systems, rather than operate the machine directly, a programmer handed the program (typically entered on punched cards) to a trained computer operator, who grouped it into a "batch"—hence the name. After a few dozen programs were collected, the operator carried this batch of cards to a small I/O computer that put these programs on tape. This tape was carried into the machine room and loaded onto the "big" computer that actually ran the users' programs, one at a time, writing the results to yet another tape. During the last stage, this output tape was carried back to the I/O computer to be printed and handed to the programmer. The entire cycle is diagrammed in Figure 6.17.

Figure 6.17 Operation of a batch computer system

This cycle might seem cumbersome and, from the programmer's point of view, it was. (Every programmer who worked in the late 1950s or early 1960s has horror stories about waiting many hours—even days—for a program to be returned, only to discover that there was a missing comma.) From the computer's point of view, however, this new batch system worked wonderfully, and system utilization increased dramatically. No longer were there delays while a programmer was setting up to perform an operation. There were no long periods of idleness while someone was mulling over what to do next. As soon as one job was either completed normally or halted because of an error, the computer went to the input tape, loaded the next job, and started execution. As long as there was work to be done, the computer was kept busy.

Now That's Big!

The most widely used measure of program size is source *lines of code* (abbreviated SLOC). This is a count of the total number of nonblank, noncomment lines in a piece of software. According to Wikipedia (www.wikipedia.org), the estimated size of Mac OS X Tiger (10.4)—an older version but the latest for which there is reliable data—is 86 million SLOC. If you were to print out the entire program, at 60 lines

per printed page, you would generate about 1,433,000 pages of output, or roughly the number of pages in 4,000 full-length novels. As an even more enormous example, the code required to run all of Google's popular applications—Gmail, Google Maps, Google Docs, Google+, and so on—contains *two billion* lines of code! If you were to store that output on a bookshelf, it would stretch for more than two miles.

It is estimated that the average programmer can produce about 40 lines of correct code per day. If that number is accurate, then the code for all Google apps represents about 50 million person-days, or (at 250 working days per year) about 200,000 person-years of effort.

Because programmers no longer operated the machine, they needed a way to communicate to the operating system what had to be done, and these early batch operating systems were the first to include a *command language*, also called a *job control language*. This was a special-purpose language in which users wrote commands specifying to the operating system (or the human operator) what operations to perform on their programs. These commands were interpreted by the operating system, which initiated the proper action. The "receptionist/dispatcher" responsibility of the operating system had been born. A typical batch job was a mix of programs, data, and commands, as shown in Figure 6.18.

Figure 6.18 Structure of a typical batch job

By the mid-1960s, integrated circuits and other new technologies had boosted computational speeds enormously. The batch operating system just described kept only a single program in memory at any one time. If that job paused for a few milliseconds to complete an I/O operation (such as read a disk sector or print a file on the printer), the processor simply waited. As computers became faster, designers began to look for ways to use those idle milliseconds. The answer they came up with led to a *third generation* of operating systems called **multiprogrammed operating systems** (1965–1985).

In a multiprogrammed operating system, many user programs, rather than just one, are simultaneously loaded into memory:

If the currently executing program pauses for I/O, one of the other ready jobs is selected for execution so that no time is wasted. As we described earlier, this cycle of running/waiting/ready states led to significantly higher processor utilization.

To make this all work properly, the operating system had to protect user programs (and itself) from damage by other programs. When there was a single program in memory, the only user program that could be damaged was your own. Now, with many programs in memory, an erroneous instruction in one user's program could wreak havoc on any of the others. For example, the seemingly harmless instruction

```
STORE 1000        --Store the contents of register R into memory cell 1000
```

should not be executed if the physical address 1000 is not located within this user's program. It could wipe out an instruction or piece of data in someone else's program, causing unexpected behavior and (probably) incorrect results.

Similarly, the operating system could no longer permit any program to execute a HALT instruction because that would shut down the processor and prevent it from finishing any other program currently in memory. These third-generation systems developed the concept of *user operation codes* that could be included in any user program and *privileged operation codes* whose use was restricted to the operating system or other system software. The HALT instruction became a privileged op code that could be executed only by a system program, not by a user program.

These multiprogrammed operating systems were the first to have extensive protection and error-detection capabilities, and the "traffic officer" responsibility began to take on much greater importance.

