INSTRUCTORS MANUAL

OPERATING SYSTEMS:
INTERNALS AND DESIGN PRINCIPLES
FOURTH EDITION

WILLIAM STALLINGS

TABLE OF CONTENTS

Part One

SOLUTIONS MANUAL

This manual contains solutions to all of the problems in *Operating Systems, Fourth 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 me at ws@shore.net. An errata sheet for this manual, if needed, is available at ftp://ftp.shore.net/members/ws/S/

W.S.

CHAPTER 1
COMPUTER SYSTEM OVERVIEW

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 "wordlength" of the memory will have to double to be able to send/receive 32-bit quantities.

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

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

b. The process described in (a) is very wasteful. The CPU, which is much faster than the teletype, must repeatedly check FGI and FGO. If interrupts are used,

the teletype can issue an interrupt to the CPU whenever it is ready to accept or send data. The IEN register can be set by the CPU (under programmer control)

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

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

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

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

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

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

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

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

1.11 Define

C_i = Average cost per bit, memory level i

S_i = Size of memory level i

T_i = Time to access a word in memory level i

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

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

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

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

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

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

We can write:

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

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

$$T_2 = B_1 + T_1$$

Further

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

Generalizing:

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

So

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

But

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

Finally

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

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

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

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

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

H = 1190/1200

1.13 There are three cases to consider:

Location of referenced word	Probability	Total time for access in ns
In cache	0.9	20
Not in cache, but in main memory	(0.1)(0.6) = 0.06	60 + 20 = 80
Not in cache or main memory	(0.1)(0.4) = 0.04	12ms + 60 + 20 = 12000080

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 PROBLEMS

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

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

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

2.3 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 = 12000080

So the average access time would be:

$$\text{Avg} = (0.9)(20) + (0.06)(80) + (0.04)(12000080) = 480026 \text{ ns}$$

2.4 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.

2.5 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.

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

CHAPTER 3
PROCESS DESCRIPTION AND CONTROL

ANSWERS TO QUESTIONS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ANSWERS TO PROBLEMS

3.1

- **Creation and deletion of both user and system processes.** The processes in the system can execute concurrently for information sharing, computation speedup, modularity, and convenience. Concurrent execution requires a mechanism for process creation and deletion. The required resources are given to the process when it is created, or allocated to it while it is running. When the process terminates, the OS needs to reclaim any reusable resources.
- **Suspension and resumption of processes.** In process scheduling, the OS needs to change the process's state to waiting or ready state when it is waiting for some resources. When the required resources are available, OS needs to change its state to running state to resume its execution.
- **Provision of mechanism for process synchronization.** Cooperating processes may share data. Concurrent access to shared data may result in data inconsistency. OS has to provide mechanisms for processes synchronization to ensure the orderly execution of cooperating processes, so that data consistency is maintained.
- **Provision of mechanism for process communication.** The processes executing under the OS may be either independent processes or cooperating processes. Cooperating processes must have the means to communicate with each other.

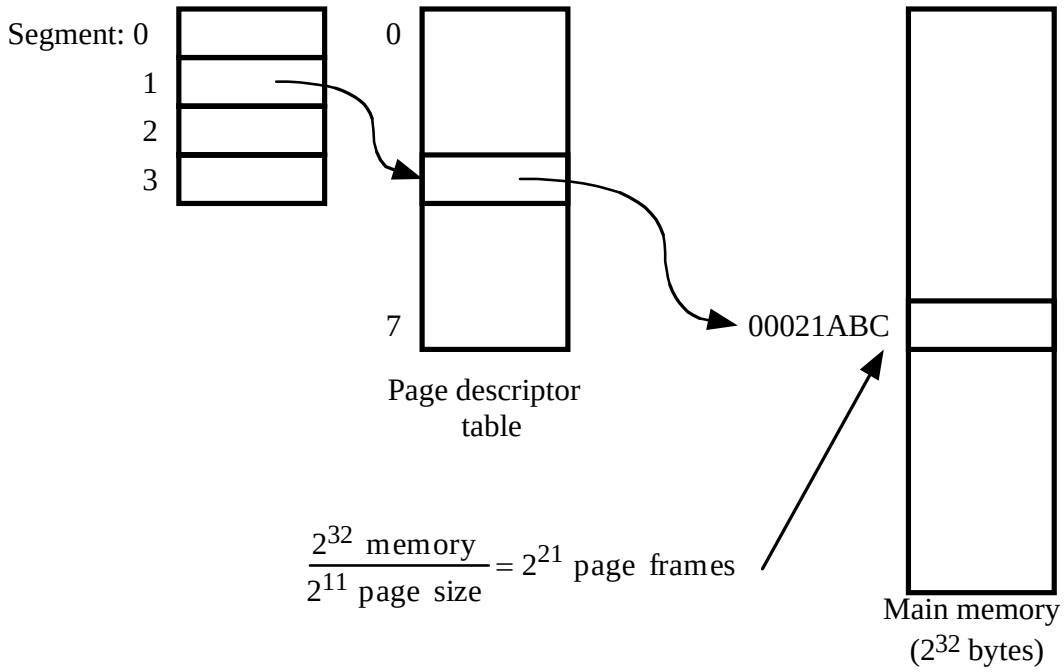
- **Provision of mechanisms for deadlock handling.** In a multiprogramming environment, several processes may compete for a finite number of resources. If a deadlock occurs, all waiting processes will never change their waiting state to running state again, resources are wasted and jobs will never be completed.

