

SOLUTIONS MANUAL

OPERATING SYSTEMS

NINTH EDITION

GLOBAL EDITION

CHAPTERS 1–9

WILLIAM STALLINGS

Do Not Post on Web

Copyright 2018: William Stallings

© 2018 by William Stallings

All rights reserved. No part of this document may be reproduced, in any form or by any means, or posted on the Internet, without permission in writing from the author. Selected solutions may be shared with students, provided that they are not available, unsecured, on the Web.

NOTICE

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

W.S.

TABLE OF CONTENTS

Chapter 1	Computer System Overview	5
Chapter 2	Operating System Overview	11
Chapter 3	Process Description and Control.....	15
Chapter 4	Threads	22
Chapter 5	Mutual Exclusion and Synchronization	29
Chapter 6	Deadlock and Starvation.....	49
Chapter 7	Memory Management.....	65
Chapter 8	Virtual Memory	71
Chapter 9	Uniprocessor Scheduling.....	82

CHAPTER 1 COMPUTER SYSTEM OVERVIEW

ANSWERS TO QUESTIONS

- 1.1** A processor, which controls the operation of the computer and performs its data processing functions ; a **main memory**, which stores both data and instructions; **I/O modules**, which move data between the computer and its external environment; and the system bus, which provides for communication among processors, main memory, and I/O modules.
- 1.2 User-visible registers:** Enable the machine- or assembly-language programmer to minimize main memory references by optimizing register use. For high-level languages, an optimizing compiler will attempt to make intelligent choices of which variables to assign to registers and which to main memory locations. Some high-level languages, such as C, allow the programmer to suggest to the compiler which variables should be held in registers. **Control and status registers:** Used by the processor to control the operation of the processor and by privileged, operating system routines to control the execution of programs.
- 1.3** These actions fall into four categories: **Processor-memory:** Data may be transferred from processor to memory or from memory to processor. **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module. **Data processing:** The processor may perform some arithmetic or logic operation on data. **Control:** An instruction may specify that the sequence of execution be altered.
- 1.4** An interrupt is a mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor.
- 1.5** Multiple interrupts may be serviced by assigning different priorities to interrupts arising from different sources. This enables a higher-priority interrupt to be serviced first when multiple requests arrive simultaneously; it also allows a higher-priority interrupt to preempt a lower-priority interrupt. For example, suppose a system has assigned a higher priority to a communication line and a lower priority to a magnetic disk. When two simultaneous requests arrive, the computer services the communication line. Similarly, if some disk operations are

ongoing when a request for the communication line arrives, the state of the disk is put in a stack and the communication line operations are catered to.

- 1.6** The characteristics observed while going up the memory hierarchy are **a.** increasing cost per bit, **b.** decreasing capacity, **c.** decreasing access time, and **d.** increasing frequency of access to the memory by the processor.
- 1.7** The main trade-offs for determining the cache size are the speed and the cost of the cache.
- 1.8** A multicore computer is a special case of a multiprocessor, in which all of the processors are on a single chip.
- 1.9 Spatial locality** refers to the tendency of execution to involve a number of memory locations that are clustered. **Temporal locality** refers to the tendency for a processor to access memory locations that have been used recently.
- 1.10 Spatial locality** is generally exploited by using larger cache blocks and by incorporating prefetching mechanisms (fetching items of anticipated use) into the cache control logic. **Temporal locality** is exploited by keeping recently used instruction and data values in cache memory and by exploiting a cache hierarchy.

ANSWERS TO PROBLEMS

- 1.1** Memory (contents in hex) : 300: 3007; 301: 4880; 302: 2881;
Step 1: 3007 → IR **Step 2:** 6 → AC
Step 3: 4880 → IR **Step 4:** 6 SUB 5 (0110 – 0101 = 0001) = 1 → AC
Step 5: 2881 → IR **Step 6:** AC → Memory Location 881
- 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. Number of bits for memory address is $64 - 4 \times 8 = 32$. Hence, the maximum addressable memory capacity is $2^{32} = 4$ GBytes.
 - b. The address buses must be ideally 64 bits so that the whole address can be transferred at once and decoded in the memory without requiring any additional memory control logic.
In case of 64-bit data buses, the whole instruction or operand can be transferred in one cycle. 32-bit data buses will require 2 fetch cycles and 16-bit data buses will require 4 fetch cycles. Hence, system speed will be reduced for lesser capacity buses.
 - c. If the IR is to contain only the opcode, it should contain 32 bits. However, if it contains the whole instruction, it should contain 64 bits.

- 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 = $1/16 \text{ MHz} = 62500 \text{ ps} = 62.5 \text{ ns}$
 Bus cycle = $4 \times 62.5 \text{ ns} = 250 \text{ ns}$
 4 bytes transferred every 250 ns; thus transfer rate = 16 MBytes/sec.

Doubling the frequency may mean adopting a new chip manufacturing technology (assuming each instructions will have the same number of clock cycles); doubling the external data bus means wider (maybe newer) on-chip data bus drivers/latches and modifications to the bus control logic. In the first case, the speed of the memory chips will also need to double (roughly) not to slow down the microprocessor; in the second case, the "word length" of the memory will have to double to be able to send/receive 64-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 OUTF 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 1350 characters per second, or one every 740 μs . The DMA therefore "steals" every 740th cycle. This slows down the processor

approximately $\frac{1}{740} \times 100\% = 0.14\%$.

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 Let the three memory hierarchies be M1, M2, and M3.

Let us define the following parameters:

T_s = average system access time

T_1 , T_2 , and T_3 = access time of M1, M2, and M3 respectively.

h_1 , h_2 = hit ratios of memories M1 and M2.

C_s = average cost per bit of combined memory.

C_1 , C_2 , and C_3 = cost per bit of M1, M2, and M3 respectively.

Extension of Equation (1.1) to 3-level memory hierarchy:

We can say that a word is found in M1 with a probability h_1 .

So the word is not found in M1 with a probability $(1 - h_1)$.

Or in other words, memory M2 is accessed with a probability $(1 - h_1)$.

As the hit ratio of M2 is h_2 , the word is found in M2 with a probability $(1 - h_1)h_2$.

So, memory M3 is accessed with a probability $(1 - h_1)(1 - h_2)$.

If we multiply the probabilities of access with the access times and sum up, we will get the average system access time.

Hence, $T_s = h_1 \cdot T_1 + (1 - h_1)h_2 \cdot T_2 + (1 - h_1)(1 - h_2) \cdot T_3$

Extension of Equation (1.2) to 3-level memory hierarchy:

Average cost = Total cost/Total size of memory

$$= \frac{C_1 S_1 + C_2 S_2 + C_3 S_3}{S_1 + S_2 + S_3}$$

1.12 a. Cost of 1 GByte of main memory = $C_m \times 8 \times 10^9 = \$64,000$

b. Cost of 1 MByte of cache memory = $C_c \times 8 \times 10^6 \text{ ¢} = \400

c. From Equation 1.1:

$$2 \times T_c = T_c + (1 - H)T_m$$

$$2 \times 120 = (1 - H) \times 1500$$

$$H = 1260/1500 = 0.84$$

1.13 There are three cases to consider:

Location of the preferred word	Probability	Total time for access in ns
In cache	0.85	25
Not in cache, but in main memory	$(1 - 0.85) \times 0.8 = 0.12$	$25 + 100 = 125$
Neither in cache nor in main memory	$(1 - 0.85) \times (1 - 0.8) = 0.03$	$10 \text{ ms} + 100 + 25 = 1,000,125$

$$\begin{aligned}\text{Average Access Time} &= 0.85 \times 25 + 0.12 \times 125 + 0.03 \times 1000125 \\ &= 21.25 + 15 + 30003.75 \\ &= 30040 \text{ ns}\end{aligned}$$

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

CHAPTER 2 OPERATING SYSTEM OVERVIEW

ANSWERS TO QUESTIONS

- 2.1 Convenience:** An operating system makes a computer more convenient to use. **Efficiency:** An operating system allows the computer system resources to be used in an efficient manner. **Ability to evolve:** An operating system should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.
- 2.2** The kernel is a portion of the operating system that includes the most heavily used portions of software. Generally, the kernel is maintained permanently in main memory. The kernel runs in a privileged mode and responds to calls from processes and interrupts from devices.
- 2.3** Multiprogramming is a mode of operation that provides for the interleaved execution of two or more computer programs by a single processor.
- 2.4** A process is a program in execution. A process is controlled and scheduled by the operating system.
- 2.5** The **execution context**, or **process state**, is the internal data by which the operating system is able to supervise and control the process. This internal information is separated from the process, because the operating system has information not permitted to the process. The context includes all of the information that the operating system needs to manage the process and that the processor needs to execute the process properly. The context includes the contents of the various processor registers, such as the program counter and data registers. It also includes information of use to the operating system, such as the priority of the process and whether the process is waiting for the completion of a particular I/O event.
- 2.6 Process isolation:** The operating system must prevent independent processes from interfering with each other's memory, both data and instructions. **Automatic allocation and management:** Programs

should be dynamically allocated across the memory hierarchy as required. Allocation should be transparent to the programmer. Thus, the programmer is relieved of concerns relating to memory limitations, and the operating system can achieve efficiency by assigning memory to jobs only as needed. **Support of modular programming:**

Programmers should be able to define program modules, and to create, destroy, and alter the size of modules dynamically. **Protection and access control:** Sharing of memory, at any level of the memory hierarchy, creates the potential for one program to address the memory space of another. This is desirable when sharing is needed by particular applications. At other times, it threatens the integrity of programs and even of the operating system itself. The operating system must allow portions of memory to be accessible in various ways by various users.

Long-term storage: Many application programs require means for storing information for extended periods of time, after the computer has been powered down.

2.7 Time slicing is a technique adopted in time-sharing systems to distribute CPU time to multiple users. In this technique, a system clock generates interrupts at a particular rate. At each clock interrupt, the OS regains control and can assign the processor to another user. Thus, at regular time intervals, the current user is preempted and another user is loaded in. To preserve the old user program status for later resumption, the old user programs and data are written out to disk before the new user programs and data are read in. Subsequently, the old user program code and data are restored in main memory when that program is given a turn in a subsequent time-slice.

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

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

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

2.11 A distributed operating system is a model where distributed applications are run on multiple computers linked by communications. It essentially comprises a set of software that operate over a collection of independent, networked, communicating, and physically separate computational nodes. Each individual node holds a specific software subset of the global aggregate operating system. It provides the illusion of a single main memory space and a single secondary memory space, plus other unified access facilities, such as a distributed file system.

ANSWERS TO PROBLEMS

- 2.1 a.** In uniprogramming system, the jobs JOB1, JOB2, JOB3 and JOB4 executes in a sequential manner. JOB1 completes in $8 + 8 = 16s$, then JOB2 completes in $4 + 14 = 18s$, then JOB3 completes in $6s$ and then JOB4 completes in $4 + 16 = 20s$; i.e. in total of $60s$.
- b.** The multiprogramming system with RR scheduling with $2s$ CPU time for each process can be illustrated as follows:

JOB1	JOB2	JOB3	JOB4	JOB1	JOB2	JOB3	JOB4	JOB1	JOB3	JOB1	
2	4	6	8	10	12	14	16	18	20	22	24 26 28 30

----- time -----

The quantities for both (a) and (b) are as tabulated below:

Computer System	Elapsed Time	Throughput	Processor Utilization
Uni-programming	60s	4 jobs/min	$22/60 = 36.67\%$
Multi-programming	22s	$10.9 \sim 10$ jobs/min	100%

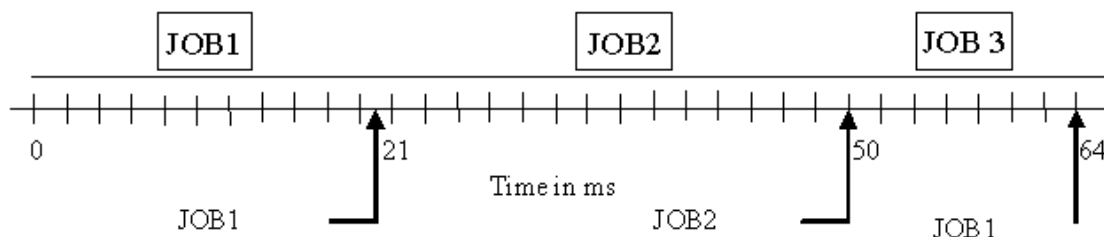
2.2 Time break up for the three jobs JOB1, JOB2 and JOB3 is:

JOB1 : 9ms I/O , 3ms CPU, 9ms I/O

JOB2 : 12ms I/O, 5ms CPU, 12ms I/O

JOB3 : 5ms I/O, 4ms CPU, 5ms I/O

Illustration of execution in uni-programming system:

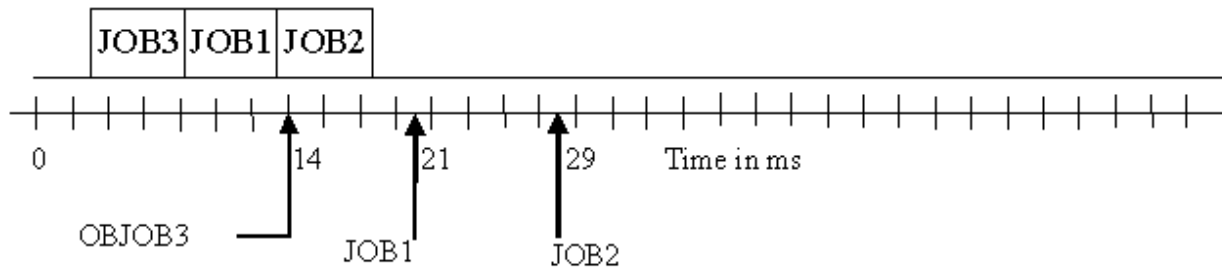


Hence, the total time required for completion of the jobs = $21 + 29 + 14 = 64\text{ms}$

Total execution time in CPU = $3 + 5 + 4 = 12\text{ ms}$

CPU utilization = $12 / 64 = 0.1875 = 18.75\%$

Illustration of execution in multiprogramming environment:



From the above diagram, it is evident that, total time = 29 ms

Total execution time in CPU = 12 ms

CPU utilization = $12 / 29 = 0.4138 = 41.38\%$

- 2.3** With time sharing, the concern is turnaround time. Time-slicing is preferred because it gives all processes access to the processor over a short period of time. In a batch system, the concern is with throughput, and the less context switching, the more processing time is available for the processes. Therefore, policies that minimize context switching are favored.
- 2.4** The two modes of operation in an operating system are the user mode and the kernel mode. At system boot time, the hardware starts in kernel mode and loads the operating system. The user application then starts in user mode. When the interrupt occurs, the operating system switches to kernel mode to service the interrupt.
- 2.5** The system operator can review this quantity to determine the degree of "stress" on the system. By reducing the number of active jobs allowed on the system, this average can be kept high. A typical guideline is that this average should be kept above 2 minutes. This may seem like a lot, but it isn't.
- 2.6 a.** If a conservative policy is used, at most $24/6 = 4$ processes can be active simultaneously. Because two of the drives allocated to each process can be idle most of the time, at most $4 \times 2 = 8$ 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 four tape drives. The remaining two drives will be allocated on demand. In this policy, at most $\lfloor 24/4 \rfloor = 6$ processes can be active simultaneously. The minimum number of idle drives is 0 and the maximum number is also 0.

