**Chapter 6**

**An Introduction to System Software and Virtual Machines**

**A Guide to this Instructor’s Manual:**

We have designed this Instructor’s Manual to supplement and enhance your teaching experience through classroom activities and a cohesive chapter summary.

This document is organized chronologically, using the same headings that you see in the textbook. Under the headings you will find: lecture notes that summarize the section, Teaching Tips, Class Discussion Topics, and Additional Projects and Resources. Pay special attention to teaching tips and activities geared towards quizzing your students and enhancing their critical thinking skills.

In addition to this Instructor’s Manual, our Instructor’s Resources also contain PowerPoint Presentations, Test Banks, and other supplements to aid in your teaching experience.

|  |
| --- |
| **At a Glance** |

**Instructor’s Manual Table of Contents**

* Overview
* Learning Objectives
* Teaching Tips and Quick Quizzes
* Class Discussion Topics
* Additional Projects
* Additional Resources
* Key Terms
* Solutions to End-of-Chapter Exercises

|  |
| --- |
| Lecture Notes |

**Overview**

Chapter 6 introduces the programs and tasks that make up system software for a computer. It describes in detail how and why assembly language is used for programming a computer, rather than programming directly in machine language. It describes the main tasks an operating system needs to perform and how the operating system interacts with the user and with other subsidiary programs to perform its tasks. It provides a historical overview of operating system development, emphasizing how each generation sought to improve on previous systems.

# **Learning Objectives**

* 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

# **Teaching Tips**

**6.1 Introduction**

1. Introduce the term **naked machine,** and describe what it would be like to work with a machine with no system software to help. Reference the early days of computing when all programs had to be entered in machine language.
2. The operating system and other system software creates an interface for the user that hides ugly details, makes it easier to see what is going on, and protects the computer from errant damage by the user. See the analogy with the dashboard and controls of a car versus what is happening under the hood.

**6.2 System Software**

1. Introduce the term **system software**, emphasizing that it is software, not hardware. List the purposes of system software: hide complex details, present important information in easy format, permit access to machine resources in an easy way, and provide a safe and secure environment. Note the system software creates a **virtual machine** or a **virtual environment** for the user.
2. Introduce the term **operating system**, and distinguish it from some other software packages, including the **graphical user interface (GUI)**, language services, memory managers, information managers, I/O system, scheduler, and utilities—which are sometimes called **program libraries**.
3. Compare the work required to write, load, and run a program on a naked machine with the work required when using tools provided by an operating system and applications program.

**6.3 Assemblers and Assembly Language**

1. Describe the shortcomings of machine language, and introduce the term **assembly language,** known more commonly now as **low-level programming language**, for the first step toward human-friendly notation. Emphasize that each low-level programming language instruction translates to exactly one machine language instruction.
2. Briefly discuss **high-level programming languages** such as C++ and Python, where the languages are more user oriented and further away from machine language. Highlight that these languages use a combination of natural language and mathematical notation. A single high-level language instruction translates into many machine language instructions.
3. Assembly language must be translated into machine language: the processor does not understand it directly. Introduce the terms **source program**, **object program**, **assembler**, and **compiler**. Go into detail on each of these terms.
4. Emphasize the three advantages of assembly language: (1) use of symbolic op codes, (2) use of symbolic memory addresses, and (3) “pseudo-operations” that generate data, and so on. Note that the preceding chapter used the symbolic names for operations already. Symbolic labels make assembly code easier to read and easier to modify. **Pseudo-ops** are actually instructions to the assembler and not to the machine; use examples with data generation and .BEGIN and .END.
5. Go over several examples of assembly language programs. Start small and end with full-scale algorithms turned into assembly. Ask students to read and write assembly programs as well.

|  |  |
| --- | --- |
| ***Teaching Tip*** | For students to truly understand assembly language programs, they must develop two skills: reading and understanding assembly programs and writing them. Divide students into groups, and ask them to work through novel assembly programs until they can quickly and accurately determine what the programs do. Then start with tiny examples and have the groups develop assembly programs from pseudocode. At the end, have students switch programs with different groups to see if they make sense to others. |