During the 1960s and 1970s, computer networks and telecommunications systems (which are discussed in detail in Chapter 7) developed and grew rapidly. Another form of third-generation operating system evolved to take advantage of this new technology. It is called a **time-sharing system**, and it is a variation of the multiprogrammed operating system just described.

In a time-sharing system, many programs can be stored in memory rather than just one. However, instead of requiring the programmer to load all system commands, programs, and data in advance, a time-sharing system allows them to be entered online—that is, entered dynamically by users sitting at terminals and communicating interactively with the operating system. This configuration is shown in Figure 6.19.

Figure 6.19 Configuration of a time-shared computing system

The terminals are connected to the central computer via communication links and can be located anywhere. This new system design freed users from the "tyranny of geography." No longer did they have to go to the computer to hand in their deck of cards; the services of the computer were delivered directly to them via their terminal. However, now that the walls and doors of the computer center no longer provided security and access control, the "security guard/watchman" responsibility became an extremely important part of operating system design. (We will discuss the topic of computer security at length in Chapter 8.)

In a time-sharing system, a user would sit down at a terminal, log in, and initiate a program or make a request by entering a command:

```
>run    MyJob
```

In this example, the program called MyJob would be loaded into memory and would compete for the processor with all other ready programs. When the program was finished running, the system would again display a prompt (">") and wait for the next command. The user could examine the results of the last program, think for a while, and decide what to do next, rather than having to determine the entire sequence of operations in advance.

However, one minor change was needed to make this new system work efficiently. In a "true" multiprogrammed environment, the only event, other than termination, that causes a program to be *suspended* (taken off the processor) is the execution of a slow I/O

operation. What if the program currently executing is heavily *compute-bound*? That is, it does mostly computation and little or no I/O (for example, computing the value of π to a million decimal places). It could run for minutes or even hours before it is suspended and the processor is given to another program. During that time, all other programs would have to sit in the ready queue, waiting their turn. This is analogous to being in line at a bank behind someone depositing thousands of checks.

To design a smooth and efficient time-sharing system, we must make the following change to the multiprogrammed operating system described earlier. A program can keep the processor until *either* of the following two events occurs:

- It initiates an I/O operation.
- It has run for a maximum length of time, called a *time slice*.
  Typically, this time slice is on the order of about a tenth of a second. This might seem like a minuscule amount of time, but it isn't. As we saw in Chapter 5, a typical time to fetch and execute a machine language instruction is about 1 nsec. Thus, in the 0.1-second time slice allocated to a program, a modern processor could execute roughly 100 million machine language instructions.

The basic idea in a time-sharing system is to service many users in a circular, round-robin fashion, giving each one a small amount of time and then moving on to the next. If there are not too many users on the system, the processor can get back to a user before he or she even notices any delay. Each one will believe that they have the entire system to themselves. Time-sharing was the dominant form of operating system during the 1970s and 1980s, and time-sharing terminals appeared throughout government offices, businesses, and campuses.

The early 1980s saw the appearance of the first personal computers (known as PCs or microcomputers), and in many business and academic environments the "dumb" terminal began to be replaced by these PCs. Initially, the PC was viewed as simply another type of terminal, and during its early days it was used primarily to access a central time-sharing system. However, as PCs became faster and more powerful, people soon realized that much of the computing being done on the centralized machine could be done much more conveniently and cheaply by the microcomputers sitting on their desktops.

During the late 1980s and the 1990s, computing rapidly changed from the centralized environment typical of batch, multiprogramming, and time-sharing systems to a *distributed environment* in which much of the computing was done remotely in the office, laboratory, classroom, and factory. Computing moved from the computer center out to where the real work was being done. The operating systems available for early personal computers were simple *single-user operating systems* that gave one user total access to the entire system. Because personal computers were so cheap, there was really no need for many users to share their resources, and the time-sharing and multiprogramming designs of the third generation became less important.

Although personal computers were relatively cheap (and were becoming cheaper all the time), many of the peripherals and supporting gear—laser printers, large disk drives, tape backup units, and specialized software packages—were not. In addition, email was growing in importance, and stand-alone PCs were unable to communicate easily with other users and partake in this important new application. The personal computer era

required a new approach to operating system design. It needed a virtual environment that supported both *local computation* and *remote access* to other users and shared resources.