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** A process is an instance of a program being executed. A program is essentially a set of instructions written in any high-level language; it is generally in the form of an executable file and is stored in a secondary storage device. When this executable file is loaded on to the main memory in order that the instructions can actually be executed, it becomes a process. Apart from the program code, a process includes the current activity (represented by a value in program counter), registers, and process stacks. A program is a passive entity whereas a process is an active entity.
- 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** When an OS creates a process at the explicit request of another process, this action is called process spawning. When one process spawns another, the former is called the parent process and the one spawned is called the child. Typically, these processes need to communicate and cooperate with each other.
- 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

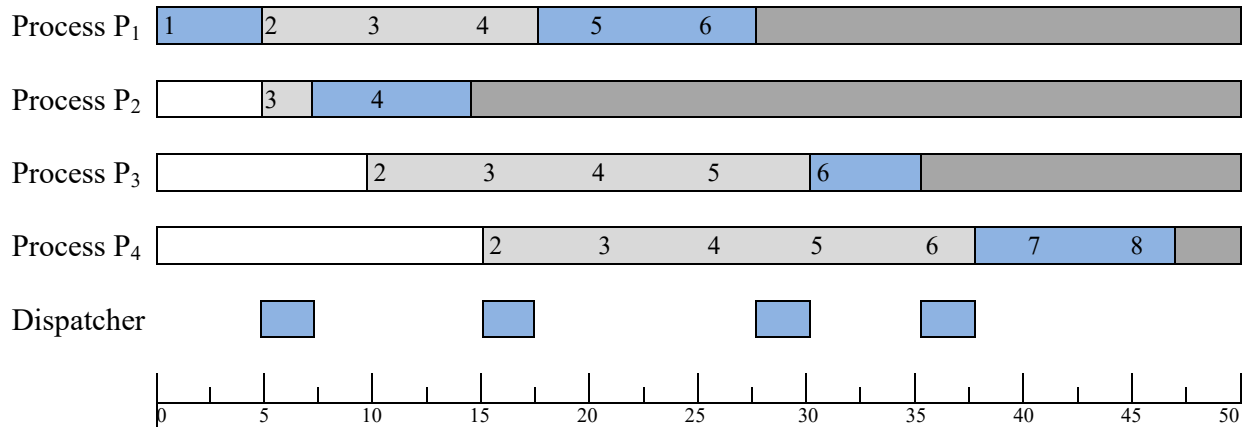
main memory (suspended or not). To accommodate this 2×2 combination, we need two Ready states and two Blocked states.

- 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** The elements of a process image are: **User Data:** This is the modifiable part of the user space. It generally includes a user stack area, program data, and programs that can be modified. **User Program:** This comprises the program to be executed. **Stack:** They are used to store parameters and calling addresses for procedure and system calls. Each process has a LIFO stack associated with it. **Process Control Block:** It contains many pieces of information associated with a specific process, like the process identifier, process 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 Timing diagram for the scenario:



The numbers denote the priorities of the processes at the time instance.

Running Ready Blocked / Completed

Turnaround time for P₁ = 27.5 ms

Turnaround time for P₂ = (15 - 5) = 10 ms

Turnaround time for P₃ = (35 - 10) = 25 ms

Turnaround time for P₄ = (47.5 - 15) = 32.5 ms

3.2 Let the starting address of the dispatcher be 200. Traces of the interleaved processes will be as follows:

1. 4050	18. 203	35. 4059	42. 200
2. 4051	19. 5000	---Time Slice Expires	43. 201
3. 4052	20. 5001	36. 200	44. 202
4. 4053	21. 5002	37. 201	45. 203
5. 4054	22. 5003	38. 202	46. 4060
---Time Slice Expires	23. 5004	39. 203	---P1 Exits
6. 200	---Time Slice Expires	40. 3025	47. 200
7. 201	24. 200	41. 3026	48. 201
8. 202	25. 201	--- I/O Request	49. 202
9. 203	26. 202	42. 200	50. 203
10. 3200	27. 203	43. 201	51. 5010
11. 3201	28. 6700	44. 202	---P3 Exits

12. 3202	29. 6701	45. 203
13. 3203	30. 6702	46. 5005
14. 3204	--- I/O Request	47. 5006
---Time Slice	31. 4055	48. 5007
Expires		
15. 200	32. 4056	49. 5008
16. 201	33. 4057	50. 5009
17. 202	34. 4058	---Time Slice
		Expires

3.3 a. New → Ready or Ready/Suspend: covered in text

Ready → Running or Ready/Suspend: covered in text

Ready/Suspend → Ready: covered in text

Blocked → Ready or Blocked/Suspend: covered in text

Blocked/Suspend → Ready /Suspend or Blocked: covered in text

Running → Ready, Ready/Suspend, or Blocked: covered in text

Any State → Exit: covered in text

b. New → Blocked, Blocked/Suspend, or Running: A newly created process remains in the new state until the processor is ready to take on an additional process, at which time it goes to one of the Ready states.

Ready → Blocked or Blocked/Suspend: Typically, a process that is ready cannot subsequently be blocked until it has run. Some systems may allow the OS to block a process that is currently ready, perhaps to free up resources committed to the ready process.

Ready/Suspend → Blocked or Blocked/Suspend: Same reasoning as preceding entry.

Ready/Suspend → Running: The OS first brings the process into memory, which puts it into the Ready state.

Blocked → Ready /Suspend: this transition would be done in 2 stages. A blocked process cannot at the same time be made ready and suspended, because these transitions are triggered by two different causes.

Blocked → Running: When a process is unblocked, it is put into the Ready state. The dispatcher will only choose a process from the Ready state to run

Blocked/Suspend → Ready: same reasoning as Blocked → Ready /Suspend

Blocked/Suspend → Running: same reasoning as Blocked → Running

Running → Blocked/Suspend: this transition would be done in 2 stages

Exit → Any State: Can't turn back the clock

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

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

- 3.6 a.** A separate queue is associated with each wait state. The differentiation of waiting processes into queues reduces the work needed to locate a waiting process when an event occurs that affects it. For example, when a page fault completes, the scheduler know that the waiting process can be found on the Page Fault Wait queue.
- b.** In each case, it would be less efficient to allow the process to be swapped out while in this state. For example, on a page fault wait, it makes no sense to swap out a process when we are waiting to bring in another page so that it can execute.
- c.** The state transition diagram can be derived from the following state transition table:

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

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

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

3.8 With $j < i$, a process running in D_i is prevented from accessing objects in D_j . Thus, if D_j contains information that is more privileged or is to be

kept more secure than information in D_i , this restriction is appropriate. However, this security policy can be circumvented in the following way. A process running in D_j could read data in D_j and then copy that data into D_i . Subsequently, a process running in D_i could access the information.

- 3.9 a.** An application may be processing data received from another process and storing the results on disk. If there is data waiting to be taken from the other process, the application may proceed to get that data and process it. If a previous disk write has completed and there is processed data to write out, the application may proceed to write to disk. There may be a point where the process is waiting both for additional data from the input process and for disk availability.
- b.** There are several ways that could be handled. A special type of either/or queue could be used. Or the process could be put in two separate queues. In either case, the operating system would have to handle the details of alerting the process to the occurrence of both events, one after the other.

3.10 The purpose of system call `fork()` is to create processes. It takes no arguments and returns a process ID. The new process becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the `fork()` system call.

The following C routine creates a child process in UNIX:

```
#include <stdio.h>
#include <sys/types.h> /* pid_t */
#include <unistd.h> /* _exit, fork */
#include <stdlib.h> /* exit */
#include <errno.h> /* errno */

int main(void)
{
    int pid_t, pid;

    /* Output from both the child and the parent process will be
       written to the standard output, as they both run at the
       same time. */
    pid = fork();
    if (pid == -1)
    {
        /* Error: When fork() returns -1, an error happened */
        fprintf(stderr, "Can't fork, error %d\n", errno);
        exit(EXIT_FAILURE);
    }

    if (pid == 0)
    {
```

```

        /* Child process: When fork() returns 0, we are in the
        child process. */
        printf("child process");
        _exit(0);
    }
    else
    {

        /* When fork() returns a positive number, we are in the
        parent process (the fork return value is the PID of the newly
        created child process) */
        printf("Parent Process");
        printf("Child Process id = %d", pid);
        exit(0);
    }
    return 0;
}

```

3.11 Process 0 is the first process that is created when the system boots. It is automatically created from a predefined data structure that is loaded at boot time. Process 1 is spawned by Process 0. It is referred as the init process. This init process spawns other processes that may in turn spawn child processes in a recursive manner. All processes in the system have init as an ancestor. Process 1 is also responsible for creating a user process for any new user who logs in to the system.

The `ps` command is used to get a list of the active processes in the system. To get information about the running processes, `ps -el` or `ps -ef` are used.

3.12

```

0
<child pid>
or
<child pid>
0

```

CHAPTER 4 THREADS

ANSWERS TO QUESTIONS

- 4.1** This will differ from system to system, but in general, resources are owned by the process and each thread has its own execution state. A few general comments about each category in Table 3.5:
Identification: the process must be identified but each thread within the process must have its own ID. **Processor State Information:** these are generally process-related. **Process control information:** scheduling and state information would mostly be at the thread level; data structuring could appear at both levels; interprocess communication and interthread communication may both be supported; privileges may be at both levels; memory management would generally be at the process level; and resource info would generally be at the process level.
- 4.2** Less state information is involved.
- 4.3** Resource ownership and scheduling/execution.
- 4.4** Foreground/background work; asynchronous processing; speedup of execution by parallel processing of data; modular program structure.
- 4.5** The notable differences between a process and a thread can be tabulated as follows:

	Process	Thread
1.	Generally, more than one process cannot share the same memory. Sharing memory among processes requires additional memory-management schemes.	Threads of the same process can share the same memory unless they are specially allotted separate memory locations.
2.	Process creation, process execution, and process switch are time consuming.	Thread creation, thread execution, and thread switch are much faster in comparison.
3.	Processes are generally loosely coupled and so a lesser amount of resource sharing is possible.	As the threads of a process are tightly coupled; a greater amount of resource sharing is possible.
4.	Communication between processes is difficult and requires system calls.	Communication between threads is much easier and more efficient.

- 4.6** Multi-threading refers to an application with multiple threads running within a process, while multi-processing refers to an application organized across multiple OS-level processes. The advantages of multi-threading over multi-processing are:
- a.** As multi-threading is a more light-weight form of concurrency than multi-processing, the costs associated with context switching, communication, data sharing, and synchronization are lower.
 - b.** On single-processor machines, multithreading is particularly advantageous when the jobs are varied in terms of time and resource requirements and when some jobs require concurrent processing.
 - c.** On multiprocessor systems, multithreading takes advantage of the additional hardware, thus resulting in better overall performance.

4.7 Advantages: **1.** Since the operating system has knowledge of all the threads in the system, if a thread of a process gets blocked, the OS may choose to run the next one either from the same process or from a different one. **2.** The kernel can simultaneously schedule multiple threads from the same process on multiple processors. **3.** Kernel routines can themselves be multithreaded. **Disadvantages:** **1.** The transfer of control from one thread to another within the same process requires a mode switch to the kernel and so is time consuming. **2.** Creation and destruction of threads in the kernel is costlier.

- 4.8** The Clouds operating system is an operating system that is designed to support a variety of distributed-object programming models. It is based on an object-thread paradigm. Cloud objects are abstractions of protected, passive storage and threads are the activity abstractions that exist independent of objects. Upon creation, a thread starts executing in a process by invoking an entry point to a procedure in that process. Threads may move from one address space to another address space, an activity that is called "thread migration." During movement, a thread carries certain information like the controlling terminal, global parameters, and scheduling guidance.

ANSWERS TO PROBLEMS

- 4.1** Some examples of programs where performance is not improved by multithreading are:
- a.** Sequential programs where no two portions can be executed in parallel.
 - b.** A shell script of UNIX that has to monitor its own working space (such as files, environment variables, and current working directory).
 - c.** Any computation-centric system that has a single data stream and possible multiple instruction streams where the output of one instruction may be required as the input or control variable of a subsequent instruction.

Some examples of programs in which multi-threaded solutions provide improved performances (when compared to single-threaded solutions) are:

- a.** A parallelized application, like matrix multiplication or vector processing, where the different parts (of the matrix or vector) can be worked on in parallel on separate threads.
 - b.** A Web server that spawns a separate thread for each request and services each thread individually.
 - c.** An interactive GUI program where separate threads may be created for separate activities (like the user input, the program execution in the processor, the output, and the performance analysis).
 - d.** All systems that have a single instruction stream and multiple data streams.
- 4.2 The implementation without using threads:** The main program and the two sub-routines can be considered as three different processes and can be executed simultaneously. However, this incurs the overhead of creating as well as executing different processes.

