

操作系统 习题与解答


(英文版)

OPERATING SYSTEMS

 **Covers principles underlying all the major operating systems**

 **Includes DOS, UNIX, & Linux**

 **Solved examples from simple to comprehensive**

 **Perfect for professionals refreshing core concepts**

(美) J. Archer Harris 著

全美经典
学习指导系列

操作系统 习题与解答

(英文版)

OPERATING SYSTEMS

(美) J. Archer Harris 著

 机械工业出版社
China Machine Press

John R. Hubbard: Data Structures With C++(ISBN 0-07-118358-2).

Copyright © 2002 by The McGraw-Hill Companies, Inc. All rights reserved.

Jointly published by China Machine Press/McGraw-Hill. This edition may be sold in the People's Republic of China only. This book cannot be re-exported and is not for sale outside the People's Republic of China.

本书英文影印版由美国McGraw-Hill公司授权机械工业出版社在中国大陆境内独家出版发行, 未经出版者许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封面贴有McGraw-Hill公司激光防伪标签, 无标签者不得销售。

版权所有, 侵权必究。

本书版权登记号: 图字: 01-2002-1883

图书在版编目(CIP)数据

操作系统习题与解答 / (美) 哈里斯 (Harris, J.A.) 著. - 北京: 机械工业出版社, 2002.8

(全美经典学习指导系列)

书名原文: Operating Systems

ISBN 7-111-10617-2

I. 操… II. 哈… III. 操作系统(软件) - 习题 - 英文 IV. TP316-44

中国版本图书馆CIP数据核字 (2002) 第051850号

机械工业出版社 (北京市西城区百万庄大街22号·邮政编码 100037)

责任编辑: 华章

北京牛山世兴印刷厂印刷·新华书店北京发行所发行

2002年8月第1版第1次印刷

787mm × 1092mm 1/16 · 15.25印张

印数: 0 001-3000册

定价: 25.00元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

PREFACE

My first experience with operating systems was on a DEC PDP-11 computer running Unix Version 6. With the source code in hand, I learned about a fascinating piece of software which was both elegant and complex. An operating system is the ultimate challenge for a programmer, encompassing everything from low-level device manipulation, to concurrency, to object-oriented design.

This book explores the design principles found in modern operating systems. It is intended for those wishing to learn more about operating systems in general or for those with interest in a particular system who desire a broader perspective on its operation. As in all Schaum's Outline Series books, each chapter includes a concise presentation of the material and numerous solved problems. The book is suitable for use as a companion to a standard operating systems text, as a supplement to a course, or as a review guide for students preparing for graduate entrance or comprehensive exams.

The emphasis of this book is on design principles, not the detailed characteristics of any particular operating system. However, examples from various operating systems are cited, with DOS, Windows, and Unix being the most frequently referenced. The book concentrates on generally applicable design features and does not cover more specialized topics such as real-time or distributed systems.

I wish to thank all the staff at McGraw-Hill who helped produce this effort. I thank my students and my colleagues at James Madison University for their understanding and support. A special acknowledgment to Dr. Ramón Mata-Toledo for his part in bringing this book to life.

Finally, I express my profound gratitude to my wife and colleague, Nancy Harris. In addition to emotional support, her critical review was invaluable.

Although it is hoped all material in this book is accurate, the possibility does exist that some errors are present. Notification of errors, omissions, or suggested improvements should be sent to schaumos@mailgate.cs.jmu.edu. Updates on the book may be found at URL <http://www.cs.jmu.edu/users/harrisja/schaumos>.

J. ARCHER HARRIS

CONTENTS

CHAPTER 1	Introduction	1
	1.1 Machine Hardware	1
	1.2 Operating System Structure	5
	1.3 Outline of the Rest of This Book	8
CHAPTER 2	Process Management	14
	2.1 Process Scheduling	15
	2.2 Process State	15
	2.3 Scheduling Criteria	18
	2.4 Scheduling Algorithms	19
	2.5 Scheduling Algorithm Performance	23
	2.6 Process Attributes	24
	2.7 Process Supervisor Calls	26
CHAPTER 3	Interprocess Communication and Synchronization	45
	3.1 Interprocess Communication	45
	3.2 Process Synchronization	49
	3.3 Deadlock	59
CHAPTER 4	Memory Management	87
	4.1 Single Absolute Partition	87
	4.2 Single Relocatable Partition	88
	4.3 Multiprogramming	88
	4.4 Multiple Partitions	89
	4.5 Simple Paging	92
	4.6 Simple Segmentation	93
	4.7 Segmentation with Paging	95
	4.8 Page and Segment Tables	95
	4.9 Swapping	97
	4.10 Overlaying	97