This led to the development of a *fourth-generation* operating system called a **network operating system** (1985–present). A network operating system manages not only the resources of a single computer but also the capabilities of a telecommunications system called a *local area network*, or *LAN* for short. (We will take a much closer look at these types of networks in Chapter 7.) A LAN is a network that is located in a geographically contiguous area such as a room, a building, or a campus. It is composed of personal computers (workstations), and special shared resources called *servers*, all interconnected via a high-speed link, either wireless or constructed from coaxial or fiber-optic cable. A typical LAN configuration is shown in Figure 6.20.

Figure 6.20 A local area network

The users of the individual computers in Figure 6.20, called *clients*, can perform local computations without accessing the network. In this mode, the operating system provides exactly the same services described earlier: loading and executing programs and managing the resources of this one machine.

However, a user can also access any one of the shared network resources just as though it were local. These resources are provided by a computer called a *server* and can include a special high-quality laser printer, a shared file system, a large computational node, or access to an international network such as the Internet. The system software does all the work, hiding the complex details of communication and competition with other nodes for shared resources.

Network operating systems create a virtual machine that extends beyond the boundaries of the local system on which the user is working. They let us access a huge pool of resources—computers, servers, and other users—exactly as though they were connected to our own computers. This fourth-generation virtual environment, exemplified by operating systems such as Windows 10, macOS, Linux, iOS, and Android, is diagrammed in Figure 6.21.

Figure 6.21 The virtual environment created by a network operating system

One important variation of the network operating system is called a **real-time operating system**. During the 1980s and 1990s, computers got smaller and smaller, and it became common to place them inside other pieces of equipment to control their operation. These types of computers are called **embedded systems**; examples include computers placed inside automobile engines, microwave ovens, thermostats, assembly lines, airplanes, homes, and even the treadmill at your local fitness center.

For example, the Boeing 787 Dreamliner jet contains hundreds of embedded computer systems inside its engines, braking system, wings, landing gear, and cabin. The central

computer controlling the overall operation of the airplane is connected to these embedded computers that monitor system functions and send status information.

In all the operating systems described thus far, we have implied that the system satisfies requests for services and resources in the order received. In some systems, however, certain requests are much more important than others, and when these important requests arrive, we must drop everything else to service them. Imagine that the central computer on our Boeing 787 receives two requests. The first request is from a cabin monitoring sensor that wants the central system to raise the cabin temperature a little for passenger comfort. The second message comes from the onboard collision-detection system and says that another plane is approaching on the same flight path, and there is about to be a midair collision. It would like the central computer to take evasive action. Which request should be serviced next? Of course, the collision-detection message, even though it arrived second.

A real-time operating system manages the resources of embedded computers that are controlling ongoing physical processes and that have requests that must be serviced within fixed time constraints. This type of operating system guarantees that it can service important requests within a fixed amount of time. For example, it may guarantee that, regardless of what else it is currently doing, if a collision-detection message arrives, the software implementing collision avoidance will be scheduled, activated, and executed within 50 milliseconds. Typically, the way that this guarantee is implemented is that all requests to a real-time operating system are *prioritized*. Instead of being handled in first-come, first-served order, they are handled in priority sequence, from most important to least important, where "importance" is defined in terms of the time-critical nature of the request. A real-time operating system lets passengers be uncomfortably cool for a few more seconds while it handles the problem of avoiding a midair collision.

Main content

Chapter Contents

6.4.3 The Future

The discussions in this chapter show that, just as there have been huge changes in hardware over the last 50 years, there have been equally huge changes in system software. We have progressed from a first-generation environment in which a user had to personally manage the computing hardware, to current fourth-generation systems in which users can request services from anywhere in the world using networking capabilities and powerful and easy-to-use graphical user interfaces.

And just as hardware capabilities continue to improve, there is a good deal of computer science research directed at further improving the high-level virtual environment created by a modern fourth-generation operating system. A fifth-generation operating system is certainly not far off.

Like Apple's personal assistant Siri and Google's Android Assistant, these next-generation operating systems will have powerful user interfaces that incorporate not only text, touch, and graphics but voice-activated commands, photography, facial and body gestures, video and TV, along with artificial intelligence capabilities that allow it to learn about your personal and professional characteristics. These *intelligent multimedia*

user interfaces will interact with users and solicit requests in a variety of ways. Instead of point-and-click, a fully integrated fifth-generation system might allow you to speak the command, "What do I have planned for today?" The results may include a verbal reminder of an important meeting, a map of how to reach the meeting site, the agenda, articles that the system automatically determined you should read to prepare, and photographs of important people who will be attending. It may even make a comment on whether you are dressed appropriately! Just as text-only systems are now viewed as totally outmoded, today's GUIs will be viewed as far too limiting in terms of their user/system interaction.

