SOLUTIONS MANUAL

OPERATING SYSTEMS NINTH EDITION

CHAPTERS 1–9

WILLIAM STALLINGS

NOTICE

This manual contains solutions to the review questions and homework problems in *Operating Systems, Ninth Edition*. If you spot an error in a solution or in the wording of a problem, I would greatly appreciate it if you would forward the information via email to wllmst@me.net. An errata sheet for this manual, if needed, is available at http://www.box.net/shared/fa8a0oyxxl . File name is S-OS9e-mmyy.

W.S.

TABLE OF CONTENTS

Chapter 1 Computer System Overview 5
Chapter 2 Operating System Overview 12
Chapter 3 Process Description and Control 15
Chapter 4 Threads 21
Chapter 5 Mutual Exclusion and Synchronization.........................25
Chapter 6 Deadlock and Starvation 42
Chapter 7 Memory Management 54
Chapter 8 Virtual Memory 60
Chapter 9 Uniprocessor Scheduling 69

Chapter 1 Computer System Overview

Answers to Questions

1.1 A processor, which controls the operation of the computer and performs its data processing functions ; a **main memory**, which stores both data and instructions; **I/O modules**, which move data between the computer and its external environment; and the system bus, which provides for communication among processors, main memory, and I/O modules.

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 A multicore computer is a special case of a multiprocessor, in which all of the processors are on a single chip.

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 = $\frac{1}{8 \text{ MHz}} = 125 \text{ ns}$

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 "word length" 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.

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

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.

Number of jobs	TAT	Throughput	Processor utilization
1	NT	1/N	50%
2	NT	2/N	100%
4	(2N – 1)T	4/(2N – 1)	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. This may seem like a lot, but it isn't.

2.6 **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.

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 RUN to READY can be caused by a time-quantum expiration
READY to NONRESIDENT occurs if memory is overcommitted, and a process is temporarily swapped out of memory
READY to RUN occurs only if a process is allocated the CPU by the dispatcher
RUN to BLOCKED can occur if a process issues an I/O or other kernel request.
BLOCKED to READY occurs if the awaited event completes (perhaps I/O completion)
BLOCKED to NONRESIDENT - same as READY to NONRESIDENT.

3.2 At time 22:

- P1: blocked for I/O
- P3: blocked for I/O
- P5: ready/running
- P7: blocked for I/O
- P8: ready/running

At time 37

- P1: ready/running
- P3: ready/running
- P5: blocked suspend
- P7: blocked for I/O
- P8: ready/running

At time 47

- P1: ready/running
- P3: ready/running
- P5: ready suspend
- P7: blocked for I/O
- P8: exit

3.3 a. New → Ready or Ready/Suspend: covered in text
Ready → Running or Ready/Suspend: covered in text
Ready/Suspend → Ready: covered in text
Blocked → Ready or Blocked/Suspend: covered in text
Blocked/Suspend → Ready /Suspend or Blocked: covered in text
Running → Ready, Ready/Suspend, or Blocked: covered in text
Any State → Exit: covered in text

b. New → Blocked, Blocked/Suspend, or Running: A newly created process remains in the new state until the processor is ready to take on an additional process, at which time it goes to one of the Ready states.
Ready → Blocked or Blocked/Suspend: Typically, a process that is ready cannot subsequently be blocked until it has run. Some systems may allow the OS to block a process that is currently ready, perhaps to free up resources committed to the ready process.
Ready/Suspend → Blocked or Blocked/Suspend: Same reasoning as preceding entry.
Ready/Suspend → Running: The OS first brings the process into memory, which puts it into the Ready state.
Blocked → Ready /Suspend: this transition would be done in 2 stages. A blocked process cannot at the same time be made ready and suspended, because these transitions are triggered by two different causes.

Blocked → Running: When a process is unblocked, it is put into the Ready state. The dispatcher will only choose a process from the Ready state to run
Blocked/Suspend → Ready: same reasoning as Blocked → Ready /Suspend
Blocked/Suspend → Running: same reasoning as Blocked → Running
Running → Blocked/Suspend: this transition would be done in 2 stages
Exit → Any State: Can't turn back the clock

3.4 Figure 9.3 in Chapter 9 shows the result for a single blocked queue. The figure readily generalizes to multiple blocked queues.

3.5 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.

3.6 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		

3.7 **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.

3.8 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.

3.9 **a.** An 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.

3.10 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.

3.11 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.

3.12

```
0
<child pid>
or
<child pid>
0
```

CHAPTER 4 THREADS

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.

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.

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

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.** The function counts the number of positive elements in a list.
b. This should work correctly, because `count_positives` in this specific case does not update `global_positives`, and hence the two threads operate on distinct global data and require no locking. Source: Boehn, H. et al. "Multithreading in C and C++." *;login*, February 2007.

4.8 This transformation is clearly consistent with the C language specification, which addresses only single-threaded execution. In a single-threaded environment, it is indistinguishable from the original. The pthread specification also contains no clear prohibition against this kind of transformation. And since it is a library and not a language specification, it is not clear that it could. However, in a multithreaded environment, the transformed version is quite different, in that it assigns to `global_positives`, even if the list contains only negative elements. The original program is now broken, because the update of `global_positives` by thread B may be lost, as a result of thread A writing back an earlier value of `global_positives`. By pthread rules, a thread-unaware compiler has turned a perfectly legitimate program into one with undefined semantics. Source: Boehn, H. et al. "Multithreading in C and C++." *;login*, February 2007.

4.9 **a.** This program creates a new thread. Both the main thread and the new thread then increment the global variable `myglobal` 20 times.

b. Quite unexpected! Because `myglobal` starts at zero, and both the main thread and the new thread each increment it by 20, we should see `myglobal` equaling 40 at the end of the program. But `myglobal` equals 21, so we know that something fishy is going on here. In thread_function(), notice that we copy `myglobal` into a local variable called `j`. The program increments `j`, then sleeps for one second, and only then copies the new `j` value into `myglobal`. That's the key. Imagine what happens if our main thread increments `myglobal` just after our new thread copies the value of `myglobal` into `j`. When `thread_function()` writes the value of `j` back to `myglobal`, it will overwrite the modification that the main thread made. Source: Robbins, D. "POSIX Threads Explained." *IBM Developer Works*, July 2000. www.ibm.com/developerworks/library/l-posix1.html

4.10 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.