The implementation using threads: Two threads can be created for this purpose. The main program and one of the sub-routines can be considered as a single activity and, hence, can be implemented as a thread. The other sub-routine can be executed as another thread. Both threads may use the same address space.

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

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

4.5 Portion of code that is inherently serial = 20% = 0.2

Portion that is parallelizable = 1 – 0.2 = 0.8

Number of processors, $N = 4$

According to Amdahl's law,

$$\begin{aligned}\text{Speedup} &= \frac{1}{(1-f) + \frac{f}{N}} \\ &= \frac{1}{(1-0.8) + \frac{0.8}{4}} = 2.5\end{aligned}$$

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 The two possible outputs are:

Run 1:

```
Thread X : x = 10
Thread X : x = 11
Thread Y : y = 100
Main Thread Exiting.....
Thread Y : y = 99
Thread Y : y = 98
Thread X : x = 12
Thread Y : y = 97
Thread Y : y = 96
Thread X : x = 13
Thread X : x = 14
Thread Y : y = 95
Thread Y : y = 94
Thread Y : y = 93
Thread Y : y = 92
Thread Y : y = 91
```

Run 2:

```
Thread X : x = 10
Thread Y : y = 100
Thread Y : y = 99
Thread X : x = 11
Thread Y : y = 98
Thread X : x = 12
Thread Y : y = 97
Thread Y : y = 96
Thread Y : y = 95
Main Thread Exiting.....
Thread Y : y = 94
Thread Y : y = 93
Thread X : x = 13
Thread X : x = 14
Thread Y : y = 92
Thread Y : y = 93
```

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

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

4.10 Let the two threads be `th1` and `th2`. The thread `th2` is assigned the maximum priority whereas `th1` is assigned a normal priority. So, even if thread `th1` is set to start earlier, `th2` preempts `th1` and runs to completion. It is only after `th2` has completed its execution that `th1` resumes its activity.

The following Java program implements the scenario:

```
class X extends Thread {
    public void run() {
        for(int i=0;i<10;i++)
            System.out.println("\nThread X : i = " + i);
        System.out.println("\nExiting Thread X");
    }
}

class Y extends Thread {
    public void run() {
        for(int j=100;j<110;j++)
            System.out.println("\nThread Y : j = " + j );
        System.out.println("\nExiting Thread Y");
    }
}

class Runthreads {
    public static void main( String args[]) {
        X th1 = new X();
        Y th2 = new Y();
        th2.setPriority(Thread.MAX_PRIORITY);
        th1.start();
        th2.start();
    }
}
```

- 4.11 a.** Some programs have logical parallelism that can be exploited to simplify and structure the code but do not need hardware parallelism. For example, an application that employs multiple windows, only one of which is active at a time, could with advantage be implemented as a set of ULTs on a single LWP. The advantage of restricting such applications to ULTs is efficiency. ULTs may be created, destroyed, blocked, activated, and so on. without involving the kernel. If each ULT were known to the kernel, the kernel would have to allocate kernel data structures for each one and perform thread switching. Kernel-level thread switching is more expensive than user-level thread switching.
- b.** The execution of user-level threads is managed by the threads library whereas the LWP is managed by the kernel.
- c.** An unbound thread can be in one of four states: runnable, active, sleeping, or stopped. These states are managed by the threads library. A ULT in the active state is currently assigned to a LWP and executes while the underlying kernel thread executes. We can view the LWP state diagram as a detailed description of the ULT active state, because an thread only has an LWP assigned to it when it is

in the Active state. The LWP state diagram is reasonably self-explanatory. An active thread is only executing when its LWP is in the Running state. When an active thread executes a blocking system call, the LWP enters the Blocked state. However, the ULT remains bound to that LWP and, as far as the threads library is concerned, that ULT remains active.

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

CHAPTER 5 MUTUAL EXCLUSION AND SYNCHRONIZATION

ANSWERS TO QUESTIONS

- 5.1** Communication among processes, sharing of and competing for resources, synchronization of the activities of multiple processes, and allocation of processor time to processes.
- 5.2** Multiple applications, structured applications, operating-system structure.
- 5.3** A race condition occurs when multiple processes or threads read and write data items so that the final outcome depends on the order of execution of instructions in the multiple processes. For example, suppose there are n number of processes that share a global variable c that has an initial value $c = 1$. At some point in the execution, a process, say P_i , updates $c = 2$; at some other point in the execution, another process P_j increments c ; and at some other point, another process P_k updates $c = 5$. The final value of c will depend on the order in which these processes execute their tasks. If the order is $\langle P_i, P_j, P_k \rangle$ then the value of c will be 5 and the updates by P_i and P_j will be lost. On the other hand, if the order is $\langle P_k, P_i, P_j \rangle$, then the value of c will be 3, and the update by P_k will be lost. In the latter case, the update by P_j depends on the previous update by P_i .
- 5.4 Processes unaware of each other:** These are independent processes that are not intended to work together. **Processes indirectly aware of each other:** These are processes that are not necessarily aware of each other by their respective process IDs, but that share access to some object, such as an I/O buffer. **Processes directly aware of each other:** These are processes that are able to communicate with each other by process ID and which are designed to work jointly on some activity.
- 5.5 Competing processes** need access to the same resource at the same time, such as a disk, file, or printer. **Cooperating processes** either share access to a common object, such as a memory buffer or are able

to communicate with each other, and cooperate in the performance of some application or activity.

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

5.7 Starvation refers to a situation where a runnable process is infinitely overlooked by the scheduler for performance of a certain activity. In the context of concurrency control using mutual exclusion, this situation occurs when many processes are contending to enter in the critical section and a process is indefinitely denied access. Although this process is ready to execute in its critical section, it is never chosen and as an outcome never runs into completion.

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 mutex is a mutual exclusion object that is created so that multiple processes can take turns in accessing a shared variable or resource. A binary semaphore is a synchronization variable used for signalling among processes; it can take on only two values: 0 and 1. The mutex and the binary semaphore are used for similar purposes.

The key difference between the two is that the process that locks a mutex must be the one to unlock it; in a semaphore implementation, however, if the operation `wait(s)` is executed by one process, the operation `signal(s)` can be executed by either that process or by any another process

5.11 A monitor is a collection of procedures, variables, and data structures which are grouped together in a module. The characteristics that mark it as a high-level synchronization tool and give it an edge over primitive tools are:

- a.** As the variables and procedures are encapsulated, local data variables are accessible only by the monitor's procedures and not by any external procedure, thus eliminating the erroneous updating of variables.
- b.** A process enters the monitor by invoking one of its procedures. Therefore, not all processes can use the monitor, and those that can must do so only in the manner defined in its construct.
- c.** Only one process may execute in the monitor at a time; all other processes that invoke the monitor are blocked, and wait for the monitor to become available.
- d.** Monitors can control the time of accessing a variable by inserting appropriate functions.

5.12 In direct addressing, each communicating process has to name the recipient or the sender of the message explicitly. In this scheme `send()` and `receive()` primitives are defined such that the identities of the communicating processes are included in it as:

- `send(x, msg)` – sends message to process `x`
- `receive(y, msg)` – receives message from process `y`

Thus, a link is automatically established between exactly one pair of communicating processes, `x` and `y` in this case. Here, each process knows the ID of the other and each pair of processes can have only a single link. The receiving process may or may not designate the sending process.

On the other hand, in indirect addressing, messages are sent and received from ports or mailboxes (which are shared data-structures consisting of queues that can temporarily hold messages). In this scheme `send()` and `receive()` primitives are defined as follows:

- `send(M, msg)` – sends message to mailbox `M`
- `receive(M, msg)` – receives message from mailbox `M`

Thus, a link can be established between more than one processes if they have a shared mailbox. Between each pair of communicating processes, more than one link may exist, where each link corresponds to one mailbox.

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

ANSWERS TO PROBLEMS

- 5.1 a.** Process P1 will only enter its critical section if $\text{flag}[0] = \text{false}$. Only P1 may modify $\text{flag}[1]$, and P1 tests $\text{flag}[0]$ only when $\text{flag}[1] = \text{true}$. It follows that when P1 enters its critical section we have:

$$(\text{flag}[1] \text{ and } (\text{not } \text{flag}[0])) = \text{true}$$

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

$$(\text{flag}[1] \text{ and } (\text{not } \text{flag}[0])) = \text{true}$$

- b. Case 1:** A single process P(i) is attempting to enter its critical section. It will find $\text{flag}[1-i]$ set to false, and enters the section without difficulty.

Case 2: Both processes are attempting to enter their critical section, and $\text{turn} = 0$ (a similar reasoning applies to the case of $\text{turn} = 1$). 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: $\text{flag}[0] = \text{false}$. P1 finds $\text{flag}[0] = 0$, and can enter its critical section immediately.

Subcase 2b: $\text{flag}[0] = \text{true}$. Since $\text{turn} = 0$, P0 will wait in its external loop for $\text{flag}[1]$ to be set to false (without modifying the value of $\text{flag}[0]$). Meanwhile, P1 sets $\text{flag}[1]$ to false (and will wait in its internal loop because $\text{turn} = 0$). At that point, P0 will enter the critical section.

Thus, if both processes are attempting to enter their critical section, there is no deadlock.

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

- 5.3 a.** There is no variable that is both read and written by more than one process (like the variable turn in Dekker's algorithm). Therefore, the bakery algorithm does not require atomic load and store to the same global variable.

- b.** Because of the use of flag to control the reading of turn , we again do not require atomic load and store to the same global variable.

- 5.4 a.** The requirements that should be met in order to provide support for mutual exclusion are: **1.** Only one process should be executing in its critical section at a time among many contending processes, i.e., mutual exclusion is enforced. **2.** A process executing in its non-critical section should not interfere with any other process. **3.** No deadlocks or livelocks should exist. **4.** A process should be allowed to enter its critical section within a finite amount of time (or, in other

words, it should satisfy the bounded-waiting condition). **5.** A process cannot execute in its critical section for an indefinite amount of time. **6.** When no process is in its critical section, any process that wishes to enter the critical section should be allowed to do so immediately. **7.** No assumptions should be made about the relative speeds or the number of processors.

When interrupt disabling is used, mutual exclusion is guaranteed since a critical section cannot be interrupted. Also, the question of deadlocks does not arise. However, the condition of bounded-waiting is not met and so starvation may occur. Further, the time that a process stays in the critical section cannot be made finite.

- b.** Two major problems are associated with the interrupt-disabling mechanism: **1.** Since all interrupts are disabled before entry into the critical section, the ability of the processor to interleave processes is greatly restricted. **2.** This fails to work in a multiprocessor environment since the scope of an interrupt capability is within one processor only. Thus, mutual exclusion cannot be guaranteed.

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

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

5.6 a. The initial values of x and y are 2 and 3 respectively.

After executing P1, $x = 6$, $y = 4$

After executing P2 after P1, $x = 7$, $y = 28$

Hence, at the end of a serial schedule, the values of x and y are 7 and 28 respectively.

- b. An interleaved schedule that gives same output as the serial schedule:

Process	Statement	Contents of Registers				Value of x	Value of y
		R1	R2	R3	R4		
P1	LOAD R1, X	2	-	-	-	2	3
P1	LOAD R2, Y	2	3	-	-	2	3
P1	MUL R1, R2	6	3	-	-	2	3
P1	STORE X, R1	6	3	-	-	6	3
P2	LOAD R3, X	6	3	6	-	6	3
P2	INC R3	6	3	7	-	6	3
P1	INC R2	6	4	7	-	6	3
P1	STORE Y, R2	6	4	7	-	6	4
P2	LOAD R4, Y	6	4	7	4	6	4
P2	MUL R4, R3	6	4	7	28	6	4
P2	STORE X, R3	6	4	7	28	7	4
P2	STORE Y, R4	6	4	7	28	7	28

After execution of the above schedule, the values of x and y are 7 and 28 respectively, which is same as the output after serial execution of P1 and P2.

- c. An interleaved schedule that gives output different from the serial schedule:

Process	Statement	Contents of Registers				Value of x	Value of y
		R1	R2	R3	R4		
P1	LOAD R1, X	2	-	-	-	2	3
P1	LOAD R2, Y	2	3	-	-	2	3
P1	MUL R1, R2	6	3	-	-	2	3
P2	LOAD R3, X	6	3	2	-	2	3
P2	INC R3	6	3	3	-	2	3
P2	LOAD R4, Y	6	3	3	3	2	3
P1	STORE X, R1	6	3	3	3	6	3
P1	INC R2	6	4	3	3	6	3
P1	STORE Y, R2	6	4	3	3	6	4
P2	MUL R4, R3	6	4	9	3	6	4
P2	STORE X, R3	6	4	9	3	9	4
P2	STORE Y, R4	6	4	9	3	9	3

The above schedule gives the output $x = 9$ and $y = 3$, which is not equivalent to any serial schedule.

- 5.7 a.** On casual inspection, it appears that t_{ally} will fall in the range $50 \leq t_{ally} \leq 100$ 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 $2 \leq \text{tally} \leq 100$.

- b. For the generalized case of N processes, the range of final values is $2 \leq \text{tally} \leq (N \times 50)$, 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.