A fifth-generation operating system will typically be running on a massively parallel processor, and it will need to efficiently manage a computer system containing hundreds or even thousands of processors (today's multicore machines typically contain only two, four, or six processors per chip). Such an operating system will need to recognize opportunities for parallel execution, send the separate tasks to the appropriate processor, and coordinate their concurrent execution, all in a way that is transparent to the user. On this virtual machine, a user will be unaware that multiple processors even exist except that programs now run 10, 100, or 1,000 times faster.

Finally, new fifth-generation operating systems will create a truly **distributed computing environment** in which users do not need to know the location of a given resource within the network. This is analogous to the way that the manager of a business gives instructions to an assistant: "Get this job done. I don't care how or where. Just do it, and when you are done, give me the results." The details of how and where to get the job done are left to the underling. The manager is concerned only with the final results.

In a truly distributed operating system, the user is the manager, the operating system is the assistant, and the user does not care where or how the system satisfies a request as long as it gets done correctly. The users of a distributed system do not see a network of distinct sites or "local" and "remote" nodes. Instead, they see a single logical system that provides resources and services. The individual nodes and the boundaries between them are no longer visible to the user, who thinks only in terms of *what* must be done, not *where* it will be done or *which* node will do it. This situation is diagrammed in Figure 6.22. The concept of a single large collection of accessible resources whose location is not known to the user is often referred to as *cloud computing* after the model of the computational cloud shown in Figure 6.22. Cloud computing will be discussed in much greater detail in the next chapter.

Figure 6.22 Structure of a distributed system

This is certainly the most powerful virtual environment we have yet described, and an operating system that creates such an environment would significantly enhance the productivity of all its users. These "fifth-generation dashboards" will make using the most powerful and most complex computer system as easy as driving a car—perhaps even easier. Surfing the web provides a good indicator of what it is like to work on a distributed system. When we click on a link, we have no idea where that webpage is located and, moreover, we don't care. We simply want it to appear on our screen. Similarly, when we back up our smartphone to the cloud, we have no idea where our

data will be stored. We only know that we will be able to retrieve it whenever we need to. To us, both the web and the cloud behave like one giant logical system even though they may be spread out across hundreds of countries and millions of computers.

Figure 6.23 summarizes the historical evolution of operating systems, much as Figure 1.8 summarized the historical development of computer hardware.

Figure 6.23

Some of the major advances in operating systems development

Generation	Approximate Dates	Major Advances
First	1945–1955	No operating system available
		Programmers operated the machine
themselves		
Second	1955–1965	Batch operating systems
		Improved system utilization
		Development of the first command language
Third	1965–1985	Multiprogrammed operating systems
		Time-sharing operating systems
		Increasing concern for protecting programs from damage by other programs
		Creation of privileged instructions and user instructions
		Interactive use of computers
		Increasing concern for security and access
control		
		First personal computer operating systems
Fourth	1985–present	Network operating systems
		Client-server computing
		Remote access to resources
		Graphical user interfaces
		Real-time operating systems
		Embedded systems
Fifth	??	Multimedia user interfaces
		Massively parallel operating systems
		Distributed computing environments

Gesture-Based Computing

There have been enormous changes in the interface between humans and computers. From typed commands to mouse clicks to finger taps, the ability to select and activate operations has become ever more simple. Today, virtually everyone is familiar with the use of multitouch operations to rotate the screen, swipes to move from one image to the next, and shaking to randomize a playlist.

The use of body and facial motions to communicate computational requests is called **gesture-based computing**, and it is similar to what is being done today using game controllers such as Microsoft Kinect®. However, computer scientists want to develop algorithms that teach computers to understand human body language to broaden the possible applications of gesture-based computing. Think how convenient it would be to turn on a light by simulating the motion of flipping a switch; lower the volume of a sound system by moving your hand in a downward direction; select an option on the screen by pointing to it; or decline to answer the phone simply by moving your head left and right to indicate no. Researchers at the University of Minnesota have used Microsoft Kinect to evaluate a range of motion-related symptoms in children, including autism, attention-deficit disorder, and obsessive-compulsive disorder.

The operating system user interface of the future will involve even more intuitive and easy-to-learn movements. Many people have postulated that gesture-based computing will soon make “regular” input devices, such as the mouse and keypad, obsolete.