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

> Suppose a process has been executing for a while and needs an additional magnetic tape drive so that it can write out a temporary file. Before it can initiate a write to tape, it must be given permission to use one of the drives. When it makes its request, a tape drive may not be available, and if that is the case, the process will be placed in the blocked state. At some point, we assume the system will allocate the tape drive to the process; at that time the process will be moved back to the active state. When the process is placed into the execute state again it will request a write operation to its newly acquired tape drive. At this point, the process will be move to the suspend state, where it waits for the completion of the current write on the tape drive that it now owns.

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

3.3 We show the result for a single blocked queue. The figure readily generalizes to multiple blocked queues.



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

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

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

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

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

Variety of wait states (resident)		Event satisfied	Outswap		
Variety of wait states (outswapped)			Event satisfied		

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

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

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

b. An approach to dealing with this problem, known as a trusted system, is discussed in Chapter 15.

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

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

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

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

CHAPTER 4
THREADS, SMP, AND MICROKERNELS

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 ownereship 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 all of the threads within the process are blocked. **2.** In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing. A kernel assigns one process to only

one processor at a time. Therefore, only a single thread within a process can execute at a time.

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

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

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

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

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

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

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

4.15 Messages.

ANSWERS TO PROBLEMS

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

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

4.3 **a.** The use of sessions is well suited to the needs of an interactive graphics interface for personal computer and workstation use. It provides a uniform mechanism for keeping track of where graphics output and keyboard/mouse input should be directed, easing the task of the operating system.
b. The split would be the same as any other process/thread scheme, with address space and files assigned at the process level.

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

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

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

4.7 **a.** If a conservative policy is used, at most 20/4 = 5 processes can be active simultaneously. Because one of the drives allocated to each process can be idle most of the time, at most 5 drives will be idle at a time. In the best case, none of the drives will be idle.
b. To improve drive utilization, each process can be initially allocated with three tape drives. The fourth one will be allocated on demand. In this policy, at most $\lfloor 20/3 \rfloor = 6$ processes can be active simultaneously. The minimum number of idle drives is 0 and the maximum number is 2. Source: *Advanced Computer Architecture*, K. Hwang, 1993.

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

CHAPTER 5
CONCURRENCY: MUTUAL EXCLUSION AND SYNCHRONIZATION

ANSWERS TO QUESTIONS

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

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

5.3 The ability to enforce mutual exclusion.

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

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

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

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

assumptions are made about relative process speeds or number of processors. **6.** A process remains inside its critical section for a finite time only.

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

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

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

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

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

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

Answers to Problems

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

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

5.3 **a.** On casual inspection, it appears that *tally* will fall in the range $50 \leq tally \leq 100$ 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 $2 \leq tally \leq 100$.

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

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

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

5.5 Consider the case in which *turn* equals 0 and P(1) sets blocked[1] to true and then finds blocked[0] set to false. P(0) will then set blocked[0] to true, find *turn* = 0, and

enter its critical section. P(1) will then assign 1 to *turn* and will also enter its critical section.

5.6 **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.

5.7 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.