5.8 Spinlocks refer to a mutual exclusion mechanism in which a process executes in an infinite loop while waiting for the value of a shared variable `lock` to indicate availability. In this method, a Boolean variable `lock` is defined whose value can be either `TRUE` or `FALSE`. A contending process continually checks for the value of `lock`. If the value of `lock` is `TRUE`, this implies that some other process is executing in its critical section and so this process waits. If the value of `lock` is `FALSE`, then the process sets `lock` as `TRUE` and enters its critical section. After the process has completed its execution in the critical section, it resets the value of `lock` to `FALSE`. For implementation, a `test_and_set_lock()` instruction can be defined that sets the value of `lock` to `TRUE` and returns the original value of `lock` as follows:

```

boolean test_and_set_lock(boolean *lockval)
{
    boolean retval= *lockval;
    if (retval == FALSE) *lockval = TRUE;
    return retval;
}

```

This instruction is continually executed by a contending process till it can gain access to its critical section. This can be implemented as follows:

```

/* program mutualexclusion */
const int n = /* number of processes */
boolean lock;

void P(int i)
{
    while(true) {
        while(test_and_set_lock(&lock)==TRUE)
            /* do nothing */;
        /* critical section */
        lock = FALSE;
        /* remainder */
    }
}

void main()
{
    lock = FALSE;
    parbegin(P(1), P(2), . . . , P(n));
}

```

5.9 Consider the case in which turn equals 0 and P(1) sets blocked[1] to true and then finds blocked[0] set to false. P(0) will then set blocked[0] to true, find turn = 0, and enter its critical section. P(1) will then assign 1 to turn and will also enter its critical section.

- 5.10 a.** When a process wishes to enter its critical section, it is assigned a ticket number. The ticket number assigned is calculated by adding one to the largest of the ticket numbers currently held by the processes waiting to enter their critical section and the process already in its critical section. The process with the smallest ticket number has the highest precedence for entering its critical section. In case more than one process receives the same ticket number, the process with the smallest numerical name enters its critical section. When a process exits its critical section, it resets its ticket number to zero.
- b.** If each process is assigned a unique process number, then there is a unique, strict ordering of processes at all times. Therefore, deadlock cannot occur.
- c.** To demonstrate mutual exclusion, we first need to prove the following lemma: if P_i is in its critical section, and P_k has calculated

its number[k] and is attempting to enter its critical section, then the following relationship holds:

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

To prove the lemma, define the following times:

- T_{w1} P_i 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} P_i begins its final execution, for $j = k$, of the second **while** loop. We therefore have $T_{w1} < T_{w2}$.
- T_{k1} P_k enters the beginning of the **repeat** loop.
- T_{k2} P_k finishes calculating number[k].
- T_{k3} P_k sets choosing[k] to false. We have $T_{k1} < T_{k2} < T_{k3}$.

Since at T_{w1} , 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 P_i was assigned its number prior to P_k ; 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} , P_i 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 $(\text{number}[i], i) < (\text{number}[k], k)$, which completes the proof of the lemma.

It is now easy to show the mutual exclusion is enforced. Assume that P_i is in its critical section and P_k is attempting to enter its critical section. P_k will be unable to enter its critical section, as it will find number[i] $\neq 0$ and $(\text{number}[i], i) < (\text{number}[k], k)$.

5.11 Suppose we have two processes just beginning; call them p_0 and p_1 . Both reach line 3 at the same time. Now, we'll assume both read number[0] and number[1] before either addition takes place. Let p_1 complete the line, assigning 1 to number[1], but p_0 block before the assignment. Then p_1 gets through the while loop at line 5 and enters the critical section. While in the critical section, it blocks; p_0 unblocks, and assigns 1 to number[0] at line 3. It proceeds to the while loop at line 5. When it goes through that loop for $j = 1$, the first condition on line 5 is true. Further, the second condition on line 5 is false, so p_0 enters the critical section. Now p_0 and p_1 are both in the critical section, violating mutual exclusion. The reason for choosing is to prevent the while loop in line 5 from being entered when process j is setting its number[j]. Note that if the loop is entered and then process

j reaches line 3, one of two situations arises. Either `number[j]` has the value 0 when the first test is executed, in which case process i moves on to the next process, or `number[j]` has a non-zero value, in which case at some point `number[j]` will be greater than `number[i]` (since process i finished executing statement 3 before process j began). Either way, process i will enter the critical section before process j, and when process j reaches the while loop, it will loop at least until process i leaves the critical section.

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

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

5.13 To incorporate bounded waiting, we can define an additional boolean variable `waiting` associated with each process. A process `P[i]` can enter its critical section only if `waiting[i]` is `FALSE` or `key` is 1. An implementation is as follows:

```
/* program mutual exclusion with bounded waiting */
int const n = /* number of processes */
int bolt;
boolean waiting[n];

void P(int i)
{
    while (TRUE) {
        waiting[i]=TRUE;
        key = 1;
        do exchange(&key, &bolt)
        while ((waiting[i]!= FALSE) && (key != 0));
        waiting[i]= FALSE;

        /* critical section */

        j = (i + 1) % n;
        while ((j != i) && (waiting[j]== FALSE))
            j = (j + 1) % n;

        if (j == i)
            bolt = 0;
        else
            waiting[j] = FALSE;

        /* remainder */
    }
}

void main()
{
    bolt=0;
    parbegin (P(1), P(2), . . . , P(n));
}
```

In the above algorithm, we see that when a process leaves its critical section, it scans the array `waiting[n]` in a cyclic order and halts at the first process in the order that is waiting for the critical section. It then appoints this process as the next entrant to the critical section. Thus the condition of being bounded is met.

```

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

```

The algorithm uses the common data structures

```

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

```

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

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

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

gain access to the resource. When the unblocked ones eventually resume execution, they will also begin accessing the resource. The solution has failed because it has allowed four processes to access the resource together.

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

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

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

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

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

26, the process leaves without opening the mutex) or block yourself.

- b. Only one waiting process can be unblocked even if several are waiting—to unblock more would violate the mutual exclusion of the status variables. This problem is solved by having the newly unblocked process check whether more processes should be unblocked (line 14). If so, it passes the baton to one of them (line 15); if not, it opens up the mutual exclusion for new arrivals (line 17).
- c. This pattern synchronizes processes like runners in a relay race. As each runner finishes her laps, she passes the baton to the next runner. “Having the baton” is like having permission to be on the track. In the synchronization world, being in the mutual exclusion is analogous to having the baton—only one person can have it..

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

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

5.20

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

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

5.22

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

Both producer and consumer are blocked.

5.23

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

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

5.25

Scheduled step of execution	full's state & queue	Buffer	empty's state & queue
Initialization	full = 0	000	empty = +3
Ca executes c1	full = -1 (Ca)	000	empty = +3
Cb executes c1	full = -2 (Ca, Cb)	000	empty = +3
Pa executes p1	full = -2 (Ca, Cb)	000	empty = +2
Pa executes p2	full = -2 (Ca, Cb)	X00	empty = +2
Pa executes p3	full = -1 (Cb) Ca	X00	empty = +2
Ca executes c2	full = -1 (Cb)	000	empty = +2
Ca executes c3	full = -1 (Cb)	000	empty = +3
Pb executes p1	full = -1 (Cb)	000	empty = +2
Pa executes p1	full = -1 (Cb)	000	empty = +1
Pa executes p2	full = -1 (Cb)	X00	empty = +1
Pb executes p2	full = -1 (Cb)	XX0	empty = +1
Pb executes p3	full = 0 (Cb)	XX0	empty = +1
Pc executes p1	full = 0 (Cb)	XX0	empty = 0
Cb executes c2	full = 0	X00	empty = 0
Pc executes p2	full = 0	XX0	empty = 0
Cb executes c3	full = 0	XX0	empty = +1
Pa executes p3	full = +1	XX0	empty = +1
Pb executes p1-p3	full = +2	XXX	empty = 0
Pc executes p3	full = +3	XXX	empty = 0
Pa executes p1	full = +3	XXX	empty = -1(Pa)
Pd executes p1	full = +3	XXX	Empty = -2(Pa, Pd)
Ca executes c1-c3	full = +2	XX0	empty = -1(Pd) Pa
Pa executes p2	full = +2	XXX	empty = -1(Pd)
Cc executes c1-c2	full = +1	XX0	empty = -1(Pd)
Pa executes p3	full = +2	XX0	empty = -1(Pd)
Cc executes c3	full = +2	XX0	empty = 0(Pd)
Pd executes p2-p3	full = +3	XXX	empty = 0

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

5.26

```

#define REINDEER 9 /* max # of reindeer */
/*
#define ELVES      3 /* size of elf group */
/* Semaphores */
only_elves = 3, /* 3 go to Santa */
emutex = 1, /* update elf_cnt */
rmutex = 1, /* update rein_ct */
rein_semWait = 0, /* block early arrivals
back from islands */
sleigh = 0, /* all reindeer
semWait
around the sleigh */
done = 0, /* toys all delivered */
santa_semSignal = 0, /* 1st 2 elves semWait
on
this outside Santa's shop
*/
santa = 0, /* Santa sleeps on this
blocked semaphore
*/
problem = 0, /* semWait to pose
the
question to Santa */
elf_done = 0; /* receive reply */
/* Shared Integers */
rein_ct = 0; /* # of reindeer back
*/
elf_ct = 0; /* # of elves with problem
*/

/* Reindeer Process */
for (;;) {
    tan on the beaches in the Pacific until
    Christmas is close
    semWait (rmutex)
    rein_ct++
    if (rein_ct == REINDEER) {
        semSignal (rmutex)
        semSignal (santa)
    }
    else {
        semSignal (rmutex)
        semWait (rein_semWait)
    }
}
/* all reindeer semWaiting to be attached to
sleigh */
semWait (sleigh)
fly off to deliver toys
semWait (done)
head back to the Pacific islands
} /* end "forever" loop */

/* Elf Process */
for (;;) {
    semWait (only_elves) /* only 3 elves
"in" */
    semWait (emutex)
    elf_ct++
    if (elf_ct == ELVES) {
        semSignal (emutex)
        semSignal (santa) /* 3rd elf wakes
Santa */
    }
    else {
        semSignal (emutex)
        semWait (santa_semSignal) /*
semWait outside
Santa's shop door */
    }
}
semWait (problem)
ask question /* Santa woke elf up */
semWait (elf_done)
semSignal (only_elves)
} /* end "forever" loop */

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

```

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

send (message, dest)	
semWait (mbuf)	semWait for message buffer available
semWait (mutex)	mutual exclusion on message queue
acquire free buffer slot	
copy message to slot	
link slot to other messages	
semSignal (dest.sem)	wake destination process
semSignal (mutex)	release mutual exclusion
receive (message)	
semWait (own.sem)	semWait for message to arrive
semWait (mutex)	mutual exclusion on message queue
unlink slot from own.queue	
copy buffer slot to message	
add buffer slot to freelist	
semSignal (mbuf)	indicate message slot freed
semSignal (mutex)	release mutual exclusion

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

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

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

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

CHAPTER 6 DEADLOCK AND STARVATION

ANSWERS TO QUESTIONS

- 6.1** Examples of reusable resources are processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores. Examples of consumable resources are interrupts, signals, messages, and information in I/O buffers.
- 6.2** **Mutual exclusion.** Only one process may use a resource at a time. **Hold and wait.** A process may hold allocated resources while awaiting assignment of others. **No preemption.** No resource can be forcibly removed from a process holding it.
- 6.3** The above three conditions, plus: **Circular wait.** A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.
- 6.4** Deadlocks can be dealt with in any of the following methods:
- a. Deadlock prevention:** Deadlocks can be prevented by stopping the system from allowing one of the four necessary conditions for their occurrence. This is done either indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption), or directly, by preventing the occurrence of a circular wait. Deadlock prevention leads to an inefficient use of resources and an inefficient execution of processes.
 - b. Deadlock avoidance:** This method permits the three necessary conditions but makes judicious and dynamic choices to ensure that the deadlock point is never reached. As such, avoidance allows more concurrency than prevention. With deadlock avoidance, a decision is taken on whether granting the current resource allocation request can potentially lead to a deadlock.
 - c. Deadlock detection and recovery:** In contrast to the above strategies, the deadlock detection and recovery strategy does not limit resource access or restrict process actions. Instead, the OS periodically checks whether a deadlock has occurred in the system and tries to recover it if one has.
- 6.5** Mutual exclusion restricts the usage of a resource to one user at a time. If mutual exclusion is disallowed, then all non-sharable resources

become sharable. While this may not hamper some activities (like a read-only file being accessed by a number of users), it poses serious problems for activities that require non-sharable resources (like writing to a file). Preventing mutual exclusion in these situations gives undesirable results. Also, there are some resources (like printers) that are inherently non-sharable, and it is impossible to disallow mutual exclusion. Thus, in general, mutual exclusion cannot be disallowed for practical purposes.

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 Some methods to recover from deadlocks are:

- a. Abort all deadlocked processes. Though this is a common solution adopted in operating systems, the overhead is very high in this case.
- b. Back up each deadlocked process to some previously defined checkpoint and restart all processes. This requires that rollback and restart mechanisms be built into the system. The risk in this approach lies in the fact that the original deadlock may recur.
- c. Detect the deadlocked processes in a circular-wait condition. Successively abort deadlocked processes until the circular wait is eliminated and the deadlock no longer exists. The order in which processes are selected for abortion should be on the basis of some criterion of minimum cost.
- d. Successively preempt resources until the deadlock no longer exists. A process that has a resource preempted from it must be rolled back to a point prior to its acquisition of that resource.

ANSWERS TO PROBLEMS

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

6.2 Prevention: Hold-and-wait approach: Require that a car request both quadrants that it needs and blocking the car until both quadrants can be granted. No preemption approach: releasing an assigned quadrant is problematic, because this means backing up, which may not be possible if there is another car behind this car. Circular-wait approach: assign a linear ordering to the quadrants.

Avoidance: The algorithms discussed in the chapter apply to this problem. Essentially, deadlock is avoided by not granting requests that might lead to deadlock.

Detection: The problem here again is one of backup.

- 6.3**
1. Q acquires B and A, and then releases B and A. When P resumes execution, it will be able to acquire both resources.
 2. Q acquires B and A. P executes and blocks on a request for A. Q releases B and A. When P resumes execution, it will be able to acquire both resources.
 3. Q acquires B and then P acquires and releases A. Q acquires A and then releases B and A. When P resumes execution, it will be able to acquire B.
 4. P acquires A and then Q acquires B. P releases A. Q acquires A and then releases B. P acquires B and then releases B.
 5. P acquires and then releases A. P acquires B. Q executes and blocks on request for B. P releases B. When Q resumes execution, it will be able to acquire both resources.
 6. P acquires A and releases A and then acquires and releases B. When Q resumes execution, it will be able to acquire both resources.