4.11 **a.** Some programs have logical parallelism that can be exploited to simplify and structure the code but do not need hardware parallelism. For example, an application that employs multiple windows, only one of which is active at a time, could with advantage be implemented as a set of ULTs on a single LWP. The advantage of restricting such applications to ULTs is efficiency. ULTs may be created, destroyed, blocked, activated, and so on. without involving the kernel. If each ULT were known to the kernel, the kernel would have to allocate kernel data structures for each one and perform thread switching. Kernel-level thread switching is more expensive than user-level thread switching.

b. The execution of user-level threads is managed by the threads library whereas the LWP is managed by the kernel.

c. An unbound thread can be in one of four states: runnable, active, sleeping, or stopped. These states are managed by the threads library. A ULT in the active state is currently assigned to a LWP and executes while the underlying kernel thread executes. We can view the LWP state diagram as a detailed description of the ULT active state, because an thread only has an LWP assigned to it when it is in the Active state. The LWP state diagram is reasonably self-explanatory. An active thread is only executing when its LWP is in the Running state. When an active thread executes a blocking system call, the LWP enters the Blocked state. However, the ULT remains bound to that LWP and, as far as the threads library is concerned, that ULT remains active.

4.12 As the text describes, the Uninterruptible state is another blocked state. The difference between this and the Interruptible state is that in an uninterruptible state, a process is waiting directly on hardware conditions and therefore will not handle any signals. This is used in situations where the process must wait without interruption or when the event is expected to occur quite quickly. For example, this state may be used when a process opens a device file and the corresponding device driver starts probing for a corresponding hardware device. The device driver must not be interrupted until the probing is complete, or the hardware device could be left in an unpredictable state.

Chapter 5 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 **a.** Process P1 will only enter its critical section if flag[0] = false. Only P1 may modify flag[1], and P1 tests flag[0] only when flag[1] = true. It follows that when P1 enters its critical section we have:

(flag[1] **and** (**not** flag[0])) = true

Similarly, we can show that when P0 enters its critical section:

(flag[1] **and** (**not** flag[0])) = true

b. **Case 1:** A single process P(i) is attempting to enter its critical section. It will find flag[1-i] set to false, and enters the section without difficulty.
Case 2: Both process are attempting to enter their critical section, and *turn* = 0 (a similar reasoning applies to the case of *turn* = 1). Note that once both processes enter the **while** loop, the value of *turn* is modified only after one process has exited its critical section.
Subcase 2a: flag[0] = false. P1 finds flag[0] = 0, and can enter its critical section immediately.
Subcase 2b: flag[0] = true. Since *turn* = 0, P0 will wait in its external loop for flag[1] to be set to false (without modifying the value of flag[0]. Meanwhile, P1 sets flag[1] to false (and will wait in its internal loop because *turn* = 0). At that point, P0 will enter the critical section.
Thus, if both processes are attempting to enter their critical section, there is no deadlock.

5.2 It doesn't work. There is no deadlock; mutual exclusion is enforced; but starvation is possible if *turn* is set to a non-contending process.

5.3 **a.** There is no variable that is both read and written by more than one process (like the variable `turn` in Dekker's algorithm). Therefore, the bakery algorithm does not require atomic load and store to the same global variable.
b. Because of the use of `flag` to control the reading of `turn`, we again do not require atomic load and store to the same global variable.

5.4 On uniprocessors you can avoid interruption and thus concurrency by disabling interrupts. Also on multiprocessor machines another problem arises: memory ordering (multiple processors accessing the memory unit).

5.5 **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.

d. This can be accomplished using three concurrent processes. One of these, Input, reads the cards and simply passes the characters (with the additional trailing space) through a finite buffer, say InBuffer, to a process Squash which simply looks for double asterisks and passes a stream of modified characters through a second finite buffer, say OutBuffer, to a process Output, which extracts the characters from the second buffer and prints them in 15 column lines. A producer/consumer semaphore approach can be used.

5.6 a. For "x is 10", the interleaving producing the required behavior is easy to find since it requires only an interleaving at the source language statement level. The essential fact here is that the test for the value of x is interleaved with the increment of x by the other process. Thus, x was not equal to 10 when the test was performed, but was equal to 10 by the time the value of x was read from memory for printing.

```
                                 M(x)
P1: x = x - 1;                    9
P1: x = x + 1;                    10
P2: x = x - 1;                    9
P1: if(x != 10)                   9
P2: x = x + 1;                    10
P1: printf("x is %d", x);         10
```

"x is 10" is printed.

b. For "x is 8" we need to be more inventive, since we need to use interleavings of the machine instructions to find a way for the value of x to be established as 9 so it can then be evaluated as 8 in a later cycle. Notice how the first two blocks of statements correspond to C source lines, but how later blocks of machine language statements interleave portions of a source language statement.

```
Instruction                     M(x)    P1-R0   P2-R0
P1: LD   R0, x                  10      10      --
P1: DECR R0                     10      9       --
P1: STO  R0, x                  9       9       --

P2: LD   R0, x                  9       9       9
P2: DECR R0                     9       9       8
P2: STO  R0, x                  8       9       8
```

```
P1: LD   R0, x                  8       8       8
P1: INCR R0                     8       9       --

P2: LD   R0, x                  8       9       8
P2: INCR R0                     8       9       9
P2: STO  R0, x                  9       9       9
P2: if(x != 10) printf("x is %d", x);
P2: "x is 9" is printed.

P1: STO  R0, x                  9       9       9
P1: if(x != 10) printf("x is %d", x);
P1: "x is 9" is printed.

P1: LD   R0, x                  9       9       9
P1: DECR R0                     9       8       --
P1: STO  R0, x                  8       8       --

P2: LD   R0, x                  8       8       8
P2: DECR R0                     8       8       7
P2: STO  R0, x                  7       8       7

P1: LD   R0, x                  7       7       7
P1: INCR R0                     8       8       7
P1: STO  R0, x                  8       8       7
P1: if(x != 10) printf("x is %d", x);
P1: "x is 8" is printed.
```

5.7 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.

5.8 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.

5.9 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.

5.10 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:

$$(\text{number}[i], i) < (\text{number}[k], k)$$

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] $\neq$ 0 and
(number[i], i) < (number[k], k).

5.11 Suppose we have two processes just beginning; call them p0 and p1. Both reach line 3 at the same time. Now, we'll assume both read number[0] and number[1] before either addition takes place. Let p1 complete the line, assigning 1 to number[1], but p0 block before the assignment. Then p1 gets through the while loop at line 5 and enters the critical section. While in the critical section, it blocks; p0 unblocks, and assigns 1 to number[0] at line 3. It proceeds to the while loop at line 5. When it goes through that loop for j = 1, the first condition on line 5 is true. Further, the second condition on line 5 is false, so p0 enters the critical section. Now p0 and p1 are both in the critical section, violating mutual exclusion. The reason for choosing is to prevent the while loop in line 5 from being entered when process j is setting its number[j]. Note that if the loop is entered and then process j reaches line 3, one of two situations arises. Either number[j] has the value 0 when the first test is executed, in which case process i moves on to the next process, or number[j] has a non-zero value, in which case at some point number[j] will be greater than number[i] (since process i finished executing statement 3 before process j began). Either way, process i will enter the critical section before process j, and when process j reaches the while loop, it will loop at least until process i leaves the critical section.

