SOLUTIONS MANUAL

OPERATING SYSTEMS: INTERNALS AND DESIGN PRINCIPLES

FIFTH EDITION

WILLIAM STALLINGS

Author me

TABLE OF CONTENTS

Chapter 1: Computer System Overview 5
Chapter 2: Operating System Overview 11
Chapter 3: Process Description and Control 14
Chapter 4: Threads, SMP, and Microkernels..........18
Chapter 5: Concurrency: Mutual Exclusion and Synchronization..........21
Chapter 6: Concurrency: Deadlock and Starvation..........30
Chapter 7: Memory Management..........38
Chapter 8: Virtual Memory..........43
Chapter 9: Uniprocessor Scheduling..........51
Chapter 10: Multiprocessor and Real-Time Scheduling..........62
Chapter 11: I/O Management and Disk Scheduling..........65
Chapter 12: File Management..........71
Chapter 13: Networking..........74
Chapter 14: Distributed Processing, Client/Server, and Clusters..........76
Chapter 15: Distributed Process Management..........79
Chapter 16: Security 82

Chapter 1
Computer System Overview

Answers to Questions

1.1 A **main memory**, which stores both data and instructions: an **arithmetic and logic unit** (ALU) capable of operating on binary data; a **control unit**, which interprets the instructions in memory and causes them to be executed; and **input and output (I/O) equipment** operated by the control unit.

1.2 **User-visible registers:** Enable the machine- or assembly-language programmer to minimize main memory references by optimizing register use. For high-level languages, an optimizing compiler will attempt to make intelligent choices of which variables to assign to registers and which to main memory locations. Some high-level languages, such as C, allow the programmer to suggest to the compiler which variables should be held in registers. **Control and status registers:** Used by the processor to control the operation of the processor and by privileged, operating system routines to control the execution of programs.

1.3 These actions fall into four categories: **Processor-memory:** Data may be transferred from processor to memory or from memory to processor. **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module. **Data processing:** The processor may perform some arithmetic or logic operation on data. **Control:** An instruction may specify that the sequence of execution be altered.

1.4 An interrupt is a mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor.

1.5 Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be interrupted.

1.6 The three key characteristics of memory are cost, capacity, and access time.

1.7 Cache memory is a memory that is smaller and faster than main memory and that is interposed between the processor and main memory. The cache acts as a buffer for recently used memory locations.

1.8 **Programmed I/O:** The processor issues an I/O command, on behalf of a process, to an I/O module; that process then busy-waits for the operation to be completed before proceeding. **Interrupt-driven I/O:** The processor issues an I/O command on behalf of a process, continues to execute subsequent instructions, and is interrupted by the I/O module when the latter has completed its work. The subsequent instructions may be in the same process, if it is not necessary for that process to wait for the completion of the I/O. Otherwise, the process is suspended pending the interrupt and other work is performed. **Direct memory access (DMA):** A DMA module controls the exchange of data between main memory and an I/O module. The processor sends a request for the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

1.9 **Spatial locality** refers to the tendency of execution to involve a number of memory locations that are clustered. **Temporal locality** refers to the tendency for a processor to access memory locations that have been used recently.

1.10 **Spatial locality** is generally exploited by using larger cache blocks and by incorporating prefetching mechanisms (fetching items of anticipated use) into the cache control logic. **Temporal locality** is exploited by keeping recently used instruction and data values in cache memory and by exploiting a cache hierarchy.

Answers to Problems

1.1 Memory (contents in hex): 300: 3005;   301: 5940;   302: 7006
Step 1: 3005 → IR;   **Step 2:** 3 → AC
Step 3: 5940 → IR;   **Step 4:** 3 + 2 = 5 → AC
Step 5: 7006 → IR;   **Step 6:** AC → Device 6

1.2 **1.** **a.** The PC contains 300, the address of the first instruction. This value is loaded in to the MAR.
b. The value in location 300 (which is the instruction with the value 1940 in hexadecimal) is loaded into the MBR, and the PC is incremented. These two steps can be done in parallel.
c. The value in the MBR is loaded into the IR.
2. **a.** The address portion of the IR (940) is loaded into the MAR.
b. The value in location 940 is loaded into the MBR.
c. The value in the MBR is loaded into the AC.
3. **a.** The value in the PC (301) is loaded in to the MAR.
b. The value in location 301 (which is the instruction with the value 5941) is loaded into the MBR, and the PC is incremented.
c. The value in the MBR is loaded into the IR.
4. **a.** The address portion of the IR (941) is loaded into the MAR.
b. The value in location 941 is loaded into the MBR.
c. The old value of the AC and the value of location MBR are added and the result is stored in the AC.
5. **a.** The value in the PC (302) is loaded in to the MAR.
b. The value in location 302 (which is the instruction with the value 2941) is loaded into the MBR, and the PC is incremented.
c. The value in the MBR is loaded into the IR.
6. **a.** The address portion of the IR (941) is loaded into the MAR.
b. The value in the AC is loaded into the MBR.
c. The value in the MBR is stored in location 941.

1.3 **a.** 2^{24} = 16 MBytes
b. **(1)** If the local address bus is 32 bits, the whole address can be transferred at once and decoded in memory. However, since the data bus is only 16 bits, it will require 2 cycles to fetch a 32-bit instruction or operand.

(2) The 16 bits of the address placed on the address bus can't access the whole memory. Thus a more complex memory interface control is needed to latch the first part of the address and then the second part (since the microprocessor will end in two steps). For a 32-bit address, one may assume the first half will decode to access a "row" in memory, while the second half is sent later to access a "column" in memory. In addition to the two-step address operation, the microprocessor will need 2 cycles to fetch the 32 bit instruction/operand.

c. The program counter must be at least 24 bits. Typically, a 32-bit microprocessor will have a 32-bit external address bus and a 32-bit program counter, unless on-chip segment registers are used that may work with a smaller program counter. If the instruction register is to contain the whole instruction, it will have to be 32-bits long; if it will contain only the op code (called the op code register) then it will have to be 8 bits long.

1.4 In cases **(a)** and **(b)**, the microprocessor will be able to access 2^{16} = 64K bytes; the only difference is that with an 8-bit memory each access will transfer a byte, while with a 16-bit memory an access may transfer a byte or a 16-byte word. For case **(c)**, separate input and output instructions are needed, whose execution will generate separate "I/O signals" (different from the "memory signals" generated with the execution of memory-type instructions); at a minimum, one additional output pin will be required to carry this new signal. For case **(d)**, it can support 2^8 = 256 input and 2^8 = 256 output byte ports and the same number of input and output 16-bit ports; in either case, the distinction between an input and an output port is defined by the different signal that the executed input or output instruction generated.

1.5 Clock cycle = $\dfrac{1}{8\text{ MHz}}$ = 125 ns

Bus cycle = 4 × 125 ns = 500 ns
2 bytes transferred every 500 ns; thus transfer rate = 4 MBytes/sec

Doubling the frequency may mean adopting a new chip manufacturing technology (assuming each instructions will have the same number of clock cycles); doubling the external data bus means wider (maybe newer) on-chip data bus drivers/latches and modifications to the bus control logic. In the first case, the speed of the memory chips will also need to double (roughly) not to slow down the microprocessor; in the second case, the "wordlength" of the memory will have to double to be able to send/receive 32-bit quantities.

1.6 **a.** Input from the Teletype is stored in INPR. The INPR will only accept data from the Teletype when FGI=0. When data arrives, it is stored in INPR, and FGI is set to 1. The CPU periodically checks FGI. If FGI =1, the CPU transfers the contents of INPR to the AC and sets FGI to 0.

When the CPU has data to send to the Teletype, it checks FGO. If FGO = 0, the CPU must wait. If FGO = 1, the CPU transfers the contents of the AC to OUTR and sets FGO to 0. The Teletype sets FGI to 1 after the word is printed.

b. The process described in **(a)** is very wasteful. The CPU, which is much faster than the Teletype, must repeatedly check FGI and FGO. If interrupts are used, the Teletype can issue an interrupt to the CPU whenever it is ready to accept or send data. The IEN register can be set by the CPU (under programmer control)

1.7 If a processor is held up in attempting to read or write memory, usually no damage occurs except a slight loss of time. However, a DMA transfer may be to or from a device that is receiving or sending data in a stream (e.g., disk or tape), and cannot be stopped. Thus, if the DMA module is held up (denied continuing access to main memory), data will be lost.

1.8 Let us ignore data read/write operations and assume the processor only fetches instructions. Then the processor needs access to main memory once every microsecond. The DMA module is transferring characters at a rate of 1200 characters per second, or one every 833 μs. The DMA therefore "steals" every 833rd cycle. This slows down the processor approximately $\frac{1}{833} \times 100\% = 0.12\%$

1.9 **a.** The processor can only devote 5% of its time to I/O. Thus the maximum I/O instruction execution rate is $10^6 \times 0.05 = 50{,}000$ instructions per second. The I/O transfer rate is therefore 25,000 words/second.

b. The number of machine cycles available for DMA control is

$$10^6(0.05 \times 5 + 0.95 \times 2) = 2.15 \times 10^6$$

If we assume that the DMA module can use all of these cycles, and ignore any setup or status-checking time, then this value is the maximum I/O transfer rate.

1.10 **a.** A reference to the first instruction is immediately followed by a reference to the second.

b. The ten accesses to `a[i]` within the inner for loop which occur within a short interval of time.

1.11 Define

C_i = Average cost per bit, memory level i
S_i = Size of memory level i
T_i = Time to access a word in memory level i
H_i = Probability that a word is in memory i and in no higher-level memory
B_i = Time to transfer a block of data from memory level (i + 1) to memory level i

Let cache be memory level 1; main memory, memory level 2; and so on, for a total of N levels of memory. Then

$$C_S = \frac{\sum_{i=1}^{N} C_i S_i}{\sum_{i=1}^{N} S_i}$$

The derivation of T_s is more complicated. We begin with the result from probability theory that:

$$\text{Expected Value of } x = \sum_{i=1}^{N} i \Pr[x = 1]$$

We can write:

$$T_s = \sum_{i=1}^{N} T_i H_i$$

We need to realize that if a word is in M_1 (cache), it is read immediately. If it is in M_2 but not M_1, then a block of data is transferred from M_2 to M_1 and then read. Thus:

$$T_2 = B_1 + T_1$$

Further

$$T_3 = B_2 + T_2 = B_1 + B_2 + T_1$$

Generalizing:

$$T_i = \sum_{j=1}^{i-1} B_j + T_1$$

So

$$T_s = \sum_{i=2}^{N}\sum_{j=1}^{i-1}\left(B_j H_i\right) + T_1 \sum_{i=1}^{N} H_i$$

But $\sum_{i=1}^{N} H_i = 1$

Finally

$$T_s = \sum_{i=2}^{N}\sum_{j=1}^{i-1}\left(B_j H_i\right) + T_1$$

1.12 **a.** Cost = $C_m \times 8 \times 10^6 = 8 \times 10^3$ ¢ = \$80

b. Cost = $C_c \times 8 \times 10^6 = 8 \times 10^4$ ¢ = \$800

c. From Equation 1.1 : $1.1 \times T_1 = T_1 + (1 - H)T_2$

$(0.1)(100) = (1 - H)(1200)$

$H = 1190/1200$

1.13 There are three cases to consider:

Location of referenced word	Probability	Total time for access in ns
In cache	0.9	20
Not in cache, but in main memory	(0.1)(0.6) = 0.06	60 + 20 = 80
Not in cache or main memory	(0.1)(0.4) = 0.04	12ms + 60 + 20 = 12,000,080

So the average access time would be:

Avg = (0.9)(20) + (0.06)(80) + (0.04)(12000080) = 480026 ns

1.14 Yes, if the stack is only used to hold the return address. If the stack is also used to pass parameters, then the scheme will work only if it is the control unit that removes parameters, rather than machine instructions. In the latter case, the processor would need both a parameter and the PC on top of the stack at the same time.

CHAPTER 2 OPERATING SYSTEM OVERVIEW

ANSWERS TO QUESTIONS

2.1 **Convenience:** An operating system makes a computer more convenient to use. **Efficiency:** An operating system allows the computer system resources to be used in an efficient manner. **Ability to evolve:** An operating system should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

2.2 The kernel is a portion of the operating system that includes the most heavily used portions of software. Generally, the kernel is maintained permanently in main memory. The kernel runs in a privileged mode and responds to calls from processes and interrupts from devices.

2.3 Multiprogramming is a mode of operation that provides for the interleaved execution of two or more computer programs by a single processor.

2.4 A process is a program in execution. A process is controlled and scheduled by the operating system.

2.5 The **execution context,** or **process state,** is the internal data by which the operating system is able to supervise and control the process. This internal information is separated from the process, because the operating system has information not permitted to the process. The context includes all of the information that the operating system needs to manage the process and that the processor needs to execute the process properly. The context includes the contents of the various processor registers, such as the program counter and data registers. It also includes information of use to the operating system, such as the priority of the process and whether the process is waiting for the completion of a particular I/O event.

2.6 **Process isolation:** The operating system must prevent independent processes from interfering with each other's memory, both data and instructions. **Automatic allocation and management:** Programs should be dynamically allocated across the memory hierarchy as required. Allocation should be transparent to the programmer. Thus, the programmer is relieved of concerns relating to memory limitations, and the operating system can achieve efficiency by assigning memory to jobs only as needed. **Support of modular programming:** Programmers should be able to define program modules, and to create, destroy, and alter the size of modules dynamically. **Protection and access control:** Sharing of memory, at any level of the memory hierarchy, creates the potential for one program to address the memory space of another. This is desirable when sharing is needed by particular applications. At other times, it threatens the integrity of programs and even of the operating system itself. The operating system must allow portions of memory to be accessible in various ways by various users. **Long-term storage:** Many application programs require means for storing information for extended periods of time, after the computer has been powered down.

2.7 A **virtual address** refers to a memory location in virtual memory. That location is on disk and at some times in main memory. A real address is an address in main memory.

2.8 Round robin is a scheduling algorithm in which processes are activated in a fixed cyclic order; that is, all processes are in a circular queue. A process that cannot proceed because it is waiting for some event (e.g. termination of a child process or an input/output operation) returns control to the scheduler.

2.9 A **monolithic kernel** is a large kernel containing virtually the complete operating system, including scheduling, file system, device drivers, and memory management. All the functional components of the kernel have access to all of its internal data structures and routines. Typically, a monolithic kernel is implemented as a single process, with all elements sharing the same address space. A **microkernel** is a small privileged operating system core that provides process scheduling, memory management, and communication services and relies on other processes to perform some of the functions traditionally associated with the operating system kernel.

2.10 Multithreading is a technique in which a process, executing an application, is divided into threads that can run concurrently.

Answers to Problems

2.1 The answers are the same for **(a)** and **(b)**. Assume that although processor operations cannot overlap, I/O operations can.

1 Job:	TAT	=	NT	Processor utilization	=	50%
2 Jobs:	TAT	=	NT	Processor utilization	=	100%
4 Jobs:	TAT	=	(2N – 1)NT	Processor utilization	=	100%

2.2 I/O-bound programs use relatively little processor time and are therefore favored by the algorithm. However, if a processor-bound process is denied processor time for a sufficiently long period of time, the same algorithm will grant the processor to that process since it has not used the processor at all in the recent past. Therefore, a processor-bound process will not be permanently denied access.

2.3 With time sharing, the concern is turnaround time. Time-slicing is preferred because it gives all processes access to the processor over a short period of time. In a batch system, the concern is with throughput, and the less context switching, the more processing time is available for the processes. Therefore, policies that minimize context switching are favored.

2.4 A system call is used by an application program to invoke a function provided by the operating system. Typically, the system call results in transfer to a system program that runs in kernel mode.

2.5 The system operator can review this quantity to determine the degree of "stress" on the system. By reducing the number of active jobs allowed on the system, this average can be kept high. A typical guideline is that this average should be kept above 2 minutes [IBM86]. This may seem like a lot, but it isn't.

CHAPTER 3
PROCESS DESCRIPTION AND CONTROL

ANSWERS TO QUESTIONS

3.1 An instruction trace for a program is the sequence of instructions that execute for that process.

3.2 New batch job; interactive logon; created by OS to provide a service; spawned by existing process. See Table 3.1 for details.

3.3 **Running:** The process that is currently being executed. **Ready:** A process that is prepared to execute when given the opportunity. **Blocked:** A process that cannot execute until some event occurs, such as the completion of an I/O operation. **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the operating system. **Exit:** A process that has been released from the pool of executable processes by the operating system, either because it halted or because it aborted for some reason.

3.4 Process preemption occurs when an executing process is interrupted by the processor so that another process can be executed.

3.5 Swapping involves moving part or all of a process from main memory to disk. When none of the processes in main memory is in the Ready state, the operating system swaps one of the blocked processes out onto disk into a suspend queue, so that another process may be brought into main memory to execute.

3.6 There are two independent concepts: whether a process is waiting on an event (blocked or not), and whether a process has been swapped out of main memory (suspended or not). To accommodate this 2 × 2 combination, we need two Ready states and two Blocked states.

3.7 **1.** The process is not immediately available for execution. **2.** The process may or may not be waiting on an event. If it is, this blocked condition is independent of the suspend condition, and occurrence of the blocking event does not enable the process to be executed. **3.** The process was placed in a suspended state by an agent; either itself, a parent process, or the operating system, for the purpose of preventing its execution. **4.** The process may not be removed from this state until the agent explicitly orders the removal.

3.8 The OS maintains tables for entities related to memory, I/O, files, and processes. See Table 3.10 for details.

3.9 Process identification, processor state information, and process control information.

3.10 The user mode has restrictions on the instructions that can be executed and the memory areas that can be accessed. This is to protect the operating system from damage or alteration. In kernel mode, the operating system does not have these restrictions, so that it can perform its tasks.

3.11 **1.** Assign a unique process identifier to the new process. **2.** Allocate space for the process. **3.** Initialize the process control block. **4.** Set the appropriate linkages. **5.** Create or expand other data structures.

3.12 An interrupt is due to some sort of event that is external to and independent of the currently running process, such as the completion of an I/O operation. A trap relates to an error or exception condition generated within the currently running process, such as an illegal file access attempt.

3.13 Clock interrupt, I/O interrupt, memory fault.

3.14 A mode switch may occur without changing the state of the process that is currently in the Running state. A process switch involves taking the currently executing process out of the Running state in favor of another process. The process switch involves saving more state information.

Answers to Problems

3.1
- **Creation and deletion of both user and system processes.** The processes in the system can execute concurrently for information sharing, computation speedup, modularity, and convenience. Concurrent execution requires a mechanism for process creation and deletion. The required resources are given to the process when it is created, or allocated to it while it is running. When the process terminates, the OS needs to reclaim any reusable resources.
- **Suspension and resumption of processes.** In process scheduling, the OS needs to change the process's state to waiting or ready state when it is waiting for some resources. When the required resources are available, OS needs to change its state to running state to resume its execution.
- **Provision of mechanism for process synchronization.** Cooperating processes may share data. Concurrent access to shared data may result in data inconsistency. OS has to provide mechanisms for processes synchronization to ensure the orderly execution of cooperating processes, so that data consistency is maintained.
- **Provision of mechanism for process communication.** The processes executing under the OS may be either independent processes or cooperating processes. Cooperating processes must have the means to communicate with each other.
- **Provision of mechanisms for deadlock handling.** In a multiprogramming environment, several processes may compete for a finite number of resources. If a deadlock occurs, all waiting processes will never change their waiting state to running state again, resources are wasted and jobs will never be completed.

3.2 The following example is used in [PINK89] to clarify their definition of block and suspend:

> Suppose a process has been executing for a while and needs an additional magnetic tape drive so that it can write out a temporary file. Before it can initiate a write to tape, it must be given permission to use one of the drives. When it makes its request, a tape drive may not be available, and if that is the

case, the process will be placed in the blocked state. At some point, we assume the system will allocate the tape drive to the process; at that time the process will be moved back to the active state. When the process is placed into the execute state again it will request a write operation to its newly acquired tape drive. At this point, the process will be move to the suspend state, where it waits for the completion of the current write on the tape drive that it now owns.