5.8 **a.** With this inequality, we can state that the condition in lines 4-5 is not satisfied and Pi can advance to stage j+1. Since the act of checking the condition is not a primitive, equation (1) may become untrue during the check: some Pr may set q[r] = j; several could do so, but as soon as the first of them also modifies turn[j], Pi can proceed (assuming it tries; this assumption is present throughout the proof, but will be kept tacit from now on). Moreover, once more than one additional process joins stage j, Pi can be overtaken.

b. Then either condition (1) holds, and therefore Pi precedes all other processes; or turn[j] ≠ i, with the implication the Pi is not the last, among all processes currently in lines 1-6 of their programs, to enter stage j. Regardless of how many processes modified turn[j] since Pi did, there is a lost one, Pr, for which the condition is its line 5 is true. This process makes the second line of the lemma hold (Pi is not alone at stage j). Note that it is possible that Pi proceeds to modify q[i] on the strength of its finding that condition (1) is true, and in the

meantime another process destroys this condition, thereby establishing the possibility of the second line of the lemma.

c. The claim is void for j = 1. For j = 2, we use Lemma 2: when there is a process at stage 2, another one (or more) will join it only when the joiner leaves behind a process in stage 1. That one, so long as it is alone there cannot advance, again by Lemma 2. Assume the Lemma holds for stage j-1; if there are two (or more) at stage j, consider the instant that the last of them joined in. At that time, there were (at least) two at stage j-1 (Lemma 2), and by the induction assumption, all preceding stages were occupied. By Lemma 2, none of these stages could have vacated since.

d. If stages 1 through j-1 contain at least one process, there are N – (j – 1) left at most for stage j. If any of those stages is "empty" Lemma 3 implies there is at most one process at stage j.

e. From the above, stage N-1 contains at most two processes. If there is only one there, and another is at its critical section, Lemma 2 says it cannot advance to enter its critical section. When there are two processes at stage N-1, there is no process left to be at stage N (critical section), and one of the two may enter its critical section. For the one remaining process, the condition in its line 5 holds. Hence there is mutual exclusion.

There is no deadlock: There is one process the precedes all others or is with company at the highest occupied stage, which it was not the last to enter, and for such a process the condition of its line 5 does not hold.

There is no starvation. If a process tries continually to advance, no other process can pass it; at worst, it entered stage 1 when all others were in their entry protocols; they may all enter stage N before it does—but no more.

Source: [HOFR90].

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

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

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

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

To prove the lemma, define the following times:

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

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

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

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

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

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

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

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

5.10 a. There is no variable which 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.

5.11 The following program is provided in [SILB98]:

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

The algorithm uses the common data structures

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

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

5.12 The two are equivalent. In the definition of Figure 5.8, 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.

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

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

5.14 The program is found in [RAYN86]:

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

5.15 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().

5.16

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

Both producer and consumer are blocked.

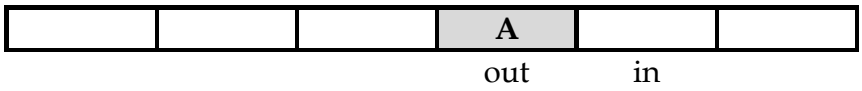
5.17 This solution is from [BEN82].

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

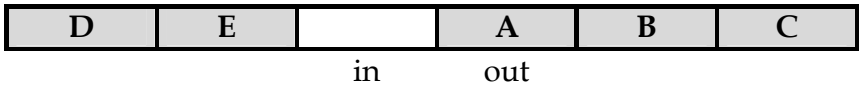
```
                signalB(s);
                waitB(delay);
                waitB(s)
            end;
        consume;
        signalB(s)
    forever
end;
begin (*main program*)
    n := 0;
    parbegin
        producer; consumer
    parend
end.
```

5.18 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.

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



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



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

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

5.20 The answer is no for both questions.

5.21 **a.** Change *receipt* to an array of semaphores all initialized to 0 and use *enqueue2*, *queue2*, and *dequeue2* to pass the customer numbers.

b. Change *leave_b_chair* to an array of semaphores all initialized to 0 and use enqueue1(custnr), queue1, and dequeue1(b_cust) to release the right barber.