1. Introduce the key tasks of the assembler: translating op codes, translating symbolic addresses, performing pseudo-op tasks, and creating an object file. Emphasize how the assembler is a program, written using all the tools of design and analysis from earlier chapters. In particular, note the discussion of sequential versus binary search for looking up symbolic op codes.
2. Introduce the term **pass**; assemblers frequently perform more than one pass over the assembly language program.
3. Introduce the term **symbol table**, and describe how assemblers work in one pass to collect the symbolic labels and their relative position with respect to the start of the program. Then in a second pass, the assembler substitutes the position for the symbol. Introduce the term **binding** for the process of associating symbolic names with memory addresses.
4. Introduce the term **object file**. Remind students that an example of this on a Windows machine is the .exe (executable file) at the end of a program file.
5. Introduce the term **loader**, and describe its purpose: to place the machine language program into memory and send it to the hardware.

**Quick Quiz 1**

1. A program written in assembly language is called a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Answer: source program

1. (True or false) Pseudo-ops are instructions for services provided by the assembler.

Answer: True

1. (True or false) A pass is when the assembler chooses not to do anything with a particular program.

Answer: False

1. Name two advantages of assembly language programs over machine instruction programs.

Answer: Readability, maintainability, symbolic labels, program clarity, symbolic op codes, or pseudo-ops

1. The .exe at the end of a Windows program file is an example of a(n) \_\_\_\_\_\_\_.

Answer: object file

**6.4 Operating Systems**

1. Introduce the term **system commands**; system commands may be given by text typed into a terminal or by interacting with the computer’s GUI. Most students will never have seen or used terminal commands; demonstrate it for them. Discuss the power of terminal commands.
2. The **operating system** provides a variety of functions. Focus the students on the five most relevant ones: the user interface, system security and protection, program scheduling and activation, efficient allocation of resources, and the safe use of resources. You see us how you always see us: a receptionist, a security guard, a dispatcher, an efficiency expert, and a traffic officer.
3. User interface algorithm is an endless loop of waiting for input from users, performing the appropriate action, and repeating. Text-oriented user interfaces require users to learn a frequently cryptic **command language** that will control the operating system. Graphical user interfaces use standardized metaphors and icons to make interactions easier.
4. System security and protection ensures that only authorized actions are permitted. Emphasize the role of usernames and passwords. Introduce the term **superusers** for administrators who have greater access to the system, especially through the terminal. Discuss how operating systems **encrypt** critical data like password files to increase security.
5. Discuss the efficient allocation of resources and its importance for enabling multitasking on the computer. Multiple programs can run at the same time by maintaining a queue of ready programs; a program is either running, ready, or waiting for some I/O. A running program gets a short amount of time, then another is selected.
6. Safe use of resources refers to the problem of **deadlock**—when different programs require the same resources, each one has one resource and is waiting for the other. Discuss deadlock prevention and deadlock recovery algorithms.
7. When discussing the historical development of operating systems, emphasize (1) how this shows the iterative process of design and development and (2) how understanding of more simple operating system designs will make it easier to understand modern, complex operating systems.

|  |  |
| --- | --- |
| ***Teaching Tip*** | Great posters are available that show time lines for operating systems:  <http://www.levenez.com/> |

1. First generation computer operating systems were almost naked machines. Introduce the term **batch operating systems**, the second generation of operating systems. Discuss the difference between first and second generation, making careful note of how the role of programmers changed from first generation to second. Discuss command language, also called job control language, and how it allowed the programmers, not at the machine, to specify basic operating system commands.
2. Introduce the term **multiprogrammed operating systems**, the third generation.   
   Emphasize the new need for system security and safety. Introduce the term **time-sharing system**. Discuss how the illusion of sole access was created by the use of time slices, and mention the modern application: a single computer running iTunes, a web browser, and a word processor.
3. Introduce the term **network operating system**, the fourth generation. Communication with other computers over a network becomes critical, and accessing shared resources (i.e., printers and servers). Introduce the terms **real-time operating system** and **embedded systems** as examples of network operating systems.
4. Fifth generation is the future: multimedia interfaces, massively **parallel processing systems**, and a **distributed computing environment**. Resources are seamlessly local or not. Introduce the term **cloud computing**.