5.12 This is a program that provides mutual exclusion for access to a critical resource among N processes, which can only use the resource one at a time. The unique feature of this algorithm is that a process need wait no more then N - 1 turns for access. The values of `control[i]` for process `i` are interpreted as follows: 0 = outside the critical section and not seeking entry; 1 = wants to access critical section; 2 = has claimed precedence for entering the critical section. The value of `k` reflects whose turn it is to enter the critical section. **Entry algorithm:** Process `i` expresses the intention to enter the critical section by setting `control[i] = 1`. If no other process between `k` and `i` (in circular order) has expressed a similar intention then process i claims its precedence for entering the critical section by setting `control[i] = 2`. If `i` is the only process to have made a claim, it enters the critical section by setting `k = 1`; if there is contention, `i` restarts the entry algorithm. **Exit algorithm:** Process `i` examines the array `control` in circular fashion beginning with entry `i + 1`. If process i finds a process with a nonzero `control` entry, then *k* is set to the identifier of that process.

The original paper makes the following observations: First observe that no two processes can be simultaneously processing between their statements L3 and L6. Secondly, observe that the system cannot be blocked; for if none of the processes contending for access to its critical section has yet passed its statement L3, then after a point, the value of k will be constant, and the first contending process in the cyclic ordering (k, k + 1, ..., N, 1, ..., k – 1) will meet no resistance. Finally, observe that no single process can be blocked. Before any process having executed its critical section can exit the area protected from simultaneous processing, it must designate as its unique successor the first contending process in the cyclic ordering, assuring the choice of any individual contending process within N – 1 turns. Original paper: Eisenberg, A., and McGuire, M. "Other Comments on Dijkstra's Concurrent Programming Control Problem." *Communications of the ACM*, November 1972.

5.13 a. `exchange(keyi, bolt)`

b. The statement `bolt = 0` is preferable. An atomic statement such as exchange will use more resources.

5.14
```
var  j: 0..n-1;
     key: boolean;
while (true) {
  waiting[i] = true;
  key := true;
  while (waiting[i] && key)
    key = (compare_and_swap(lock, 0, 1) == 0);
  waiting[i] = false;
  < critical section >
  j = i + 1 mod n;
  while (j != i && !waiting[j]) j = j + 1 mod n;
  if (j == i) lock := false
  else waiting = false;
  < remainder section >
}
```

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.

5.15 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.

5.16 a. There are two problems. First, because unblocked processes must reenter the mutual exclusion (line 10) there is a chance that newly arriving processes (at line 5) will beat them into the critical section. Second, there is a time delay between when the waiting processes are unblocked and when they resume execution and update the counters. The waiting processes must be accounted for as soon as they are unblocked (because they might resume execution at any time), but it may be some time before the processes actually do resume and update the counters to reflect this. To illustrate, consider the case where three processes are blocked at line 9. The last active process will unblock them (lines 25-28) as it departs. But there is no way to predict when these processes will resume executing and update the counters to reflect the fact that they have become active. If a new process reaches line 6 before the unblocked ones resume, the new one should be blocked. But the status variables have not yet been updated so the new process will gain access to the resource. When the unblocked ones eventually resume execution, they will also begin accessing the resource. The solution has failed because it has allowed four processes to access the resource together.

b. This forces unblocked processes to recheck whether they can begin using the resource. But this solution is more prone to starvation because it encourages new arrivals to “cut in line” ahead of those that were already waiting.

5.17 a. This approach is to eliminate the time delay. If the departing process updates the waiting and active counters as it unblocks waiting processes, the counters will accurately reflect the new state of the system before any new processes can get into the mutual exclusion. Because the updating is already done, the unblocked processes need not reenter the critical section at all. Implementing this pattern is easy. Identify all of the work that would have been done by an unblocked process and make the unblocking process do it instead.

b. Suppose three processes arrived when the resource was busy, but one of them lost its quantum just before blocking itself at line 9 (which is unlikely, but certainly possible). When the last active process departs, it will do three `semSignal` operations and set must_wait to true. If a new process arrives before the older ones resume, the new one will decide to block itself. However, it will breeze past the `semWait` in line 9 without blocking, and when the process that lost its quantum earlier runs it will block itself instead. This is not an error—the problem doesn’t dictate which processes access the resource, only how many are allowed to access it. Indeed, because the unblocking order of semaphores is implementation dependent, the only portable way to ensure that processes proceed in a particular order is to block each on its own semaphore.

c. The departing process updates the system state on behalf of the processes it unblocks.

5.18 a. After you unblock a waiting process, you leave the critical section (or block yourself) without opening the mutual exclusion. The unblocked process doesn’t reenter the mutual exclusion—it takes over your ownership of it. The process can therefore safely update the system state on its own. When it is finished, it reopens the mutual exclusion. Newly arriving processes can no longer cut in line because they cannot enter the mutual exclusion until the unblocked process has finished. Because the unblocked process takes care of its own updating, the cohesion of this solution is better. However, once you have unblocked a process, you must immediately stop accessing the variables protected by the mutual exclusion. The safest approach is to immediately leave (after line 26, the process leaves without opening the mutex) or block yourself.

b. Only one waiting process can be unblocked even if several are waiting—to unblock more would violate the mutual exclusion of the status variables. This problem is solved by having the newly unblocked process check whether more processes should be unblocked (line 14). If so, it passes the baton to one of them (line 15); if not, it opens up the mutual exclusion for new arrivals (line 17).

c. This pattern synchronizes processes like runners in a relay race. As each runner finishes her laps, she passes the baton to the next runner. “Having the baton” is like having permission to be on the track. In the synchronization world, being in the mutual exclusion is analogous to having the baton—only one person can have it..

5.19 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.

5.20

```
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;
```

5.21 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()`.

5.22

	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.

5.23

```
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.
```

5.24 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.