The distinction made between two different reasons for waiting for a device could be useful to the operating system in organizing its work. However, it is no substitute for a knowledge of which processes are swapped out and which processes are swapped in. This latter distinction is a necessity and must be reflected in some fashion in the process state.

3.3 Figure 9.3 shows the result for a single blocked queue. The figure readily generalizes to multiple blocked queues.

3.4 Penalize the Ready, suspend processes by some fixed amount, such as one or two priority levels, so that a Ready, suspend process is chosen next only if it has a higher priority than the highest-priority Ready process by several levels of priority.

3.5 **a.** A separate queue is associated with each wait state. The differentiation of waiting processes into queues reduces the work needed to locate a waiting process when an event occurs that affects it. For example, when a page fault completes, the scheduler know that the waiting process can be found on the Page Fault Wait queue.

b. In each case, it would be less efficient to allow the process to be swapped out while in this state. For example, on a page fault wait, it makes no sense to swap out a process when we are waiting to bring in another page so that it can execute.

c. The state transition diagram can be derived from the following state transition table:

	Next State				
Current State	**Currently Executing**	**Computable (resident)**	**Computable (outswapped)**	**Variety of wait states (resident)**	**Variety of wait states (outswapped)**
Currently Executing		Rescheduled		Wait	
Computable (resident)	Scheduled		Outswap		
Computable (outswapped)		Inswap			
Variety of wait states (resident)		Event satisfied	Outswap		
Variety of wait states (outswapped)			Event satisfied		

3.6 **a.** The advantage of four modes is that there is more flexibility to control access to memory, allowing finer tuning of memory protection. The disadvantage is complexity and processing overhead. For example, procedures running at each of the access modes require separate stacks with appropriate accessibility.

b. In principle, the more modes, the more flexibility, but it seems difficult to justify going beyond four.

3.7 **a.** With j < i, a process running in D_i is prevented from accessing objects in D_j. Thus, if D_j contains information that is more privileged or is to be kept more secure than information in D_i, this restriction is appropriate. However, this security policy can be circumvented in the following way. A process running in D_j could read data in D_j and then copy that data into D_i. Subsequently, a process running in D_i could access the information.

b. An approach to dealing with this problem, known as a trusted system, is discussed in Chapter 16.

3.8 **a.** A application may be processing data received from another process and storing the results on disk. If there is data waiting to be taken from the other process, the application may proceed to get that data and process it. If a previous disk write has completed and there is processed data to write out, the application may proceed to write to disk. There may be a point where the process is waiting both for additional data from the input process and for disk availability.

b. There are several ways that could be handled. A special type of either/or queue could be used. Or the process could be put in two separate queues. In either case, the operating system would have to handle the details of alerting the process to the occurrence of both events, one after the other.

3.9 This technique is based on the assumption that an interrupted process *A* will continue to run after the response to an interrupt. But, in general, an interrupt may cause the basic monitor to preempt a process *A* in favor of another process *B*. It is now necessary to copy the execution state of process *A* from the location associated with the interrupt to the process description associated with *A*. The machine might as well have stored them there in the first place. Source: [BRIN73].

3.10 Because there are circumstances under which a process may not be preempted (i.e., it is executing in kernel mode), it is impossible for the operating system to respond rapidly to real-time requirements.

Chapter 4
Process Description and Control

Answers to Questions

4.1 This will differ from system to system, but in general, resources are owned by the process and each thread has its own execution state. A few general comments about each category in Table 3.5: **Identification:** the process must be identified but each thread within the process must have its own ID. **Processor State Information:** these are generally process-related. **Process control information:** scheduling and state information would mostly be at the thread level; data structuring could appear at both levels; interprocess communication and interthread communication may both be supported; privileges may be at both levels; memory management would generally be at the process level; and resource info would generally be at the process level.

4.2 Less state information is involved.

4.3 Resource ownership and scheduling/execution.

4.4 Foreground/background work; asynchronous processing; speedup of execution by parallel processing of data; modular program structure.

4.5 Address space, file resources, execution privileges are examples.

4.6 **1.** Thread switching does not require kernel mode privileges because all of the thread management data structures are within the user address space of a single process. Therefore, the process does not switch to the kernel mode to do thread management. This saves the overhead of two mode switches (user to kernel; kernel back to user). **2.** Scheduling can be application specific. One application may benefit most from a simple round-robin scheduling algorithm, while another might benefit from a priority-based scheduling algorithm. The scheduling algorithm can be tailored to the application without disturbing the underlying OS scheduler. **3.** ULTs can run on any operating system. No changes are required to the underlying kernel to support ULTs. The threads library is a set of application-level utilities shared by all applications.

4.7 **1.** In a typical operating system, many system calls are blocking. Thus, when a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked. **2.** In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing. A kernel assigns one process to only one processor at a time. Therefore, only a single thread within a process can execute at a time.

4.8 Jacketing converts a blocking system call into a nonblocking system call by using an application-level I/O routine which checks the status of the I/O device.

4.9 **SIMD:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that each instruction is executed on a different set of data by the different processors.. **MIMD:** A set of processors simultaneously execute different instruction sequences on different data sets. **Master/slave:** The operating system kernel always runs on a particular processor. The other processors may only execute user programs and perhaps operating system utilities. **SMP:** the kernel can execute on any processor, and typically each processor does self-scheduling from the pool of available processes or threads. **Cluster:** Each processor has a dedicated memory, and is a self-contained computer.

4.10 Simultaneous concurrent processes or threads; scheduling; synchronization; memory management; reliability and fault tolerance.

4.11 Device drivers, file systems, virtual memory manager, windowing system, and security services.

4.12 **Uniform interfaces:** Processes need not distinguish between kernel-level and user-level services because all such services are provided by means of message passing. **Extensibility:** facilitates the addition of new services as well as the provision of multiple services in the same functional area. **Flexibility:** not only can new features be added to the operating system, but existing features can be subtracted to produce a smaller, more efficient implementation. **Portability:** all or at least much of the processor-specific code is in the microkernel; thus, changes needed to port the system to a new processor are fewer and tend to be arranged in logical groupings. **Reliability:** A small microkernel can be rigorously tested. Its use of a small number of application programming interfaces (APIs) improves the chance of producing quality code for the operating-system services outside the kernel. **Distributed system support:** the message orientation of microkernal communication lends itself to extension to distributed systems. **Support for object-oriented operating system (OOOS):** An object-oriented approach can lend discipline to the design of the microkernel and to the development of modular extensions to the operating system.

4.13 It takes longer to build and send a message via the microkernel, and accept and decode the reply, than to make a single service call.

4.14 These functions fall into the general categories of low-level memory management, inter-process communication (IPC), and I/O and interrupt management.

4.15 Messages.

Answers to Problems

4.1 Yes, because more state information must be saved to switch from one process to another.

4.2 Because, with ULTs, the thread structure of a process is not visible to the operating system, which only schedules on the basis of processes.

4.3 **a.** The use of sessions is well suited to the needs of an interactive graphics interface for personal computer and workstation use. It provides a uniform mechanism for keeping track of where graphics output and keyboard/mouse input should be directed, easing the task of the operating system.

b. The split would be the same as any other process/thread scheme, with address space and files assigned at the process level.

4.4 The issue here is that a machine spends a considerable amount of its waking hours waiting for I/O to complete. In a multithreaded program, one KLT can make the blocking system call, while the other KLTs can continue to run. On uniprocessors, a process that would otherwise have to block for all these calls can continue to run its other threads. Source: [LEWI96]

4.5 No. When a process exits, it takes everything with it—the KLTs, the process structure, the memory space, everything—including threads. Source: [LEWI96]

4.6 As much information as possible about an address space can be swapped out with the address space, thus conserving main memory.

4.7 **a.** If a conservative policy is used, at most 20/4 = 5 processes can be active simultaneously. Because one of the drives allocated to each process can be idle most of the time, at most 5 drives will be idle at a time. In the best case, none of the drives will be idle.

b. To improve drive utilization, each process can be initially allocated with three tape drives. The fourth one will be allocated on demand. In this policy, at most $\lfloor 20/3 \rfloor = 6$ processes can be active simultaneously. The minimum number of idle drives is 0 and the maximum number is 2. Source: *Advanced Computer Architecture*, K. Hwang, 1993.

4.8 Every call that can possibly change the priority of a thread or make a higher-priority thread runnable will also call the scheduler, and it in turn will preempt the lower-priority active thread. So there will never be a runnable, higher-priority thread. Source: [LEWI96]

CHAPTER 5
CONCURRENCY: MUTUAL EXCLUSION AND SYNCHRONIZATION

ANSWERS TO QUESTIONS

5.1 Communication among processes, sharing of and competing for resources, synchronization of the activities of multiple processes, and allocation of processor time to processes.

5.2 Multiple applications, structured applications, operating-system structure.

5.3 The ability to enforce mutual exclusion.

5.4 **Processes unaware of each other:** These are independent processes that are not intended to work together. **Processes indirectly aware of each other:** These are processes that are not necessarily aware of each other by their respective process IDs, but that share access to some object, such as an I/O buffer. **Processes directly aware of each other:** These are processes that are able to communicate with each other by process ID and which are designed to work jointly on some activity.

5.5 Competing processes need access to the same resource at the same time, such as a disk, file, or printer. Cooperating processes either share access to a common object, such as a memory buffer or are able to communicate with each other, and cooperate in the performance of some application or activity.

5.6 **Mutual exclusion:** competing processes can only access a resource that both wish to access one at a time; mutual exclusion mechanisms must enforce this one-at-a-time policy. **Deadlock:** if competing processes need exclusive access to more than one resource then deadlock can occur if each processes gained control of one resource and is waiting for the other resource. **Starvation:** one of a set of competing processes may be indefinitely denied access to a needed resource because other members of the set are monopolizing that resource.

5.7 **1.** Mutual exclusion must be enforced: only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object. **2.** A process that halts in its non-critical section must do so without interfering with other processes. **3.** It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation. **4.** When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay. **5.** No assumptions are made about relative process speeds or number of processors. **6.** A process remains inside its critical section for a finite time only.

5.8 **1.** A semaphore may be initialized to a nonnegative value. **2.** The *wait* operation decrements the semaphore value. If the value becomes negative, then the process executing the *wait* is blocked. **3.** The *signal* operation increments the semaphore value. If the value is not positive, then a process blocked by a *wait* operation is unblocked.

5.9 A binary semaphore may only take on the values 0 and 1. A general semaphore may take on any integer value.

5.10 A strong semaphore requires that processes that are blocked on that semaphore are unblocked using a first-in-first-out policy. A weak semaphore does not dictate the order in which blocked processes are unblocked.

5.11 A monitor is a programming language construct providing abstract data types and mutually exclusive access to a set of procedures

5.12 There are two aspects, the send and receive primitives. When a *send* primitive is executed in a process, there are two possibilities: either the sending process is blocked until the message is received, or it is not. Similarly, when a process issues a *receive* primitive, there are two possibilities: If a message has previously been sent, the message is received and execution continues. If there is no waiting message, then either (a) the process is blocked until a message arrives, or (b) the process continues to execute, abandoning the attempt to receive.

5.13 **1.** Any number of readers may simultaneously read the file. **2.** Only one writer at a time may write to the file. **3.** If a writer is writing to the file, no reader may read it.

Answers to Problems

5.1 **b.** The `read` coroutine reads the cards and passes characters through a one-character buffer, `rs`, to the `squash` coroutine. The `read` coroutine also passes the extra blank at the end of every card image. The `squash` coroutine need known nothing about the 80-character structure of the input; it simply looks for double asterisks and passes a stream of modified characters to the `print` coroutine via a one-character buffer, `sp`. Finally, `print` simply accepts an incoming stream of characters and prints it as a sequence of 125-character lines.

5.2 ABCDE; ABDCE; ABDEC; ADBCE; ADBEC; ADEBC;
DEABC; DAEBC; DABEC; DABCE

5.3 **a.** On casual inspection, it appears that `tally` will fall in the range 50 ≤ `tally` ≤ 100 since from 0 to 50 increments could go unrecorded due to the lack of mutual exclusion. The basic argument contends that by running these two processes concurrently we should not be able to derive a result lower than the result produced by executing just one of these processes sequentially. But consider the following interleaved sequence of the load, increment, and store operations performed by these two processes when altering the value of the shared variable:

1. Process A loads the value of `tally`, increments *tally*, but then loses the processor (it has incremented its register to 1, but has not yet stored this value.

2. Process B loads the value of `tally` (still zero) and performs forty-nine complete increment operations, losing the processor after it has stored the value 49 into the shared variable `tally`.
3. Process A regains control long enough to perform its first store operation (replacing the previous `tally` value of 49 with 1) but is then immediately forced to relinquish the processor.
4. Process B resumes long enough to load 1 (the current value of *tally*) into its register, but then it too is forced to give up the processor (note that this was B's final load).
5. Process A is rescheduled, but this time it is not interrupted and runs to completion, performing its remaining 49 load, increment, and store operations, which results in setting the value of `tally` to 50.
6. Process B is reactivated with only one increment and store operation to perform before it terminates. It increments its register value to 2 and stores this value as the final value of the shared variable.

Some thought will reveal that a value lower than 2 cannot occur. Thus, the proper range of final values is 2 ≤ `tally` ≤ 100.

b. For the generalized case of N processes, the range of final values is 2 ≤ `tally` ≤ (N × 50), since it is possible for all other processes to be initially scheduled and run to completion in step (5) before Process B would finally destroy their work by finishing last.

Source: [RUDO90]. A slightly different formulation of the same problem appears in [BEN98]

5.4 On average, yes, because busy-waiting consumes useless instruction cycles. However, in a particular case, if a process comes to a point in the program where it must wait for a condition to be satisfied, and if that condition is already satisfied, then the busy-wait will find that out immediately, whereas, the blocking wait will consume OS resources switching out of and back into the process.

5.5 Consider the case in which `turn` equals 0 and `P(1)` sets `blocked[1]` to `true` and then finds `blocked[0]` set to `false`. `P(0)` will then set `blocked[0]` to `true`, find `turn` = 0, and enter its critical section. `P(1)` will then assign 1 to `turn` and will also enter its critical section. The error was pointed out in [RAYN86].

5.6 **a.** When a process wishes to enter its critical section, it is assigned a ticket number. The ticket number assigned is calculated by adding one to the largest of the ticket numbers currently held by the processes waiting to enter their critical section and the process already in its critical section. The process with the smallest ticket number has the highest precedence for entering its critical section. In case more than one process receives the same ticket number, the process with the smallest numerical name enters its critical section. When a process exits its critical section, it resets its ticket number to zero.

b. If each process is assigned a unique process number, then there is a unique, strict ordering of processes at all times. Therefore, deadlock cannot occur.

c. To demonstrate mutual exclusion, we first need to prove the following lemma: if Pi is in its critical section, and Pk has calculated its number[k] and is attempting to enter its critical section, then the following relationship holds:

(number[i], i) < (number[k], k)

To prove the lemma, define the following times:

T_{w1} Pi reads choosing[k] for the last time, for j = k, in its first wait, so we have choosing[k] = false at T_{w1}.

T_{w2} Pi begins its final execution, for j = k, of the second **while** loop. We therefore have $T_{w1} < T_{w2}$.

T_{k1} Pk enters the beginning of the **repeat** loop.

T_{k2} Pk finishes calculating number[k].

T_{k3} Pk sets choosing[k] to false. We have $T_{k1} < T_{k2} < T_{k3}$.

Since at Tw1, choosing[k] = false, we have either $T_{w1} < T_{k1}$ or $T_{k3} < T_{w1}$. In the first case, we have number[i] < number[k], since Pi was assigned its number prior to Pk; this satisfies the condition of the lemma.

In the second case, we have $T_{k2} < T_{k3} < T_{w1} < T_{w2}$, and therefore $T_{k2} < T_{w2}$. This means that at T_{w2}, Pi has read the current value of number[k]. Moreover, as T_{w2} is the moment at which the final execution of the second **while** for j = k takes place, we have (number[i], i) < (number[k], k), which completes the proof of the lemma.

It is now easy to show the mutual exclusion is enforced. Assume that Pi is in its critical section and Pk is attempting to enter its critical section. Pk will be unable to enter its critical section, as it will find number[i] ≠ 0 and (number[i], i) < (number[k], k).

5.7 The following program is provided in [SILB98]:

```
var    j: 0..n-1;
       key: boolean;
repeat
    waiting[i] := true;
    key := true;
    while waiting[i] and key do key := testset(lock);
    waiting[i] := false;
    < critical section >
    j := i + 1 mod n;
    while (j ≠ i) and (not waiting[j]) do j := j + 1 mod n;
    if j = i then lock := false
        else waiting := false;
    < remainder section >
until false;
```

The algorithm uses the common data structures

```
var  waiting: array [0..n – 1] of boolean
     lock: boolean
```

These data structures are initialized to false. When a process leaves its critical section, it scans the array *waiting* in the cyclic ordering (i + 1, i + 2, ..., n – 1, 0, ..., i – 1). It designates the first process in this ordering that is in the entry section (waiting[j] = true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n – 1 turns.

5.8 The two are equivalent. In the definition of Figure 5.3, when the value of the semaphore is negative, its value tells you how many processes are waiting. With the definition of this problem, you don't have that information readily available. However, the two versions function the same.

5.9 Suppose two processes each call `semWait(s)` when `s` is initially 0, and after the first has just done `semSignalB(mutex)` but not done `semWaitB(delay)`, the second call to `semWait(s)` proceeds to the same point. Because `s` = –2 and `mutex` is unlocked, if two other processes then successively execute their calls to `semSignal(s)` at that moment, they will each do `semSignalB(delay)`, but the effect of the second `semSignalB` is not defined.

The solution is to move the **else** line, which appears just before the **end** line in `semWait` to just before the **end** line in `semSignal`. Thus, the last `semSignalB(mutex)` in `semWait` becomes unconditional and the `semSignalB(mutex)` in `semSignal` becomes conditional. For a discussion, see "A Correct Implementation of General Semaphores," by Hemmendinger, *Operating Systems Review*, July 1988.

5.10 The program is found in [RAYN86]:

```
var a, b, m: semaphore;
     na, nm: 0 … +∞;
a := 1; b := 1; m := 0; na := 0; nm := 0;
semWait(b); na ← na + 1; semSignal(b);
semWait(a); nm ← nm + 1;
            semwait(b); na ← na – 1;
            if na = 0 then semSignal(b); semSignal(m)
                       else semSignal(b); semSignal(a)
            endif;
semWait(m); nm ← nm – 1;
<critical section>;
if nm = 0 then semSignal(a)
             else semSignal(m)
endif;
```

5.11 The code has a major problem. The `V(passenger_released)` in the car code can unblock a passenger blocked on `P(passenger_released)` that is NOT the one riding in the car that did the `V()`.

5.12

	Producer	Consumer	s	n	delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		**if** (n==0) (semWaitB(delay))			
10	semWaitB(s)				

Both producer and consumer are blocked.

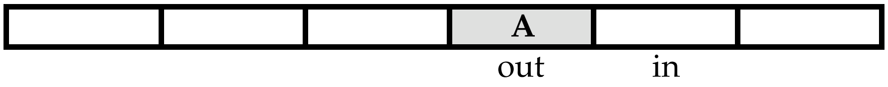
5.13 This solution is from [BEN82].

```
program     producerconsumer;
var         n: integer;
            s: (*binary*) semaphore (:= 1);
            delay: (*binary*) semaphore (:= 0);
procedure producer;
begin
    repeat
        produce;
        semWaitB(s);
        append;
        n := n + 1;
        if n=0 then semSignalB(delay);
        semSignalB(s)
    forever
end;
procedure consumer;
begin
    repeat
        semWaitB(s);
        take;
        n := n – 1;
        if n = -1 then
            begin
                semSignalB(s);
                semWaitB(delay);
                semWaitB(s)
            end;
        consume;
        semSignalB(s)
    forever
end;
begin (*main program*)
```

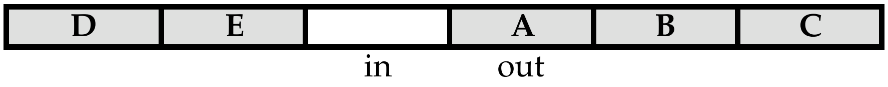
```
     n := 0;
     parbegin
          producer; consumer
     parend
end.
```

5.14 Any of the interchanges listed would result in an incorrect program. The semaphore `s` controls access to the critical region and you only want the critical region to include the `append` or `take` function.