**Quick Quiz 2**

1. Name two important operating system tasks.  
   Answer: User interface, security and safety, program scheduling, efficient use of resources, safe use of resources, communications with external devices, or communications with computer network
2. (True or false) Multiprogrammed operating systems are future systems that use many processors to run programs at the same time.

Answer: False

1. A(n) \_\_\_\_\_\_\_\_\_\_\_\_\_ consists of text instructions that tell the operating system what to do.

Answer: command language or job control language

1. A collection of resources whose location is not known to the user is called \_\_\_\_\_\_\_\_\_.

Answer: cloud computing

1. (True or false) A batch operating system could run multiple programs at the same time.

Answer: False

# **Class Discussion Topics**

1. Look at the list of common operating system tasks in Section 6.4.1. Which of these tasks have you performed in the past several weeks, and how did you perform them?
2. Why does an assembler use two passes to handle symbolic labels in assembly language programs? Why can’t it just do everything in one pass?
3. What advantages did each generation of operating system have over the previous ones? What problems was the next generation trying to solve?

# **Additional Projects**

1. Take a program written in assembly language (given to you by your instructor) and determine what the program is calculating. What answer will it produce for specific starting values, but also, at a high level, what is its purpose?
2. Take the brief assembly language program (given to you by your instructor) and perform the two passes described in the book to translate the program into its machine language equivalent, starting the program at location 0 in memory.
3. Take a program written in pseudocode and construct an assembly language program for it. What parts of the program are easy or difficult to construct?

# **Additional Resources**

1. History of the development of graphical user interfaces: <http://arstechnica.com/old/content/2005/05/gui.ars>
2. Tutorial to terminal-based UNIX commands: <http://www.ee.surrey.ac.uk/Teaching/Unix/>
3. Tutorial to terminal-based Windows commands: <http://www.bleepingcomputer.com/tutorials/windows-command-prompt-introduction/>
4. A basic introduction to basic terminal commands on macOS:

<http://www.macworld.co.uk/feature/mac-software/how-use-terminal-on-mac-3608274/>

**Key Terms**

* **Algorithmic problem-solving cycle:** The sequence of designing an algorithm, coding it into a programming language, translating it into machine language, and then running it on a Von Neumann computer to solve the problem.
* **Assembler:** The program that translates a source assembly language program into machine language.
* **Assembly language:** A low-level programming language that maps 1:1 into machine language.
* **Batch operating system:** A type of operating system in which a batch of programs are collected and then run as a group, all at once, one after the other.
* **Binding:** The process of associating a symbolic name with a physical memory address.
* **Cloud computing:** A computing system in which the user can be completely unaware of where data is stored and where services are being provided.
* **Command language:** The language used to enter system commands. Today, it is more commonly a set of actions, such as mouse clicks or finger taps.
* **Compiler:** The program that translates a high-level language into machine language.
* **Deadlock:** When a computer is frozen because it is waiting for an event to occur that never will.
* **Distributed computing environment:** A system that hides the exact location of specific pieces of information and allows the user to view the system as simply one large collection of resources.
* **Embedded systems:** Computers that are placed within other types of devices to control their operation.
* **Encrypt:** To change plain text into coded text that cannot be understood without specific details about the encoding algorithm.
* **Graphical user interface (GUI):** The visual overview of the computer that is provided by the operating system.
* **High-level programming language:** A programming language that uses both natural language constructs and mathematical notation.
* **Loader:** System software that reads instructions from the object file and stores them into memory for execution.
* **Low-level programming language:** The general term for a programming language that is close in design to machine languages.
* **Multiprogrammed operating system:** A type of operating system in which multiple user programs are loaded into memory at the same time, and the computer takes turns running them.
* **Naked machine:** A computer without any helpful system software to facilitate its usage.
* **Network operating system:** An operating system that manages the resources of both the local computer as well as providing efficient access to a collection of remote resources via a computer network.
* **Object file:** The file produced by the assembler or compiler that contains the translated machine language instructions and the address of where each instruction is to be loaded.
* **Object program:** The translated source program produced by the assembler or compiler.
* **Operating system:** The main piece of system software that helps to run and manage the computer system.
* **Parallel processing system:** An operating system that controls the operation of computers with multiple processors.
* **Pass:** An examination of every statement in the source program by an assembler or compiler.
* **Program library:** A collection of software utilities provided for the user.
* **Pseudo-op:** An assembly language command that does not actually produce a machine-language instruction but performs a service on behalf of the user.
* **Real-time operating system:** An operating system that must provide access to particular resources or respond to system problems within a well-defined time limit.
* **Source program:** The original program as written by the user.
* **Superuser:** A privileged user who has access to virtually all services and information stored on the computer; the system administrator.
* **Symbol table:** A table that contains the name of every symbolic variable in a program and its equivalent binary memory address.
* **System commands:** Commands given by the user to the operating system to perform a service on the user’s behalf.
* **System software:** A collection of computer programs that manage the resources of a computer and facilitate access to those resources.
* **Time-sharing system:** Another type of multiprogrammed operating system but one in which the user can interact with the running program.
* **Virtual environment:** Another term for a virtual machine.
* **Virtual machine:** The computer system as perceived by the user as opposed to the hardware that actually exists; the set of services and resources created by the software and seen by the user. Also called a virtual environment.