5.25

Scheduled step of execution	full's state & queue	Buffer	empty's state & queue
Initialization	full = 0	OOO	empty = +3
Ca executes c1	full = −1 (Ca)	OOO	empty = +3
Cb executes c1	full = −2 (Ca, Cb)	OOO	empty = +3
Pa executes p1	full = −2 (Ca, Cb)	OOO	empty = +2
Pa executes p2	full = −2 (Ca, Cb)	XOO	empty = +2
Pa executes p3	full = −1 (Cb) Ca	XOO	empty = +2
Ca executes c2	full = −1 (Cb)	OOO	empty = +2
Ca executes c3	full = −1 (Cb)	OOO	empty = +3
Pb executes p1	full = −1 (Cb)	OOO	empty = +2
Pa executes p1	full = −1 (Cb)	OOO	empty = +1
Pa executes p2	full = −1 (Cb)	XOO	empty = +1
Pb executes p2	full = −1 (Cb)	XXO	empty = +1
Pb executes p3	full = 0 (Cb)	XXO	empty = +1
Pc executes p1	full = 0 (Cb)	XXO	empty = 0
Cb executes c2	full = 0	XOO	empty = 0
Pc executes p2	full = 0	XXO	empty = 0
Cb executes c3	full = 0	XXO	empty = +1
Pa executes p3	full = +1	XXO	empty = +1
Pb executes p1-p3	full = +2	XXX	empty = 0
Pc executes p3	full = +3	XXX	empty = 0
Pa executes p1	full = +3	XXX	empty = −1(Pa)
Pd executes p1	full = +3	XXX	Empty = −2(Pa, Pd)
Ca executes c1-c3	full = +2	XXO	empty = −1(Pd) Pa
Pa executes p2	full = +2	XXX	empty = −1(Pd)
Cc executes c1-c2	full = +1	XXO	empty = −1(Pd)
Pa executes p3	full = +2	XXO	empty = −1(Pd)
Cc executes c3	full = +2	XXO	empty =0(Pd
Pd executes p2-p3	full = +3	XXX	empty = 0

Differences from one step to the next are highlighted in red.

5.26

```
#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)
     }
   }
```

5.27 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.

5.28 The code for the one-writer many readers is fine if we assume that the readers have always priority. The problem is that the readers can starve the writer(s) since they may never all leave the critical region, i.e., there is always at least one reader in the critical region, hence the ‘wrt’ semaphore may never be signaled to writers and the writer process does not get access to ‘wrt’ semaphore and writes into the critical region.

CHAPTER 6 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 **Prevention:** Hold-and-wait approach: Require that a car request both quadrants that it needs and blocking the car until both quadrants can be granted. No preemption approach: releasing an assigned quadrant is problematic, because this means backing up, which may not be possible if there is another car behind this car. Circular-wait approach: assign a linear ordering to the quadrants.
Avoidance: The algorithms discussed in the chapter apply to this problem. Essentially, deadlock is avoided by not granting requests that might lead to deadlock.
Detection: The problem here again is one of backup.

6.3
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.

6.4 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.

6.5 a. 15 – (2+0+4+1+1+1) = 6
6 – (0+1+1+0+1+0) = 3
9 – (2+1+0+0+0+1) = 5
10 – (1+1+2+1+0+1) = 4

b. Need Matrix = Max Matrix – Allocation Matrix

	need			
process	A	B	C	D
P0	7	5	3	4
P1	2	1	2	2
P2	3	4	4	2
P3	2	3	3	1
P4	4	1	2	1
P5	3	4	3	3

c. The following matrix shows the order in which the processes finish and shows what is available once each process finishes

	available			
process	A	B	C	D
P5	7	3	6	5
P4	8	4	6	5
P3	9	4	6	6
P2	13	5	6	8
P1	13	6	7	9
P0	15	6	9	10

d. ANSWER is NO for the following reasons: If this request were granted, then the new allocation matrix would be:

	allocation			
process	A	B	C	D
P0	2	0	2	1
P1	0	1	1	1
P2	4	1	0	2
P3	1	0	0	1
P4	1	1	0	0
P5	4	2	4	4

Then the new need matrix would be

process	allocation A	B	C	D
P0	7	5	3	4
P1	2	1	2	2
P2	3	4	4	2
P3	2	3	3	1
P4	4	1	2	1
P5	0	2	0	0

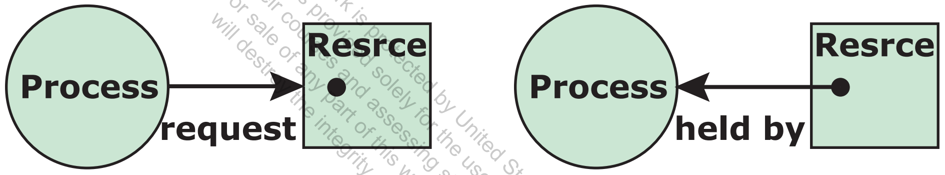
And Available is then:

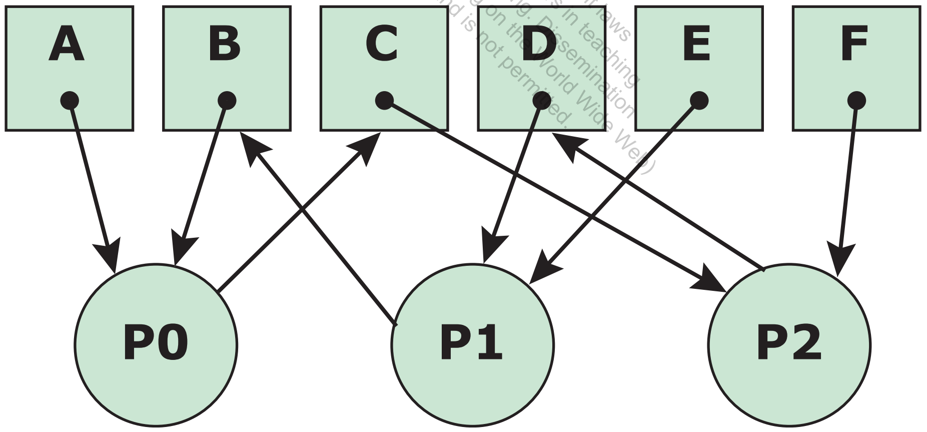
Available

A	B	C	D
3	1	2	1

Which means we could NOT satisfy ANY process' need.

6.6 a.