Contents

CHAPTER 5	Virtual Memory	124
	5.1 Demand Paging	124
	5.2 Segmentation	130
CHAPTER 6	File System Management	152
	6.1 Directories and Names	152
	6.2 Types of File System Objects	157
	6.3 File System Functions	157
	6.4 Information Types	158
	6.5 File System Architecture	159
CHAPTER 7	Device Management	185
	7.1 Hardware I/O Organization	185
	7.2 Software Organization	190
	7.3 Devices	195
CHAPTER 8	Security	218
	8.1 Authentication	218
	8.2 Prevention	220
	8.3 Detection	221
	8.4 Correction	221
	8.5 Identification	221
	8.6 Threat Categories	222
	8.7 Program Threats	222
CHAPTER 9	Bibliography	229
INDEX		232

Introduction

Computer systems provide a capability for gathering data, performing computations, storing information, communicating with other computer systems, and generating output reports. Some of those capabilities are best implemented in hardware, others in software. An operating system is the software that takes the raw capabilities of the hardware and builds a more practical platform for the execution of programs. The operating system manages hardware resources, provides services for accessing those resources, and creates higher-level abstractions such as files, directories, and processes.

A typical computer system contains five major components: the hardware, the operating system, systems programs, application programs, and users (Fig. 1-1). The hardware does all the actual work and includes the memory, the *central processing unit* (CPU), and the input and output (I/O) devices. The operating system provides a set of services to programs. The user interacts with the operating system indirectly, through the programs. Systems programs are a set of utility programs supplied with an operating system to provide basic services for users. Examples of systems programs include a window manager for a *graphical user interface* (GUI), a command interpreter, and programs to rename, copy, or delete files. Application programs provide the computer with the functionality the users require. Examples of application programs include tax preparation software, a financial planner, a word processor, and a spreadsheet.

1.1 Machine Hardware

The CPU is the heart and brain of a computer system. It contains a number of special-purpose registers, an *arithmetic logic unit* (ALU), and the control logic necessary to decode and execute instructions (Fig. 1-2). Connected to the CPU by way of a communication *bus* are the memory and the I/O devices. The operation of the CPU is controlled by the instructions the CPU fetches from memory. The I/O devices, in turn, are commanded by the CPU.

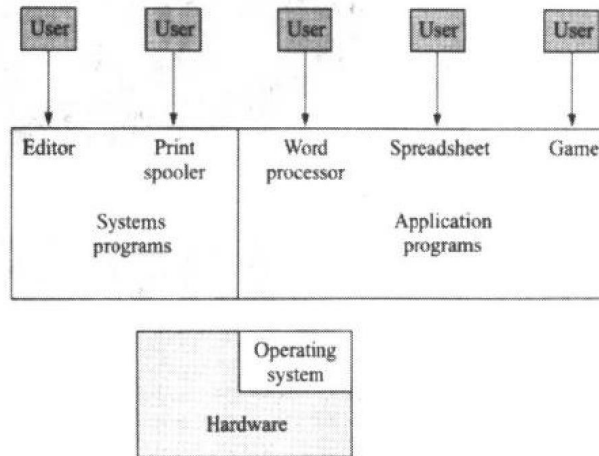


Fig. 1-1. Computer system components.

The operation of a CPU can be described in terms of a simple loop, where each time through the loop one instruction is executed. The basic process by which instructions execute never varies.

- An instruction is fetched from the memory location specified by the special register called the *program counter*. All instructions to be executed are fetched from main memory.
- The instruction is placed in a special register called the *instruction register*.
- The program counter is incremented so it points to the next instruction to be executed.
- The instruction is decoded to determine what action is to be performed. The action is specified by the instruction's *opcode* (operation code) bits. The machine architecture defines which bits contain the opcode.

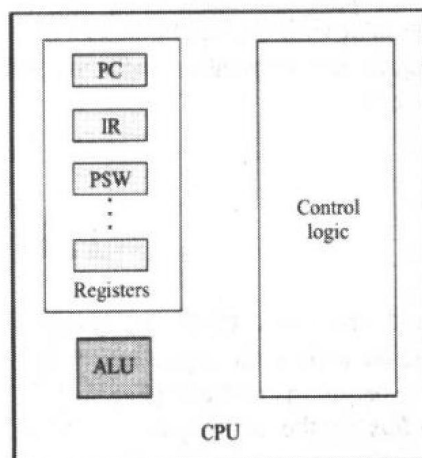


Fig. 1-2. CPU organization.