5.15 **a.** If the buffer is allowed to contain n entries, then the problem is to distinguish an empty buffer from a full one. Consider a buffer of six slots, with only one entry, as follows:



Then, when that one element is removed, out = in. Now suppose that the buffer is one element shy of being full:



Here, out = in + 1. But then, when an element is added, in is incremented by 1 and out = in, the same as when the buffer is empty.

b. You could use an auxiliary variable, `count`, which is incremented and decremented appropriately.

5.16

```
#define REINDEER 9   /* max # of reindeer
*/
#define ELVES       3   /* size of elf group */
               /* Semaphores */
only_elves = 3,            /* 3 go to Santa */
emutex = 1,                /* update elf_cnt */
rmutex = 1,                /* update rein_ct */
rein_semWait = 0,          /* block early arrivals
                              back from islands */
sleigh = 0,                /*all reindeer semWait
                             around the sleigh */
done = 0,                  /* toys all delivered */
santa_semSignal = 0,       /* 1st 2 elves semWait
on
                        this outside Santa's shop
*/
santa = 0,                 /* Santa sleeps on this
                             blocked semaphore
*/
problem = 0,               /* semWait to pose
the
                               question to Santa */
elf_done = 0;              /* receive reply */
              /* Shared Integers */
rein_ct = 0;               /* # of reindeer back
*/
elf_ct = 0;            /* # of elves with problem
*/
              /* Reindeer Process */
for (;;) {
  tan on the beaches in the Pacific until
      Christmas is close
  semWait (rmutex)
      rein_ct++
      if (rein_ct == REINDEER) {
          semSignal (rmutex)
          semSignal (santa)
      }
      else {
           semSignal (rmutex)
           semWait (rein_semWait)
      }
/* all reindeer semWaiting to be attached to
sleigh */
  semWait (sleigh)
   fly off to deliver toys
   semWait (done)
   head back to the Pacific islands
} /* end "forever" loop */
```

```
                 /* Elf Process */
for (;;) {
  semWait (only_elves)          /* only 3 elves
"in" */
     semWait (emutex)
         elf_ct++
         if (elf_ct == ELVES) {
            semSignal (emutex)
            semSignal (santa)  /* 3rd elf wakes
Santa */
         }
         else {
             semSignal (emutex)
             semWait (santa _semSignal)  /*
semWait outside
                                    Santa's shop door */
         }
     semWait (problem)
      ask question      /* Santa woke elf up */
      semWait (elf_done)
  semSignal (only_elves)
} /* end "forever" loop */
               /* Santa Process */
for (;;) {
  semWait (santa)                 /* Santa "rests" */
  /* mutual exclusion is not needed on rein_ct
     because if it is not equal to REINDEER,
     then elves woke up Santa */
   if (rein_ct == REINDEER) {
     semWait (rmutex)
     rein_ct = 0            /* reset while blocked */
     semSignal (rmutex)
     for (i = 0; i < REINDEER – 1; i++)
          semSignal (rein_semWait)
     for (i = 0; i < REINDEER; i++)
          semSignal (sleigh)
     deliver all the toys and return
     for (i = 0; i < REINDEER; i++)
          semSignal (done)
   }
    else {                  /* 3 elves have arrive */
     for (i = 0; i < ELVES – 1; i++)
          semSignal (santa_semSignal)
     semWait (emutex)
           elf_ct = 0
     semSignal (emutex)
     for (i = 0; i < ELVES; i++)  {
          semSignal (problem)
          answer that question
          semSignal (elf_done)
     }
   }
} /* end "forever" loop */
```

5.17 **a.** There is an array of message slots that constitutes the buffer. Each process maintains a linked list of slots in the buffer that constitute the mailbox for that process. The message operations can implemented as:

```
send (message, dest)
semWait (mbuf)                        semWait for message buffer available
semWait (mutex)                       mutual exclusion on message queue
acquire free buffer slog
copy message to slot
link slot to other messages
semSignal (dest.sem)                  wake destination process
semSignal (mutex)                     release mutual exclusion

receive (message)
semWait (own.sem)                     semWait for message to arrive
semWait (mutex)                       mutual exclusion on message queue
unlink slot from own.queue
copy buffer slot to message
add buffer slot to freelist
semSignal (mbuf)                      indicate message slot freed
semSignal (mutex)                     release mutual exclusion
```

where mbuf is initialized to the total number of message slots available; own and dest refer to the queue of messages for each process, and are initially zero.

b. This solution is taken from [TANE97]. The synchronization process maintains a counter and a linked list of waiting processes for each semaphore. To do a WAIT or SIGNAL, a process calls the corresponding library procedure, *wait* or *signal*, which sends a message to the synchronization process specifying both the operation desired and the semaphore to be used. The library procedure then does a RECEIVE to get the reply from the synchronization process.

When the message arrives, the synchronization process checks the counter to see if the required operation can be completed. SIGNALs can always complete, but WAITs will block if the value of the semaphore is 0. If the operation is allowed, the synchronization process sends back an empty message, thus unblocking the caller. If, however, the operation is a WAIT and the semaphore is 0, the synchronization process enters the caller onto the queue and does not send a reply. The result is that the process doing the WAIT is blocked, just as it should be. Later, when a SIGNAL is done, the synchronization process picks one of the processes blocked on the semaphore, either in FIFO order, priority order, or some other order, and sends a reply. Race conditions are avoided here because the synchronization process handles only one request at a time.

CHAPTER 6
CONCURRENCY: DEADLOCK AND STARVATION

ANSWERS TO QUESTIONS

6.1 Examples of reusable resources are processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores. Examples of consumable resources are interrupts, signals, messages, and information in I/O buffers.

6.2 **Mutual exclusion.** Only one process may use a resource at a time. **Hold and wait.** A process may hold allocated resources while awaiting assignment of others. **No preemption.** No resource can be forcibly removed from a process holding it.

6.3 The above three conditions, plus: **Circular wait.** A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

6.4 The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time, and blocking the process until all requests can be granted simultaneously.

6.5 First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource. Alternatively, if a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.

6.6 The circular-wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.

6.7 **Deadlock prevention** constrains resource requests to prevent at least one of the four conditions of deadlock; this is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption), or directly, by preventing circular wait. **Deadlock avoidance** allows the three necessary conditions, but makes judicious choices to assure that the deadlock point is never reached. With **deadlock detection**, requested resources are granted to processes whenever possible.; periodically, the operating system performs an algorithm that allows it to detect the circular wait condition.

ANSWERS TO PROBLEMS

6.1 **Mutual exclusion:** Only one car can occupy a given quadrant of the intersection at a time. **Hold and wait:** No car ever backs up; each car in the intersection waits until the quadrant in front of it is available. **No preemption:** No car is allowed to force another car out of its way. **Circular wait:** Each car is waiting for a quadrant of the intersection occupied by another car.

6.2 **1.** Q acquires B and A, and then releases B and A. When P resumes execution, it will be able to acquire both resources.
2. Q acquires B and A. P executes and blocks on a request for A. Q releases B and A. When P resumes execution, it will be able to acquire both resources.
3. Q acquires B and then P acquires and releases A. Q acquires A and then releases B and A. When P resumes execution, it will be able to acquire B.
4. P acquires A and then Q acquires B. P releases A. Q acquires A and then releases B. P acquires B and then releases B.
5. P acquires and then releases A. P acquires B. Q executes and blocks on request for B. P releases B. When Q resumes execution, it will be able to acquire both resources.
6. P acquires A and releases A and then acquires and releases B. When Q resumes execution, it will be able to acquire both resources.

6.3 If Q acquires B and A before P requests A, then Q can use these two resources and then release them, allowing A to proceed. If P acquires A before Q requests A, then at most Q can proceed to the point of requesting A and then is blocked. However, once P releases A, Q can proceed. Once Q releases B, A can proceed.

6.4 **a.** 0 0 0 0
0 7 5 0
6 6 2 2
2 0 0 2
0 3 2 0

b. to d. Running the banker's algorithm, we see processes can finish in the order p1, p4, p5, p2, p3.

e. Change available to (2,0,0,0) and p3's row of "still needs" to (6,5,2,2). Now p1, p4, p5 can finish, but with available now (4,6,9,8) neither p2 nor p3's "still needs" can be satisfied. So it is not safe to grant p3's request.

6.5 1. W = (2 1 0 0)
2. Mark P3; W = (2 1 0 0) + (0 1 2 0) = (2 2 2 0)
3. Mark P2; W = (2 2 2 0) + (2 0 0 1) = (4 2 2 1)
4. Mark P1; no deadlock detected

6.6 A deadlock occurs when process I has filled the disk with input ($i = max$) and process i is waiting to transfer more input to the disk, while process P is waiting to transfer more output to the disk and process O is waiting to transfer more output from the disk. Source: [BRIN73].

6.7 Reserve a minimum number of blocks (called *reso*) permanently for output buffering, but permit the number of output blocks to exceed this limit when disk space is available. The resource constraints now become:

$$i + o \le max$$
$$i \le max - reso$$

where

$$0 < reso < max$$

If process P is waiting to deliver output to the disk, process O will eventually consume all previous output and make at least *reso* pages available for further output, thus enabling P to continue. So P cannot be delayed indefinitely by O. Process I can be delayed if the disk is full of I/O; but sooner or later, all previous input will be consumed by P and the corresponding output will be consumed by O, thus enabling I to continue. Source: [BRIN73].

6.8

$$\begin{array}{ll} i + o + p & \leq max - \\ i + o & \leq max - resp \\ i + p & \leq max - reso \\ i & \leq max - (reso + resp) \end{array}$$

Source: [BRIN73].

6.9 **a.**
1. $i \leftarrow i + 1$
2. $i \leftarrow i - 1;\ p \leftarrow p + 1$
3. $p \leftarrow p - 1;\ o \leftarrow o + 1$
4. $o \leftarrow o - 1$
5. $p \leftarrow p + 1$
6. $p \leftarrow p - 1$

b. By examining the resource constraints listed in the solution to problem 6.7, we can conclude the following:

6. Procedure returns can take place immediately because they only release resources.
5. Procedure calls may exhaust the disk ($p = max - reso$) and lead to deadlock.
4. Output consumption can take place immediately after output becomes available.
3. Output production can be delayed temporarily until all previous output has been consumed and made at least *reso* pages available for further output.
2. Input consumption can take place immediately after input becomes available.
1. Input production can be delayed until all previous input and the corresponding output has been consumed. At this point, when $i = o = 0$, input can be produced provided the user processes have not exhausted the disk ($p < max - reso$).

Conclusion: the uncontrolled amount of storage assigned to the user processes is the only possible source of a storage deadlock. Source: [BRIN73].

6.10 **a.** Creating the process would result in the state:

Process	Max	Hold	Claim	Free
1	70	45	25	25
2	60	40	20	
3	60	15	45	
4	60	25	35	

There is sufficient free memory to guarantee the termination of either P1 or P2. After that, the remaining three jobs can be completed in any order.

b. Creating the process would result in the trivially unsafe state:

Process	Max	Hold	Claim	Free
1	70	45	25	15
2	60	40	20	
3	60	15	45	
4	60	35	25	

6.11 It is unrealistic: don't know max demands in advance, number of processes can change over time, number of resources can change over time (something can break). Most OS's ignore deadlock. But Solaris only lets the superuser use the last process table slot.

6.12 a. The buffer is declared to be an array of shared elements of type T. Another array defines the number of input elements *available* to each process. Each process keeps track of the index *j* of the buffer element it is referring to at the moment.

```
var  buffer: array 0..max-1 of shared T;
     available: shared array 0..n-1 of 0..max;

"Initialization"
var K: 1..n-1;
region available do
begin
    available(0) := max;
    for every k do available (k) := 0;
end

"Process i"
var j: 0..max-1; succ: 0..n-1;
begin
    j := 0; succ := (i+1) mod n;
    repeat
        region available do
        await available (i) > 0;
        region buffer(j) do consume element;
        region available do
        begin
            available (i) := available(i) – 1;
            available (succ) := available (succ) + 1;
        end
        j := (j+1) mod max;
    forever
end
```

In the above program, the construct region defines a critical region using some appropriate mutual-exclusion mechanism. The notation

region v **do** S

means that at most one process at a time can enter the critical region associated with variable v to perform statement S.

b. A deadlock is a situation in which:

P_0 waits for P_{n-1} AND
P_1 waits for P_0 AND
. . . . .
P_{n-1} waits for P_{n-2}

because

(available (0) = 0) AND
(available (1) = 0) AND
. . . . .
(available (n-1) = 0)

But if max > 0, this condition cannot hold because the critical regions satisfy the following invariant:

$$\sum_{i=1}^{N} claim(i) < N \sum_{i=0}^{n-1} available(i) = \text{max}$$

Source: [BRIN73].

6.13 a. Deadlock occurs if all resource units are reserved while one or more processes are waiting indefinitely for more units. But, if all 4 units are reserved, at least one process has acquired 2 units. Therefore, that process will be able to complete its work and release both units, thus enabling another process to continue.

b. Using terminology similar to that used for the banker's algorithm, define claim[i] = total amount of resource units needed by process i; allocation[i] = current number of resource units allocated to process i; and deficit[i] = amount of resource units still needed by i. Then we have:

$$\sum_{i}^{N} claim[i] = \sum_{i}^{N} deficit[i] + \sum_{i}^{N} allocation[i] < M + N$$

In a deadlock situation, all resource units are reserved:

$$\sum_{i}^{N} allocation[i] = M$$

and some processes are waiting for more units indefinitely. But from the two preceding equations, we find

$$\sum_{i}^{N} deficit[i] < N$$

This means that at least one process j has acquired all its resources (deficit[j] = 0) and will be able to complete its task and release all its resources again, thus ensuring further progress in the system. So a deadlock cannot occur.

6.14 The number of available units required for the state to be safe is 3, making a total of 10 units in the system. In the state shown in the problem, if one additional unit is available, P2 can run to completion, releasing its resources, making 2 units available. This would allow P1 to run to completion making 3 units available. But at this point P3 needs 6 units and P4 needs 5 units. If to begin with, there had been 3 units available instead of 1 unit, there would now be 5 units available. This would allow P4 to run to completion, making 7 units available, which would allow P3 to run to completion.

6.15 **a.** In order from most-concurrent to least, there is a rough partial order on the deadlock-handling algorithms:

1. detect deadlock and kill thread, releasing its resources
detect deadlock and roll back thread's actions
restart thread and release all resources if thread needs to wait

None of these algorithms limit concurrency before deadlock occurs, because they rely on runtime checks rather than static restrictions. Their effects after deadlock is detected are harder to characterize: they still allow lots of concurrency (in some cases they enhance it), but the computation may no longer be sensible or efficient. The third algorithm is the strangest, since so much of its concurrency will be useless repetition; because threads compete for execution time, this algorithm also prevents useful computation from advancing. Hence it is listed twice in this ordering, at both extremes.

2. banker's algorithm
resource ordering

These algorithms cause more unnecessary waiting than the previous two by restricting the range of allowable computations. The banker's algorithm prevents unsafe allocations (a proper superset of deadlock-producing allocations) and resource ordering restricts allocation sequences so that threads have fewer options as to whether they must wait or not.

3. reserve all resources in advance

This algorithm allows less concurrency than the previous two, but is less pathological than the worst one. By reserving all resources in advance, threads have to wait longer and are more likely to block other threads while they work, so the system-wide execution is in effect more linear.

4. restart thread and release all resources if thread needs to wait

As noted above, this algorithm has the dubious distinction of allowing both the most and the least amount of concurrency, depending on the definition of concurrency.

b. In order from most-efficient to least, there is a rough partial order on the deadlock-handling algorithms:

1. reserve all resources in advance
resource ordering

These algorithms are most efficient because they involve no runtime overhead. Notice that this is a result of the same static restrictions that made these rank poorly in concurrency.

2. banker's algorithm
detect deadlock and kill thread, releasing its resources

These algorithms involve runtime checks on allocations which are roughly equivalent; the banker's algorithm performs a search to verify safety which is O(n m) in the number of threads and allocations, and deadlock detection performs a cycle-detection search which is O(n) in the length of resource-dependency chains. Resource-dependency chains are bounded by the number of threads, the number of resources, and the number of allocations.

3. detect deadlock and roll back thread's actions

This algorithm performs the same runtime check discussed previously but also entails a logging cost which is O(n) in the total number of memory writes performed.

4. restart thread and release all resources if thread needs to wait

This algorithm is grossly inefficient for two reasons. First, because threads run the risk of restarting, they have a low probability of completing. Second, they are competing with other restarting threads for finite execution time, so the entire system advances towards completion slowly if at all.

This ordering does not change when deadlock is more likely. The algorithms in the first group incur no additional runtime penalty because they statically disallow deadlock-producing execution. The second group incurs a minimal, bounded penalty when deadlock occurs. The algorithm in the third tier incurs the unrolling cost, which is O(n) in the number of memory writes performed between checkpoints. The status of the final algorithm is questionable because the algorithm does not allow deadlock to occur; it might be the case that unrolling becomes more expensive, but the behavior of this restart algorithm is so variable that accurate comparative analysis is nearly impossible.

6.16 The philosophers can starve while repeatedly picking up and putting down their left forks in perfect unison. Source: [BRIN73].

6.17 **a.** Assume that the table is in deadlock, i.e., there is a nonempty set D of philosophers such that each Pi in D holds one fork and waits for a fork held by neighbor. Without loss of generality, assume that Pj Œ D is a lefty. Since Pj clutches his left fork and cannot have his right fork, his right neighbor Pk never completes his dinner and is also a lefty. Therefore, Pk Œ D. Continuing the argument rightward around the table shows that all philosophers in D are lefties. This contradicts the existence of at least one righty. Therefore deadlock is not possible.

b. Assume that lefty Pj starves, i.e., there is a stable pattern of dining in which Pj never eats. Suppose Pj holds no fork. Then Pj's left neighbor Pi must continually hold his right fork and never finishes eating. Thus Pi is a righty holding his right fork, but never getting his left fork to complete a meal, i.e., Pi also starves. Now Pi's left neighbor must be a righty who continually holds his right fork. Proceeding leftward around the table with this argument shows that all philosophers are (starving) righties. But Pj is a lefty: a contradiction. Thus Pj must hold one fork.

As Pj continually holds one fork and waits for his right fork, Pj's right neighbor Pk never sets his left fork down and never completes a meal, i.e., Pk is also a lefty who starves. If Pk did not continually hold his left fork, Pj could eat; therefore Pk holds his left fork. Carrying the argument rightward around the table shows that all philosophers are (starving) lefties: a contradiction. Starvation is thus precluded.

Source: [GING90].

6.18 One solution (6.14) waits on available forks; the other solutions (6.17) waits for the neighboring philosophers to be free. The logic is essentially the same. The solution of Figure 6.17 is slightly more compact.

6.19 Atomic operations operate on atomic data types, which have their own internal format. Therefore, a simple read operation cannot be used, but a special read operation for the atomic data type is needed.

6.20 This code causes a deadlock, because the writer lock will spin, waiting for all readers to release the lock, including this thread.

6.21 Without using the memory barriers, on some processors it is possible that `c` receives the *new* value of `b`, while `d` receives the *old* value of `a`. For example, `c` could equal 4 (what we expect), yet `d` could equal 1 (not what we expect). Using the `mb()` insures `a` and `b` are written in the intended order, while the `rmb()` insures `c` and `d` are read in the intended order. This example is from [LOVE04].

Chapter 7
Memory Management

Answers to Questions

7.1 Relocation, protection, sharing, logical organization, physical organization.

7.2 Typically, it is not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution of his or her program. In addition, we would like to be able to swap active processes in and out of main memory to maximize processor utilization by providing a large pool of ready processes to execute. In both these cases, the specific location of the process in main memory is unpredictable.

7.3 Because the location of a program in main memory is unpredictable, it is impossible to check absolute addresses at compile time to assure protection. Furthermore, most programming languages allow the dynamic calculation of addresses at run time, for example by computing an array subscript or a pointer into a data structure. Hence all memory references generated by a process must be checked at run time to ensure that they refer only to the memory space allocated to that process.