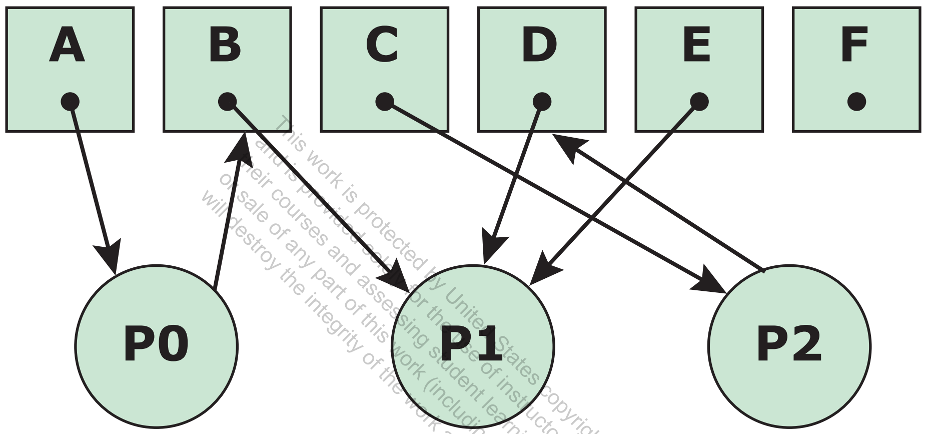
There is a deadlock if the scheduler goes, for example: P0-P1-P2-P0-P1-P2 (line by line): Each of the 6 resources will then be held by one process, so all 3 processes are now blocked at their third line inside the loop, waiting for a resource that another process holds. This is illustrated by the circular wait (thick arrows) in the RAG above: P0→C→P2→D→P1→B→P0.

b. Any change in the order of the get() calls that alphabetizes the resources inside each process code will avoid deadlocks. More generally, it can be a direct or reverse alphabet order, or any arbitrary but predefined ordered list of the resources that should be respected inside each process.

Explanation: if resources are uniquely ordered, cycles are not possible any more because a process cannot hold a resource that comes after another resource it is holding in the ordered list. See this remark in Section 6.2 about Circular Wait Prevention. For example:

A	B	C
B	D	D
C	E	F

With this code, and starting with the same worst-case scheduling scenario P0-P1-P2, we can only continue with either P1-P1-CR1… or P2-P2-CR2…. For example, in the case P1-P1, we get the following RAG without circular wait:



After entering CR1, P1 then releases all its resources and P0 and P2 are free to go. Generally the same thing would happen with any fixed ordering of the resources: one of the 3 processes will always be able to enter its critical area and, upon exit, let the other two progress.

6.7 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.

6.8 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:

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

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.

6.9

$$\begin{aligned} i + o + p &\le max \\ i + o &\le max - resp \\ i + p &\le max - reso \\ i &\le max - (reso + resp) \end{aligned}$$

6.10 **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.

6.11 **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	

6.12 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.

6.13 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:

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

6.14 a. Yes. If foo( ) executes semWait(S) and then bar( ) executes semWait(R) both processes will then block when each executes its next instruction. Since each will then be waiting for a semSignal( ) call from the other, neither will ever resume execution.

b. No. If either process blocks on a semWait( ) call then either the other process will also block as described in (a) or the other process is executing in its critical section. In the latter case, when the running process leaves its critical section, it will execute a semSignal( ) call, which will awaken the blocked process.

6.15 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.

6.16 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.

6.17 The philosophers can starve while repeatedly picking up and putting down their left forks in perfect unison.

6.18 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 $\in$ 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.

6.19 One solution (6.14) waits on available forks; the other solution (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.

6.20 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.

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

6.22 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.

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

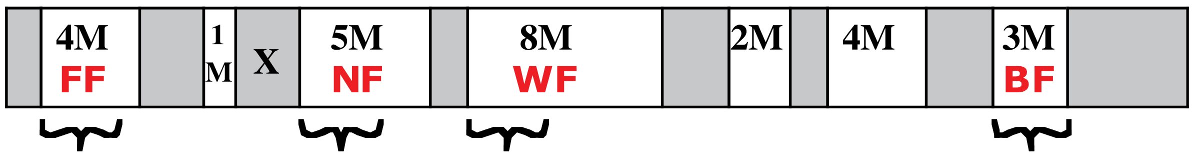
7.2 The number of partitions equals the number of bytes of main memory divided by the number of bytes in each partition: $2^{24}/2^{16} = 2^8$. Eight bits are needed to identify one of the 2^8 partitions.

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 Let N be the length of list of free blocks.
Best-fit: Average length of search = N, as each free block in the list is considered, to find the best fit.
First-fit: The probability of each free block in the list to be large enough or not large enough, for a memory request is equally likely. Thus the probability of first free block in the list to be first fit is 1/2. For the second free block to be first fit, the first free block should be smaller, and the second free block should be large enough, for the memory request. Thus the probability of second free block to be first fit is 1/2 x 1/2 = 1/4. Proceeding in the same way, probability of ith free block in the list to be first fit is $1/2^i$. Thus the average length of search = $1/2 + 2/2^2 + 3/2^3 + \ldots \ldots + N/2^N + N/2^N$
(the last term corresponds to the case, when no free block fits the request). Above length of search has a value between 1 and 2.
Next-fit: Same as first-fit, except for the fact that search starts where the previous first-fit search ended.

7.5 a. 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.
b. Same as best fit.

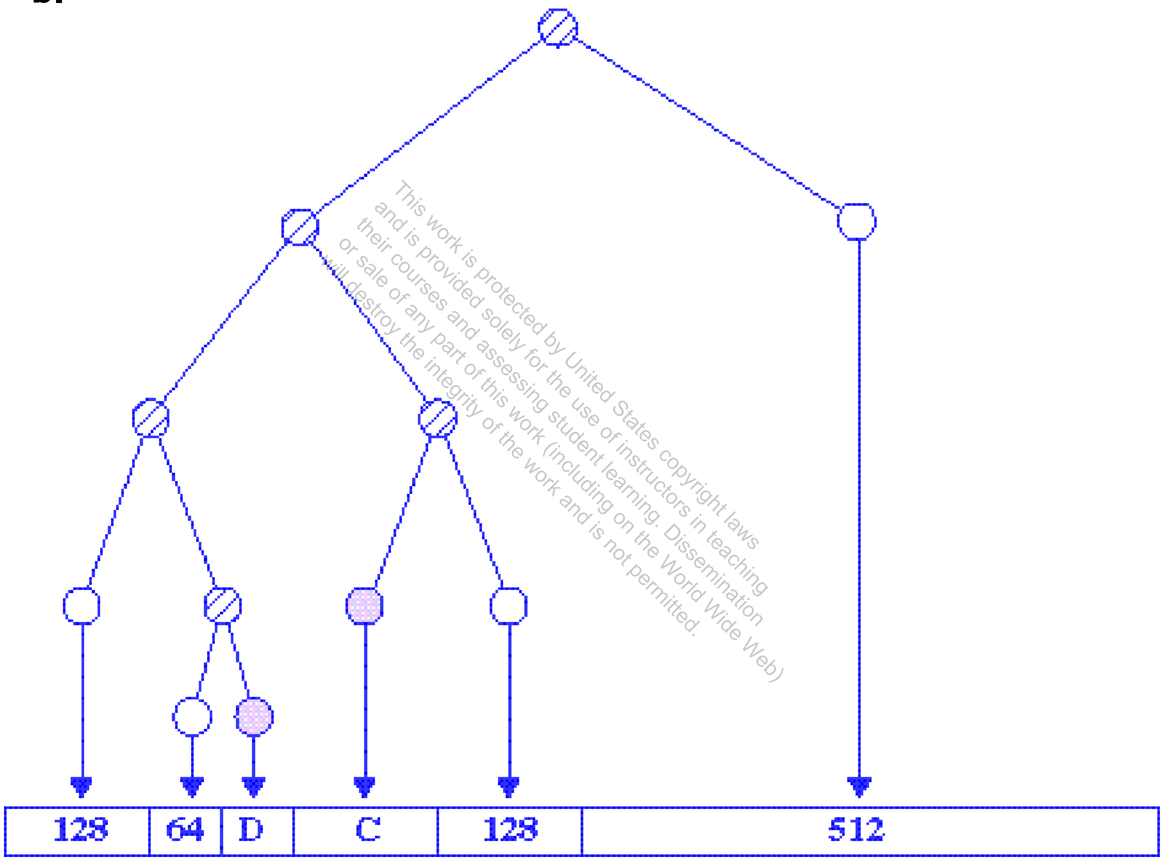
7.6 a. When the 2-MB process is placed, it fills the leftmost portion of the free block selected for placement. Because the diagram shows an empty block to the left of X, the process swapped out after X was placed must have created that empty block. Therefore, the maximum size of the swapped out process is 1M.
b. The free block consisted of the 5M still empty plus the space occupied by X, for a total of 7M.
c. The answers are indicated in the following figure:



7.7 a.

Request 70	A	128		256		512
Request 35	A	B	64	256		512
Request 80	A	B	64	C	128	512
Return A	128	B	64	C	128	512
Request 60	128	B	D	C	128	512
Return B	128	64	D	C	128	512
Return D	256			C	128	512
Return C	1024					

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) $\times$ (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 **a.** The page number is in the higher 8 bits: 00010100. We chop it off from the address and replace it with the frame number, which is 4 times less, that is, shifted 2 bits to the right: 00000101. Therefore the result is this frame number concatenated with the offset 10111010:
binary physical address = 0000010110111010
b. The segment number is in the higher 6 bits: 000101. We chop it off from the address and add the remaining offset $\underline{0010111010}$ to the base of the segment. The base is 22 = 10110 added to the segment number times 4,096, that is, shifted 12 bits to the left: 10110 + 0101000000000000 = $\underline{0101000000010110}$. So adding up the 2 two underlined numbers gives:
binary physical address = 0101000011010000

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

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 Thrashing is 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, ($7 \times 1024+28 = 7196$)

(ii) 2221 = $2 \times 1024 + 173$ maps to VPN 2, page fault

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

8.2 a. 3 page faults for every 4 executions of C[i, j] = A[i, j] +B[i, j].

b. Yes. The page fault frequency can be minimized by switching the inner and outer loops.

c. After modification, there are 3 page faults for every 256 executions.

8.3 a. 4 MByte

b. Number of rows: 2^6 x 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

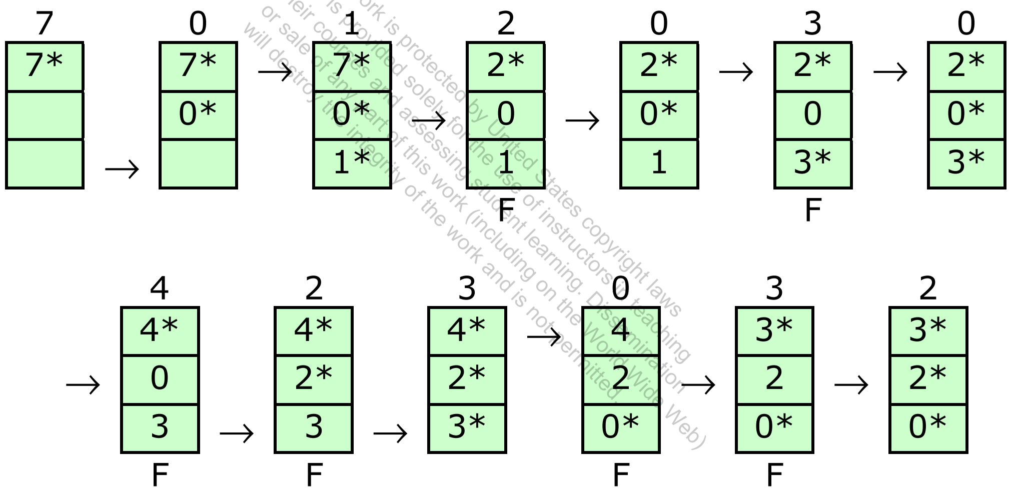
8.4 a. FIFO:

7	0	1	2	0	3	0	4	2	3	0	3	2
7	7	7	2	2	2	2	4	4	4	0	0	0
	0	0	0	0	3	3	3	2	2	2	2	2
		1	1	1	1	0	0	0	3	3	3	3
			F		F	F	F	F	F	F		

b. LRU:

7	0	1	2	0	3	0	4	2	3	0	3	2
7	7	7	2	2	2	2	4	4	4	0	0	0
	0	0	0	0	0	0	0	0	3	3	3	3
		1	1	1	3	3	3	2	2	2	2	2
			F		F		F	F	F	F		

c. Clock:



d. OPT:

7	0	1	2	0	3	0	4	2	3	0	3	2
7	7	7	2	2	2	2	2	2	2	2	2	2
	0	0	0	0	0	0	4	4	4	0	0	0
		1	1	3	3	3	3	3	3	3	3	3
			F		F					F		

e. FIFO: page faults = 7   miss rate = 70%
LRU: page faults = 6   miss rate = 60%
Clock: page faults = 6   miss rate = 60%
OPT: page faults = 3   miss rate = 30%

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	2	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	4	4	4	4	4	4	4	4
-	-	2	2	2	2	2	2	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	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	2	2	2	2	2	2	2	2	2	2	2	3	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.