Figure 1 shows the program with both of the above modifications. Note: The barbershop example in the book and Problems 5.21 and 5.22 are based on the following article, used with permission:

Hilzer, P. "Concurrency with Semaphores." *SIGSCE Bulletin*, September 1992.

```
program   barbershop2;
var   max_capacity: semaphore (:= 20);
      sofa: semaphore (:= 4);
      barber_chair, coord: semaphore (:= 3);
      mutex1, mutex2, mutex3: semaphore (:=1);
      cust_ready, payment: semaphore (:= 0);
      finished, leave_b_chair, receipt: array[1..50] of semaphore (:=0);
      count: integer;

procedure customer;
var  custnr:  integer;
begin
wait(max_capacity);
enter shop;
wait(mutex1);
count := count + 1;
custnr := count;
signal(mutex1);
wait(sofa);
sit on sofa;
wait(barber_chair);
get up from sofa;
signal(sofa);
sit in barber chair;
wait(mutex2);
enqueue1(custnr);
signal(cust_ready);
signal(mutex2);
wait(finished[custnr]);
signal(leave_b_chair[custnr]);
pay;
wait(mutex3);
enqueue2(custnr);
signal(payment);
signal(mutex3);
wait(receipt[custnr]);
exit shop;
signal(max_capacity)
end;

procedure barber;
var  b_cust:  integer;
begin
repeat
wait(cust_ready);
wait(mutex2);
dequeue1(b_cust);
signal(mutex2);
wait(coord);
cut hair;
signal(coord);
signal(finished[b_cust]);
wait(leave_b_chair[custnr]);
signal(barber_chair);
forever
end;

procedure cashier;
var  b_cust:  integer;
begin
repeat
wait(payment);
wait(mutex3);
dequeue2(c_cust);
signal(mutex3);
wait(coord);
accept pay;
signal(coord);
signal(receipt[c_cust]);
forever
end;

begin (*main program*)
    count := 0;
    parbegin
      customer; . . . 50 times; . . . customer;
      barber; barber; barber;
      cashier
    parend
end.
```

Figure 1 A Fair Barbershop, with Modifications

5.22

```
#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_wait = 0,              /* block early arrivals
                               back from islands */
sleigh = 0,                 /*all reindeer wait
                              around the sleigh */
done = 0,                   /* toys all delivered */
santa_signal = 0,           /* 1st 2 elves wait on
                          this outside Santa's shop
*/
santa = 0,                  /* Santa sleeps on this
                              blocked semaphore
*/
problem = 0,                /* wait 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
  wait (rmutex)
      rein_ct++
      if (rein_ct == REINDEER) {
         signal (rmutex)
         signal (santa)
      }
      else {
          signal (rmutex)
          wait (rein_wait)
      }
/* all reindeer waiting to be attached to sleigh
*/
  wait (sleigh)
   fly off to deliver toys
   wait (done)
   head back to the Pacific islands
} /* end "forever" loop */
```

```
                /* Elf Process */
for (;;) {
  wait (only_elves)          /* only 3 elves "in" */
     wait (emutex)
        elf_ct++
        if (elf_ct == ELVES) {
           signal (emutex)
           signal (santa)  /* 3rd elf wakes Santa
*/
        }
        else {
            signal (emutex)
            wait (santa _signal)  /* wait outside
                                  Santa's shop door */
        }
     wait (problem)
      ask question      /* Santa woke elf up */
      wait (elf_done)
  signal (only_elves)
} /* end "forever" loop */
                /* Santa Process */
for (;;) {
  wait (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) {
     wait (rmutex)
     rein_ct = 0           /* reset while blocked */
     signal (rmutex)
     for (i = 0; i < REINDEER – 1; i++)
         signal (rein_wait)
     for (i = 0; i < REINDEER; i++)
         signal (sleigh)
     deliver all the toys and return
     for (i = 0; i < REINDEER; i++)
         signal (done)
   }
    else {                    /* 3 elves have arrive */
     for (i = 0; i < ELVES – 1; i++)
         signal (santa_signal)
     wait (emutex)
          elf_ct = 0
     signal (emutex)
     for (i = 0; i < ELVES; i++)  {
         signal (problem)
         answer that question
         signal (elf_done)
     }
    }
  } /* end "forever" loop */
```

5.23 **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)
wait (mbuf)                     wait for message buffer available
wait (mutex)                    mutual exclusion on message queue
acquire free buffer slog
copy message to slot
link slot to other messages
signal (dest.sem)               wake destination process
signal (mutex)                  release mutual exclusion

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

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

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

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

CHAPTER 6
CONCURRENCY: DEADLOCK AND STARVATION

ANSWERS TO QUESTIONS

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

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

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

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

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

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

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

Answers to Problems

6.1 **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.

6.2 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.

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

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

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

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

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

6.6 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. Source: [BRIN73].

6.7 $i + o + p \leq max -$
$i + o \leq max - resp$
$i + p \leq max - reso$
$i \leq max - (reso + resp)$
Source: [BRIN73].