6.4 A sequence that does not cause deadlock:

Process P	Process Q	Resources held by P	Resources held by Q	Resources requested by P	Resources requested by Q
		-	-	A, B	A, B
Get A		A	-	B	A, B
Release A		-	-	B	A, B
	Get B	-	B	B	A
	Get A	-	A, B	B	-
	Release B	-	A	B	-
Get B	-	B	A	-	-
Release B	-	-	A	-	-
	Release A	-	-	-	-

P completes and exits

Q completes and exits

Another sequence that does not cause deadlock:

Process P	Process Q	Resources held by P	Resources held by Q	Resources requested by P	Resources requested by Q
		-	-	A, B	A, B
	Get B	-	B	A, B	A
Get A		A	B	B	A
Release A		-	B	B	A
	Get A	-	A, B	B	-
	Release B	-	A	B	-
	Release A	-	-	B	-
Get B		B	-	-	-
Release B		-	-	-	-

Q completes and exits

P completes and exits

6.5 Given that

Total number of existing resources:

R1	R2	R3	R4
6	3	4	3

Snapshot at the initial time stage:

	Allocation					Claim			
	R1	R2	R3	R4		R1	R2	R3	R4
P1	3	0	1	1		6	2	1	1
P2	0	1	0	0		0	2	1	2
P3	1	1	1	0		3	2	1	0
P4	1	1	0	1		1	1	1	1
P5	0	0	0	0		2	1	1	1

a. Total number of resources allocated to different processes:

R1	R2	R3	R4
3 + 1 + 1 = 5	1 + 1 + 1 = 3	1 + 1 = 2	1 + 1 = 2

Available matrix = Total Existing – Total Allocated

R1	R2	R3	R4
1	0	2	1

b. Need matrix = Claim – Allocation

Process	Need			
	R1	R2	R3	R4
P1	3	2	0	0
P2	0	1	1	2
P3	2	1	0	0
P4	0	0	1	0
P5	2	1	1	1

c. Safety algorithm:

The following matrix shows the order in which the processes can run to completion. It also shows the resources that become available once a given process completes and releases the resources held by it.

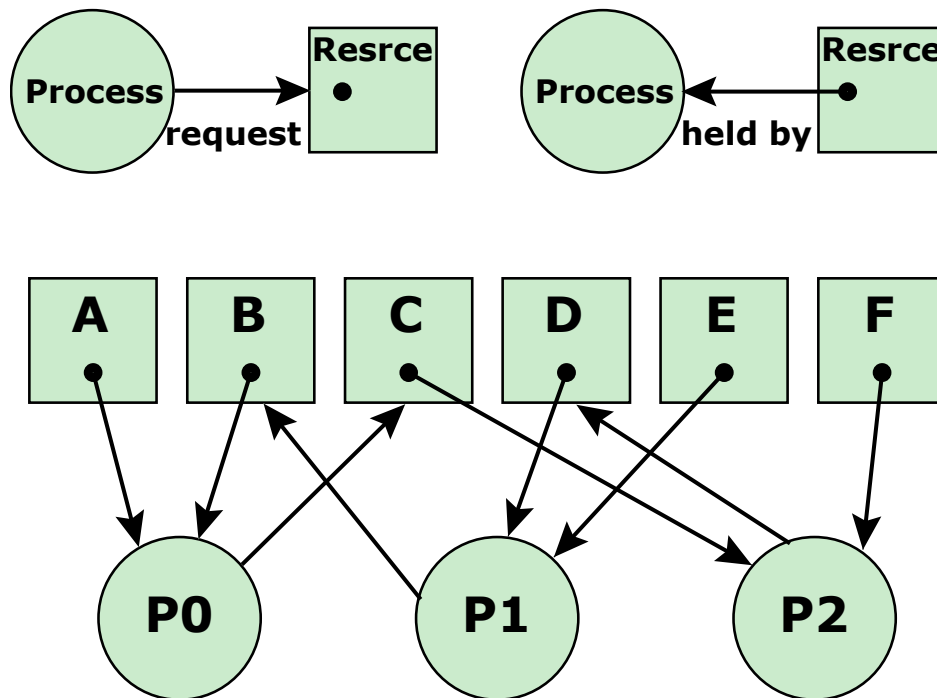
At each step, a process P_i can be completed if $\text{Need}[i] \leq \text{Available}$.
The Available matrix is updated as $\text{Available} = \text{Available} + \text{Allocation}$

Process	Available			
	R1	R2	R3	R4
P4	2	1	2	2
P2	2	2	2	2
P5	2	2	2	2
P3	3	3	3	2
P1	6	3	4	3

Hence, the system is in a safe state and $\langle P4, P2, P5, P3, P1 \rangle$ is a safe sequence.

d. Request from P1 is (1, 1, 0, 0), whereas Available is (1, 0, 2, 1). As Request is greater than Available, this request cannot be granted.

6.6 a.



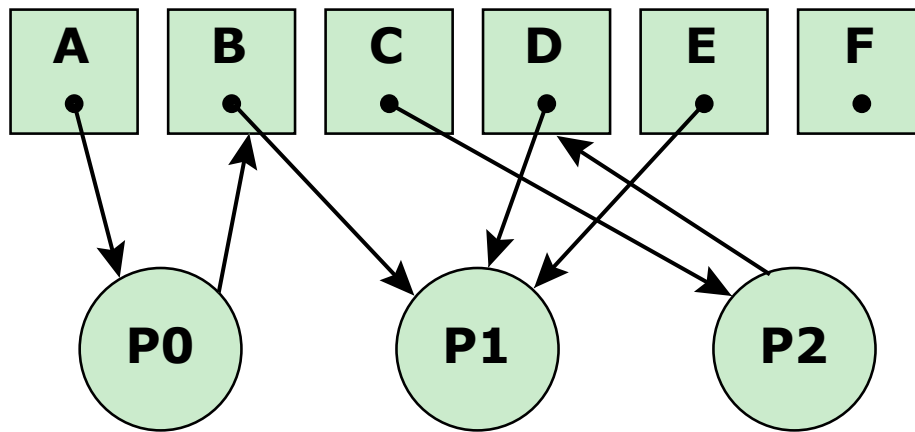
There is a deadlock if the scheduler goes, for example: P0-P1-P2-P0-P1-P2 (line by line): Each of the 6 resources will then be held by one process, so all 3 processes are now blocked at their third line inside the loop, waiting for a resource that another process holds. This is illustrated by the circular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

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

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

A	B	C
B	D	D
C	E	F

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



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

6.7 A deadlock occurs when process I has filled the disk with input ($i = \text{max}$) and process i is waiting to transfer more input to the disk, while process P is waiting to transfer more output to the disk and process O is waiting to transfer more output from the disk.

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

$$i + o \leq \text{max}$$

$$i \leq \text{max} - \text{reso}$$

where

$$0 < \text{reso} < \text{max}$$

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

6.9

$$i + o + p \leq \text{max}$$

$$i + o \leq \text{max} - \text{resp}$$

$$i + p \leq \text{max} - \text{reso}$$

$$i \leq \text{max} - (\text{reso} + \text{resp})$$

- 6.10 a.**
1. $i \leftarrow i + 1$
 2. $i \leftarrow i - 1; p \leftarrow p + 1$
 3. $p \leftarrow p - 1; o \leftarrow o + 1$
 4. $o \leftarrow o - 1$
 5. $p \leftarrow p + 1$
 6. $p \leftarrow p - 1$
- b.** By examining the resource constraints listed in the solution to problem 6.7, we can conclude the following:
6. Procedure returns can take place immediately because they only release resources.
 5. Procedure calls may exhaust the disk ($p = \text{max-reso}$) 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 ($p < \text{max-reso}$).

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

6.11 Given allocation state is:

	Allocation				Request				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	0	2	0	0	2	2	1	1	2
P2	2	0	0	1	1	3	0	1				
P3	0	1	1	0	2	1	1	0				
P4	1	1	0	0	4	0	3	1				

Since $\text{Request}[3] \leq \text{Available}$, P3 can run into completion. So, mark P3.

New Available = Available + Allocation[3]

Available			
R1	R2	R3	R4
2	2	2	2

Since $\text{Request}[1] \leq \text{Available}$, P1 can run into completion. So, mark P1.

New Available = Available + Allocation[1]

Available			
R1	R2	R3	R4
2	2	3	2

At this stage, neither P2 nor P4 can run to completion as the request vectors of P2 and P4 have at least one element greater than available. Hence, the system is deadlocked and the deadlocked processes are P2 and P4.

- 6.12** A recovery strategy that may be adapted is to preempt some resources so that one of the two processes, P1 and P2, can run to completion. Subsequently, the other will also run to completion.

The available vector, after P3 runs to completion and releases its allocated resources, is:

Available				
R1	R2	R3	R4	R5
0	0	0	1	1

Two preemptions can be identified here:

1. Preempt one instance of resource R2 from process P2. The available vector will now be as follows:

Available				
R1	R2	R3	R4	R5
0	1	0	1	1

Since $\text{Request}[1] \leq \text{Available}$, P1 can run into completion. After P1 releases its allocated resources, P2 can also run into completion.

2. Preempt 1 instance of resource R3 from process P1. The available vector will be:

Available				
R1	R2	R3	R4	R5
0	0	1	1	1

Since $\text{Request}[2] \leq \text{Available}$, P2 can run into completion. After P2 releases its resources, P1 can also run into completion.

Either of the options mentioned above can be used for deadlock recovery. However, the actual decision depends upon a number of factors, such as processor time consumed by each process thus far,

output produced by each process thus far, relative priority of the processes, the remaining time, and the overhead associated with the preemption of a resource.

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

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

"Initialization"

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

"Process i"

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

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

region v do S

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

b. A deadlock is a situation in which:

P_0 waits for P_{n-1} AND
 P_1 waits for P_0 AND
 $\dots\dots\dots$
 P_{n-1} waits for P_{n-2}

because

(available (0) = 0) AND
 (available (1) = 0) AND
 $\dots\dots\dots$
 (available (n-1) = 0)

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

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

- 6.14 a.** Yes. If `foo()` executes `semWait(S)` and then `bar()` executes `semWait(R)` both processes will then block when each executes its next instruction. Since each will then be waiting for a `semSignal()` call from the other, neither will ever resume execution.
- b.** No. If either process blocks on a `semWait()` call then either the other process will also block as described in (a) or the other process is executing in its critical section. In the latter case, when the running process leaves its critical section, it will execute a `semSignal()` call, which will awaken the blocked process.

6.15 The current states of the claim and allocation matrices are:

$$C = \begin{pmatrix} 2 \\ 6 \\ 9 \\ 5 \end{pmatrix} \quad A = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 3 \end{pmatrix}$$

Total number of instances of the resource = 9

Number of available resources, $a = 9 - (1 + 2 + 2 + 3) = 1$

Let the processes be P_1, P_2, P_3 , and P_4 .

Let a sequence of allocation be:

P_1 since $C_1 - A_1 = 1 \leq a$
 Update a as, $a = a + A_1 = 2$

P_4 since $C_4 - A_4 = 2 \leq a$
 Update a as, $a = a + A_4 = 5$

P2 since $C2 - A2 = 4 \leq a$
 Update a as, $a = a + A2 = 7$

P3 since $C3 - A3 = 7 \leq a$
 Update a as, $a = a + A4 = 9$

Because all the processes can run into completion, the system is in a safe state.

If P4 is allocated a resource, the number of available resources will be 0. At this stage, no process can run into completion and so the system will be deadlocked.

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

1. detect deadlock and kill thread, releasing its resources

 detect deadlock and roll back thread's actions

 restart thread and release all resources if thread needs to wait

None of these algorithms limit concurrency before deadlock occurs, because they rely on runtime checks rather than static restrictions.

Their effects after deadlock is detected are harder to characterize: they still allow lots of concurrency (in some cases they enhance it),

but the computation may no longer be sensible or efficient. The third algorithm is the strangest, since so much of its concurrency

will be useless repetition; because threads compete for execution time, this algorithm also prevents useful computation from

advancing. Hence it is listed twice in this ordering, at both extremes.

2. banker's algorithm

 resource ordering

These algorithms cause more unnecessary waiting than the previous two by restricting the range of allowable computations.

The banker's algorithm prevents unsafe allocations (a proper superset of deadlock-producing allocations) and resource ordering restricts allocation sequences so that threads have fewer options as to whether they must wait or not.

3. reserve all resources in advance

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

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

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

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

- 1.** reserve all resources in advance
resource ordering

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

- 2.** banker's algorithm

detect deadlock and kill thread, releasing its resources

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

- 3.** detect deadlock and roll back thread's actions

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

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

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

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

6.17 The problem can be solved by assigning two different rules, one for the odd philosophers and one for the even philosophers. If all the odd philosophers pick up the left fork first and then the right fork; and all the even philosophers pick up the right fork first and then the left fork, the system will be deadlock free.

The following algorithms incorporate the stated logic. For the implementation, it has been assumed that the philosophers are

numbered in an anti-clockwise manner, with the fork to the left having same index as that of the philosopher.

```

/* program evendiningphilosophers */
#define N 10
/* N can be any even number denoting the number of philosophers */
int i;
void philosopher (int i)
{
    while (true)
    {
        think();
        if((i%2)==0)    /* even philosopher */
        {
            wait (fork [(i+1) mod N]);
            wait (fork[i]);
            eat();
            signal(fork[i]);
            signal(fork [(i+1) mod N]);
        }
        else    /* odd philosopher */
        {
            wait (fork[i]);
            wait (fork [(i+1) mod N]);
            eat();
            signal(fork [(i+1) mod N]);
            signal(fork[i]);
        }
    }
}
void main()
{
    for(i=0;i<N;i++)
        parbegin (philosopher (i));
}

```

- 6.18 a.** Assume that the table is in deadlock, i.e., there is a nonempty set D of philosophers such that each P_i in D holds one fork and waits for a fork held by neighbor. Without loss of generality, assume that $P_j \in D$ is a lefty. Since P_j clutches his left fork and cannot have his right fork, his right neighbor P_k never completes his dinner and is also a lefty. Therefore, $P_k \in D$. 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 P_j starves, i.e., there is a stable pattern of dining in which P_j never eats. Suppose P_j holds no fork. Then P_j 's left neighbor P_i must continually hold his right fork and never finishes eating. Thus P_i is a righty holding his right fork, but never getting his left fork to complete a meal, i.e., P_i also starves. Now P_i '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 P_j is a lefty: a contradiction. Thus P_j must hold one fork.