- Depending on the operation to be performed, the value of one or more operands are fetched from memory.
- The operation specified in the opcode is performed. Five basic categories of operations exist.
 - (1) **Movement:** Move a value from one location to another. The locations involved may be registers or memory locations.
 - (2) **Computation:** Send one or more operand values to the ALU and have a computation performed.
 - (3) **Conditional Branch:** If the branch condition is true, reset the program counter to point to the branch address. For an unconditional branch, the branch condition is always true.
 - (4) **Procedure Call:** Save the current value of the program counter. Then reset the program counter to point to the beginning of a procedure. At the end of the procedure, a branch instruction specifying the saved program counter will allow the program to return to the current point of execution. The saved program counter may be stored in a register, in memory, or on the stack.
 - (5) **Input/Output:** Transfer information concerning an input or output operation between the CPU and an I/O device.
- If required, a value is stored back into main memory.

1.1.1 TRAPS AND INTERRUPTS

Traps and *interrupts* are events that disrupt the normal sequence of instructions executed by the CPU. A trap is an abnormal condition detected by the CPU that usually is indicative of an error. Examples of trap conditions include dividing by zero, trying to access a memory location that does not exist or for which the program does not have access, executing an instruction with an undefined opcode, or trying to access a nonexistent I/O device.

An interrupt is a signal sent to the CPU by an external device, typically an I/O device. It is the CPU's equivalent of a pager, a signal requesting the CPU to interrupt its current activities to attend to the interrupting device's needs. A CPU will check interrupts only after it has completed the processing of one instruction and before it fetches a subsequent one.

The CPU responds to traps and interrupts by saving the current value of the program counter and resetting the program counter to a new address. This allows the CPU to return to executing at the point the trap or interrupt occurred, after it has executed a procedure for handling the trap or interrupt. The address the CPU jumps to is determined by the hardware architecture. On some machines, a unique address is associated with each trap and interrupt. More commonly, the architecture defines an address in memory to be the location of an *interrupt vector*. Each trap and interrupt is associated with an index into that vector. The branch address is determined by the contents of the memory location in the vector pointed to by the trap or interrupt's index.

Other state information is also typically saved when a trap or interrupt occurs. On many machines, that information is stored in a special *program status word* (PSW) register.

Interrupts may be associated with a hardware *priority level*. The CPU is also associated with a priority. Only interrupts with priorities higher than the CPU's are processed. An interrupt with a lower priority is left unprocessed until the CPU lowers its priority. CPU priority level is stored in the PSW and can be reset by changing the appropriate bits in the PSW.

One difference between traps and interrupts is traps are synchronous and interrupts are asynchronous. Given the same machine state and input data, a trap will occur at the exact same point of program execution each time a program runs. The occurrence of an interrupt, however, is dependent on the relative timing between the interrupting device and the CPU. Interrupts present a challenge to the debugging process since errors affected by the timing of the interrupt may not be easily repeatable.

1.1.2 MULTIMODE EXECUTION

To provide an operating system with privileges not granted to application programs, the hardware must support multiple modes of execution. Most commonly, two modes of execution are supported: *kernel* (or *supervisor*) *mode* and *user mode*. A single bit in the PSW records the system's execution mode. Attempts to perform certain activities while in user mode result in a trap. The restricted activities consist of those things normally reserved for the operating systems such as the execution of certain instructions (*privileged instructions*), accessing certain registers, and accessing I/O devices.

A system can enter kernel mode from user mode in one of three ways. A special instruction called a *supervisor call* (SVC) or a *system call* is similar to a procedure call except it sets the system's state to kernel mode. Unlike procedure call instructions, supervisor call instructions are not supplied with a branch address. The instruction's operand is a number that serves as an index into a vector similar to the interrupt vector. The branch address is determined by the contents of the memory location pointed to by the supervisor call operand. The vector is located in memory controlled by the operating system so the switch into kernel mode coincides with a jump to an operating system entry point.

Traps and interrupts are the other two mechanisms for switching into kernel mode. Like supervisor calls, the switch coincides with a jump to a kernel entry point. Application programs cannot change the system into kernel mode and remain executing in their own code.