6.8 **a.** **1.** i ← i + 1
2. i ← i – 1; p ← p + 1
3. p ← p – 1; o ← o + 1
4. o ← o – 1
5. p ← p + 1
6. p ← p – 1

b. By examining the resource constraints listed in the solution to problem 6.7, we can conclude the following:
6. Procedure returns can take place immediately because they only release resources.
5. Procedure calls may exhaust the disk ($p = max - reso$) and lead to deadlock.
4. Output consumption can take place immediately after output becomes available.
3. Output production can be delayed temporarily until all previous output has been consumed and made at least *reso* pages available for further output.
2. Input consumption can take place immediately after input becomes available.
1. Input production can be delayed until all previous input and the corresponding output has been consumed. At this point, when $i = o = 0$, input can be produced provided the user processes have not exhausted the disk ($p < max - reso$).

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

6.9 **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	

6.10 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.

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

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

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

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

```
        region available do
        begin
            available (i) := available(i) – 1;
            available (succ) := available (succ) + 1;
        end
        j := (j+1) mod max;
    forever
end
```

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

region v **do** S

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

b. A deadlock is a situation in which:

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

because

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

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

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

Source: [BRIN73].

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

b. Using terminology similar to that used for the banker's algorithm, define claim[i] = total amount of resource units needed by process i; allocation[i] =

current number of resource units allocated to process i; and deficit[i] = amount of resource units still needed by i. Then we have:

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

In a deadlock situation, all resource units are reserved:

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

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

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

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

6.13 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 which 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.

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

6.15 a. When a philosopher finishes eating, he allows his left neighbor to proceed if possible, then permits his right neighbor to proceed. The solution uses an array, *state*, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move only into the eating state if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT.

b. This counterexample is due to [GING90]. Assume that philosophers P0, P1, and P3 are waiting with hunger while philosophers P2 and P4 dine at leisure. Now consider the following admittedly unlikely sequence of philosophers' completions of their suppers.

EATING	HUNGRY
4 2	0 1 3
2 0	1 3 4
3 0	1 2 4
0 2	1 3 4
4 2	0 1 3

Each line of this table is intended to indicate the philosophers that are presently eating and those that are in a state of hunger. The dining philosopher listed first on each line is the one who finishes his meal next. For example, from the initial configuration, philosopher P4 finishes eating first, which permits P0 to commence eating. Notice that the pattern folds in on itself and can repeat forever with the consequent starvation of philosopher P1.

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

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

As Pj continually holds one fork and waits for his right fork, Pj's right neighbor Pk never sets his left fork down and never completes a meal, i.e., Pk is also a lefty who starves. If Pk did not continually hold his left fork, Pj could eat; therefore Pk holds his left fork. Carrying the argument rightward around the

table shows that all philosophers are (starving) lefties: a contradiction. Starvation is thus precluded.

Source: [GING90].

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 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.