7.4 If a number of processes are executing the same program, it is advantageous to allow each process to access the same copy of the program rather than have its own separate copy. Also, processes that are cooperating on some task may need to share access to the same data structure.

7.5 By using unequal-size fixed partitions: **1.** It is possible to provide one or two quite large partitions and still have a large number of partitions. The large partitions can allow the entire loading of large programs. **2.** Internal fragmentation is reduced because a small program can be put into a small partition.

7.6 Internal fragmentation refers to the wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition. External fragmentation is a phenomenon associated with dynamic partitioning, and refers to the fact that a large number of small areas of main memory external to any partition accumulates.

7.7 A **logical address** is a reference to a memory location independent of the current assignment of data to memory; a translation must be made to a physical address before the memory access can be achieved. A **relative address** is a particular example of logical address, in which the address is expressed as a location relative to some known point, usually the beginning of the program. A **physical address**, or absolute address, is an actual location in main memory.

7.8 In a paging system, programs and data stored on disk or divided into equal, fixed-sized blocks called pages, and main memory is divided into blocks of the same size called frames. Exactly one page can fit in one frame.

7.9 An alternative way in which the user program can be subdivided is segmentation. In this case, the program and its associated data are divided into a number of segments. It is not required that all segments of all programs be of the same length, although there is a maximum segment length.

Answers to Problems

7.1 Here is a rough equivalence:

Relocation	≈	support modular programming
Protection	≈	process isolation; protection and access control
Sharing	≈	protection and access control
Logical Organization	≈	support of modular programming
Physical Organization	≈	long-term storage; automatic allocation and management

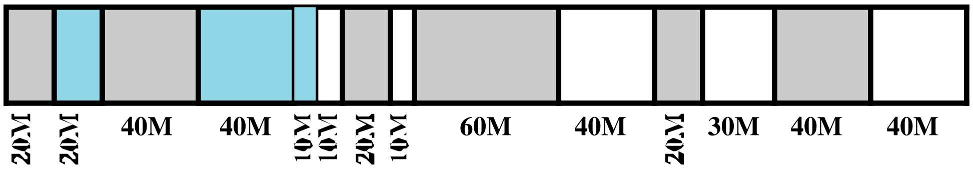
7.2 The number of partitions equals the number of bytes of main memory divided by the number of bytes in each partition: $2^{24}/2^{16} = 2^8$. Eight bits are needed to identify one of the 2^8 partitions.

7.3 Let *s* and *h* denote the average number of segments and holes, respectively. The probability that a given segment is followed by a hole in memory (and not by another segment) is 0.5, because deletions and creations are equally probable in equilibrium. so with *s* segments in memory, the average number of holes must be $s/2$. It is intuitively reasonable that the number of holes must be less than the number of segments because neighboring segments can be combined into a single hole on deletion.

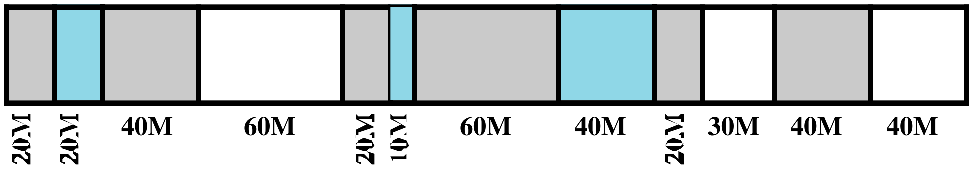
7.4 By problem 7.3, we know that the average number of holes is $s/2$, where *s* is the number of resident segments. Regardless of fit strategy, in equilibrium, the average search length is $s/4$.

7.5 A criticism of the best fit algorithm is that the space remaining after allocating a block of the required size is so small that in general it is of no real use. The worst fit algorithm maximizes the chance that the free space left after a placement will be large enough to satisfy another request, thus minimizing the frequency of compaction. The disadvantage of this approach is that the largest blocks are allocated first; therefore a request for a large area is more likely to fail.

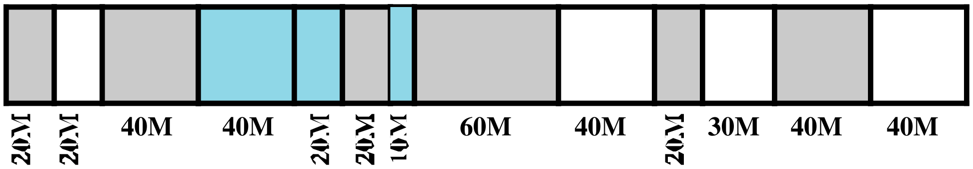
7.6 **a.** The 40 M block fits into the second hole, with a starting address of 80M. The 20M block fits into the first hole, with a starting address of 20M. The 10M block is placed at location 120M.



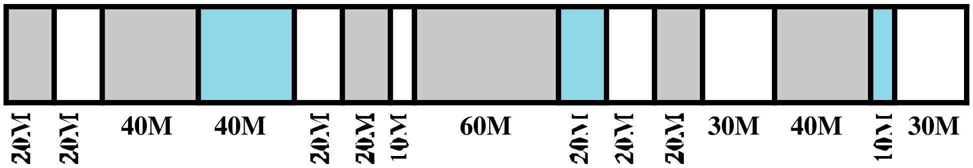
b. The three starting addresses are 230M, 20M, and 160M, for the 40M, 20M, and 10M blocks, respectively.



c. The three starting addresses are 80M, 120M, and 160M, for the 40M, 20M, and 10M blocks, respectively.



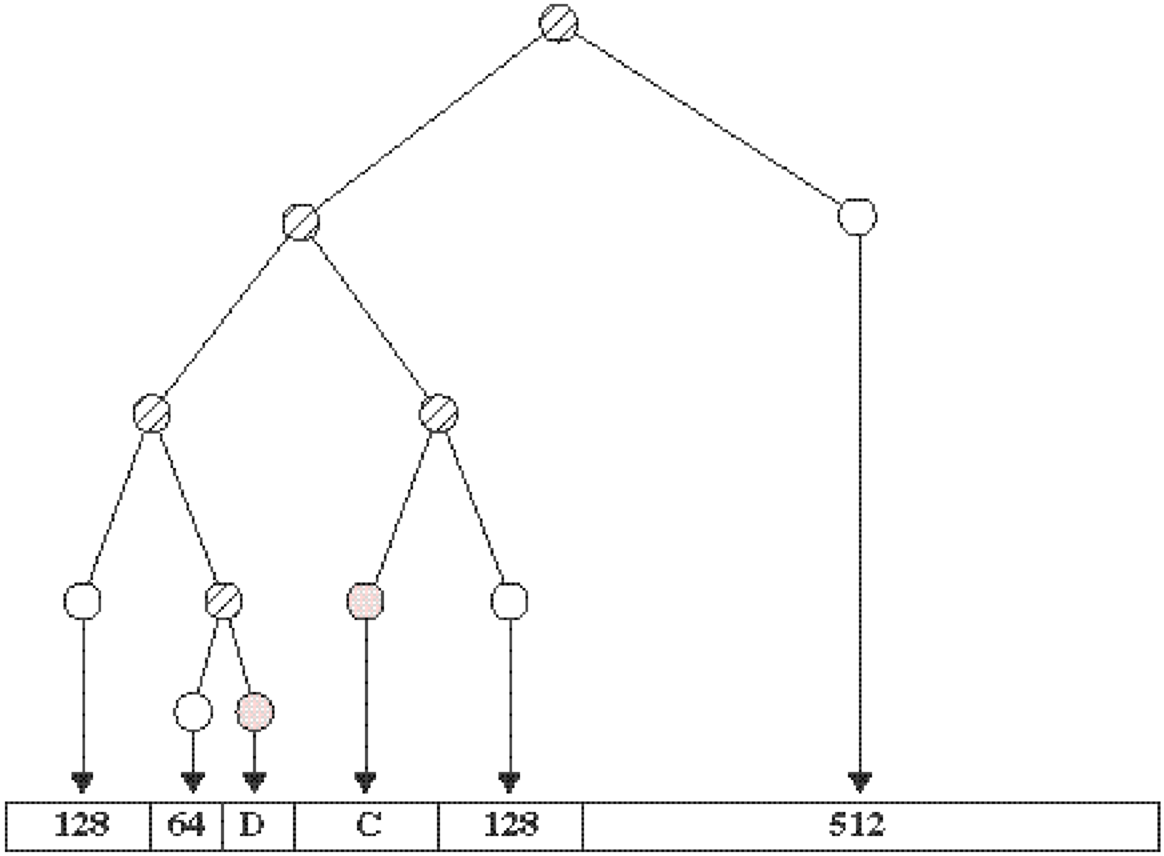
d. The three starting addresses are 80M, 230M, and 360M, for the 40M, 20M, and 10M blocks, respectively.



7.7 a.

<table>
<tr><td>Request 70</td><td>A</td><td colspan="2">128</td><td colspan="2">256</td><td>512</td></tr>
<tr><td>Request 35</td><td>A</td><td>B</td><td>64</td><td colspan="2">256</td><td>512</td></tr>
<tr><td>Request 80</td><td>A</td><td>B</td><td>64</td><td>C</td><td>128</td><td>512</td></tr>
<tr><td>Return A</td><td>128</td><td>B</td><td>64</td><td>C</td><td>128</td><td>512</td></tr>
<tr><td>Request 60</td><td>128</td><td>B</td><td>D</td><td>C</td><td>128</td><td>512</td></tr>
<tr><td>Return B</td><td>128</td><td>64</td><td>D</td><td>C</td><td>128</td><td>512</td></tr>
<tr><td>Return D</td><td colspan="3">256</td><td>C</td><td>128</td><td>512</td></tr>
<tr><td>Return C</td><td colspan="6">1024</td></tr>
</table>

b.



7.8 **a.** 011011110100
b. 011011100000

7.9 $\text{buddy}_k(x) = \begin{cases} x + 2^k & \text{if } x \bmod 2^{k+1} = 0 \\ x - 2^k & \text{if } x \bmod 2^{k+1} = 2^k \end{cases}$

7.10 **a.** Yes, the block sizes could satisfy $F_n = F_{n-1} + F_{n-2}$.
b. This scheme offers more block sizes than a binary buddy system, and so has the potential for less internal fragmentation, but can cause additional external fragmentation because many uselessly small blocks are created.

7.11 The use of absolute addresses reduces the number of times that dynamic address translation has to be done. However, we wish the program to be relocatable. Therefore, it might be preferable to use relative addresses in the instruction register. Alternatively, the address in the instruction register can be converted to relative when a process is swapped out of memory.

7.12 **a.** The number of bytes in the logical address space is (2^{16} pages) × (2^{10} bytes/page) = 2^{26} bytes. Therefore, 26 bits are required for the logical address.
b. A frame is the same size as a page, 2^{10} bytes.
c. The number of frames in main memory is (2^{32} bytes of main memory)/(2^{10} bytes/frame) = 2^{22} frames. So 22 bits is needed to specify the frame.
d. There is one entry for each page in the logical address space. Therefore there are 2^{16} entries.

e. In addition to the valid/invalid bit, 22 bits are needed to specify the frame location in main memory, for a total of 23 bits.

7.13 The relationship is $a = pz + w$, $0 \le w < z$, which can be stated as:
$p = \lfloor a/z \rfloor$, the integer part of a/z.
$w = R_z(a)$, the remainder obtained in dividing a by z.

7.14 **a.** Segment 0 starts at location 660. With the offset, we have a physical address of $660 + 198 = 858$

b. $222 + 156 = 378$

c. Segment 1 has a length of 422 bytes, so this address triggers a segment fault.

d. $996 + 444 = 1440$

e. $660 + 222 = 882$

7.15 **a.** Observe that a reference occurs to some segment in memory each time unit, and that one segment is deleted every t references. Because the system is in equilibrium, a new segment must be inserted every t references; therefore, the rate of the boundary's movement is s/t words per unit time. The system's operation time t_0 is then the time required for the boundary to cross the hole, i.e., $t_0 = fmr/s$, where m = size of memory. The compaction operation requires two memory references—a fetch and a store—plus overhead for each of the $(1 - f)m$ words to be moved, i.e., the compaction time t_c is at least $2(1 - f)m$. The fraction F of the time spent compacting is $F = 1 - t_0/(t_0 + t_c)$, which reduces to the expression given.

b. $k = (t/2s) - 1 = 9$; $F \ge (1 - 0.2)/(1 + 1.8) = 0.29$

CHAPTER 8
VIRTUAL MEMORY

ANSWERS TO QUESTIONS

8.1 **Simple paging:** all the pages of a process must be in main memory for process to run, unless overlays are used. **Virtual memory paging:** not all pages of a process need be in main memory frames for the process to run.; pages may be read in as needed

8.2 A phenomenon in virtual memory schemes, in which the processor spends most of its time swapping pieces rather than executing instructions.

8.3 Algorithms can be designed to exploit the principle of locality to avoid thrashing. In general, the principle of locality allows the algorithm to predict which resident pages are least likely to be referenced in the near future and are therefore good candidates for being swapped out.

8.4 **Frame number:** the sequential number that identifies a page in main memory; **present bit:** indicates whether this page is currently in main memory; **modify bit:** indicates whether this page has been modified since being brought into main memory.

8.5 The TLB is a cache that contains those page table entries that have been most recently used. Its purpose is to avoid, most of the time, having to go to disk to retrieve a page table entry.

8.6 With **demand paging**, a page is brought into main memory only when a reference is made to a location on that page. With **prepaging**, pages other than the one demanded by a page fault are brought in.

8.7 **Resident set management** deals with the following two issues: (1) how many page frames are to be allocated to each active process; and (2) whether the set of pages to be considered for replacement should be limited to those of the process that caused the page fault or encompass all the page frames in main memory. **Page replacement policy** deals with the following issue: among the set of pages considered, which particular page should be selected for replacement.

8.8 The clock policy is similar to FIFO, except that in the clock policy, any frame with a use bit of 1 is passed over by the algorithm.

8.9 (1) If a page is taken out of a resident set but is soon needed, it is still in main memory, saving a disk read. (2) Modified page can be written out in clusters rather than one at a time, significantly reducing the number of I/O operations and therefore the amount of disk access time.

8.10 Because a fixed allocation policy requires that the number of frames allocated to a process is fixed, when it comes time to bring in a new page for a process, one of the resident pages for that process must be swapped out (to maintain the number of frames allocated at the same amount), which is a local replacement policy.

8.11 The resident set of a process is the current number of pages of that process in main memory. The working set of a process is the number of pages of that process that have been referenced recently.

8.12 With **demand cleaning**, a page is written out to secondary memory only when it has been selected for replacement. A **precleaning** policy writes modified pages before their page frames are needed so that pages can be written out in batches.

Answers to Problems

8.1 **a.** Split binary address into virtual page number and offset; use VPN as index into page table; extract page frame number; concatenate offset to get physical memory address

b. **(i)** 1052 = 1024 + 28 maps to VPN 1 in PFN 7, (7 × 1024+28 = 7196)

(ii) 2221 = 2 × 1024 + 173 maps to VPN 2, page fault

(iii) 5499 = 5 × 1024 + 379 maps to VPN 5 in PFN 0, (0 × 1024+379 = 379)

8.2 **a.** Virtual memory can hold (2^{32} bytes of main memory)/(2^{10} bytes/page) = 2^{22} pages, so 22 bits are needed to specify a page in virtual memory. Each page table contains (2^{10} bytes per page table)/(4 bytes/entry) = 2^{8} entries. Thus, each page table can handle 8 of the required 22 bits. Therefore, 3 levels of page tables are needed.

b. Tables at two of the levels have 2^{8} entries; tables at one level have 2^{6} entries. (8 + 8 + 6 = 22).

c. Less space is consumed if the top level has 2^{6} entries. In that case, the second level has 2^{6} pages with 2^{8} entries each, and the bottom level has 2^{14} pages with 2^{8} entries each, for a total of 1 + 2^{6} + 2^{14} pages = 16,449 pages. If the middle level has 26 entries, then the number of pages is 1 + 2^{8} + 2^{14} pages = 16,641 pages. If the bottom level has 2^{6} entries, then the number of tables is 1 + 2^{8} + 2^{16} pages = 65,973 pages.

8.3 **a.** 4 Mbyte

b. Number of rows: 26+2=128 entries. Each entry consist of: 20 (page number) + 20 (frame number) + 8 bits (chain index) = 48 bits = 6 bytes.
Total: 128 × 6= 768 bytes

8.4 **a.** PFN 3 since loaded longest ago at time 20

b. PFN 1 since referenced longest ago at time 160

c. Clear R in PFN 3 (oldest loaded), clear R in PFN 2 (next oldest loaded), victim PFN is 0 since R=0

d. Replace the page in PFN 3 since VPN 3 (in PFN 3) is used furthest in the future

e. There are 6 faults, indicated by *

		*				*	*		*	*	*	
		4	0	0	0	2	4	2	1	0	3	2
VPN of	3	4	0	0	0	2	4	2	1	0	3	2
pages in	0	3	4	4	4	0	2	4	2	1	0	
memory in	2	0	3	3			0	0	4	2	1	
LRU order	1	2								4	2	

8.5 9 and 10 page transfers, respectively. This is referred to as "Belady's anomaly," and was reported in "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine," by Belady et al, *Communications of the ACM*, June 1969.

8.6 **a.** LRU: Hit ratio = 16/33

```
1 0 2 2 1 7 6 7 0 1 2 0 3 0 4 5 1 5 2 4 5 6 7 6 7 2 4 2 7 3 3 2 3

1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 2 2 2 2 2 2 2 2
- 0 0 0 0 0 6 6 6 6 2 2 2 2 2 5 5 5 5 5 5 5 5 5 5 5 4 4 4 4 4 4 4
- - 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 2 2 2 2 7 7 7 7 7 7 7 7 7 7 7
- - - - - 7 7 7 7 7 7 7 3 3 3 3 1 1 1 1 1 6 6 6 6 6 6 6 6 3 3 3 3
F F F     F F   F   F   F   F F F   F     F F     F F     F
```

b. FIFO: Hit ratio = 16/33

```
1 0 2 2 1 7 6 7 0 1 2 0 3 0 4 5 1 5 2 4 5 6 7 6 7 2 4 2 7 3 3 2 3

1 1 1 1 1 1 6 6 6 6 6 6 6 6 4 4 4 4 4 4 4 6 6 6 6 6 6 6 6 6 6 2 2
- 0 0 0 0 0 0 0 0 1 1 1 1 1 1 5 5 5 5 5 5 5 7 7 7 7 7 7 7 7 7 7 7
- - 2 2 2 2 2 2 2 2 2 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4
- - - - - 7 7 7 7 7 7 7 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3
F F F     F F     F   F F   F F F   F     F F       F     F   F
```

c. These two policies are equally effective for this particular page trace.

8.7 The principal advantage is a savings in physical memory space. This occurs for two reasons: (1) a user page table can be paged in to memory only when it is needed. (2) The operating system can allocate user page tables dynamically, creating one only when the process is created.

Of course, there is a disadvantage: address translation requires extra work.

8.8 The machine language version of this program, loaded in main memory starting at address 4000, might appear as:

4000	(R1) ← ONE	Establish index register for i
4001	(R1) ← n	Establish n in R2
4002	compare R1, R2	Test i > n
4003	branch greater 4009	
4004	(R3) ← B(R1)	Access B[i] using index register R1
4005	(R3) ← (R3) + C(R1)	Add C[i] using index register R1
4006	A(R1) ← (R3)	Store sum in A[i] using index register R1
4007	(R1) ← (R1) + ONE	Increment i

```
4008        branch 4002
6000-6999   storage for A
7000-7999   storage for B
8000-8999   storage for C
9000        storage for ONE
9001        storage for n
```

The reference string generated by this loop is

$$494944(47484649444)^{1000}$$

consisting of over 11,000 references, but involving only five distinct pages. Source: [MAEK87].

8.9 The S/370 segments are fixed in size and not visible to the programmer. Thus, none of the benefits listed for segmentation are realized on the S/370, with the exception of protection. The P bit in each segment table entry provides protection for the entire segment.