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

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

6.20 Atomic operations are used in LINUX systems to avoid simple race conditions. In a uniprocessor system, a thread performing an atomic operation cannot be interrupted once the operation has started until it has finished. In a multiprocessor system, the variable being operated is locked from access by other threads until this operation is completed. Some of the benefits of the implementation are:

1. Taking locks for small operations can be avoided, thus increasing the performance.
2. It enables code portability since the behaviour of the atomic functions is guaranteed to be unique across any architecture that Linux supports.

A simple program to implement counters using atomic variables is:

```
atomic_t *mycount;

void main()
{
    printf("Increment Counter");
    mycondition=get_condition();
    if(mycondition == 1)
        atomic_inc(&mycount);
    printf("Count value = %d", *mycount);

    printf("Decrement Counter");
    mycondition=get_condition();
    if(mycondition == 1)
        atomic_dec(&mycount);
    printf("Count value = %d", *mycount);
}
```

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

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

CHAPTER 7 MEMORY MANAGEMENT

ANSWERS TO QUESTIONS

- 7.1** Relocation, protection, sharing, logical organization, physical organization.
- 7.2** To relocate a program is to load and execute a given program to an arbitrary place in the memory; therefore, once a program is swapped out to the disk, it may be swapped back anywhere in the main memory. To allow this, each program is associated with a logical address. The logical address is generated by the CPU and is converted, with the aid of the memory manager, to the physical address in the main memory. A CPU register contains the values that are added to the logical address to generate the physical address.
- 7.3** Some of the advantages of organizing programs and data into modules are: **1.** Modules can be written and compiled independently. All references from one module to another can be resolved by the system at run time. **2.** Each module can be given different degrees of protection (like read only, read-write, execute only, read-write-execute, etc.). The overhead associated with this is quite nominal. **3.** A module can be shared among different processes by incorporating appropriate mechanisms.
- 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** Address binding is the process of associating program instructions and data with physical memory addresses so that the program can be executed. In effect, it is the mapping of a logical address generated by the CPU to the physical address.

This binding can be done at the following times:

Programming time: All actual physical addresses are directly specified by the programmer in the program itself.

Compile or assembly time: The program contains symbolic address references, and these are converted to actual physical addresses by the compiler or assembler.

Load time: The compiler or assembler produces relative addresses. The loader translates these to absolute addresses at the time of program loading.

Run time: The loaded program retains relative addresses. These are converted dynamically to absolute addresses by processor hardware.

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

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

ANSWERS TO PROBLEMS

- 7.1** Here is a rough equivalence:

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

- 7.2** The number of partitions equals the number of bytes of main memory divided by the number of bytes in each partition: $2^{24}/2^{16} = 2^8$. Eight bits are needed to identify one of the 2^8 partitions.
- 7.3** Let s and h denote the average number of segments and holes, respectively. The probability that a given segment is followed by a hole in memory (and not by another segment) is 0.5, because deletions and creations are equally probable in equilibrium. so with s segments in memory, the average number of holes must be $s/2$. It is intuitively reasonable that the number of holes must be less than the number of segments because neighboring segments can be combined into a single hole on deletion.
- 7.4** Let N be the length of list of free blocks.
Best-fit: Average length of search = N , as each free block in the list is considered, to find the best fit.
First-fit: The probability of each free block in the list to be large enough or not large enough, for a memory request is equally likely. Thus the probability of first free block in the list to be first fit is $1/2$. For the second free block to be first fit, the first free block should be smaller, and the second free block should be large enough, for the memory request. Thus the probability of second free block to be first fit is $1/2 \times 1/2 = 1/4$. Proceeding in the same way, probability of i th free block in the list to be first fit is $1/2^i$. Thus the average length of search = $1/2 + 2/2^2 + 3/2^3 + \dots + N/2^N + N/2^N$
 (the last term corresponds to the case, when no free block fits the request). Above length of search has a value between 1 and 2.
Next-fit: Same as first-fit, except for the fact that search starts where the previous first-fit search ended.
- 7.5 a.** A criticism of the best-fit algorithm is that the space remaining after allocating a block of the required size is so small that in general it is of no real use. The worst fit algorithm maximizes the chance that the free space left after a placement will be large enough to satisfy another request, thus minimizing the frequency of compaction. The disadvantage of this approach is that the largest blocks are allocated first; therefore a request for a large area is more likely to fail.
- b.** Same as best fit.
- 7.6 a.** When the 2-MB process is placed, it fills the leftmost portion of the free block selected for placement. Because the diagram shows an empty block to the left of X, the process swapped out after X was placed must have created that empty block. Therefore, the maximum size of the swapped out process is 1M.
- b.** The free block consisted of the 5M still empty plus the space occupied by X, for a total of 7M.

c. The answers are indicated in the following figure:



7.7 The results of allocation/de-allocation at each stage:

	Memory Map				Internal Fragmentation
Initial	512				0
Request A:100	A	128	256		28 (A: 28)
Request B:40	A	B	64	256	52 (A:28, B:24)
Request C:190	A	B	64	C	118 (A:28, B:24, C:66)
Return A	128	B	64	C	90 (B:24, C:66)
Request D:60	128	B	D	C	94 (B:24, C:66, D:4)
Return B	128	64	D	C	70 (D:4, C:66)
Return D	256			C	66 (C:66)
Return C	512				0

7.8 It takes 120 ns for a page lookup in case of a hit in the registers and 600 ns if there is a page-miss.

Hit ratio = 80%

Thus, the average page-lookup time = $0.8 \times 120 + 0.2 \times 600 = 216$ ns.

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

7.10 a. Yes, the block sizes could satisfy $F_n = F_{n-1} + F_{n-2}$.

b. This scheme offers more block sizes than a binary buddy system, and so has the potential for less internal fragmentation, but can cause additional external fragmentation because many uselessly small blocks are created.

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

- 7.12 a.** The total physical memory size is 2GB.
The number of bits in the physical address = $\log_2 (2 \times 2^{30}) = 31$ bits
- b.** The page size is 8KB.
The number of bits specifying page displacement = $\log_2 (8 \times 2^{10}) = 13$ bits
Thus, the number of bits for the page frame number = $31 - 13 = 18$ bits
- c.** The number of physical frames = $2^{18} = 262144$
- d.** The logical address space for each process is 128MB, requiring a total of 27 bits. The page size is the same as that of the physical pages (i.e., 8KB). Therefore, the logical address layout is 27 bits, with 14 bits for the page number and 13 bits for displacement.
- 7.13 a.** The page size is 512 bytes that requires 9 lower bits. So, the page number is in the higher 7 bits: 0011000. We chop it off from the address and replace it with the frame number, which is half the page number, that is, shifted 1 bit to the right: 0001100. Therefore the result is this frame number concatenated with the offset 000110011: binary physical address = 0001100000110011.
- b.** The segment number is in the higher 5 bits: 00110. We chop it off from the address and add the remaining offset 000 0011 0011 to the base of the segment. The base is $20 = 10100$ added to the segment number times 4,096, that is, shifted 12 bits to the left: $10100 + 0110\ 0000\ 0000\ 0000 = 0110\ 0000\ 0001\ 0100$. So adding up the 2 two underlined numbers gives: binary physical address = 0110 0000 0100 0111.
- 7.14 a.** Segment 0 starts at location 830. With the offset, we have a physical address of $830 + 228 = 1058$
- b.** Segment 2 has a length of 408 bytes which is less than 684, so this address triggers a segment fault.
- c.** $770 + 776 = 1546$
- d.** $648 + 98 = 746$
- e.** Segment 1 has a length of 110 bytes which is less than 240, so this address triggers a segment fault.
- 7.15 a.** Observe that a reference occurs to some segment in memory each time unit, and that one segment is deleted every t references. Because the system is in equilibrium, a new segment must be inserted every t references; therefore, the rate of the boundary's movement is s/t words per unit time. The system's operation time t_0 is then the time required for the boundary to cross the hole, i.e., $t_0 = fm/s$, 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. $k = (t/2s) - 1 = 7$; $F \geq (1 - 0.25)/(1 + 7 \times 0.25) = 0.75/2.75 = 0.28$

CHAPTER 8 VIRTUAL MEMORY

ANSWERS TO QUESTIONS

- 8.1** The use of virtual memory improves system utilization in the following ways:
- a.** More processes may be maintained in main memory: The use of virtual memory allows the loading of only portions of a process into the main memory. Therefore more processes can enter the system, thus resulting in higher CPU utilization.
 - b.** A process may be larger than all of main memory: The use of virtual memory theoretically allows a process to be as large as the disk storage available, without taking heed of the size of the main memory.
- 8.2** Thrashing is a phenomenon in virtual memory schemes, in which the processor spends most of its time swapping pieces rather than executing instructions.
- 8.3** Algorithms can be designed to exploit the principle of locality to avoid thrashing. In general, the principle of locality allows the algorithm to predict which resident pages are least likely to be referenced in the near future and are therefore good candidates for being swapped out.
- 8.4** Factors that determine page size are:
- a. Page size versus page table size:** A system with a smaller page size uses more pages, thus requiring a page table that occupies more space. For example, if a 2^{32} virtual address space is mapped to 4KB (2^{12} bytes) pages, the number of virtual pages is 2^{20} . However, if the page size is increased to 32KB (2^{15} bytes), only 2^{17} pages are required.
 - b. Page size versus TLB usage:** Since every access to memory must be mapped from a virtual to a physical address, reading the page table every time can be quite costly. Therefore, the translation lookaside buffer (TLB) is often used. The TLB is typically of a limited size, and when it cannot satisfy a given request (a TLB miss) the page tables must be searched manually (either in the hardware or the software, depending on the architecture) for the correct mapping.

Larger page sizes mean that a TLB cache of the same size can keep track of larger amounts of memory, which avoids the costly TLB misses.

- c. Internal fragmentation of pages:** Rarely do processes require the use of an exact number of pages. As a result, it is likely that the last page will be only partially full, causing internal fragmentation. Larger page sizes clearly increase the potential for wasted memory in this way since more potentially unused portions of memory are loaded into main memory. Smaller page sizes ensure a closer match to the actual amount of memory required in an allocation.
- d. Page size versus disk access:** When transferring from disk, much of the delay is caused by seek time—the time it takes to correctly position the read/write heads above the disk platters. Because of this, large sequential transfers are more efficient than several smaller transfers; transferring the same amount of data from disk to memory often requires less time with larger pages than with smaller pages.

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 Demand paging is a page fetch policy in which a page is brought into the main memory only when it is referenced, i.e., the pages are loaded only when they are demanded during program execution. The pages that are not accessed are thus never brought into the main memory. When a process starts, there is a flurry of page faults. As more pages are brought in, the principle of locality suggests that most future references will be to those pages that have been brought in recently. After a while, the system generally settles down and the number of page faults drops to a low level.

8.7 Cleaning refers to determining when a modified page should be written out to secondary memory. Two common cleaning policies are demand cleaning and precleaning.

There are problems with using either of the two policies exclusively. This is because, on the one hand, precleaning involves a page being written out but remaining in the main memory until the page replacement algorithm dictates that it can be removed. Therefore, while precleaning allows the writing of pages in batches, it makes little sense to write out hundreds or thousands of pages only to find that the majority of them have been modified again before they are replaced. The transfer capacity of secondary memory is limited in this method; it is wasted in unnecessary cleaning operations.

On the other hand, with demand cleaning, the writing of a dirty page is coupled to, and precedes, the reading in of a new page. This technique may minimize page writes, but it results in the fact that a process that suffers a page fault may have to wait for two-page transfers before it can be unblocked. This may decrease processor utilization.

- 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** The following steps are generally used to deal with page fault traps: **1.** An internal table in the PCB (process control block) of the process is checked to determine whether the reference is a valid or an invalid memory access. **2.** If the reference is invalid, the process is terminated. **3.** If it is valid, but the page is not in memory, it is brought in. **4.** A free frame is brought from the free-frame list. **5.** A disk operation is scheduled to read the desired page into the newly allocated frame. **6.** When the disk read is complete, the internal table (which is kept with the process and the page table) is modified. **7.** The instruction interrupted by the trap is restarted.
- 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)** $6,204 = 3 \times 2,048 + 60$ maps to VPN 3 in PFN 6
Physical address = $6 \times 2,048 + 60 = 12,348$
- (ii)** $3,021 = 1 \times 2,048 + 973$ maps to VPN 1. Page Fault occurs.
- (iii)** $9,000 = 4 \times 2,048 + 808$ maps to VPN 4 in PFN 0
Physical Address = $0 \times 2,048 + 808 = 808$

- 8.2 a.** 3 page faults for every 4 executions of $C[i, j] = A[i, j] + B[i, j]$.
b. Yes. The page fault frequency can be minimized by switching the inner and outer loops.
c. After modification, there are 3 page faults for every 256 executions.