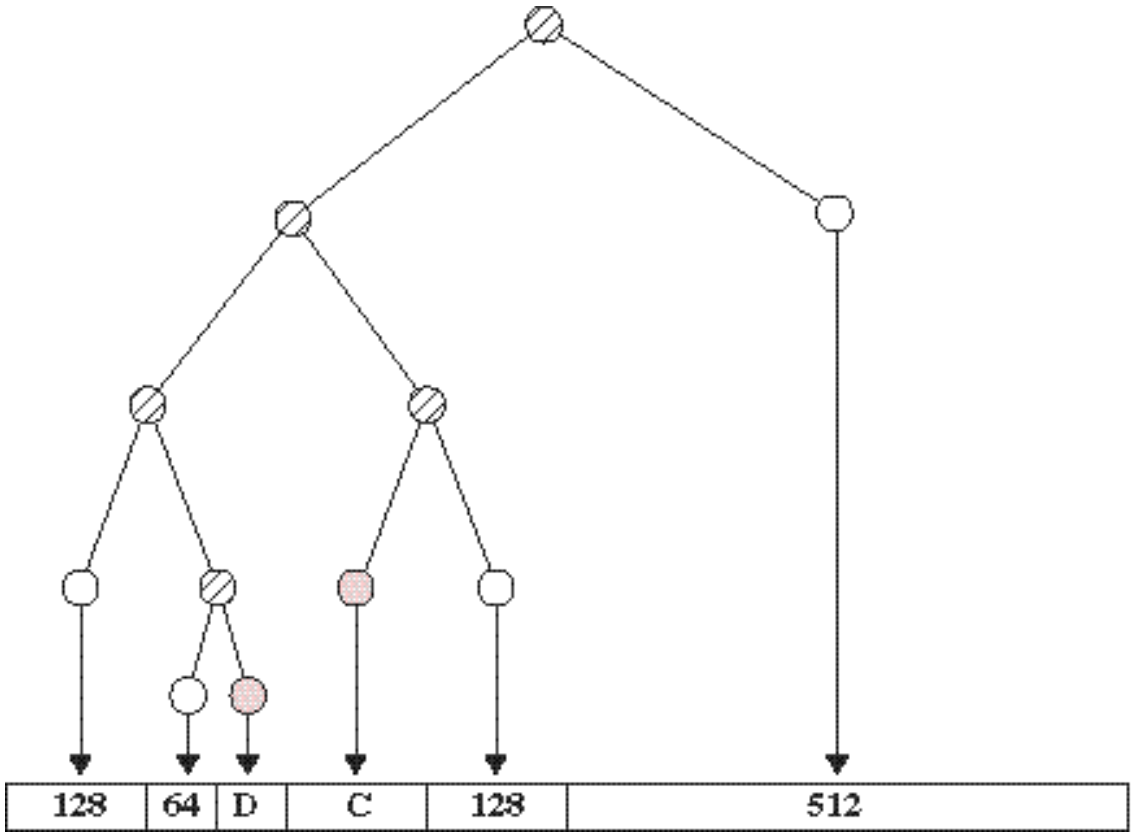
7.3 By problem 7.2, we know that the average number of holes is $s/2$, where s is the number of resident segments. Regardless of fit strategy, in equilibrium, the average search length is $s/4$.

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

7.5 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.



7.6 **a.** 011011110100
b. 011011100000

7.7 $$\text{buddy}_k = \begin{cases} x + 2^k & \text{if } x \bmod 2^{k+1} = 0 \\ x - 2^k & \text{if } x \bmod 2^{k+1} = 2^k \end{cases}$$

7.8 **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.

7.9 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.

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

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

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

CHAPTER 8
VIRTUAL MEMORY

ANSWERS TO QUESTIONS

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

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

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

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

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

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

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

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

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

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

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

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

Answers to Problems

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

b. (i) 1052 = 1024 + 28 maps to VPN 1 in PFN 7, (7 × 1024+28 = 7196)
(ii) 2221 = 2 × 1024 + 173 maps to VPN 2, page fault
(iii) 5499 = 5 × 1024 + 379 maps to VPN 5 in PFN 0, (0 × 1024+379 = 379)

8.2 **a.** PFN 3 since loaded longest ago at time 60

b. PFN 1 since referenced longest ago at time 160

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

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

e. There are 6 faults, indicated by *

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

8.3 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.