8.10 Since each page table entry is 4 bytes and each page contains 4 Kbytes, then a one-page page table would point to 1024 = 2^{10} pages, addressing a total of $2^{10} \times 2^{12} = 2^{22}$ bytes. The address space however is 2^{64} bytes. Adding a second layer of page tables, the top page table would point to 2^{10} page tables, addressing a total of 2^{32} bytes. Continuing this process,

Depth	Address Space
1	2^{22} bytes
2	2^{32} bytes
3	2^{42} bytes
4	2^{52} bytes
5	2^{62} bytes
6	2^{72} bytes (> 2^{64} bytes)

we can see that 5 levels do not address the full 64 bit address space, so a 6th level is required. But only 2 bits of the 6th level are required, not the entire 10 bits. So instead of requiring your virtual addresses be 72 bits long, you could mask out and ignore all but the 2 lowest order bits of the 6th level. This would give you a 64 bit address. Your top level page table then would have only 4 entries. Yet another option is to revise the criteria that the top level page table fit into a single physical page and instead make it fit into 4 pages. This would save a physical page, which is not much.

8.11 **a.** 400 nanoseconds. 200 to get the page table entry, and 200 to access the memory location.

b. This is a familiar effective time calculation:

$$(220 \times 0.85) + (420 \times 0.15) = 250$$

Two cases: First, when the TLB contains the entry required. In that case we pay the 20 ns overhead on top of the 200 ns memory access time. Second, when the TLB does not contain the item. Then we pay an additional 200 ns to get the required entry into the TLB.

c. The higher the TLB hit rate is, the smaller the EMAT is, because the additional 200 ns penalty to get the entry into the TLB contributes less to the EMAT.

8.12 a. N

b. P

8.13 a. This is a good analogy to the CLOCK algorithm. Snow falling on the track is analogous to page hits on the circular clock buffer. The movement of the CLOCK pointer is analogous to the movement of the plow.

b. Note that the density of replaceable pages is highest immediately in front of the clock pointer, just as the density of snow is highest immediately in front of the plow. Thus, we can expect the CLOCK algorithm to be quite efficient in finding pages to replace. In fact, it can be shown that the depth of the snow in front of the plow is twice the average depth on the track as a whole. By this analogy, the number of pages replaced by the CLOCK policy on a single circuit should be twice the number that are replaceable at a random time. The analogy is imperfect because the CLOCK pointer does not move at a constant rate, but the intuitive idea remains.

The snowplow analogy to the CLOCK algorithm comes from [CARR84]; the depth analysis comes from Knuth, D. *The Art of Computer Programming, Volume 2: Sorting and Searching*. Reading, MA: Addison-Wesley, 1997 (page 256).

8.14 The processor hardware sets the reference bit to 0 when a new page is loaded into the frame, and to 1 when a location within the frame is referenced. The operating system can maintain a number of queues of page-frame tables. A page-frame table entry moves from one queue to another according to how long the reference bit from that page frame stays set to zero. When pages must be replaced, the pages to be replaced are chosen from the queue of the longest-life nonreferenced frames.

8.15 a.

Seq of page refs	**Window Size, Δ**					
	1	**2**	**3**	**4**	**5**	**6**
1	1	1	1	1	1	1
2	2	1 2	1 2	1 2	1 2	1 2
3	3	2 3	1 2 3	1 2 3	1 2 3	1 2 3
4	4	3 4	2 3 4	1 2 3 4	1 2 3 4	1 2 3 4
5	5	4 5	3 4 5	2 3 4 5	1 2 3 4 5	1 2 3 4 5
2	2	5 2	4 5 2	3 4 5 2	3 4 5 2	1 3 4 5 2
1	1	2 1	5 2 1	4 5 2 1	3 4 5 2 1	3 4 5 2 1
3	3	1 3	2 1 3	5 2 1 3	4 5 2 1 3	4 5 2 1 3
3	3	3	1 3	2 1 3	5 2 1 3	4 5 2 1 3
2	2	3 2	3 2	1 3 2	1 3 2	5 1 3 2
3	3	2 3	2 3	2 3	1 2 3	1 2 3
4	4	3 4	2 3 4	2 3 4	2 3 4	1 2 3 4
5	5	4 5	3 4 5	2 3 4 5	2 3 4 5	2 3 4 5
4	4	5 4	5 4	3 5 4	2 3 5 4	2 3 5 4
5	5	4 5	4 5	4 5	3 4 5	2 3 4 5
1	1	5 1	4 5 1	4 5 1	4 5 1	3 4 5 1
1	1	1	5 1	4 5 1	4 5 1	4 5 1
3	3	1 3	1 3	5 1 3	4 5 1 3	4 5 1 3
2	2	3 2	1 3 2	1 3 2	5 1 3 2	4 5 1 3 2
5	5	2 5	3 2 5	1 3 2 5	1 3 2 5	1 3 2 5

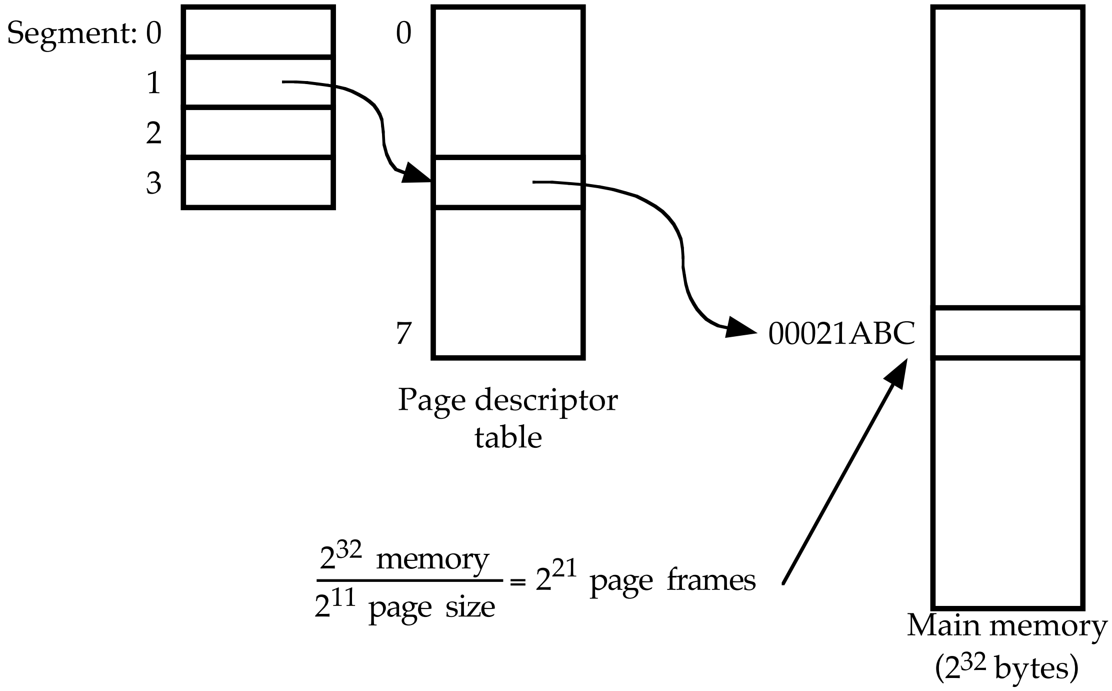
b., c.

Δ	**1**	**2**	**3**	**4**	**5**	**6**
$s_{20}(\Delta)$	1	1.85	2.5	3.1	3.55	3.9
$m_{20}(\Delta)$	0.9	0.75	0.75	0.65	0.55	0.5

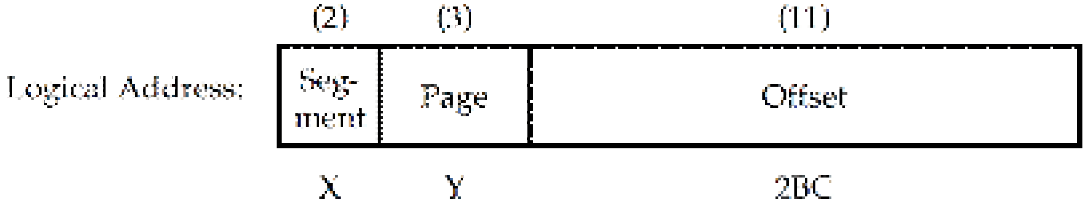
$s_{20}(\Delta)$ is an increasing function of Δ. $m_{20}(\Delta)$ is a nonincreasing function of Δ.

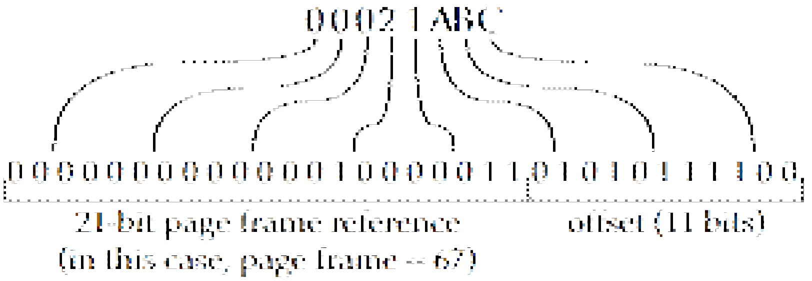
8.16 [PIZZ89] suggests the following strategy. Use a mechanism that adjusts the value of Q at each window time as a function of the actual page fault rate experienced during the window. The page fault rate is computed and compared with a system-wide value for "desirable" page fault rate for a job. The value of Q is adjusted upward (downward) whenever the actual page fault rate of a job is higher (lower) than the desirable value. Experimentation using this adjustment mechanism showed that execution of the test jobs with dynamic adjustment of Q consistently produced a lower number of page faults per execution and a decreased average resident set size than the execution with a constant value of Q (within a very broad range). The memory time product (MT) versus Q using the adjustment mechanism also produced a consistent and considerable improvement over the previous test results using a constant value of Q.

8.17 $\dfrac{2^{32} \text{ memory}}{2^{11} \text{ page size}} = 2^{21}$ page frames

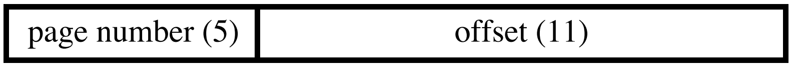

Segment: 0
1
2
3
0
7
Page descriptor table
00021ABC
$\dfrac{2^{32} \text{ memory}}{2^{11} \text{ page size}} = 2^{21}$ page frames
Main memory (2^{32} bytes)

a. 8 × 2K = 16K
b. 16K × 4 = 64K
c. 2^{32} = 4 GBytes





8.18 **a.**



b. 32 entries, each entry is 9 bits wide.

c. If total number of entries stays at 32 and the page size does not change, then each entry becomes 8 bits wide.

8.19 It is possible to shrink a process's stack by deallocating the unused pages. By convention, the contents of memory beyond the current top of the stack are undefined. On almost all architectures, the current top of stack pointer is kept in a well-defined register. Therefore, the kernel can read its contents and deallocate any unused pages as needed. The reason that this is not done is that little is gained by the effort. If the user program will repeatedly call subroutines that need additional space for local variables (a very likely case), then much time will be wasted deallocating stack space in between calls and then reallocating it later on. If the subroutine called is only used once during the life of the program and no other subroutine will ever be called that needs the stack space, then eventually the kernel will page out the unused portion of the space if it needs the memory for other purposes. In either case, the extra logic needed to recognize the case where a stack could be shrunk is unwarranted. Source: [SCHI94].

CHAPTER 9 UNIPROCESSOR SCHEDULING

ANSWERS TO QUESTIONS

9.1 **Long-term scheduling:** The decision to add to the pool of processes to be executed. **Medium-term scheduling:** The decision to add to the number of processes that are partially or fully in main memory. **Short-term scheduling:** The decision as to which available process will be executed by the processor

9.2 Response time.

9.3 **Turnaround time** is the total time that a request spends in the system (waiting time plus service time. **Response time** is the elapsed time between the submission of a request until the response begins to appear as output.

9.4 In UNIX and many other systems, larger priority values represent lower priority processes. Some systems, such as Windows, use the opposite convention: a higher number means a higher priority

9.5 **Nonpreemptive:** If a process is in the Running state, it continues to execute until (a) it terminates or (b) blocks itself to wait for I/O or to request some operating system service. **Preemptive:** The currently running process may be interrupted and moved to the Ready state by the operating system. The decision to preempt may be performed when a new process arrives, when an interrupt occurs that places a blocked process in the Ready state, or periodically based on a clock interrupt.

9.6 As each process becomes ready, it joins the ready queue. When the currently-running process ceases to execute, the process that has been in the ready queue the longest is selected for running.

9.7 A clock interrupt is generated at periodic intervals. When the interrupt occurs, the currently running process is placed in the ready queue, and the next ready job is selected on a FCFS basis.

9.8 This is a nonpreemptive policy in which the process with the shortest expected processing time is selected next.

9.9 This is a preemptive version of SPN. In this case, the scheduler always chooses the process that has the shortest expected remaining processing time. When a new process joins the ready queue, it may in fact have a shorter remaining time than the currently running process. Accordingly, the scheduler may preempt whenever a new process becomes ready.

9.10 When the current process completes or is blocked, choose the ready process with the greatest value of R, where R = (w + s)/s, with w = time spent waiting for the processor and s = expected service time.

9.11 Scheduling is done on a preemptive (at time quantum) basis, and a dynamic priority mechanism is used. When a process first enters the system, it is placed in RQ0 (see Figure 9.4). After its first execution, when it returns to the Ready state, it is placed in RQ1. Each subsequent time that it is preempted, it is demoted to the next lower-priority queue. A shorter process will complete quickly, without migrating very far down the hierarchy of ready queues. A longer process will gradually drift downward. Thus, newer, shorter processes are favored over older, longer processes. Within each queue, except the lowest-priority queue, a simple FCFS mechanism is used. Once in the lowest-priority queue, a process cannot go lower, but is returned to this queue repeatedly until it completes execution.

Answers to Problems

9.1 Each square represents one time unit; the number in the square refers to the currently-running process.

FCFS	A	A	A	B	B	B	B	B	C	C	D	D	D	D	D	E	E	E	E	E
RR, q = 1	A	B	A	B	C	A	B	C	B	D	B	D	E	D	E	D	E	D	E	E
RR, q = 4	A	A	A	B	B	B	B	C	C	B	D	D	D	D	E	E	E	E	D	E
SPN	A	A	A	C	C	B	B	B	B	B	D	D	D	D	D	E	E	E	E	E
SRT	A	A	A	C	C	B	B	B	B	B	D	D	D	D	D	E	E	E	E	E
HRRN	A	A	A	B	B	B	B	B	C	C	D	D	D	D	D	E	E	E	E	E
Feedback, q = 1	A	B	A	C	B	C	A	B	B	D	B	D	E	D	E	D	E	D	E	E
Feedback, q = 2^i	A	B	A	A	C	B	B	C	B	B	D	D	E	D	D	E	E	D	E	E

		A	B	C	D	E	
	T_a	0	1	3	9	12	
	T_s	3	5	2	5	5	
FCFS	T_f	3	8	10	15	20	
	T_r	3.00	7.00	7.00	6.00	8.00	6.20
	T_r/T_s	1.00	1.40	3.50	1.20	1.60	1.74
RR $q = 1$	T_f	6.00	11.00	8.00	18.00	20.00	
	T_r	6.00	10.00	5.00	9.00	8.00	7.60
	T_r/T_s	2.00	2.00	2.50	1.80	1.60	1.98
RR $q = 4$	T_f	3.00	10.00	9.00	19.00	20.00	
	T_r	3.00	9.00	6.00	10.00	8.00	7.20
	T_r/T_s	1.00	1.80	3.00	2.00	1.60	1.88
SPN	T_f	3.00	10.00	5.00	15.00	20.00	
	T_r	3.00	9.00	2.00	6.00	8.00	5.60
	T_r/T_s	1.00	1.80	1.00	1.20	1.60	1.32
SRT	T_f	3.00	10.00	5.00	15.00	20.00	
	T_r	3.00	9.00	2.00	6.00	8.00	5.60
	T_r/T_s	1.00	1.80	1.00	1.20	1.60	1.32
HRRN	T_f	3.00	8.00	10.00	15.00	20.00	
	T_r	3.00	7.00	7.00	6.00	8.00	6.20
	T_r/T_s	1.00	1.40	3.50	1.20	1.60	1.74
FB q = 1	T_f	7.00	11.00	6.00	18.00	20.00	
	T_r	7.00	10.00	3.00	9.00	8.00	7.40
	T_r/T_s	2.33	2.00	1.50	1.80	1.60	1.85
FB	T_f	4.00	10.00	8.00	18.00	20.00	
$q = 2^i$	T_r	4.00	9.00	5.00	9.00	8.00	7.00
	T_r/T_s	1.33	1.80	2.50	1.80	1.60	1.81

9.2

FCFS	A	B	B	B	B	B	B	B	B	B	C	D	D	D	D	D	D	D	D	D
RR, q = 1	A	B	C	B	D	B	D	B	D	B	D	B	D	B	D	B	D	B	D	D
RR, q = 4	A	B	B	B	B	C	B	B	B	B	D	D	D	D	B	D	D	D	D	D
SPN	A	B	B	B	B	B	B	B	B	B	C	D	D	D	D	D	D	D	D	D
SRT	A	B	C	B	B	B	B	B	B	B	B	D	D	D	D	D	D	D	D	D
HRRN	A	B	B	B	B	B	B	B	B	B	C	D	D	D	D	D	D	D	D	D
Feedback, q = 1	A	B	C	D	B	D	B	D	B	D	B	D	B	D	B	D	B	D	B	D
Feedback, $q = 2^i$	A	B	C	D	B	B	D	D	B	B	B	B	D	D	D	D	B	B	D	D

		A	B	C	D	
	T_a	0	1	2	3	
	T_s	1	9	1	9	
FCFS	T_f	1.00	10.00	11.00	20.00	
	T_r	1.00	9.00	9.00	17.00	9.00
	T_r/T_s	1.00	1.00	9.00	1.89	3.22
RR *q* = 1	T_f	1.00	18.00	3.00	20.00	
	T_r	1.00	17.00	1.00	17.00	9.00
	T_r/T_s	1.00	1.89	1.00	1.89	1.44
RR *q* = 4	T_f	1.00	15.00	6.00	20.00	
	T_r	1.00	14.00	4.00	17.00	9.00
	T_r/T_s	1.00	1.56	4.00	1.89	2.11
SPN	T_f	1.00	10.00	11.00	20.00	
	T_r	1.00	9.00	9.00	17.00	9.00
	T_r/T_s	1.00	1.00	9.00	1.89	3.22
SRT	T_f	1.00	11.00	3.00	20.00	
	T_r	1.00	10.00	1.00	17.00	7.25
	T_r/T_s	1.00	1.11	1.00	1.89	1.25
HRRN	T_f	1.00	10.00	11.00	20.00	
	T_r	1.00	9.00	9.00	17.00	9.00
	T_r/T_s	1.00	1.00	9.00	1.89	3.22
FB q = 1	T_f	1.00	19.00	3.00	20.00	
	T_r	1.00	18.00	1.00	17.00	9.25
	T_r/T_s	1.00	2.00	1.00	1.89	1.47
FB	T_f	1.00	18.00	3.00	20.00	
q = 2^i	T_r	1.00	17.00	1.00	17.00	9.00
	T_r/T_s	1.00	1.89	1.00	1.89	1.44

9.3 We will prove the assertion for the case in which a batch of *n* jobs arrive at the same time, and ignoring further arrivals. The proof can be extended to cover later arrivals.

Let the service times of the jobs be

$$t_1 \leq t_2 \leq . . . \leq t_n$$

Then, *n* users must wait for the execution of job 1; *n* – 1 users must wait for the execution of job 2, and so on. Therefore, the average response time is

$$\frac{n \times t_1 + (n-1) \times t_2 + \cdots + t_n}{n}$$

If we make any changes in this schedule, for example by exchanging jobs j and k (where j < k), the average response time is increased by the amount

$$\frac{(k-j)\times(t_k-t_j)}{n} \geq 0$$

In other words, the average response time can only increase if the SPN algorithm is not used. Source: [BRIN73].