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

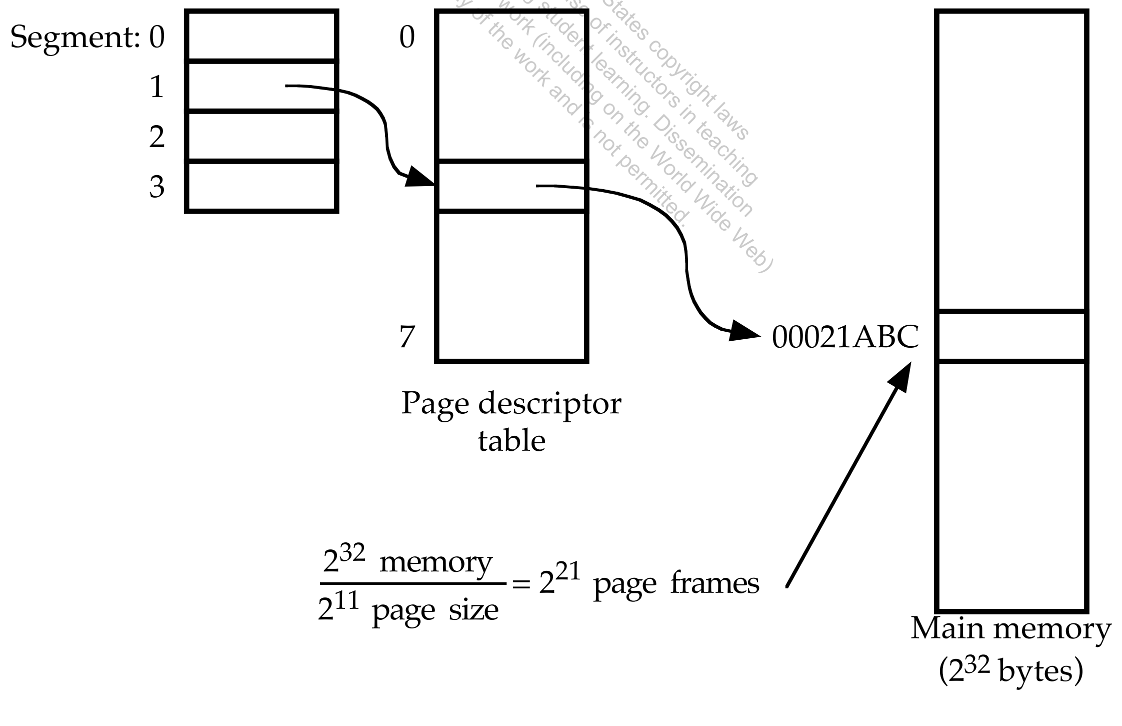
b., c.

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

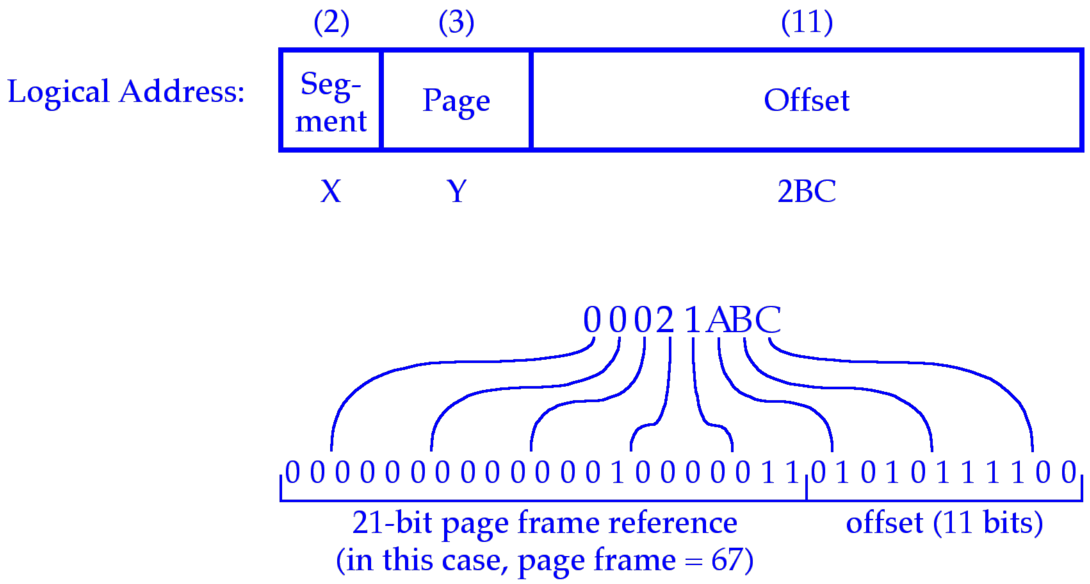
$s_{20}(\Delta)$ is an increasing function of Δ. $m_{20}(\Delta)$ is a nonincreasing function of Δ.

8.16 Consider this 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.

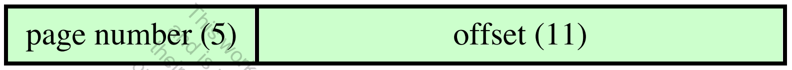
8.17 $\dfrac{2^{32} \text{ memory}}{2^{11} \text{ page size}} = 2^{21} \text{ page frames}$


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

a. $8 \times 2K = 16K$
b. $16K \times 4 = 64K$
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.

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 a. Shortest Remaining Time:

P1	P1	P2	P2	P1	P1	P1	P4	P4	P4	P4	P3	P3	P3	P3	P3	P3	P3	P3	P3	P3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Explanation: P1 starts but is preempted after 20ms when P2 arrives and has shorter burst time (20ms) than the remaining burst time of P1 (30 ms) . So, P1 is preempted. P2 runs to completion. At 40ms P3 arrives, but it has a longer burst time than P1, so P1 will run. At 60ms P4 arrives. At this point P1 has a remaining burst time of 10 ms, which is the shortest time, so it continues to run. Once P1 finishes, P4 starts to run since it has shorter burst time than P3.

Non-preemptive Priority:

P1	P1	P1	P1	P1	P2	P2	P4	P4	P4	P4	P3	P3	P3	P3	P3	P3	P3	P3	P3	P3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Explanation: P1 starts, but as the scheduler is non-preemptive, it continues executing even though it has lower priority than P2. When

P1 finishes, P2 and P3 have arrived. Among these two, P2 has higher priority, so P2 will be scheduled, and it keeps the processor until it finishes. Now we have P3 and P4 in the ready queue. Among these two, P4 has higher priority, so it will be scheduled. After P4 finishes, P3 is scheduled to run.