8.4 **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 4 4 4 4 4 4 4 4 4 4 4 2 2 2 2 2 2 2 2
- 0 0 0 0 0 6 6 6 6 2 2 2 2 2 5 5 5 5 5 5 5 5 5 5 4 4 4 4 4 4 4
- - 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 2 2 2 2 7 7 7 7 7 7 7 7 7 7 7
- - - - - 7 7 7 7 7 7 7 3 3 3 3 1 1 1 1 1 6 6 6 6 6 6 6 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 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 5 5 5 5 5 5 5 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 4 4 4 4 4 4 4- - - -
    - 7 7 7 7 7 7 7 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 3 3 3 3F F F
    F F     F   F F   F F F   F     F F       F     F   F
```

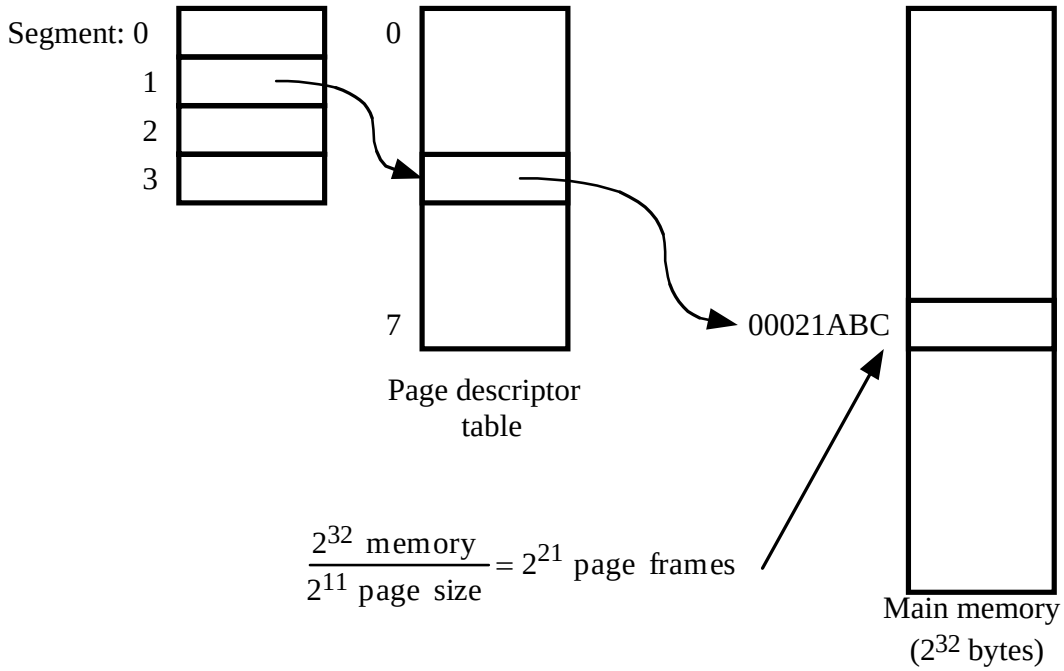
c. These two policies are equally effective for this particular page trace. Source: [HWAN93]**8.5** The principal advantage is a savings in physical memory space. This occurs for two reasons: (1) a user page table can be paged in to memory only when it is needed. (2) The operating system can allocate user page tables dynamically, creating one only when the process is created. Of course, there is a disadvantage: address translation requires extra work.**8.6** 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 R14007 (R1) ← (R1) + ONE Increment i 4008 branch 4002 6000-6999 storage for A 7000-7999 storage for B 8000-8999 storage for C 9000 storage for ONE 9001 storage for n The reference string generated by this loop is 494944(47484649444)^{1000} consisting of over 11,000 references, but involving only five distinct pages. Source: [MAEK87].**8.7** 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.**8.8** 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 $2^{10} * 2^{12} = 2^{22}$ 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^{62} 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.**8.9a.** 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.**8.10 a.** N **b.** P**8.11 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 analagous to the movement of the plow.

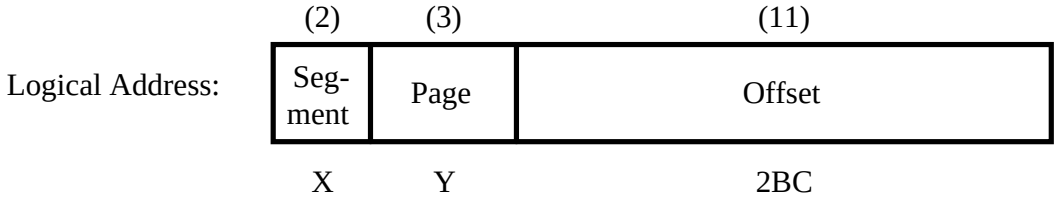