9.4 The data points for the plot:

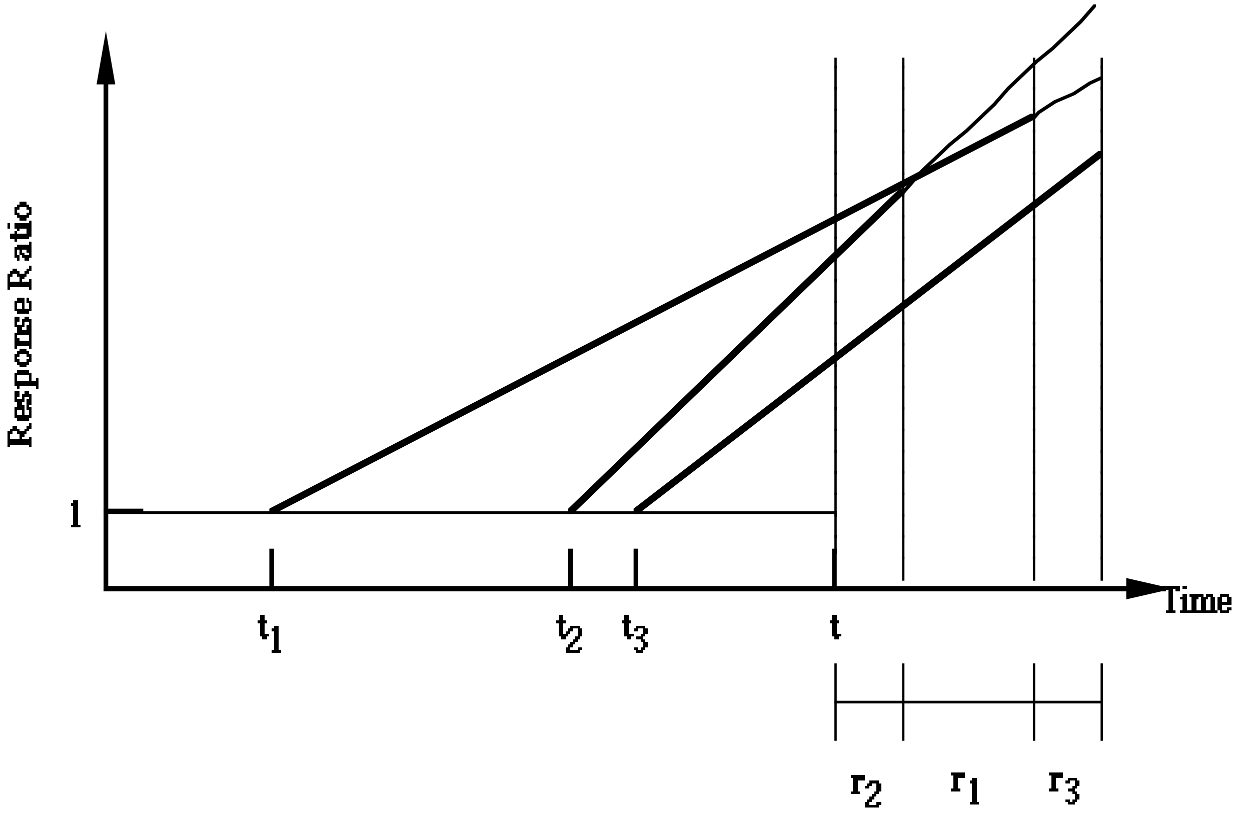
Age of Observation	Observed Value	Simple Average	alpha = 0.8	alpha = 0.5
1	6	0.00	0.00	0.00
2	4	3.00	4.80	3.00
3	6	3.33	4.16	3.50
4	4	4.00	5.63	4.75
5	13	4.00	4.33	4.38
6	13	5.50	11.27	8.69
7	13	6.57	12.65	10.84

9.5 The first equation is identical to Equation 9.3, so the parameter a provides an exponential smoothing effect. The parameter β is a delay variance factor (e.g., 1.3 to 2.0). A smaller value of β will result in faster adaptation to changes in the observed times, but also more fluctuation in the estimates.

A sophisticated analysis of this type of estimation procedure is contained in *Applied Optimal Estimation*, edited by Gelb, M.I.T. Press, 1974.

9.6 It depends on whether you put job A in a queue after the first time unit or not. If you do, then it is entitled to 2 additional time units before it can be preempted.

9.7 First, the scheduler computes the response ratios at time t + r_1 + r_2 + r_3, when all three jobs will have been finished (see figure). At that time, job 3 will have the smallest response ratio of the three: so the scheduler decides to execute this job last and proceeds to examine jobs 1 and 2 at time t + r_1 + r_2, when they will both be finished. Here the response ratio of job 1 is the smaller, and consequently job 2 is selected for service at time t. This algorithm is repeated each time a job is completed to take new arrivals into account. Note that this algorithm is not quite the same as *highest response ratio next*. The latter would schedule job 1 at time t. Intuitively, it is clear that the present algorithm attempts to minimize the maximum response ratio by consistently postponing jobs that will suffer the least increase of their response ratios.



9.8 This proof, due to P. Mondrup, is reported in [BRIN73]. Consider the queue at time t immediately after a departure and ignore further arrivals. The waiting jobs are numbered 1 to n in the order in which they will be scheduled:

job:	1	2	. . .	i	. . .	n
arrival time:	t_1	t_2	. . .	t_i	. . .	t_n
service time:	r_1	r_2	. . .	r_i	. . .	r_n

Among these we assume that job i will reach the highest response ratio before its departure. When the jobs 1 to i have been executed, time becomes

$$T_i = t + r_1 + r_2 + . . . + r_i$$

and job i has the response ratio

$$R_i(T_i) + \frac{T_i - t_i}{r_i}$$

The reason for executing job i last in the sequence 1 to i is that its response ratio will be the lowest one among these jobs at time T_i:

$$R_i(T_i) = \min [\ R_1(T_i), R_2(T_i), . . . , R_i(T_i)\ ]$$

Consider now the consequences of scheduling the same n jobs in any other sequence:

job:	a	b	. . .	j	. . .	z
arrival time:	t_a	t_b	. . .	t_j	. . .	t_z
service time:	r_a	r_b	. . .	r_j	. . .	r_z

In the new sequence, we select the smallest subsequence of jobs, a to j, that contains all the jobs, 1 to i, of the original subsequence (This implies that job j is itself one of the jobs 1 to i). When the jobs a to j have been served, time becomes

$$T_j = t + r_a + r_b + . . . + r_j$$

and job j reaches the response ratio

$$R_j\left(T_j\right) + \frac{T_j - t_j}{r_j}$$

Since the jobs 1 to i are a subset of the jobs a to j, the sum of their service times $T_i - t$ must be less than or equal to the sum of service time $T_j - t$. And since response ratios increase with time, $T_i \le T_j$ implies

$$R_j(T_j) \ge R_j(T_i)$$

It is also known that job j is one of the jobs 1 to i, of which job j has the smallest response ratio at time T_i. The above inequality can therefore be extended as follows:

$$R_j(T_j) \ge R_j(T_i) \ge R_i(T_i)$$

In other words, when the scheduling algorithm is changed, there will always be a job j that reaches response ratio $R_j(T_j)$, which is greater than or equal to the highest response ratio $R_i(T_i)$ obtained with the original algorithm.

Notice that this proof is valid in general for priorities that are non-decreasing functions of time. For example, in a FIFO system, priorities increase linearly with waiting time at the same rate for all jobs. Therefore, the present proof shows that the FIFO algorithm minimizes the maximum waiting time for a given batch of jobs.

9.9 Before we begin, there is one result that is needed, as follows. Assume that an item with service time T_s has been in service for a time h. Then, the expected remaining service time $E\,[T/T > h] = T_s$. That is, no matter how long an item has been in service, the expected remaining service time is just the average service time for the item. This result, though counter to intuition, is correct, as we now show.

Consider the exponential probability distribution function:

$$F(x) = Pr[X \le x] = 1 - e^{-\mu x}$$

Then, we have $Pr[X > x] = e^{-\mu x}$. Now let us look at the conditional probability that X is greater than x + h given that X is greater than x:

$$\Pr\left[X > x + h \middle| X > x\right] = \frac{\Pr\left[(X > x + h),(X > x)\right]}{\Pr\left[X > x\right]} = \frac{\Pr\left[X > x + h\right]}{\Pr\left[X > x\right]}$$

$$\Pr\left[X > x + h \middle| X > x\right] = \frac{e^{-\mu(x+h)}}{e^{-\mu x}} = e^{-\mu h}$$

So,

$$\Pr[X \le x + h / X > x] = 1 - e^{-\mu h} = \Pr[X \le h]$$

Thus the probability distribution for service time given that there has been service of duration x is the same as the probability distribution of total service time. Therefore the expected value of the remaining service time is the same as the original expected value of service time.

With this result, we can now proceed to the original problem. When an item arrives for service, the total response time for that item will consist of its own service time plus the service time of all items ahead of it in the queue. The total expected response time has three components.

- Expected service time of arriving process = T_s
- Expected service time of all processes currently waiting to be served. This value is simply w × T_s, where w is the mean number of items waiting to be served.
- Remaining service time for the item currently in service, if there is an item currently in service. This value can be expressed as ρ × T_s, where ρ is the utilization and therefore the probability that an item is currently in service and T_s, as we have demonstrated, is the expected remaining service time.

Thus, we have

$$R = T_s \times (1 + w + \rho) = T_s \times \left(1 + \frac{\rho^2}{1 - \rho} + \rho\right) = \frac{T_s}{1 - \rho}$$

9.10 Let us denote the time slice, or quantum, used in round robin scheduling as δ. In this problem, δ is assumed to be very small compared to the service time of a process. Now, consider a newly arrived process, which is placed at the end of the ready queue for service. We are assuming that this particular process has a service time of x, which is some multiple of δ:

$$x = m\delta$$

To begin, let us ask the question, how much time does the process spend in the queue before it receives its first quantum of service. It must wait until all q processes waiting in line ahead of it have been serviced. Thus the initial wait time = qδ, where q is the average number of items in the system (waiting and being served). We can now calculate the total time this process will spend waiting before it has received x seconds of service. Since it must pass through the active queue m times, and each time it waits qδ seconds, the total wait time is as follows:

$$\begin{aligned}\text{Wait time} &= m\,(q\delta)\\ &= \frac{x}{\delta}(q\delta)\\ &= qx\end{aligned}$$

Then, the response time is the sum of the wait time and the total service time

$$\begin{aligned}R_x &= \text{wait time + service time}\\ &= qx + x = (q + 1)\,x\end{aligned}$$

Referring to the queuing formulas in Appendix A, the mean number of items in the system, q, can be expressed as

$$q = \frac{\rho}{1-\rho}$$

Thus,

$$R_x = \left(\frac{\rho}{1-\rho} + 1\right) \times x = \frac{x}{1-\rho}\, R_x = \left(\frac{\rho}{1-\rho} + 1\right) \times x = \frac{x}{1-\rho}$$

9.11 **An argument in favor of a small quantum:** Using a small quantum will enhance responsiveness by frequently running all processes for a short time slice. When the ready queue has many processes that are interactive, responsiveness is very important e.g. general-purpose computer system.
An argument in favor of a large quantum: Using a large quantum will enhance the throughput, and the CPU utilization measured with respect to real work, because there is less context switching and therefore less overhead. e.g. batch jobs.
A system for which both might be appropriate: There are some systems for which both small and large quanta are reasonable. e.g. a long job that requires just a few user interactions. Although this type of job can be considered as a batch job, in some sense, it still has to interact with the user. Therefore, during the times when there is no user interaction, the quantum might be increased to optimize the throughout and during interactive time, the quantum might be lowered to provide better responsiveness.

9.12 First, we need to clarify the significance of the parameter λ'. The rate at which items arrive at the first box (the "queue" box) is λ. Two adjacent arrivals to the second box (the "service" box) will arrive at a slightly slower rate, since the second item is delayed in its chase of the first item. We may calculate the vertical offset y in the figure in two different ways, based on the geometry of the diagram:

$$y = \frac{\beta}{\lambda'}\, y = \frac{\beta}{\lambda'}$$

$$y = \left(\frac{1}{\lambda'} - \frac{1}{\lambda}\right)\alpha$$

which therefore gives

$$\lambda' = \lambda\left(1 - \frac{\beta}{\alpha}\right)$$

The total number of jobs q waiting or in service when the given job arrives is given by:

$$q = \frac{\rho}{1-\rho}$$

independent of the scheduling algorithm. Using Little's formula (see Appendix A):

$$R = \frac{q}{\lambda} = \frac{s}{1-\rho}$$

Now let W and V_x denote the mean times spent in the queue box and in the service box by a job of service time x. Since priorities are initially based only on elapsed waiting times, W is clearly independent of the service time x. Evidently we have

$$R_x = W + V_x$$

From problem 9.10, we have

$$V_x = \frac{t}{1-\rho'} \quad \text{where } \rho' = \lambda's$$

By taking the expected values of R_x and S_x, we have R = W + V. We have already developed the formula for R. For V, observe that the arrival rate to the service box is l', and therefore the utilization is r'. Accordingly, from our basic M/M/1 formulas, we have

$$V = \frac{s}{1-\rho'}$$

Thus,

$$W = R - V = s\left(\frac{1}{1-\rho} - \frac{1}{1-\rho'}\right)$$

which yields the desired result for R_x.

9.13 Only as long as there are comparatively few users in the system. When the quantum is decreased to satisfy more users rapidly two things happen: (1) processor utilization decreases, and (2) at a certain point, the quantum becomes too small to satisfy most trivial requests. Users will then experience a sudden increase of response times because their requests must pass through the round-robin queue several times. Source: [BRIN73].

9.14 If a process uses too much processor time, it will be moved to a lower-priority queue. This leaves I/O-bound processes in the higher-priority queues.

9.15 Dekker's algorithm relies on the fairness of the hardware and the OS. The use of priorities risks starvation as follows. It may happen if P0 is a very fast repetitive process which, as it constantly finds flag [1] = false, keeps entering its critical section, while P1, leaving the internal loop in which it was waiting for its turn, cannot set flag [1] to true, being prevented from doing so by P0's reading of the variable (remember that access to the variable takes place under mutual exclusion).

9.16 **a.** Sequence with which processes will get 1 min of processor time:

1	2	3	4	5	Elapsed time
A	B	C	D	E	5
A	B	C	D	E	10
A	B	C	D	E	15
A	B		D	E	19
A	B		D	E	23
A	B		D	E	27
A	B			E	30
A	B			E	33
A	B			E	36
A				E	38
A				E	40
A				E	42
A					43
A					44
A					45

The turnaround time for each process:
A = 45 min, B = 35 min, C = 13 min, D = 26 min, E = 42 min
The average turnaround time is = (45+35+13+26+42) / 5 = 32.2 min

b.

Priority	Job	Turnaround Time
3	B	9
4	E	9 + 12 = 21
6	A	21 + 15 = 36
7	C	36 + 3 = 39
9	D	39 + 6 = 45

The average turnaround time is: (9+21+36+39+45) / 5 = 30 min

c.

Job	Turnaround Time
A	15
B	15 + 9 = 24
C	24 + 3 = 27
D	27 + 6 = 33
E	33 + 12 = 45

The average turnaround time is: (15+24+27+33+45) / 5 = 28.8 min

d.

Running Time	Job	Turnaround Time
3	C	3
6	D	3 + 6 = 9
9	B	9 + 9 = 18
12	E	18 + 12 = 30
15	A	30 + 15 = 45

The average turnaround time is: (3+9+18+30+45) / 5 = 21 min

CHAPTER 10
MULTIPROCESSOR AND REAL-TIME SCHEDULING

ANSWERS TO QUESTIONS

10.1 **Fine:** Parallelism inherent in a single instruction stream. **Medium:** Parallel processing or multitasking within a single application. **Coarse:** Multiprocessing of concurrent processes in a multiprogramming environment. **Very Coarse:** Distributed processing across network nodes to form a single computing environment. **Independent:** Multiple unrelated processes.

10.2 **Load sharing:** Processes are not assigned to a particular processor. A global queue of ready threads is maintained, and each processor, when idle, selects a thread from the queue. The term load sharing is used to distinguish this strategy from load-balancing schemes in which work is allocated on a more permanent basis. **Gang scheduling:** A set of related threads is scheduled to run on a set of processors at the same time, on a one-to-one basis. **Dedicated processor assignment:** Each program is allocated a number of processors equal to the number of threads in the program, for the duration of the program execution. When the program terminates, the processors return to the general pool for possible allocation to another program. **Dynamic scheduling:** The number of threads in a program can be altered during the course of execution.

10.3 **First Come First Served (FCFS):** When a job arrives, each of its threads is placed consecutively at the end of the shared queue. When a processor becomes idle, it picks the next ready thread, which it executes until completion or blocking. **Smallest Number of Threads First:** The shared ready queue is organized as a priority queue, with highest priority given to threads from jobs with the smallest number of unscheduled threads. Jobs of equal priority are ordered according to which job arrives first. As with FCFS, a scheduled thread is run to completion or blocking. **Preemptive Smallest Number of Threads First:** Highest priority is given to jobs with the smallest number of unscheduled threads. An arriving job with a smaller number of threads than an executing job will preempt threads belonging to the scheduled job.

10.4 A **hard real-time task** is one that must meet its deadline; otherwise it will cause undesirable damage or a fatal error to the system. A **soft real-time task** has an associated deadline that is desirable but not mandatory; it still makes sense to schedule and complete the task even if it has passed its deadline.

10.5 An **aperiodic task** has a deadline by which it must finish or start, or it may have a constraint on both start and finish time. In the case of a **periodic task**, the requirement may be stated as "once per period *T*" or "exactly *T* units apart."

10.6 **Determinism:** An operating system is deterministic to the extent that it performs operations at fixed, predetermined times or within predetermined time intervals. **Responsiveness:** Responsiveness is concerned with how long, after acknowledgment, it takes an operating system to service the interrupt. **User control:** The user should be able to distinguish between hard and soft tasks and to specify relative priorities within each class. A real-time system may also allow the user to specify such characteristics as the use of paging or process swapping, what processes must always be resident in main memory, what disk transfer algorithms are to be used, what rights the processes in various priority bands have, and so on. **Reliability:** Reliability must be provided in such a way as to continue to meet real-time deadlines. **Fail-soft operation:** Fail-soft operation is a characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible..

10.7 **Static table-driven approaches:** These perform a static analysis of feasible schedules of dispatching. The result of the analysis is a schedule that determines, at run time, when a task must begin execution. **Static priority-driven preemptive approaches:** Again, a static analysis is performed, but no schedule is drawn up. Rather, the analysis is used to assign priorities to tasks, so that a traditional priority-driven preemptive scheduler can be used. **Dynamic planning-based approaches:** Feasibility is determined at run time (dynamically) rather than offline prior to the start of execution (statically). An arriving task is accepted for execution only if it is feasible to meet its time constraints. One of the results of the feasibility analysis is a schedule or plan that is used to decide when to dispatch this task. **Dynamic best effort approaches:** No feasibility analysis is performed. The system tries to meet all deadlines and aborts any started process whose deadline is missed.

10.8 **Ready time:** time at which task becomes ready for execution. In the case of a repetitive or periodic task, this is actually a sequence of times that is known in advance. In the case of an aperiodic task, this time may be known in advance, or the operating system may only be aware when the task is actually ready. **Starting deadline:** Time by which a task must begin. **Completion deadline:** Time by which task must be completed. The typical real-time application will either have starting deadlines or completion deadlines, but not both. **Processing time:** Time required to execute the task to completion. In some cases, this is supplied. In others, the operating system measures an exponential average. For still other scheduling systems, this information is not used. **Resource requirements:** Set of resources (other than the processor) required by the task while it is executing. **Priority:** Measures relative importance of the task. Hard real-time tasks may have an "absolute" priority, with the system failing if a deadline is missed. If the system is to continue to run no matter what, then both hard and soft real-time tasks may be assigned relative priorities as a guide to the scheduler. **Subtask structure:** A task may be decomposed into a mandatory subtask and an optional subtask. Only the mandatory subtask possesses a hard deadline.

Answers to Problems

10.1 For fixed priority, we do the case in which the priority is A, B, C. Each square represents five time units; the letter in the square refers to the currently-running process. The first row is fixed priority; the second row is earliest deadline scheduling using completion deadlines.