8.3 a.4 MByte

- b.** Number of rows: $2^8 \times 2 = 512$ entries. Each entry consist of: 24 (page number) + 24 (frame number) + 16 bits (chain index) = 64 bits = 8 bytes.
 Total: $512 \times 8 = 4096$ bytes = 4Kbytes

8.4 a. FIFO page-replacement algorithm

Page Address Stream	a	b	d	c	b	e	d	b	d	b	a	c	b	c	a	c	f	a	f	d
Frames	a	a	a	c	c	c	c	c	d	d	d	d	B	b	b	b	b	b	b	d
		b	b	b	b	e	e	e	e	e	a	a	A	a	a	a	f	f	f	f
			d	d	d	d	d	b	b	b	b	c	C	c	c	c	c	a	a	a
Faults				F		F		F	F		F	F	F				F	F		F

Number of Faults = 10 No. of Hits = 7

b. Optimal page-replacement algorithm

Page Address Stream	a	b	d	c	b	e	d	b	d	b	a	c	b	c	a	c	f	a	f	d
Frames	a	a	a	c	c	e	e	e	e	e	a	a	A	a	a	a	a	a	a	d
		b	b	b	b	b	b	b	b	b	b	b	B	b	b	b	f	f	f	f
			d	d	d	d	d	d	d	d	d	c	C	c	c	c	c	c	c	c
Faults				F		F					F	F					F			F

Number of Faults = 6 No. of Hits = 11

c. LRU page-replacement algorithm

Page Address Stream	a	b	d	c	b	e	d	b	d	b	a	c	b	c	a	c	f	a	f	d
Frames	a	a	a	c	c	c	d	d	d	d	d	c	C	c	c	c	c	c	c	d
		b	b	b	b	b	b	b	b	b	b	b	B	b	b	b	f	f	f	f
			d	d	d	e	e	e	e	e	a	a	A	a	a	a	a	a	a	a
Hits				F		F	F				F	F					F			F

Number of Faults = 7 No. of Hits = 10

The above data shows that FIFO has highest number of page faults and Optimal has the lowest. It has generally been found that this is the

trend for most reference strings. Page faults in FIFO occur more frequently since the pages that are in the memory for the longest time are replaced without regarding the fact that they may have been used quite frequently. Since this problem is overridden in LRU, the latter shows a better performance.

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

8.6 a. LRU: Hit ratio = 16/33

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

b. FIFO: Hit ratio = 16/33

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

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

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

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

8.8 Given: Total number of frames, $m = 128$

Size of process p_1 , $S_1 = 45$

Size of process p_2 , $S_2 = 75$

Size of process p_3 , $S_3 = 33$

Size of process p_4 , $S_4 = 135$

a. Equal allocation algorithm: In the equal allocation algorithm, all processes are allotted an equal number of frames, irrespective of their memory requirements.

Hence, the number of frames allotted to each process = $128/4 = 32$

b. Proportional allocation algorithm: In the proportional allocation algorithm, frames are allocated in proportion to the total memory requirements.

Allocation is given by $a_i = (S_i/S) \times m$

where a_i = number of frames allocated to process p_i

S_i = size of p_i

S = total size of all processes

m = total number of frames

Here, total memory requirement, $S = \sum S_i = 45 + 75 + 33 + 135$
 $= 288$

$a_1 = (S_1/S) \times m = (45/288) \times 128 = 20$

$a_2 = (S_2/S) \times m = (75/288) \times 128 \approx 33$

$a_3 = (S_3/S) \times m = (33/288) \times 128 \approx 15$

$a_4 = (S_4/S) \times m = (135/288) \times 128 = 60$

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

8.10 Size of page table = page size \times page table entry
 $= 8 \times 1024 \times 6$
 $= 49152 \text{ bytes}$
 $= 48\text{KB}$

No. of virtual pages = virtual space/page size
 $= 6 \times 1024 \times 1024/8$
 $= 786432$

8.11 a. 250 ns will be taken to get the page table entry, and 250 ns will be required to access the memory location. Thus, paged memory reference will take 500 ns.

b. Two cases:

First, when the TLB contains the entry required. In that case we pay the 30 ns overhead on top of the 250 ns memory access time.

Hence, for TLB hit i.e. for 85% times, the time required is $250 + 30 = 280\text{ns}$.

Second, when the TLB does not contain the item. Then we pay an additional 250 ns to get the required entry into the TLB. Hence, for TLB miss i.e. for 15% times, the time required is $250 + 250 + 30 = 530 \text{ ns}$.

Effective Memory Access Time (EMAT) = $(280 \times 0.85) + (530 \times 0.15) = 317.5 \text{ ns}$

- c. The higher the TLB hit rate is, the smaller the EMAT is, because the additional 250 ns penalty to get the entry into the TLB contributes less to the EMAT.

8.12 a. A page fault occurs when a page is brought into a frame for the first time. So, for each distinct page, a page fault occurs. If the reference string and the page replacement algorithm are such that once a page is swapped out, it is never required to be swapped in again; no further page faults will occur. Hence, the lower bound of page faults is 8.

- b. At the other extreme case, it may so happen that a referenced page is never found in the working set. In this case, a page fault occurs on all the 24 page references. Hence, the upper bound of page faults is 24.

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

- b. Note that the density of replaceable pages is highest immediately in front of the clock pointer, just as the density of snow is highest immediately in front of the plow. Thus, we can expect the CLOCK algorithm to be quite efficient in finding pages to replace. In fact, it can be shown that the depth of the snow in front of the plow is twice the average depth on the track as a whole. By this analogy, the number of pages replaced by the CLOCK policy on a single circuit should be twice the number that are replaceable at a random time. The analogy is imperfect because the CLOCK pointer does not move at a constant rate, but the intuitive idea remains. The snowplow analogy to the CLOCK algorithm comes from [CARR84]; the depth analysis comes from Knuth, D. *The Art of Computer Programming, Volume 2: Sorting and Searching*. Reading, MA: Addison-Wesley, 1997 (page 256).

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

8.15 a.

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

b., c.

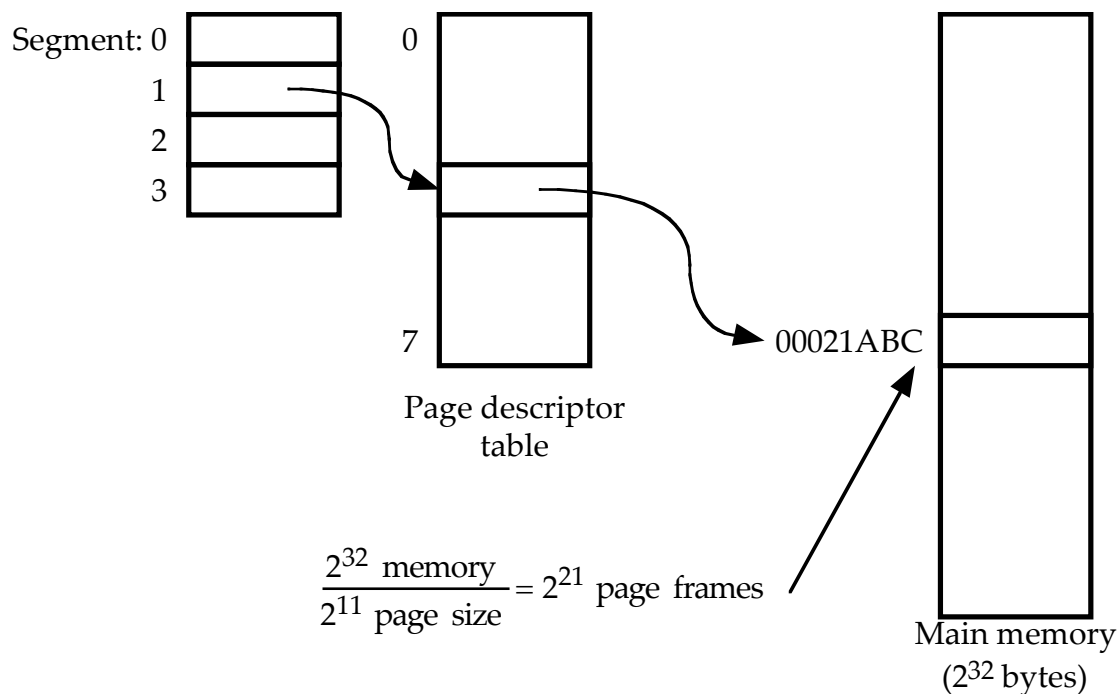
Δ	1	2	3	4	5	6
$s_{20}(\Delta)$	1	1.85	2.5	3.1	3.55	3.9
$m_{20}(\Delta)$	0.9	0.75	0.75	0.65	0.55	0.5

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

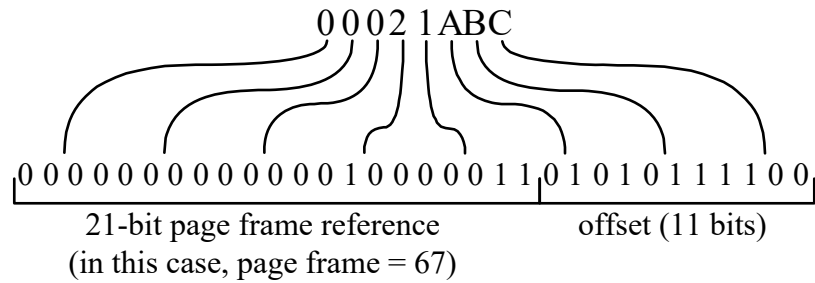
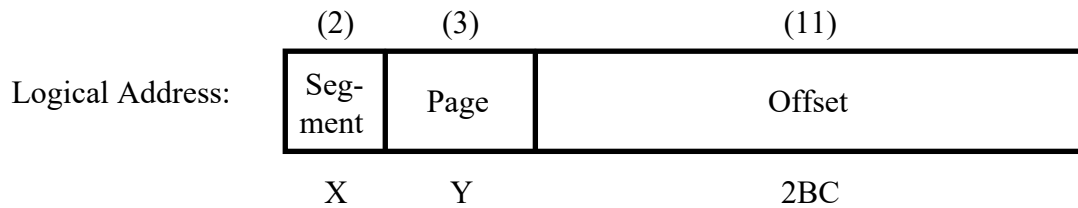
8.16 Consider this strategy. Use a mechanism that adjusts the value of Q at each window time as a function of the actual page fault rate experienced during the window. The page fault rate is computed and compared with a system-wide value for "desirable" page fault rate for a job. The value of Q is adjusted upward (downward) whenever the

actual page fault rate of a job is higher (lower) than the desirable value. Experimentation using this adjustment mechanism showed that execution of the test jobs with dynamic adjustment of Q consistently produced a lower number of page faults per execution and a decreased average resident set size than the execution with a constant value of Q (within a very broad range). The memory time product (MT) versus Q using the adjustment mechanism also produced a consistent and considerable improvement over the previous test results using a constant value of Q.

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



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



8.18 The contents of the working set, assuming that the pages are replaced using LRU if a page fault occurs, are as follows:

Sequence of Page Reference	Window Size		
	2	3	4
W			
A	A	A	A
B	A B	A B	A B
B	•	•	•
C	B C	A B C	A B C
A	C A	•	•
E	A E	A C E	A B C E
D	E D	A E D	A C E D
B	D B	E D B	A E D B
D	•	•	•
E	D E	•	•
A	E A	E D A	•
C	A C	E A C	A E D C
E	C E	•	•
B	E B	E C B	A E C B
A	B A	E B A	•
C	A C	B A C	•
A	•	•	•
F	A F	A C F	A C F
D	F D	F D	A F D
F	•	•	•

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

CHAPTER 9 UNIPROCESSOR SCHEDULING

ANSWERS TO QUESTIONS

9.1 Processor scheduling in a batch system, or in the batch portion of an OS, is done by a long-term scheduler. Newly submitted jobs are routed to disk and held in a batch queue from which the long-term scheduler creates processes and then places these in the ready queue so that they can be executed. In order to do this, the scheduler has to take two decisions:

a. When can the OS take on one or more additional processes?

This is generally determined by the desired degree of multiprogramming. The greater the number of processes created, the smaller the percentage of CPU time for each. The scheduler may decide to add one or more new jobs either when a job terminates or when the fraction of time for which the processor is idle exceeds a certain threshold.

b. Which job or jobs should it accept and turn into processes?

This decision can be based on a simple first-come-first-served (FCFS) basis. However, the scheduler may also include other criteria like priority, expected execution time, and I/O requirements.

9.2 A dispatcher, or a short-term scheduler, allocates processes in the ready queue to the CPU for immediate processing. It makes the fine-grained decision of which process to execute next. It has to work very frequently since, generally, a process is executed in the CPU for a very short interval.

The dispatcher is invoked whenever an event that may lead to the blocking of the current process occurs. It is also invoked when an event that may provide an opportunity to preempt a currently running process in favour of another occurs. Examples of such events include clock interrupts, I/O interrupts, operating system calls, and signals (e.g., semaphores).

9.3 The scheduling criteria that affect the performance of the system are:

a. Turnaround Time: This is the total time that has elapsed between the submission of a process and its completion. It is the sum of the

following: time spent waiting to get into the memory, time spent waiting in the ready queue, the CPU time, and time spent on I/O operations.

- b. Response Time:** For an interactive process, this is the time from the submission of a request to when the response begins to be received. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving an acceptable response time.
- c. Waiting Time:** This is defined as the total time spent by a job while waiting in the ready queue or in the suspended queue in a multiprogramming environment.
- d. Deadlines:** When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.
- e. Throughput:** This is defined as the average amount of work completed per unit time. The scheduling policy should attempt to maximize the number of processes completed per unit of time. This clearly depends on the average length of a process, but it is also influenced by the scheduling policy, which may affect utilization.
- f. Processor utilization:** This is defined as the average fraction of time the processor is busy executing user programs or system modules. Generally, the higher the CPU utilization, better it is. This is a significant criterion for expensive shared systems. In single-user systems, and in other systems like real-time systems, this criterion is less important than some of the others.

9.4 In pure priority-based scheduling algorithms, a process with a higher priority is always selected at the expense of a lower-priority process. The problem with a pure priority scheduling scheme is that lower-priority processes may suffer starvation. This will happen if there is always a steady supply of higher-priority ready processes. If the scheduling is nonpreemptive, a lower priority process will run into completion if it gets the CPU. However, in preemptive schemes, a lower priority process may have to wait infinitely in the ready or suspended queue.

9.5 Advantages: **1.** It ensures fairness to all processes regardless of their priority in most cases. **2.** It reduces the monopolization of the CPU by a large process. **3.** It increases the scheduling capacity of the system. **4.** Depending on the CPU time the process needs, it also gives a quick response time for processes. **Disadvantages:** **1.** Preemption is associated with extra costs due to increased context-switch, increased caching, increased bus-related costs, and the like. **2.** Preemptive scheduling results in increased dispatcher activities and, subsequently, more time for dispatching.