# **Solutions to End-of-Chapter Exercises**

1. Answers may vary. The user interface of any device is that part most visible to the user and with which the user directly interacts. Examples are the labeled buttons of a DVD’s remote control, the LED of a copier, and preset defrost/cook buttons on a microwave.
2. Answers may vary. Cases where one might like to see the underlying hardware of the computer system are when one wants to see and manipulate the details of machine operation, typically when writing system software such as an assembler or a linker or an operating system. Other instances may occur when interfacing with a device such as a mouse or a graphics tablet. There are programming languages (one example is C–see Chapter 10) that allow you to do machine-level manipulations, and one typically uses such a language for writing system software.
3. The components here are the English letter, the Spanish letter, and the translator. The English letter corresponds to the source program, the Spanish letter corresponds to the object program, and the translator corresponds to the assembler.
4. a. LOAD 60 R: 472 60: 472 61: -1   
    b. STORE 60 R: 13 60: 13 61: -1  
    c. ADD 60 R: 485 60: 472 61: -1  
    d. COMPARE 61 R: 13 60: 472 61: -1  
    e. IN 61 R: 13 60: 472 61: 50   
    f. OUT 61 R: 13 60: 472 61: -1
5. a. 6   
    b. the value in address 6  
    c. 6  
    d. the value in address 7
6. If we execute .DATA 16387 then the processor will fetch, decode, and attempt to execute the “instruction” 16387. The binary representation of 16387 is

0100 0000 0000 0011

The first 4 bits would be the op code for the INCREMENT operation, while the address field would be interpreted as the value 3. So, the contents of memory location 3 would be incremented by 1.

1. a. SUBTRACT 780  
    b. ADD 7
2. THREE: .DATA 2 is perfectly legal, but it doesn't make much sense. It assigns the label THREE to a memory cell that contains the value 2. An instruction such as ADD THREE would therefore add 2 to whatever is in register R. While the computer would have no problem, a reader of the program would probably be misled.
3. Assume that K, L, M, N have already been given values  
    a. LOAD K  
    ADD THREE  
    STORE K  
    .  
    .  
    .  
    THREE: .DATA 3  
    K: .DATA …

b. LOAD L  
 ADD ONE  
 SUBTRACT M  
 SUBTRACT N  
 STORE K  
 .  
 .  
 .  
 ONE: .DATA 1  
 K: .DATA …  
 L: .DATA …  
 M: .DATA …  
 N: .DATA …

c. LOAD TEN  
 COMPARE K  
 JUMPGT OUTPUT  
 JUMP NEXT  
 OUTPUT: OUT K  
 NEXT: next instruction goes here…  
 .  
 .  
 .   
 TEN: .DATA 10  
 K: .DATA …

1. LOAD L  
    COMPARE K  
    JUMPGT GREATER  
    OUT L  
    INCREMENT L  
    JUMP NEXT  
    GREATER: OUT K  
    INCREMENT K  
    NEXT: next instruction goes here…  
    .  
    .  
    .  
    K: .DATA …  
    L: .DATA …

e. LOAD ONE  
 STORE K  
 LOAD HUNDRED  
REPEAT: OUT K  
 INCREMENT K  
 COMPARE K   
 JUMPGT NEXT   
 JUMP REPEAT  
NEXT: next instruction goes here…  
 .  
 .  
 .  
ONE: .DATA 1  
HUNDRED: .DATA 100  
K: .DATA …