A	A	B	B	A	A	C	C	A	A	B	B	A	A	C	C	A	A		
A	A	B	B	A	C	C	A	C	A	A	B	B	A	A	C	C	C	A	A

For fixed priority scheduling, process C always misses its deadline.

10.2 Each square represents 10 time units.

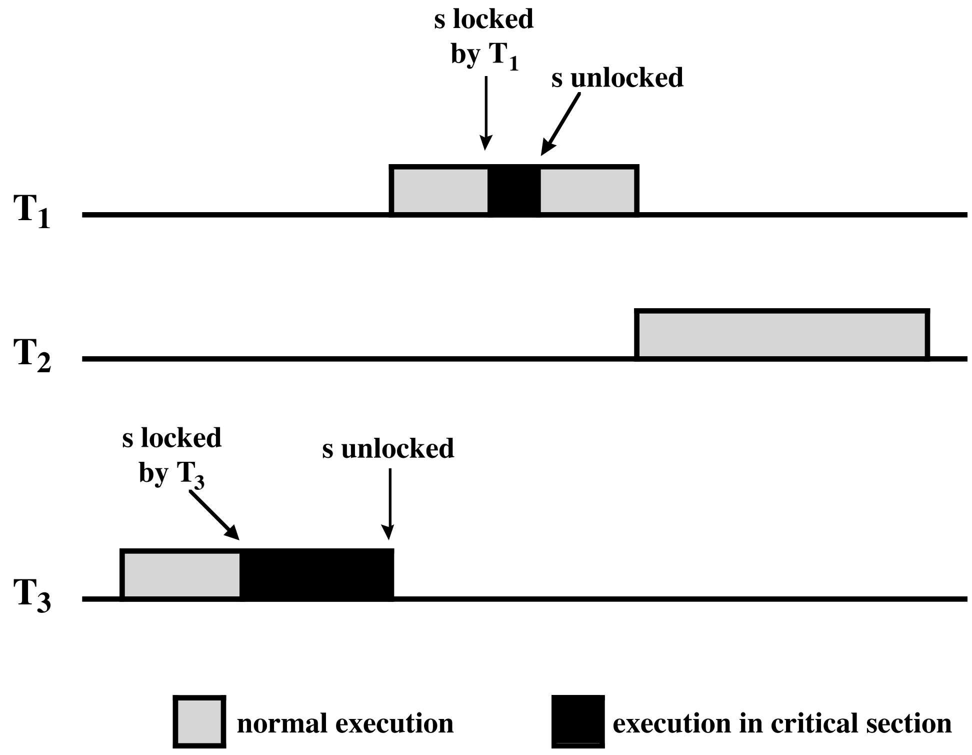
Earliest deadline		A	A		C	C	E	E	D	D		
Earliest deadline with unforced idle times			B	B	C	C	E	E	D	D	A	A
FCFS		A	A		C	C	D	D				

10.3 **a.** The total utilization of P_1 and P_2 is 0.41, which is less than 0.828, the bound given for two tasks by Equation 10.2. Therefore, these two tasks are schedulable.

b. The utilization of all the tasks is 0.86, which exceeds the bound of 0.779.

c. Observe that P_1 and P_2 must execute at least once before P_3 can begin executing. Therefore, the completion time of the first instance of P_3 can be no less than 20 + 30 + 68 = 118. However, P1 is initiated one additional time in the interval (0, 118). Therefore, P_3 does not complete its first execution until 118 + 20 = 138. This is within the deadline for P_3. By continuing this reasoning, we can see that all deadlines of all three tasks can be met.

10.4



Once T_3 enters its critical section, it is assigned a priority higher than T1. When T3 leaves its critical section, it is preempted by T_1.

Chapter 11
I/O Management and Disk Scheduling

Answers to Questions

11.1 **Programmed I/O:** The processor issues an I/O command, on behalf of a process, to an I/O module; that process then busy-waits for the operation to be completed before proceeding. **Interrupt-driven I/O:** The processor issues an I/O command on behalf of a process, continues to execute subsequent instructions, and is interrupted by the I/O module when the latter has completed its work. The subsequent instructions may be in the same process, if it is not necessary for that process to wait for the completion of the I/O. Otherwise, the process is suspended pending the interrupt and other work is performed. **Direct memory access (DMA):** A DMA module controls the exchange of data between main memory and an I/O module. The processor sends a request for the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

11.2 **Logical I/O:** The logical I/O module deals with the device as a logical resource and is not concerned with the details of actually controlling the device. The logical I/O module is concerned with managing general I/O functions on behalf of user processes, allowing them to deal with the device in terms of a device identifier and simple commands such as open, close, read, write. **Device I/O:** The requested operations and data (buffered characters, records, etc.) are converted into appropriate sequences of I/O instructions, channel commands, and controller orders. Buffering techniques may be used to improve utilization.

11.3 **Block-oriented** devices stores information in blocks that are usually of fixed size, and transfers are made one block at a time. Generally, it is possible to reference data by its block number. Disks and tapes are examples of block-oriented devices. **Stream-oriented** devices transfer data in and out as a stream of bytes, with no block structure. Terminals, printers, communications ports, mouse and other pointing devices, and most other devices that are not secondary storage are stream oriented.

11.4 Double buffering allows two operations to proceed in parallel rather than in sequence. Specifically, a process can transfer data to (or from) one buffer while the operating system empties (or fills) the other.

11.5 Seek time, rotational delay, access time.

11.6 **FIFO:** Items are processed from the queue in sequential first-come-first-served order.. **SSTF:** Select the disk I/O request that requires the least movement of the disk arm from its current position. **SCAN:** The disk arm moves in one direction only, satisfying all outstanding requests en route, until it reaches the last track in that direction or until there are no more requests in that direction. The service direction is then reversed and the scan proceeds in the opposite direction, again picking up all requests in order. **C-SCAN:** Similar to SCAN, but restricts scanning to one direction only. Thus, when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again.

11.7 **0:** Non-redundant **1:** Mirrored; every disk has a mirror disk containing the same data. **2:** Redundant via Hamming code; an error-correcting code is calculated across corresponding bits on each data disk, and the bits of the code are stored in the corresponding bit positions on multiple parity disks. **3:** Bit-interleaved parity; similar to level 2 but instead of an error-correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks. **4:** Block-interleaved parity; a bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk. **5:** Block-interleaved distributed parity; similar to level 4 but distributes the parity strips across all disks. **6:** Block-interleaved dual distributed parity; two different parity calculations are carried out and stored in separate blocks on different disks.

11.8 512 bytes.

Answers to Problems

11.1 If the calculation time exactly equals the I/O time (which is the most favorable situation), both the processor and the peripheral device running simultaneously will take half as long as if they ran separately. Formally, let C be the calculation time for the entire program and let T be the total I/O time required. Then the best possible running time with buffering is max(C, T), while the running time without buffering is C + T; and of course ((C + T)/2) ≤ max(C, T) ≤ (C + T). Source: [KNUT97].

11.2 The best ratio is (n + 1):n. Source: [KNUT97].

11.3 Disk head is initially moving in the direction of decreasing track number:

FIFO		SSTF		SCAN		C-SCAN	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
27	73	110	10	64	36	64	36
129	102	120	10	41	23	41	23
110	19	129	9	27	14	27	14
186	76	147	18	10	17	10	17
147	39	186	39	110	100	186	176
41	106	64	122	120	10	147	39
10	31	41	23	129	9	129	18
64	54	27	14	147	18	120	9
120	56	10	17	186	39	110	10
Average	61.8	Average	29.1	Average	29.6	Average	38

If the disk head is initially moving in the direction of increasing track number, only the SCAN and C-SCAN results change:

SCAN		C-SCAN	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
110	10	110	10
120	10	120	10
129	9	129	9
147	18	147	18
186	39	186	39
64	122	10	176
41	23	27	17
27	14	41	14
10	17	64	23
Average	29.1	Average	35.1

11.4 It will be useful to keep the following representation of the N tracks of a disk in mind:

0	1	• • •	j – 1	• • •	N – j	• • •	N – 2	N – 1

a. Let us use the notation Ps [j/t] = Pr [seek of length j when head is currently positioned over track t]. Recognize that each of the N tracks is equally likely to be requested. Therefore the unconditional probability of selecting any particular track is 1/N. We can then state:

$$\text{Ps}[j\ /\ t] = \frac{1}{N} \quad \text{if} \qquad t \leq j - 1 \ \text{ OR } \ t \geq N - j$$

$$\text{Ps}[j\ /\ t] = \frac{2}{N} \quad \text{if} \qquad j - 1 < t < N - j$$

In the former case, the current track is so close to one end of the disk (track 0 or track N – 1) that only one track is exactly j tracks away. In the second case, there are two tracks that are exactly j tracks away from track t, and therefore the probability of a seek of length j is the probability that either of these two tracks is selected, which is just 2/N.

b. Let Ps [K] = Pr [seek of length K, independent of current track position]. Then:

$$\begin{aligned}\text{Ps}[K] &= \sum_{t=0}^{N-1} \text{Ps}[K/t] \times \text{Pr}[\text{current track is track t}] \\ &= \frac{1}{N}\sum_{t=0}^{N-1} \text{Ps}[K/t]\end{aligned}$$

From part (a), we know that Ps[K/t] takes on the value 1/N for 2K of the tracks, and the value 2/N for (N – 2K) of the tracks. So

$$\mathrm{Ps}[K] = \frac{1}{N}\left(\frac{2K}{N} + \frac{2(N-K)}{N}\right) = \frac{2}{N^2}(N-K)$$

c.

$$\mathrm{E}[K] = \sum_{K=0}^{N-1} K \times \mathrm{Ps}[K] = \sum_{K=0}^{N-1} \frac{2K(N-K)}{N^2}$$

$$= \frac{2}{N}\sum_{K=0}^{N-1} K - \frac{2}{N^2}\sum_{K=0}^{N-1} K^2$$

$$= (N+1) - \frac{2(N+1)(2N+1)}{6N}$$

$$= \frac{N^2-1}{3N}$$

d. This follows directly from the last equation.

11.5 Define

A_i = Time to retrieve a word from memory level i

H_i = Probability that a word is in memory level i and no higher-level memory

B_i = Time to transfer a block of data from memory level (i + 1) to memory level i

Let cache be memory level 1; main memory, memory level 2, and so on, for a total of N levels of memory. Then, we can write

$$T_s = \sum_{i=1}^{N} A_i H_i$$

Now, recognize that if a word is in M_1 (cache), it is read immediately. If it is in M_2 but not in M_1, then a block of data is transferred from M_2 to M_1 and then read. Thus

$$A_2 = B_1 + A_1$$

Further

$$A_3 = B_2 + A_2 = B_1 + B_2 + A_1$$

Generalizing,

$$A_i = A_1 + \sum_{j=1}^{i-1} B_j$$

So

$$T_s = T_1 \sum_{i=1}^{N} H_i + \sum_{L=2}^{N} \sum_{J=1}^{L-1} B_j H_i$$

But $\displaystyle\sum_{i=1}^{N} H_i = 1$

Finally

$$T_s = T_1 + \sum_{L=2}^{N} \sum_{J=1}^{L-1} B_j H_i$$

11.6 **a.** The middle section in Figure 11.11b is empty. Thus, this reduces to the strategy of Figure 11.11a.

b. The old section consists of one block, and we have the LRU replacement policy.

11.7 The density can be expressed as 1600 characters per inch. Therefore the transfer rate is 1600 × 120 = 192,000 characters per second.

11.8 We provide the answers for 30 blocks per record.

b. First, let us determine how long it takes to read one physical block plus the gap to the next block.

Size of physical block = (30 logical records per physical record) × (120 bytes per logical record) = 3600 bytes

Length of physical block = $\dfrac{3600 \text{ bytes}}{1600 \text{ bytes / inch}}$ = 2.35 in

Length of gap = 0.6 in

Transfer time for one block plus gap = $\dfrac{2.25}{120} + \dfrac{0.6}{60}$ = 0.02875 seconds

The number of blocks on the tape can be computed as

$$\frac{2400 \text{ ft} \times 12 \text{ in / ft}}{2.25 + 0.6 \text{ in / record}} = 10{,}105 \text{ physical blocks}$$

Thus, the time to read a full tape is 10,105 × 0.02875 = 291 seconds

c. With a blocking factor of 30, the tape holds 10,105 physical records and 30 × 10,105 = 303,150 logical records.

d. The effective transfer rate, with a blocking factor of 30:

$$\text{R} = \frac{303{,}150 \text{records} \times 120 \text{ bytes / record}}{291 \text{seconds}} = 125{,}010 \text{ bytes/second}$$

e. Capacity = 303,150 records × 120 bytes/record = 36,378,000 bytes

11.9 Each sector can hold 4 records. The required number of sectors is 303,150/4 = 75,788 sectors. This requires 75,788/96 = 790 tracks, which in turn requires 790/110 = 8 surfaces.

11.10 There are 512 bytes/sector. Since each byte generates an interrupt, there are 512 interrupts. Total interrupt processing time = 2.5 × 512 = 1280 μs. The time to read one sector is:

((60 sec/min) / (360 rev/min)) / (96 sectors/track)
= 0.001736 sec = 1736 μs

Percentage of time processor spends handling I/O:

(100) × (1280/1736) = 74%

11.11 With DMA, there is only one interrupt of 2.5 μs. Therefore, the percentage of time the processor spends handling I/O is

(100) × (2.5/1736) = 0.14%

11.12 Only one device at a time can be serviced on a selector channel. Thus,

Maximum rate = 800 + 800 + 2 × 6.6 + 2 × 1.2 + 10 × 1 = 1625.6 KBytes/sec

11.13 It depends on the nature of the I/O request pattern. On one extreme, if only a single process is doing I/O and is only doing one large I/O at a time, then disk striping improves performance. If there are many processes making many small I/O requests, then a nonstriped array of disks should give comparable performance to RAID 0.

CHAPTER 12
FILE MANAGEMENT

ANSWERS TO QUESTIONS

12.1 A **field** is the basic element of data containing a single value. A **record** is a collection of related fields that can be treated as a unit by some application program.

12.2 A **file** is a collection of similar records, and is treated as a single entity by users and applications and may be referenced by name. A **database** is a collection of related data. The essential aspects of a database are that the relationships that exist among elements of data are explicit and that the database is designed for use by a number of different applications.

12.3 A file management system is that set of system software that provides services to users and applications in the use of files.

12.4 Rapid access, ease of update, economy of storage, simple maintenance, reliability.

12.5 **Pile:** Data are collected in the order in which they arrive. Each record consists of one burst of data. **Sequential file:** A fixed format is used for records. All records are of the same length, consisting of the same number of fixed-length fields in a particular order. Because the length and position of each field is known, only the values of fields need to be stored; the field name and length for each field are attributes of the file structure. **Indexed sequential file:** The indexed sequential file maintains the key characteristic of the sequential file: records are organized in sequence based on a key field. Two features are added; an index to the file to support random access, and an overflow file. The index provides a lookup capability to reach quickly the vicinity of a desired record. The overflow file is similar to the log file used with a sequential file, but is integrated so that records in the overflow file are located by following a pointer from their predecessor record. **Indexed file:** Records are accessed only through their indexes. The result is that there is now no restriction on the placement of records as long as a pointer in at least one index refers to that record. Furthermore, variable-length records can be employed. **Direct, or hashed, file:** The direct file makes use of hashing on the key value.

12.6 In a sequential file, a search may involve sequentially testing every record until the one with the matching key is found. The indexed sequential file provides a structure that allows a less exhaustive search to be performed.

12.7 Search, create file, delete file, list directory, update directory.

12.8 The pathname is an explicit enumeration of the path through the tree-structured directory to a particular point in the directory. The working directory is a directory within that tree structure that is the current directory that a user is working on.

12.9 None, knowledge of, read, write, execute, change protection, delete.

12.10 **Fixed blocking:** Fixed-length records are used, and an integral number of records are stored in a block. There may be unused space at the end of each block. This is referred to as internal fragmentation. **Variable-length spanned blocking:** Variable-length records are used and are packed into blocks with no unused space. Thus, some records must span two blocks, with the continuation indicated by a pointer to the successor block. **Variable-length unspanned blocking:** Variable-length records are used, but spanning is not employed. There is wasted space in most blocks because of the inability to use the remainder of a block if the next record is larger than the remaining unused space.

12.11 **Contiguous allocation:** a single contiguous set of blocks is allocated to a file at the time of file creation. **Chained allocation:** allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. **Indexed allocation:** the file allocation table contains a separate one-level index for each file; the index has one entry for each portion allocated to the file.

Answers to Problems

12.1 **Fixed blocking:** $F = \text{largest integer} \le \dfrac{B}{R}$

When records of variable length are packed into blocks, data for marking the record boundaries within the block has to be added to separate the records. When spanned records bridge block boundaries, some reference to the successor block is also needed. One possibility is a length indicator preceding each record. Another possibility is a special separator marker between records. In any case, we can assume that each record requires a marker, and we assume that the size of a marker is about equal to the size of a block pointer [WEID87]. For spanned blocking, a block pointer of size P to its successor block may be included in each block, so that the pieces of a spanned record can easily be retrieved. Then we have

Variable-length spanned blocking: $F = \dfrac{B - P}{R + P}$

With unspanned variable-length blocking, an average of R/2 will be wasted because of the fitting problem, but no successor pointer is required:

Variable-length unspanned blocking: $F = \dfrac{B - \frac{R}{2}}{R + P}$

12.2 **a.** $\log_2 \dfrac{N}{F}$

b. Less than half the allocated file space is unused at any time.

12.3 **a.** Indexed

b. Indexed sequential

c. Hashed or indexed

12.4 Clearly, security is more easily enforced if directories are easily recognized as special files by the operating system. Treating a directory as an ordinary file with certain assigned access restrictions provides a more uniform set of objects to be managed by the operating system and may make it easier to create and manage user-owned directories.

12.5 This is a rare feature. If the operating system structures the file system so that subdirectories are allowed underneath a master directory, there is little or no additional logic required to allow arbitrary depth of subdirectories. Limiting the depth of the subdirectory tree places an unnecessary limitation on the user's organization of file space.

12.6 **a.** Yes. the method employed is very similar to that used by many LISP systems for garbage collection. First we would establish a data structure representing every block on a disk supporting a file system. A bit map would be appropriate here. Then, we would start at the root of the file system (the "`/`" directory), and mark every block used by every file we could find through a recursive descent through the file system. When finished, we would create a free list from the blocks remaining as unused. This is essentially what the UNIX utility `fsck` does.

b. Keep a "backup" of the free-space list pointer at one or more places on the disk. Whenever this beginning of the list changes, the "backup" pointers are also updated. This will ensure you can always find a valid pointer value even if there is a memory or disk block failure.

12.7 **a.** Find the number of disk block pointers that fit in one block by dividing the block size by the pointer size:

8K/4 = 2K pointers per block

The maximum file size supported by the I-Node is thus:

12	+ 2K	+ (2K × 2K)	+ (2K × 2K × 2K)
Direct	Indirect – 1	Indirect – 2	Indirect – 3
12	+ 2K	+ 4M	+ 8G blocks

Which, when multiplied by the block size (8K), is

96KB + 16MB + 32GB + 64TB

Which is HUGE.

b. There are 24 bits for identifying blocks within a partition, so that leads to:

2^{24} × 8K = 16M × 8K = 128 GB

c. Using the information from (a), we see that the direct blocks only cover the first 96KB, while the first indirect block covers the next 16MB. the requested file position is 13M and change, which clearly falls within the range of the first indirect block. There will thus be two disk accesses. One for the first indirect block, and one for the block containing the required data.

CHAPTER 13
NETWORKING

ANSWERS TO QUESTIONS

13.1 The network access layer is concerned with the exchange of data between a computer and the network to which it is attached.

13.2 The transport layer is concerned with data reliability and correct sequencing.

13.3 A protocol is the set of rules or conventions governing the way in which two entities cooperate to exchange data.

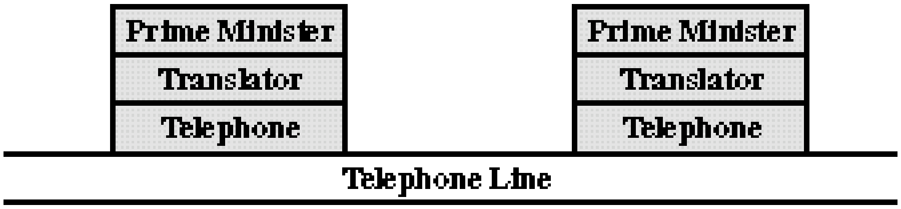
13.4 The software structure that implements the communications function. Typically, the protocol architecture consists of a layered set of protocols, with one or more protocols at each layer.