Round Robin with quantum of 30 ms:

P1	P1	P1	P2	P2	P1	P1	P3	P3	P3	P4	P4	P4	P3	P3	P3	P4	P3	P3	P3	P3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Explanation: P1 arrives first, so it will get the 30ms quantum. After that, P2 is in the ready queue, so P1 will be preempted and P2 is scheduled for 20ms. While P2 is running, P3 arrives. Note that P3 will be queued after P1 in the FIFO ready queue. So when P2 is done, P1 will be scheduled for the next quantum. It runs for 20ms. In the mean time, P4 arrives and is queued after P3. So after P1 is done, P3 runs for one 30 ms quantum. Once it is done, P4 runs for a 30ms quantum. Then again P3 runs for 30 ms, and after that P4 runs for 10 ms, and after that P3 runs for 30+10ms since there is nobody left to compete with.

b. **Shortest Remaining Time:** (20+0+70+10)/4 = 25 ms.
Explanation: P2 does not wait, but P1 waits 20ms, P3 waits 70ms and P4 waits 10ms.
Non-preemptive Priority: (0+30+10+70)/4 = 27.5ms
Explanation: P1 does not wait, P2 waits 30ms until P1 finishes, P4 waits only 10ms since it arrived at 60ms and it is scheduled at 70ms. P3 waits 70ms.
Round-Robin: (20+10+70+70)/4 = 42.5ms
Explanation: P1 waits only for P2 (for 20ms). P2 waits only 10ms until P1 finishes the quantum (it arrives at 20ms and the quantum is 30ms). P3 waits 30ms to start, then 40ms for P4 to finish. P4 waits 40ms to start and one quantum slice for P3 to finish.

9.2 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

9.3 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

$$t_1 \le t_2 \le . . . \le t_n$$

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.

9.4 The data points for the plot:

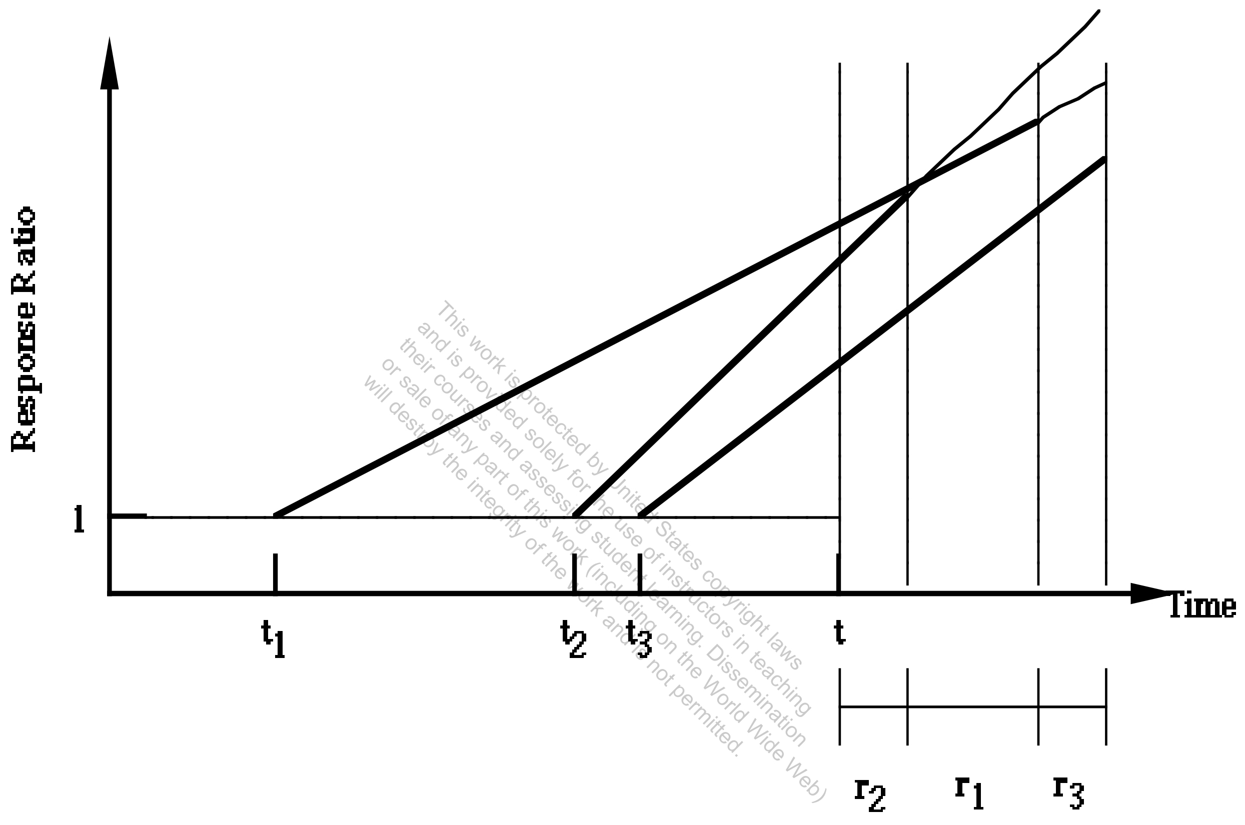
Age of Observation	Observed Value	Simple Average	alpha = 0.8	alpha = 0.5
1	6	0.00	0.00	0.00
2	4	3.00	4.80	3.00
3	6	3.33	4.16	3.50
4	4	4.00	5.63	4.75
5	13	4.00	4.33	4.38
6	13	5.50	11.27	8.69
7	13	6.57	12.65	10.84

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 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:

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

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

$$T_j = t + r_a + r_b + \ldots + r_j$$

and job j reaches the response ratio

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

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, $T_i \leq T_j$ implies

$$R_j(T_j) \geq R_j(T_i)$$

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:

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

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:

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

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

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 $w \times T_s$, 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 $\rho \times T_s$, 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 = $q\delta$, 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 $q\delta$ seconds, the total wait time is as follows:

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

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

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

Referring to the queuing formulas in Chapter 20 or Appendix H, the mean number of items in the system, q, can be expressed as

$$q = \rho/(1-\rho)$$

Thus,

$$R_x = [\rho/(1 - \rho) + 1]x = x/(1 - \rho)$$

9.11 **a.** Because the ready queue has multiple pointers to the same process, the system is giving that process preferential treatment That is, this process will get double the processor time than a process with only one pointer.

b. The advantage is that more important jobs could be given more processor time by just adding an additional pointer (i.e., very little extra overhead to implement).

c. Want longer time slice to processes deserving higher priority.
- add bit in PCB that says whether a process is allowed to execute two time slices
- add integer in PCB that indicates the number of time slices a process is allowed to execute
- have two ready queues, one of which has a longer time slice for higher priority jobs

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:

$$y = \beta/\lambda'$$

$$y = [(1/\lambda') - (1/\lambda)]\alpha$$

which therefore gives

$$\lambda' = \lambda[1 - (\beta/\alpha)]$$

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

$$q = \rho/(1 - \rho)$$

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

$$R = q/\lambda = s/(1 - \rho)$$

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

$$V = t/(1-\rho') \quad \text{where } \rho' = \lambda' s$$

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 λ', and therefore the utilization is ρ'. Accordingly, from our basic M/M/1 formulas, we have

$$V = s/(1-\rho')$$

Thus,

$$W = R - V = s/[1/(1-\rho) - 1/(1-\rho')]$$

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.

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