SOLUTIONS TO PRACTICE PROBLEMS

OPERATING SYSTEMS: INTERNALS AND DESIGN PRINCIPLES

FIFTH EDITION

WILLIAM STALLINGS

TABLE OF CONTENTS

CHAPTER 1 COMPUTER SYSTEM OVERVIEW

1.1 **In hardware:** device sends signal (voltage) on IRQ line. The signal causes a bit to be flipped in the interrupt register. At the end of an instruction cycle, the interrupt register is checked (in priority order) and, if a bit is on, the hardware places the value currently in the PC (typically) on the system stack and goes to the interrupt vector, at the location matched to the interrupt register, to get the address of the ISR. This address is placed in the PC register.

In software: the ISR begins to execute. It will save values in registers that it will need, perhaps on the stack, perhaps in the previous process's PCB. It may disable interrupts long enough to save these values. It may have to identify one of several devices using that IRQ line (if devices share a signal). It will handle the interrupt. It may restore the interrupted process's register values (note that sometimes processes are terminated, etc.).

1.2 **a.**

Addresses	**Contents**
0x0000B128	0x0200EC00
0x0000B12C	0x0300EC04
0x0000B130	0x0100EC08

(1st byte: opcode (e.g., 0x02), remaining 3 bytes are address of data)

……

0x0000EC00	0x00000016 ; (a=22=0x16)
0x0000EC04	0x0000009E ; (b=158=0x9E)
0x0000EC08	0x00000000 ; (c=0=0x00, or it can be anything)

b.

Instruction	**Contents**
PC → MAR	0x0000B128
M → MBR	0x0200EC00
MBR → IR	0x0200EC00
IR → MAR	0x0000EC00
M → MBR	0x00000016
MBR → AC	0x00000016
PC → MAR	0x0000B12C
M → MBR	0x0300EC04
MBR → IR	0x0300EC04
IR → MAR	0x0000EC04
M → MBR	0x0000009E
MBR + AC → AC	0x00000B4
PC → MAR	0x0000B130
M → MBR	0x0100EC08
MBR → IR	0x0100EC08
IR → MAR	0x0000EC08
AC → MBR	0x000000B4
MBR → M	0x000000B4

1.3 EAT = .9 (10) + .1 [.8 (10 + 100) + .2 (10 + 100 + 10000)] = 220ns
OR
EAT = 10 + .1 (100) + .02 (10000) = 220ns

1.4 **a.** Since the targeted memory module (MM) becomes available for another transaction 600 ns after the initiation of each store operation, and there are 8 MMs, it is possible to initiate a store operation every 100 ns. Thus, the maximum number of stores that can be initiated in one second would be:

$$10^9 / 10^2 = 10^7 \text{ words per second}$$

Strictly speaking, the last five stores that are initiated are not completed until sometime after the end of that second, so the maximum transfer rate is really $10^7 - 5$ words per second.

b. From the argument in part (a), it is clear that the maximum write rate will be essentially 10^7 words per second as long as a MM is free in time to avoid delaying the initiation of the next write. For clarity, let the module cycle time include the bus busy time as well as the internal processing time the MM needs. Thus in part (a), the module cycle time was 600 ns. Now, so long as the module cycle time is 800 ns or less, we can still achieve 10^7 words per second; after that, the maximum write rate will slowly drop off toward zero.

CHAPTER 2 OPERATING SYSTEM OVERVIEW

2.1 **a.** I/O-bound processes use little processor time; thus, the algorithm will favor I/O-bound processes.

b. if CPU-bound process is denied access to the processor
==> the CPU-bound process won't use the processor in the recent past.
==> the CPU-bound process won't be permanently denied access.

2.2 **a.** The time required to execute a batch is $M + (N \times T)$, and the cost of using the processor for this amount of time and letting N users wait meanwhile is $(M + (N \times T)) \times (S + (N \times W))$. The total cost of service time and waiting time per customer is

$$C = (M + (N \times T)) \times (S + (N \times W))/N$$

The result follows by setting $dC/dN = 0$

b. \$0.60/hour.

2.3 The countermeasure taken was to cancel any job request that had been waiting for more than one hour without being honored.

2.4 The problem was solved by postponing the execution of a job until all its tapes were mounted.

2.5 An effective solution is to keep a single copy of the most frequently used procedures for file manipulation, program input and editing permanently (or semi-permanently) in the internal store and thus enable user programs to call them directly. Otherwise, the system will spend a considerable amount of time multiple copies of utility programs for different users.

CHAPTER 3 PROCESS DESCRIPTION AND CONTROL

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

3.2
```
0
<child pid>
```
or
```
<child pid>
0
```

3.3 At time 22:
- P1: blocked for I/O
- P3: blocked for I/O
- P5: ready/running
- P7: blocked for I/O
- P8: ready/running

At time 37
- P1: ready/running
- P3: ready/running
- P5: blocked suspend
- P7: blocked for I/O

P8: ready/running
At time 47
- P1: ready/running
- P3: ready/running
- P5: ready suspend
- P7: blocked for I/O
- P8: exit

3.4 ==> exponential growth of processes occurs

- only a finite number of process ID's on systems
- only a finite amount of memory on systems
    however, since the size of created processes is small and since the OS has lots of swap space available
        ==> above code will most likely exhaust process table (run out of IDs) instead of run out of memory

- most systems, OS will run out of process IDs and then error "can't create a process"
    user/root can't kill the malicious process (kill needs to create a new process)
    user/root can't ps the processes in the system (ps needs to create a new process)
        ==> only choice is to re-boot the system

- some systems, OS will actually crash
- some systems, user has a pre-specified limit on the number of processes he/she can create (a long-term scheduler)
        ==> user process won't be allowed to take the system down

3.5 We need to keep the process switch time as short as possible, so keeping the part of the OS that deals with context switches always in a fixed location in memory (rather than bringing it in from disk every time or even on occasions), means that the OS instructions will execute faster and the process switch time will be predictable.