1. The difference is that the left-hand set of instructions will only add two to the register copy of X, leaving the original value of X unchanged. The right-hand set of instructions adds two to the value in memory cell X.
2. CLEAR SUM is not necessary because SUM has been set to 0 with the .DATA pseudo operation. LOAD ZERO is necessary because the register must be reset at the start of each loop to test whether each new number is nonnegative.
3. .BEGIN  
    CLEAR SUM  
    CLEAR SUM2  
    IN N --Read in next value  
    AGAIN: LOAD ZERO  
    COMPARE N  
    JUMPLT NEG   
    JUMPEQ DONE --if N = 0, finish up  
    LOAD SUM --N is a positive number  
    ADD N --add N to positive sum  
    STORE SUM  
    IN N --read next N and repeat  
    JUMP AGAIN   
    NEG: LOAD SUM2 --N is negative  
    ADD N --add N to negative sum  
    STORE SUM2  
    IN N --read next N and repeat  
    JUMP AGAIN  
    DONE: OUT SUM --write positive sum  
    OUT SUM2 --write negative sum  
    HALT  
    SUM: .DATA 0  
    SUM2: .DATA 0  
    N: .DATA 0  
    ZERO: .DATA 0
4. .BEGIN

IN MAX --read in first value as MAX  
 LOAD MAX  
 STORE MIN --store first value as MIN

AGAIN: INCREMENT COUNTER --going to read another value

IN N  
 LOAD N  
 COMPARE MAX  
 JUMPLT LARGER --MAX < N, N should be new MAX  
 COMPARE MIN  
 JUMPGT SMALLER --MIN > N, N should be new MIN  
 JUMP TEST --N is neither new MAX nor new MIN  
 LARGER: STORE MAX --store N in MAX  
 JUMP TEST  
 SMALLER: STORE MIN --store N in MIN  
 TEST: LOAD COUNTER   
 COMPARE HUNDRED  
 JUMPGT AGAIN --less than 100 values read  
 DONE: OUT MAX --write out max and min  
 OUT MIN  
 HALT  
 N: .DATA 0

MAX: .DATA 0  
 MIN: .DATA 0  
 COUNTER: .DATA 1  
 HUNDRED: .DATA 100

1. a. The list of 16 op codes would be sequentially searched for each of the 100 instructions. In the worst case each search would require examining all 16 op codes, giving 100 × 16 = 1600 comparisons. Therefore, the time taken would be 1,600 comparisons × 1/50,000 seconds/comparison = 0.032 seconds.  
     
    b. Using binary search on a list of 16 op codes (the list of op codes must be sorted) requires in the worst case 1 + ⎣lg 16⎦ = 5 comparisons (see Exercise 31 of Chapter 3). Therefore, the time taken would be 100 × 5 comparisons × 1/50,000 seconds/comparison = 0.01 seconds   
     
   Using binary search is slightly faster.

With 300 op codes, the results are:

a. 100 × 300/50000 = 0.6 seconds  
 b. 100× (1 + ⎣lg 300⎦)/50000 = 100×9/50000 = 0.018 seconds. Binary search is more than 30 times faster.

With 50,000 instructions, the results are:

a. 50000 × 16/50000 = 16 seconds  
 b. 50000 × (1 + ⎣lg 16⎦) /50000 = 50000 × 5/50000 = 5 seconds. Binary search is more than 3 times faster.

1. This is how the symbol table will look:  
     
    Symbol Address   
    AGAIN 2  
    ANS 7  
    X 8  
    ONE 9
2. This is how the symbol table will look:  
     
    Symbol Address  
    AGAIN 2  
    NEG 10  
    SUM 12  
    N 13  
    ZERO 14
3. This pair of statements is illegal – as figure 6.11 shows, the assembler checks in pass 1 for using the same symbol name twice, which is illegal. It is illegal because each symbol can be associated with only a single numeric memory address.
4. Some drawbacks to using passwords to limit access to a computer system are the possibility of having the password forgotten, stolen, or guessed. Using authorization lists, an operating system can restrict access to certain folders or files by legitimately logged-on users, as described in this chapter. This is appropriate when users of different standing have access to the same machine (for example a supervisor might have access to more files than a clerk). For highly sensitive data, the operating system might require a physical identification test such as a fingerprint or retinal scan.