- 9.6** As each process becomes ready, it joins the ready queue. When the currently-running process ceases to execute, the process that has been in the ready queue the longest is selected for running.
- 9.7** A clock interrupt is generated at periodic intervals. When the interrupt occurs, the currently running process is placed in the ready queue, and the next ready job is selected on a FCFS basis.
- 9.8** This is a nonpreemptive policy in which the process with the shortest expected processing time is selected next.
- 9.9** This is a preemptive version of SPN. In this case, the scheduler always chooses the process that has the shortest expected remaining processing time. When a new process joins the ready queue, it may in fact have a shorter remaining time than the currently running process. Accordingly, the scheduler may preempt whenever a new process becomes ready.
- 9.10** When the current process completes or is blocked, choose the ready process with the greatest value of R , where $R = (w + s)/s$, with w = time spent waiting for the processor and s = expected service time.
- 9.11** Scheduling is done on a preemptive (at time quantum) basis, and a dynamic priority mechanism is used. When a process first enters the system, it is placed in RQ0 (see Figure 9.4). After its first execution, when it returns to the Ready state, it is placed in RQ1. Each subsequent time that it is preempted, it is demoted to the next lower-priority queue. A shorter process will complete quickly, without migrating very far down the hierarchy of ready queues. A longer process will gradually drift downward. Thus, newer, shorter processes are favored over older, longer processes. Within each queue, except the lowest-priority queue, a simple FCFS mechanism is used. Once in the lowest-priority queue, a process cannot go lower, but is returned to this queue repeatedly until it completes execution.

ANSWERS TO PROBLEMS

9.1 a. Shortest Remaining Time:

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

Explanation: P1 starts but is preempted after 20ms when P2 arrives and has shorter burst time (20ms) than the remaining burst time of P1 (30 ms) . So, P1 is preempted. P2 runs to completion. At 40ms P3 arrives, but it has a longer burst time than P1, so P1 will run. At

60ms P4 arrives. At this point P1 has a remaining burst time of 10 ms, which is the shortest time, so it continues to run. Once P1 finishes, P4 starts to run since it has shorter burst time than P3.

Non-preemptive Priority:

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

Explanation: P1 starts, but as the scheduler is non-preemptive, it continues executing even though it has lower priority than P2. When P1 finishes, P2 and P3 have arrived. Among these two, P2 has higher priority, so P2 will be scheduled, and it keeps the processor until it finishes. Now we have P3 and P4 in the ready queue. Among these two, P4 has higher priority, so it will be scheduled. After P4 finishes, P3 is scheduled to run.

Round Robin with quantum of 30 ms:

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

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

b. Shortest Remaining Time: $(20+0+70+10)/4 = 25$ ms.

Explanation: P2 does not wait, but P1 waits 20ms, P3 waits 70ms and P4 waits 10ms.

Non-preemptive Priority: $(0+30+10+70)/4 = 27.5$ ms

Explanation: P1 does not wait, P2 waits 30ms until P1 finishes, P4 waits only 10ms since it arrived at 60ms and it is scheduled at 70ms. P3 waits 70ms.

Round-Robin: $(20+10+70+70)/4 = 42.5$ ms

Explanation: P1 waits only for P2 (for 20ms). P2 waits only 10ms until P1 finishes the quantum (it arrives at 20ms and the quantum is 30ms). P3 waits 30ms to start, then 40ms for P4 to finish. P4 waits 40ms to start and one quantum slice for P3 to finish.

9.2 If the time quantum q is large, round robin scheduling becomes equivalent to FCFS scheduling and thus performance degrades. If q is

small, the number of context switches increases and q almost equals the time taken to switch the CPU from one process to another. The system wastes nearly half of its time context switching, thus degrading the overall system performance. Hence, the major criteria for determining the size of a time quantum are the time spent in context switching and the burst times of the processes.

GANTT chart for time quantum $q = 2$ ms

P1	P2	P1	P2	P3	P4	P1	P2	P3	P4	P1	P3	P1	P3	P1	P3	P3	P3	P3	P3	P3
0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	

Turnaround time for a process = Total CPU time + Total context switch time

Context switch time = Number of context switches in the interval \times time for 1 context switch

Turnaround time for P1 = $(30 - 0) + 14 \times 1 = 44$ ms

Turnaround time for P2 = $(16 - 2) + 6 \times 1 = 20$ ms

Turnaround time for P3 = $(40 - 8) + 15 \times 1 = 47$ ms

Turnaround time for P4 = $(20 - 10) + 4 \times 1 = 14$ ms

GANTT chart for time quantum $q = 4$ ms

P1	P2	P3	P4	P1	P2	P3	P1	P3	P3	P3
0	4	8	12	16	20	22	26	30	34	38

Turnaround time for a process = Total CPU time + Total context switch time

Context switch time = Number of context switches in the interval \times time for 1 context switch

Turnaround time for P1 = $(30 - 0) + 7 \times 1 = 37$ ms

Turnaround time for P2 = $(22 - 2) + 4 \times 1 = 24$ ms

Turnaround time for P3 = $(40 - 8) + 8 \times 1 = 40$ ms

Turnaround time for P4 = $(16 - 10) + 1 \times 1 = 7$ ms

Average turnaround time = $(37 + 24 + 40 + 7)/4 = 27$ ms

GANTT chart for time quantum $q = 8$ ms

P1	P2	P3	P4	P1	P3
0	8	14	22	26	30

Turnaround time for a process = Total CPU time + Total context switch time

Context switch time = Number of context switches in the interval \times time for 1 context switch

Turnaround time for P1 = $(30 - 0) + 4 \times 1 = 34$ ms

Turnaround time for P2 = $(14 - 2) + 1 \times 1 = 13$ ms

Turnaround time for P3 = $(40 - 8) + 4 \times 1 = 36$ ms

Turnaround time for P4 = $(26 - 10) + 2 \times 1 = 18$ ms

Average turnaround time = $(34 + 13 + 36 + 18)/4 = 25.25$ ms

9.3 As the system allows only nonpreemptive schedules, the maximum number of schedules is nothing but the arrangement of n processes or the n -permutations of n objects, nP_n . Hence, the total number of schedules is equal to $n!$.

9.4 The data points for the plot:

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

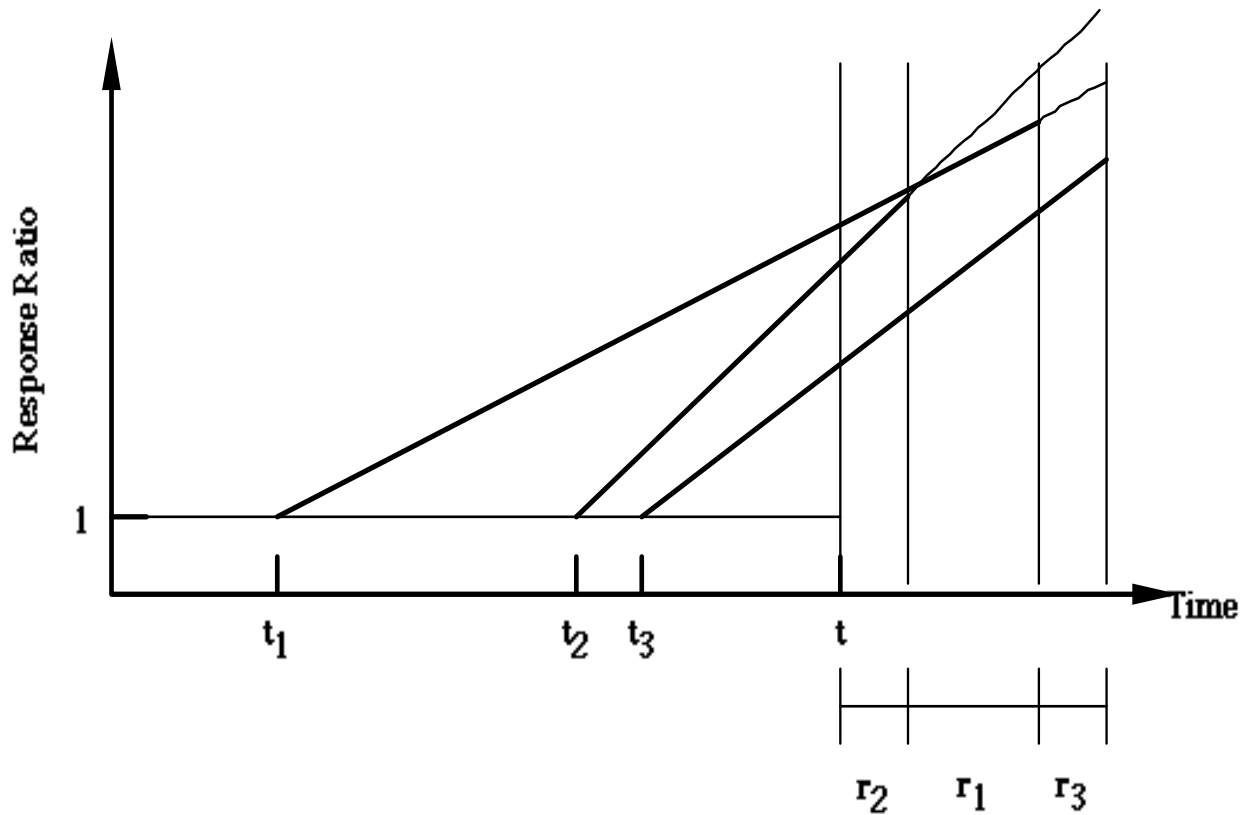
9.5 The first equation is identical to Equation 9.3, so the parameter α provides an exponential smoothing effect. The parameter β is a delay variance factor (e.g., 1.3 to 2.0). A smaller value of β will result in faster adaptation to changes in the observed times, but also more fluctuation in the estimates.

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

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

9.7 First, the scheduler computes the response ratios at time $t + r_1 + r_2 + r_3$, when all three jobs will have been finished (see figure). At that time, job 3 will have the smallest response ratio of the three: so the scheduler decides to execute this job last and proceeds to examine jobs 1 and 2 at time $t + r_1 + r_2$, when they will both be finished. Here the response ratio of job 1 is the smaller, and consequently job 2 is selected for service at time t . This algorithm is repeated each time a job is completed to take new arrivals into account. Note that this algorithm is

not quite the same as *highest response ratio next*. The latter would schedule job 1 at time t . Intuitively, it is clear that the present algorithm attempts to minimize the maximum response ratio by consistently postponing jobs that will suffer the least increase of their response ratios.



9.8 Consider the queue at time t immediately after a departure and ignore further arrivals. The waiting jobs are numbered 1 to n in the order in which they will be scheduled:

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

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

$$T_i = t + r_1 + r_2 + \dots + r_i$$

and job i has the response ratio

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

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

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

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

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

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

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

and job j reaches the response ratio

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

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

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

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

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

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

Notice that this proof is valid in general for priorities that are non-decreasing functions of time. For example, in a FIFO system, priorities

increase linearly with waiting time at the same rate for all jobs. Therefore, the present proof shows that the FIFO algorithm minimizes the maximum waiting time for a given batch of jobs.

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

Consider the exponential probability distribution function:

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

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

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

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

So,

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

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

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

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

currently in service and T_s , as we have demonstrated, is the expected remaining service time.

Thus, we have

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

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

$$x = m\delta$$

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

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

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

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

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

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

Thus,

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

- 9.11** Burst time of process A = 100 ms
 Burst time of process B = 120 ms
 Burst time of process C = 60 ms
 Analysis of system with NRR scheduling:

GANTT chart

A	B	C	A	B	C	A	B	A	B	B	B
0	40	80	120	150	180	200	220	240	250	260	270

Turnaround time for A = 240 ms
 Turnaround time for B = 280 ms
 Turnaround time for C = 200 ms
 Average turnaround time = $720/3 = 240$ ms

Waiting time for A = $240 - 100 = 140$ ms
 Waiting time for B = $280 - 120 = 160$ ms
 Waiting time for C = $200 - 60 = 140$ msec
 Average waiting time = $440/3 = 146.67$ ms

In this system, the CPU time of newer processes entering the system is greater. Thus, processes that are interactive and I/O-bound get more CPU time than CPU bound processes. Hence, a relative priority that prefers processes with less CPU time is automatically set. The turnaround time of these processes is reduced as they waste less time in context switches, which is an advantage over the normal round-robin scheduling.

The two disadvantages of this system are: **1.** it requires the overhead of an added logic in each process to maintain its time quantum; **2.** if there is a steady influx of short processes in the system, a long job may have to wait for a very long time for its completion.

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

$$y = \beta/\lambda\alpha$$

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

which therefore gives

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

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

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

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

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

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

$$R_x = W + V_x$$

From problem 9.10, we have

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

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

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

Thus,

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

which yields the desired result for R_x .

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

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

9.15 a. Given $\alpha < \beta < 0$, since both α and β are negative, the priorities decrease with time. The new process entering the system has the highest priority and so it preempts the running process and starts executing. Further, the rate of decrease of priority in the ready queue is more than the rate of decrease of priority in the running queue. Hence, the processes that have just arrived in the ready queue have greater priority over processes that were in the queue earlier. Processes that have got the CPU time have greater priority over processes that have not. All these observations imply that last-in-first-out scheduling will be observed in this case.

b. Given $\beta > \alpha > 0$, when α and β have positive values, the priorities of processes increase with the time they have been in the system. Thus, the process entering the system first will get the CPU first. Also, since the rate of increase of priority of a running process is greater than the rate of increase of priority of a waiting process, a process that has substantial CPU time will get the CPU again because of its increased priority. Hence, eventually, first-come-first-served scheduling will occur.

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

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

The turnaround time for each process:

A = 45 min, B = 35 min, C = 13 min, D = 26 min, E = 42 min

The average turnaround time is $= (45+35+13+26+42) / 5 = 32.2$ min

b.

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

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

c.

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

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

d.

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

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