On Unix systems, one user called the *superuser* is given special access privileges. The superuser may read or write any file and kill any process. The superuser should not be confused with kernel mode execution. Application programs running as the superuser are granted extraordinary access rights by the kernel, but those programs do not run in kernel mode and must use supervisor calls to request operating system services.

1.2 Operating System Structure

The operating system provides applications with a virtual machine. The supervisor calls implemented by the operating system expand the instruction set capability provided by the raw hardware. The supervisor calls support new abstractions such as processes and file systems.

In addition to providing the system call interface, the operating system has the responsibility for managing the underlying hardware resources. Applications cannot access I/O devices or execute privileged instructions. The operating system performs these tasks on behalf of the application programs. In doing so, it attempts to efficiently utilize the resources available to it and protect the integrity of the applications that must share those resources.

The tasks performed by an operating system can be divided into four areas.

- **Process Management:** A *process* is an executing program. Associated with a process are its code, its data, a set of resources allocated to it, and one or more “flows” of execution through its code. The operating system provides supervisor calls for managing processes and must manage the allocation of resources to processes. If multiple processes can exist simultaneously, the operating system must be capable of providing each process with an appropriate virtual environment in which it can run.
- **Memory Management:** At a minimum, memory must be shared by an application program and the operating system. On more sophisticated systems, memory can be shared by a number of processes. The operating system must manage the allocation of memory to processes and control the memory management hardware that determines which memory locations a process may access.
- **File System Management:** Computers process information. That information must be transmitted, processed, and stored. A file system object is an abstract entity for storing or transmitting a collection of information. The file system is an organized collection of file system objects. The operating system must provide primitives to manipulate those objects.
- **Device Management:** A computer communicates information through its input and output devices. Processes access those devices through the operating system supervisor calls provided for that purpose. The operating system attempts to manage those devices in a manner that allows them to be efficiently shared among the processes requiring them.

1.2.1 OPERATING SYSTEM TYPES

Users rarely had direct access to early computer systems. Computer input, both programs and data, was prepared on the input media, typically paper tape or punch cards. Users would hand their input to an operator and return minutes (if you were lucky) or hours later to pick up whatever output their program or “job” generated.

The operator would assemble the similar jobs into “batches” and run the batches through the computer. Each job had total control of the machine until it terminated.

A *batch* mode operating system manages a machine run in this manner. A batch operating system provides minimal functionality since it need not worry about the complications of sharing resources with multiple processes.

On *multiprogrammed batch* systems, jobs are read into a job pool stored on a disk. When one job is unable to execute because it is waiting for an I/O operation to complete, another job may be allowed to run. The sharing of computer resources by concurrently executing processes greatly increases operating system complexity.

Time-shared operating systems allow for interaction between user and process. In batch systems, all data is supplied at the time the program is input. This is fine for a program taking payroll information and printing weekly paychecks, but for programs that must interact with the user, the operating system must support an environment that allows programs to respond to user inputs in a reasonable amount of time. The operating system must not only share resources among the various processes, but it must create the illusion that processes are running simultaneously. It does this by shifting execution rapidly among all the active processes.

Increasingly, computers exist not as stand-alone entities but as part of a network of computers. At a basic level, this has limited impact on the operating system itself. An I/O device allows the computer to communicate on the network. However, network communications involve complex protocols and, for the reliability of the network, support for those protocols is built into a *networked* operating system. The general-purpose operating system of today is a networked time-shared system and is the focus of this book. Such systems may also have support for some form of batch processing.

A *real-time* operating system is designed to support execution of tasks within specific wall clock time constraints. Normally, users want the computer system to execute their programs as soon as possible, but no exact timing is required. In real-time systems, the correctness of a processing task is dependent on the wall clock time at which the processing occurred. For example, a real-time system sensing loss of coolant to a nuclear reactor may be required to initiate a backup system within a fraction of a second. The operating system must guarantee the task can be executed within a specified time constraint. Use of real-time systems is mostly limited to dedicated applications such as industrial control systems, weapon systems, and computer-controlled products.

What are desirable features in a time-shared system, resource sharing and management of I/O devices by the operating system, become detriments in a real-time system. Minimization of delays in the completion of tasks discourages resource sharing and encourages low-level access to hardware. For this reason, the design of general-purpose operating systems is much different from the design of real-time systems. The design of real-time systems will not be covered in this book.

Another specialized form of an operating system is a *distributed operating system*. With a network operating system, the resources on each machine on the network are managed by that machine’s operating system. Operating system support facilitates communications among the machines. With a distributed operating system, the operating systems on all the machines work together to manage the collective network resources. A single collective distributed operating system

manages the network resources provided by each network computer or *node*. Distributed operating systems are also beyond the scope of this book.