b. Note that the density of replaceable pages is highest immediately in front of the clock pointer, just as the density of snow is highest immediately in front of the plow. Thus, we can expect the CLOCK algorithm to be quite efficient in finding pages to replace. In fact, it can be shown that the depth 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 inuitive 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).**8.12** 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.**8.13** [PIZZ89] suggests the following strategy. Use a mechanism that adjusts the value of Q at each window time as a function of the actual page fault rate experienced during the window. The page fault rate is computed and compared with a system-wide value for "desirable" page fault rate for a job. The value of Q is adjusted upward (downward) whenever the actual page fault rate of a job is higher (lower) than the desirable value. Experimentation using this adjustment mechanism showed that execution of the test jobs with dynamic adjustment of Q consistently produced a lower number of page faults per execution and a decreased average resident set size than the execution with a constant value of Q (within a very broad range). The memory time product (MT) versus Q using the adjustment mechanism also produced a consistent and

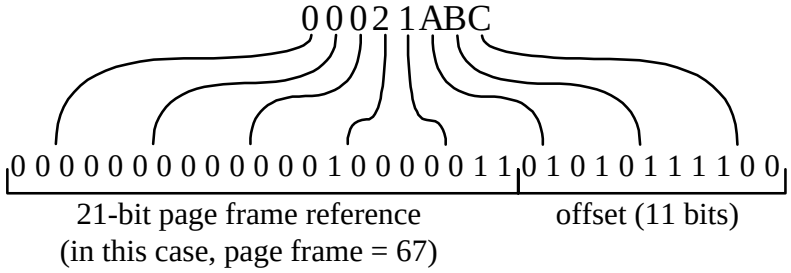
considerable improvement over the previous test results using a constant value of Q.

8.14 $\frac{2^{32}\ \text{memory}}{2^{11}\ \text{page size}} = 2^{21}\ \text{page frames}$



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





8.15 a.

page number (5)	offset (11)

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.

8.16 There are three cases to consider:

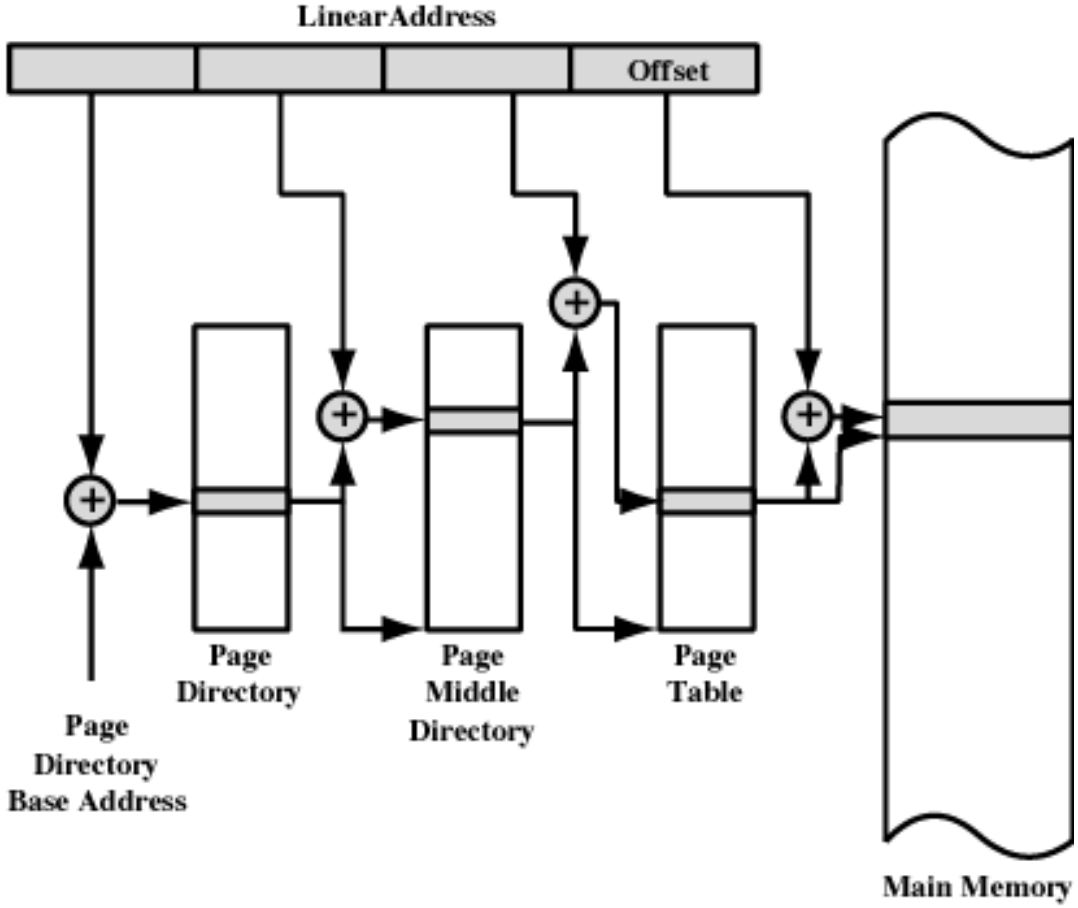
Location of referenced word	Probability	Total time for access in ns
In cache	0.9	20
Not in cache, but in main memory	(0.1)(0.6) = 0.06	60 + 20 = 80
Not in cache or main memory	(0.1)(0.4) = 0.04	12ms + 60 + 20 = 12000080

So the average access time would be:

$$\text{Avg} = (0.9)(20) + (0.06)(80) + (0.04)(12000080) = 480026 \text{ ns}$$

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

8.18 From [BECK98]:


Linear Address
Offset
Page Directory
Page Middle Directory
Page Table
Page Directory Base Address
Main Memory