19. Authorization lists must be encrypted and protected from unauthorized change because they restrict unauthorized users from viewing information that is confidential and valuable. If the authorization list is modified, it could allow users to access data they should not see and modify or even to delete it. Also, users could be denied access to files that they have a legitimate right to read and change.

20. If 3 programs, A, B, and C, are loaded into memory, then for the processor to be idle at any time, all three programs must be waiting for I/O operations. Each program spends 1/2 of its time waiting for an I/O operation, so at any moment of time, there is a (1/2)×(1/2) × (1/2) = 1/8 probability that all three programs are waiting. Therefore, the processor is doing useful work 1 - (1/8) = 7/8 of the time, which is 87.5%. With 4 programs in memory the processor is doing useful work 1 - (1/2)4 = 1 - 1/16 = 15/16 = 93.75%. With 5 programs the processor is doing useful work 1 - (1/2)5 = 1 - (1/32) = 31/32 = 96.875%, more than 95%.

21. If two friends call each other at the exact time, both will receive busy signals and will hang up. Both will redial 1 minute later and the same thing will happen again; in fact, this will go on indefinitely. This problem could be solved (or at least made highly unlikely) by waiting a random amount of time before redialing.

22. A batch operating system would be far too slow to handle such modern applications, which require real-time responses. Also, in an application such as an airline reservations system, many users need access to the software at the same time.

23. In a time-sharing operating system, if the time slice value is too large, a user may wait a long time before his or her program gets any processing time. If the time slice is too small, each user's turn comes around quickly but very little progress can be made during that turn because of the small-time slice; in addition, the processor will spend a larger proportion of its time swapping between jobs.

24. When a user has his or her own machine (i.e., a personal computer, tablet, smartphone) there is no delay to wait for the processor to turn its attention to the user's program. The user is never competing with other users for processing time and computing is slowed only by the I/O operations.

25. Answers may vary. The user may need additional privileges to access a remote resource, and the response may be slower than for a local resource.

26. The operating system should handle these requests in the following order, with the most critical or time-sensitive requests being handled first:

1. Someone pulled the plug on the power supply, and the system will run out of power in 50 msec. (This is critical or the entire system will shut down.)  
2. The program running on processor 2 is trying to perform an illegal operation code. (An illegal operation code could write over data or program instructions.)  
3. The clock in the computer has just “ticked” and we need to update a seconds counter.   
4. The disk has just read the character that passed under the read/write head, and it wants to store it in memory before the next one arrives. (Least critical - external data is accessed at a slow rate compared to processor operations and the character can be stored in a local buffer until other crises have passed and it can be written to memory.)

**Challenge Work**

**1.** This is indeed a challenging problem. Note that you do not want to repeatedly read a value and add it to a sum, similar to what is done in Figure 6.8. Instead you want to read all 50 values into memory and then access each one in turn. The assembly language program will need 50 memory cells to store the 50 values, but you do not want to make 50 .DATA pseudo-op statements nor have to write 50 input statements. The declaration

A: .DATA 0

will reserve a memory location, let's say it is memory location 42, with the label A for the first of the 50 values to be stored. Make this pseudo-op the last program instruction so that subsequent memory locations are available to store the remaining 49 numeric values.

The first of the 50 values will be read into location A using an instruction such as

REPEAT: IN A

REPEAT is then a label attached to a particular memory address. The content of that memory address is the instruction

*op code address*

1101 0000 0010 1010

but this instruction can be treated as data. In particular, the statement

INCREMENT REPEAT

says to increment the contents of memory cell REPEAT, which would change the REPEAT statement to

*op code address*

1101 0000 0010 1011

which means "input to 43, the next address after A". Hence a loop containing the INCREMENT REPEAT instruction can be used to read in the 50 values. A similar loop can be used to add each value in the list to the running sum.