1.2.2 OPERATING SYSTEM KERNEL

The operating system *kernel* is the code designed to be executed while the hardware is executing in kernel mode. The kernel should not be considered a program. It is more accurately described as a subroutine library. One or more procedures in the library execute following a trap, an interrupt, or a supervisor call. At some point, the hardware is reset into user mode and control returns to a user program (although possibly not the same program that was executing when the kernel was entered).

To some, the kernel is the operating system. Each of the four areas of operating system management is implemented in the kernel, with supervisor calls triggering execution of functions in that area. In addition, device interrupts result in the execution of device management functions, and traps generated by memory management hardware activate memory management code. Every time the kernel is entered, the process scheduler may be called as the last step before returning control to a user program.

While systems featuring such large *monolithic* kernels are common, there has also been interest in *microkernel* operating systems. In these systems, the kernel contains minimal functionality. Essential functions such as control of memory management hardware, interprocess communications facilities, and interrupt and trap handling are provided by the kernel. Higher-level functions such as file system services are provided by server processes. A program seeking file system services would use interprocess communication system calls to send requests to the file system server process.

The use of server processes enhances flexibility. New servers can be activated without even rebooting the system. The small size of the microkernel simplifies the implementation and porting to new systems. Microkernels are well suited for distributed systems since, when using interprocess communication facilities, it makes little difference whether the process is on a local or remote system.

If the definition of an operating system is expanded to include some services provided by user-level processes, the distinction between system program and operating system becomes blurred. Services such as a print spooling system or a window manager are difficult to classify. An argument could be made for considering them part of the operating system's device management capabilities. In this book, the design features typically found in a monolithic kernel are explored. Although some of those features may be implemented outside the kernel, the design principles do not change. Higher-level services commonly considered to be systems programs, like print spooling systems, are not covered.

1.2.3 THE BOOT PROCESS

When power is first supplied to a computer system, a program in read-only memory (ROM) executes. After performing some diagnostic checks, a stage-0 boot program is executed. The program checks for the presence of one or more boot devices. On a PC, a boot device could be a floppy disk, a CD-ROM, or a hard disk. Once found,

the stage-0 boot program reads the first sector from the boot device into main memory. The first sector of the boot device is known as the *boot sector* and should contain a stage-1 boot program. After it has been loaded into memory, the stage-0 boot program branches to the stage-1 boot program.

There is no guarantee the boot sector will contain a boot program. If the boot sector has never been initialized, undefined values will be copied into memory. Before transferring control to the next stage in the boot process, the current boot program will check one or more bytes at the end of the boot sector for a “magic” bit pattern which is used to indicate that the sector contains a valid boot program.

On some systems, the stage-1 boot program then reads in the operating system. On others, a series of bootstrap programs must be copied into memory and executed. In either case, the operating system eventually gets copied into memory and the final boot program branches to the operating system’s initialization entry point.

On a PC booting from a hard disk, the boot sector also contains a *partition table*. The disk may be divided into at most four primary partitions. The partition table contains the starting and ending location of each partition. The partition table also defines one partition as *active*. The stage-1 boot program reads the partition table and copies into memory the first sector of the active partition. The first sector of the active partition contains a stage-2 boot program designed to work with the operating system contained in that partition. It may also have capabilities for loading boot programs or operating systems from other partitions. At its simplest, the stage-2 boot program loads the partition’s operating system into memory.

At the end of the boot process, the operating system has been loaded into memory and the boot program has branched to the operating system. The machine is executing in kernel mode. Any data structures (such as the interrupt vector), not set when the operating system was loaded, must be initialized. System registers and devices are initialized as is needed.

Once the operating system is ready to service processes, it creates any processes that provide operating systems services and also creates one or more processes to execute initialization system programs. On a simple system like DOS, this might only be a process to execute a command interpreter. On a more complicated system, an *init* system program may be provided that creates additional processes as specified in a configuration database. In either case, once the system is switched into user mode and control switches to the user mode processes, the operating system has completed its boot responsibilities and the system is running.

1.3 Outline of the Rest of This Book

Chapter 2, “Process Management,” and Chap. 3, “Interprocess Communication and Synchronization,” explore the design issues related to processes. Process characteristics, process scheduling, and supervisor calls for controlling processes are covered in Chap. 2. Chapter 3 covers those issues that arise when processes work cooperatively. Synchronization and communication primitives are discussed, as are ways of dealing with deadlock.