13.5 Transmission Control Protocol/Internet Protocol (TCP/IP) are two protocols originally designed to provide low level support for internetworking. The term is also used generically to refer to a more comprehensive collection of protocols developed by the U.S. Department of Defense and the Internet community.

13.6 A sockets interface is an API that enables programs to be writing that make use of the TCP/IP protocol suite to establish communication between a client and server.

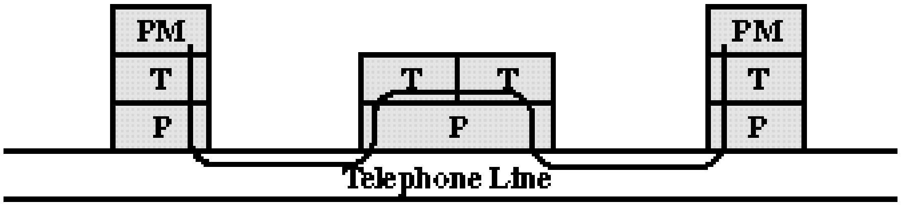
ANSWERS TO PROBLEMS

13.1 **a.**



The PMs speak as if they are speaking directly to each other. For example, when the French PM speaks, he addresses his remarks directly to the Chinese PM. However, the message is actually passed through two translators via the phone system. The French PM's translator translates his remarks into English and telephones these to the Chinese PM's translator, who translates these remarks into Chinese.

b.



An intermediate node serves to translate the message before passing it on.

13.2 Perhaps the major disadvantage is the processing and data overhead. There is processing overhead because as many as seven modules (OSI model) are invoked to move data from the application through the communications software. There is data overhead because of the appending of multiple headers to the data. Another possible disadvantage is that there must be at least one protocol standard per layer. With so many layers, it takes a long time to develop and promulgate the standards.

13.3 Data plus transport header plus internet header equals 1820 bits. This data is delivered in a sequence of packets, each of which contains 24 bits of network header and up to 776 bits of higher-layer headers and/or data. Three network packets are needed. Total bits delivered = 1820 + 3 × 24 = 1892 bits.

13.4 UDP has a fixed-sized header. The header in TCP is of variable length.

13.5 Suppose that A sends a data packet k to B and the ACK from B is delayed but not lost. A resends packet k, which B acknowledges. Eventually A receives 2 ACKs to packet k, each of which triggers transmission of packet (k + 1). B will ACK both copies of packet (k + 1), causing A to send two copies of packet (k + 2). From now on, 2 copies of every data packet and ACK will be sent.

13.6 TFTP can transfer a maximum of 512 bytes per round trip (data sent, ACK received). The maximum throughput is therefore 512 bytes divided by the round-trip time.

CHAPTER 14
DISTRIBUTED PROCESSING, CLIENT/SERVER, AND CLUSTERS

ANSWERS TO QUESTIONS

14.1 A networked environment that includes client machines that make requests of server machines.

14.2 **1.** There is a heavy reliance on bringing user-friendly applications to the user on his or her own system. This gives the user a great deal of control over the timing and style of computer usage and gives department-level managers the ability to be responsive to their local needs. **2.** Although applications are dispersed, there is an emphasis on centralizing corporate databases and many network management and utility functions. This enables corporate management to maintain overall control of the total capital investment in computing and information systems, and to provide interoperability so that systems are tied together. At the same time it relieves individual departments and divisions of much of the overhead of maintaining sophisticated computer-based facilities, but enables them to choose just about any type of machine and interface they need to access data and information. **3.** There is a commitment, both by user organizations and vendors, to open and modular systems. This means that the user has greater choice in selecting products and in mixing equipment from a number of vendors. **4.** Networking is fundamental to the operation. Thus, network management and network security have a high priority in organizing and operating information systems.

14.3 It is the communications software that enables client and server to interoperate.

14.4 **Server-based processing:** The rationale behind such configurations is that the user workstation is best suited to providing a user-friendly interface and that databases and applications can easily be maintained on central systems. Although the user gains the advantage of a better interface, this type of configuration does not generally lend itself to any significant gains in productivity or to any fundamental changes in the actual business functions that the system supports. **Client-based processing:** This architecture is enables the user to employ applications tailored to local needs. **Cooperative processing:** This type of configuration may offer greater user productivity gains and greater network efficiency than other client/server approaches.

14.5 **Fat client:** Client-based processing, with most of the software at the client. The main benefit of the fat client model is that it takes advantage of desktop power, offloading application processing from servers and making them more efficient and less likely to be bottlenecks. **Thin client:** Server-based processing, with most of the software at the server. This approach more nearly mimics the traditional host-centered approach and is often the migration path for evolving corporate wide applications from the mainframe to a distributed environment.

14.6 **Fat client:** The main benefit is that it takes advantage of desktop power, offloading application processing from servers and making them more efficient and less likely to be bottlenecks. The addition of more functions rapidly overloads the capacity of desktop machines, forcing companies to upgrade. If the model extends beyond the department to incorporate many users, the company must install high-capacity LANs to support the large volumes of transmission between the thin servers and the fat clients. Finally, it is difficult to maintain, upgrade, or replace applications distributed across tens or hundreds of desktops. **Thin client:** This approach more nearly mimics the traditional host-centered approach and is often the migration path for evolving corporate wide applications from the mainframe to a distributed environment. It does not provide the flexibility of the fat client approach.

14.7 The middle tier machines are essentially gateways between the thin user clients and a variety of backend database servers. The middle tier machines can convert protocols and map from one type of database query to another. In addition, the middle tier machine can merge/integrate results from different data sources. Finally, the middle tier machine can serve as a gateway between the desktop applications and the backend legacy applications by mediating between the two worlds.

14.8 Middleware is a set of standard programming interfaces and protocols that sit between the application above and communications software and operating system below. It provides a uniform means and style of access to system resources across all platforms

14.9 TCP/IP does not provide the APIs and the intermediate-level protocols to support a variety of applications across different hardware and OS platforms.

14.10 **Nonblocking primitives** provide for efficient, flexible use of the message-passing facility by processes. The disadvantage of this approach is that it is difficult to test and debug programs that use these primitives. Irreproducible, timing-dependent sequences can create subtle and difficult problems. **Blocking primitives** have the opposite advantages and disadvantages.

14.11 **Nonpersistent binding**: Because a connection requires the maintenance of state information on both ends, it consumes resources. The nonpersistent style is used to conserve those resources. On the other hand, the overhead involved in establishing connections makes nonpersistent binding inappropriate for remote procedures that are called frequently by the same caller. P**ersistent binding**: For applications that make many repeated calls to remote procedures, persistent binding maintains the logical connection and allows a sequence of calls and returns to use the same connection.

14.12 The **synchronous RPC** is easy to understand and program because its behavior is predictable. However, it fails to exploit fully the parallelism inherent in distributed applications. This limits the kind of interaction the distributed application can have, resulting in lower performance. To provide greater flexibility, **asynchronous RPC** facilities achieve a greater degree of parallelism while retaining the familiarity and simplicity of the RPC. Asynchronous RPCs do not block the caller; the replies can be received as and when they are needed, thus allowing client execution to proceed locally in parallel with the server invocation.

14.13 **Passive Standby:** A secondary server takes over in case of primary server failure. **Separate Servers:** Separate servers have their own disks. Data is continuously copied from primary to secondary server. **Servers Connected to Disks:** Servers are cabled to the same disks, but each server owns its disks. If one server fails, its disks are taken over by the other server. **Servers Share Disks:** Multiple servers simultaneously share access to disks.

Answers to Problems

14.1 **a.** MIPS rate = $[n\alpha + (1 - \alpha)]\, x = (n\alpha - \alpha + 1)x$

b. $\alpha = 0.6$

Source: [HWAN93]

14.2 **a.** One computer executes for a time T. Eight computers execute for a time T/4, which would take a time 2T on a single computer. Thus the total required time on a single computer is 3T. Effective speedup = 3. $\alpha = 0.75$.

b. New speedup = 3.43

c. α must be improved to 0.8.

14.3 **a.** Sequential execution time = 1,051,628 cycles

b. Speedup = 16.28

c. Each computer is assigned 32 iterations balanced between the beginning and end of the I-loop.

d. The ideal speedup of 32 is achieved.

CHAPTER 15
DISTRIBUTED PROCESS MANAGEMENT

ANSWERS TO QUESTIONS

15.1 **Load sharing:** By moving processes from heavily loaded to lightly loaded systems, the load can be balanced to improve overall performance. **Communications performance:** Processes that interact intensively can be moved to the same node to reduce communications cost for the duration of their interaction. Also, when a process is performing data analysis on some file or set of files larger than the process's size, it may be advantageous to move the process to the data rather than vice versa. **Availability:** Long-running processes may need to move to survive in the face of faults for which advance notice can be achieved or in advance of scheduled downtime. If the operating system provides such notification, a process that wants to continue can either migrate to another system or ensure that it can be restarted on the current system at some later time. **Utilizing special capabilities:** A process can move to take advantage of unique hardware or software capabilities on a particular node.

15.2 The following alternative strategies may be used. **Eager (all):** Transfer the entire address space at the time of migration. **Precopy:** The process continues to execute on the source node while the address space is copied to the target node. Pages modified on the source during the precopy operation have to be copied a second time. **Eager (dirty):** Transfer only those pages of the address space that are in main memory and have been modified. Any additional blocks of the virtual address space will be transferred on demand only. **Copy-on-reference:** This is a variation of eager (dirty) in which pages are only brought over when referenced. **Flushing:** The pages of the process are cleared from the main memory of the source by flushing dirty pages to disk. Then pages are accessed as needed from disk instead of from memory on the source node.

15.3 Nonpreemptive process migration can be useful in load balancing. It has the advantage that it avoids the overhead of full-blown process migration. The disadvantage is that such a scheme does not react well to sudden changes in load distribution.

15.4 Because of the delay in communication among systems, it is impossible to maintain a system wide clock that is instantly available to all systems. Furthermore, it is also technically impractical to maintain one central clock and to keep all local clocks synchronized precisely to that central clock; over a period of time, there will be some drift among the various local clocks that will cause a loss of synchronization.

15.5 In a fully **centralized algorithm**, one node is designated as the control node and controls access to all shared objects. When any process requires access to a critical resource, it issues a Request to its local resource-controlling process. This process, in turn, sends a Request message to the control node, which returns a Reply (permission) message when the shared object becomes available. When a process has finished with a resource, a Release message is sent to the control node. In a distributed algorithm, the mutual exclusion algorithm involves the concurrent cooperation of distributed peer entities.

15.6 Deadlock in resource allocation, deadlock in message communication.

Answers to Problems

15.1 **a.** Eager (dirty)
b. Copy on reference

15.2 Process P_1 begins with a clock value of 0. To transmit message *a*, it increments its clock by 1 and transmits (*a*, 1, 1), where the first numerical value is the timestamp and the second is the identity of the site. Similarly, P_4 increments its clock by 1 and transmits issues (*q*, 1, 4). Both messages are received by the other three sites. Both *a* and *q* have the same timestamp, but P_1's numerical identifier is less than P_4's numerical identifier (1 < 4). Therefore, the ordering is {*a*, *q*} at all four sites.

15.3 P_i can save itself the transmission of a Reply message to P_j if P_i has sent a Request message but has not yet received the corresponding Release message.

15.4 **a.** If a site *i*, which has asked to enter its critical section, has received a response from all the others, then (1) its request is the oldest (in the sense defined by the timestamp ordering) of all the requests that may be waiting; and (2) all critical sections requested earlier have been completed. If a site *j* has itself sent an earlier request, or if it was in its critical section, it would not have sent a response to *i*.

b. As incoming requests are totally ordered, they are served in that order; every request will at some stage become the oldest, and will then be served.

15.5 The algorithm makes no allowance for resetting the time stamping clocks with respect to each other. For a given process, P_i for example, *clock* is only used to update, on the one hand, *request* [*i*] variables in the other processes by way of request messages, and, on the other hand, *token* [*i*] variables, when messages of the token type are transferred. So the clocks are not used to impose a total ordering on requests. They are used simply as counters that record the number of times the various processes have asked to use the critical section, and so to find whether or not the number of times that P*i* has been given this access, recorded as the value of *token* [*i*], is less than the number of requests it has made, known to P_j by the value of $request_j$ [*i*]. The function *max* used in the processing associated with the reception of requests results in only the last request from P_j being considered if several had been delivered out of sequence.

15.6 **a.** Mutual exclusion is guaranteed if at any one time the number of variables *token_present* that have the value true cannot exceed 1. Since this is the initial condition, it suffices to show that the condition is conserved throughout the procedure. Consider first the prelude. The variable for P_i, which we write $token\_present_i$, changes its value from false to true when P_i receives the token. If we now consider the postlude for process P_j that has issued the token, we see that P_j has been able to do so only if $token\_present_j$ had the value true and P_j had changed this to false before sending the token.

b. Suppose that all the processes wish to enter the critical section but none of them has the token, so they are all halted, awaiting its arrival. The token is therefore in transit. It will after some finite time arrive at one of the processes, and so unblock it.

c. Fairness follows from the fact that all messages are delivered within a finite time of issue. The postlude requires that P_i transfers the token to the first process P_j, found in scanning the set in the order j = i+1, i+2, . . ., n, 1, . . ., i-1, whose request has reached Pi; if the transmission delays for all messages are finite (i.e., no message is lost), all the processes will learn of the wish for some P_j to enter the critical section and will agree to this when its turn comes.

15.7 The receipt of a request from P_j has the effect of updating the local variable, `request (j)`, which records the time of P_j's last request. The max operator assures that the correct order is maintained.

CHAPTER 16
SECURITY

ANSWERS TO QUESTIONS

16.1 Confidentiality, integrity, availability, authenticity.

16.2 **Passive attacks** are in the nature of eavesdropping on, or monitoring of, transmissions. The goal of the opponent is to obtain information that is being transmitted. **Active attacks** involve some modification of data or a data stream or the creation of false data or data stream.

16.3 **Passive attacks.** Release of message contents: an unauthorized person or program reads the contents of a file or message. Traffic analysis: gaining information by analyzing the pattern of data traffic.
Active attacks. Masquerade: one entity pretends to be a different entity. Replay: the passive capture of a data unit and its subsequent retransmission to produce an unauthorized effect. Message modification: some portion of a legitimate message is altered, or messages are delayed or reordered, to produce an unauthorized effect. Denial of service:

16.4 The most common technique for user access control on a shared system or server is the user logon, which requires both a user identifier (ID) and a password.

16.5 **Subject:** An entity capable of accessing objects. Generally, the concept of subject equates with that of process. Any user or application actually gains access to an object by means of a process that represents that user or application. **Object:** Anything to which access is controlled. Examples include files, portions of files, programs, and segments of memory.

16.6 The salt serves three purposes: (1) It prevents duplicate passwords from being visible in the password file. Even if two users choose the same password, those passwords will be assigned at different times. Hence, the "extended" passwords of the two users will differ. (2) It effectively increases the length of the password without requiring the user to remember two additional characters. Hence, the number of possible passwords is increased by a factor of 4096, increasing the difficulty of guessing a password. (3) It prevents the use of a hardware implementation of DES, which would ease the difficulty of a brute-force guessing attack.

16.7 **Statistical anomaly detection:** Involves the collection of data relating to the behavior of legitimate users over a period of time. Then statistical tests are applied to observed behavior to determine with a high level of confidence whether that behavior is not legitimate user behavior. **Rule-based detection:** Involves an attempt to define a set of rules that can be used to decide that a given behavior is that of an intruder.

16.8 Either term would be reasonable. The term *worm* usually connotes an independent program as opposed to a piece of a program, so virus might be a preferable term.

16.9 Encryption can be used in the following fashion. A portion of the virus, generally called a *mutation engine*, creates a random encryption key to encrypt the remainder of the virus. The key is stored with the virus, and the mutation engine itself is altered. When an infected program is invoked, the virus uses the stored random key to decrypt the virus. When the virus replicates, a different random key is selected.

16.10 **Cryptanalytic attacks** rely on the nature of the algorithm plus perhaps some knowledge of the general characteristics of the plaintext or even some sample plaintext-ciphertext pairs. This type of attack exploits the characteristics of the algorithm to attempt to deduce a specific plaintext or to deduce the key being used. In a **brute-force attack**, every possible key is tried on a piece of ciphertext until an intelligible translation into plaintext is obtained.

16.11 DES is a widely-used conventional encryption standard, standardized by NIST. The original DES specified a data encryption algorithm (DEA). The most recent version of the standard also includes the option of using triple DEA, which repeats DEA three times with two or three independent keys.

16.12 AES is expected to execute more rapidly in software than TDEA. Also, AES uses a larger block size, which should enhance security.

16.13 Evaluation criteria include security, computational efficiency, memory requirements, hardware and software suitability, and flexibility.

16.14 In conventional encryption, the same key is used for encryption and decryption. In public-key encryption, there is a pair of keys, one used for encryption and one for decryption. Only one of the two keys need be kept private.

16.15 The key used in conventional encryption is typically referred to as a **secret key**. The two keys used for public-key encryption are referred to as the **public key** and the **private key**. Invariably, the private key is kept secret, but it is referred to as a private key rather than a secret key to avoid confusion with conventional encryption.

Answers to Problems

16.1 **a.** $T = \frac{26^4}{2}$ seconds $= 63.5$ hours

b. Expect 13 tries for each digit. $T = 13 \times 4 = 52$ seconds.

16.2 **a.** $p = r^k$

b. $p = \frac{r^k - r^p}{r^{k+p}}$

c. $p = r^p$

16.3 **a.** $T = (21 \times 5 \times 21)^2 = 4{,}862{,}025$

b. $p = \frac{1}{T} \approx 2 \times 10^{-7}$

16.4 **a.** There are $95^{10} \approx 6 \times 10^{19}$ possible passwords. The time required is:

$$\frac{6 \times 10^{19} \text{passwords}}{6.4 \times 10^{6} \text{passwords / second}} = 9.4 \times 10^{12} \text{seconds}$$

$$= 300,000 \text{ years}$$

16.5 **a.** Since KUa and KRa are inverses, the value KRa can be checked to validate that Pa was correctly supplied: Simply take some arbitrary block X and verify that X = D_{KRa} [E_{KUa} [X]].

b. Since the file /etc/publickey is publicly readable, an attacker can guess P (say P') and compute KRa' = $D_{P'}$ [E_P [KUa]]. now he can choose an arbitrary block Y and check to see if Y = D_{KRa} [E_{KUa} [Y]]. If so, it is highly probable that P' = P. Additional blocks can be used to verify the equality.

16.6 Yes.

16.7 Without the salt, the attacker can guess a password and encrypt it. If ANY of the users on a system use that password, then there will be a match. With the salt, the attacker must guess a password and then encrypt it once for each user, using the particular salt for each user.

16.8 It depends on the size of the user population, not the size of the salt, since the attacker presumably has access to the salt for each user. The benefit of larger salts is that the larger the salt, the less likely it is that two users will have the same salt. If multiple users have the same salt, then the attacker can do one encryption per password guess to test all of those users.

16.9 D is supposed to examine a program P and return TRUE if P is a computer virus and FALSE if it is not. But CV calls D. If D says that CV is a virus, then CV will not infect an executable. But if D says that CV is not a virus, it infects an executable. D always returns the wrong answer.

16.10 The purpose of the "no write down" rule, or *-property is to address the problem of Trojan horse software. With the *-property, information cannot be compromised through the use of a Trojan horse. Under this property, a program operating on behalf of one user cannot be used to pass information to any user having a lower or disjoint access class.

16.11 Drake is not authorized to read the string directly, so the no-read-up rule will prevent this. Similarly, Drake is not authorized to assign a security level of sensitive to the back-pocket file, so that is prevented as well.

16.12 Yes. The eavesdropper is left with two strings, one sent in each direction, and their XOR is the secret key.