Chapter 4, “Memory Management,” and Chap. 5, “Virtual Memory,” discuss the allocation of memory to processes. The hardware support required to implement the various memory management options is discussed, as well as the operating system design issues. Chapter 5 focuses on those memory management schemes that allow the computer to operate as if it had more memory than it actually does.

Chapter 6, “File System Management,” looks at the structure of both the overall file system and the individual files within the file system.

Chapter 7, “Device Management,” covers the physical characteristics of key input/output devices and the operating system requirements for controlling those devices. Particular emphasis is placed on disks. A discussion of disk scheduling and RAID systems is included.

The book concludes with Chap. 8, “Security.” In many ways, security is more of a management and policy issue than an operating system design issue. However, at the very least, operating system design and security are dependent on each other. Chapter 8 presents a brief look at security measures and the type of threats they are designed to deal with.

Solved Problems



- 1.1 What are the two main functions of an operating system?

Answer:

The two main functions of an operating system are managing system resources and providing application programs with a set of primitives that provide higher-level services.

- 1.2 What does the CPU do when there are no programs to run?

Answer:

There is always a program to run (as long as the computer is turned on). The cycle of fetching, decoding, and executing instructions never stops. When there are no user programs to run, the operating system will execute in a loop that does nothing (called a *busy-wait loop* or *idle loop*) until an interrupt occurs.

- 1.3 What characteristic is common to traps, interrupts, supervisor calls, and subroutine calls?

Answer:

Traps, interrupts, supervisor calls, and subroutine calls all save the current value of the program counter and branch to a new location in memory.

- 1.4 What characteristic is common to traps, interrupts, and supervisor calls, but different in subroutine calls?

Answer:

Traps, interrupts, and supervisor calls cause the machine to shift into kernel mode. A subroutine call does not change the execution mode.

- 1.5 Which of the following instructions should be privileged (can only be executed in kernel mode)?
- (a) Change memory management registers
 - (b) Write the program counter
 - (c) Read the time-of-day clock
 - (d) Set the time-of-day clock
 - (e) Change processor priority

Answer:

- (a) Yes
Changing memory management registers would allow a process to access memory locations it was not authorized to access.
 - (b) No
Writing the program counter is no different than executing on unconditional branch.
 - (c) No
Although direct access to devices is usually unwise, read access of the clock should not be harmful.
 - (d) Yes
Changing the clock could disrupt scheduled events and is typically not a right granted to a user process.
 - (e) Yes
Changing the processor's priority could cause interrupts to be lost.
- 1.6 An operating system could implement a memory I/O device. I/O operations to the device cause the corresponding memory location to be read or written. What is the disadvantage of providing such a device? Should it be accessible to users or just to system administrators?

Answer:

If users could access a memory device, they could read or write any memory location, subverting the operating system's protection mechanisms. The operating system itself could be overwritten, giving user programs the ability to execute instructions in kernel mode. Providing that capability to any user, even the system administrator, threatens the integrity of the operating system.

1.7 Classify the following applications as batch-oriented or interactive.

- (a) Word processing
- (b) Generating monthly bank statements
- (c) Computing π to a million decimal places

Answer:

- (a) Interactive
- (b) Batch
- (c) Batch

This job is almost 100% CPU-bound. A batch system without multiprogramming would attain nearly 100% CPU utilization running this program.

1.8 Why must a computer start in kernel mode when power is first turned on?

Answer:

When power is first turned on, the contents of memory is undefined. The boot program in ROM must read the code from the boot device to load into memory. To perform the input operation, the hardware must be in kernel mode. Since the operating system is not loaded, there are no supervisor calls the ROM boot program could use if it was in user mode.

1.9 What is the maximum size of the stage-1 boot program at the beginning of a hard disk, assuming a 2-byte bootstrap magic bit pattern and a 512-byte sector size?

Answer:

The stage-1 boot program must fit in the initial sector. Assuming a sector size of 512 bytes, and 2 bytes for the magic bit pattern, the maximum size of the boot program is 510 bytes. (On PC systems, which also store a partition table in the first sector, the boot program can be at most 446 bytes.)

Supplementary Problems



1.10 Name hardware features designed to explicitly assist the operating system.

1.11 What characteristic is common to traps, supervisor calls, and subroutine calls, but different in interrupts?

1.12 What characteristic do subroutine calls and supervisor calls have, but traps and interrupts do not?