CHAPTER 5 CONCURRENCY: MUTUAL EXCLUSION AND SYNCHRONIZATION

5.1

Dispatcher 1	**Dispatcher 2**
Get ptr to next process to execute	Get ptr to next process to execute
Update pointer	Update pointer
Execute process	Execute process

Execute the above sequentially - no problem with consistency
Now interleave the first instruction on both dispatchers.
The processors will execute the same process and one process will be skipped.

5.2 Mutual exclusion: If all three processes try to access the resource concurrently, two processes will enter Spago's due to AND criteria (i.e., sign can only be ONE value at a time). A social disaster!

5.3 **1.** Provide mutual exclusion?

There are two cases to consider:

a. A process is inside the critical section and another tried to enter: Without loss of generality, assume Penelope is inside the critical section and Nicole tries to enter. Before entering the critical section Penelope sets her own flag to 1. When Nicole tries to enter the critical section she will see that Lope is up and will get caught in the while loop. Nicole will continue in the while loop until Penelope lowers her flag, which happens only at the end of the critical section

b. Both are trying to enter simultaneously: In this situation, if both reach their respective while loop at the top, then the SIGN will ensure that only one of them passes through. The SIGN is alternating between the two of them, and is only modified at the exit of a critical section.

2. No Deadlock?

Suppose both are trying to enter simultaneously. In this case if the first is trapped into the while loop, then the SIGN will make one of the two women lower her flag and go into a loop waiting for the SIGN to change (the inner while loop). The other woman whose turn is set by the SIGN will be able to get into Spago's.

3. No Starvation?

Assume one is blocked inside the inner while loop, while the other is in the critical section In such a case, if the one inside the critical section tries to re-enter, she will be blocked because on exit of the critical section she sets the SIGN to point to the other. Therefore, the one that just got out of the critical section will be forced to wait for her own turn. So, bounded waiting is taken care of.

4. Progress?

Suppose one of them is trying to enter with no competition: In such a case, the flag of the other is down, and thus she can enter.

In summary, ALL requirements are SATISFIED

5.4 The operation does not always cause a wait; when semaphore value is > 0. NO actual waiting occurs

5.5
```
#include <pthread.h>
pthread_mutex_t countlock = PTHREAD_MUTEX_INITIALIZER;

static int count = 0;
int increment(void)
{
    pthread_mutex_lock(&countlock);
    count++;
    if (count > 5) {
        printf(°∞counter %d reached value > 5°±, count);
        pthread_mutex_unlock(&countlock);
        return 0;
    }
    pthread_mutex_unlock(&countlock);
    return 1;
}

int decrement(void)
{
    pthread_mutex_lock(&countlock);
    while (count >5) {
    printf(°∞counter %d is > 5:, count);
    count --;
    }
    if (count ==0) {
        pthread_mutex_unlock(&countlock);
        return 0;
    }
    else {
        pthread_mutex_unlock(&countlock);
        return 1;
    }
}
```

5.6 **Identify processes:**
i. The generic teller process, parameterized by type (quick or normal service) and number
ii. The generic customer service, parameterized by type (quick or normal service) and number

Identify variables:
quick: teller or customer is quick service
number: number of the teller or customer
tn: teller number to serve this customer
t[i]: the ith teller
c[j]: the jth customer
Qserve: the current quick service customer to be served

Nserve: the current normal service customer to be served
mutexQ: semaphore for mutual exclusion on queue computations

The first two are part of each teller record, the first three are part of each customer record. The others are global.

Identify events We could approach these from either the teller point of view, or the customer point of view. The latter is preferable, as each customer transaction is unique, while teller transactions are not. Hence we will not have to worry about synchronizing with the wrong event.
—A customer arrives in the bank and joins a queue
—A customer is called to the teller
—A customer arrives at a teller window
—A customer completes a transaction and leaves the bank
We create a semaphore for each of these (except the first: that is assumed to be outside our control), as part of each customer record.

Write customer process We write this first, since the customer is the driving process for the tellers. The customer process is characterized by a) the type of customer (normal or quick), and b) the number of the customer

```
class customer(Thread):
   def init(self,quick,number):
      self.quick=quick; self.number=number
      self.tn = 0
      self.call=semaphore()
      self.customer_arrive=semaphore()
      self.transaction_complete=semaphore()
   def isQuick(self): return self.quick
   def run(self):
      self.call.wait()               # wait for our turn
      goto_teller(self.tn)           # teller number tn is
our teller
      elf.customer_arrive.signal()   # tell teller we're here
      do_transaction()
      self.transaction_complete.signal()  # tell teller we're
                                          # not here
      leave_bank
```

Write teller process The teller process is characterized by a) the type of teller (normal or quick), and b) the number of the teller

```
class teller(Thread):
 def run(self,quick,number):
   global mutexQ,c,Qserve,Nserve
    while (1):
     mutexQ.wait()
     if quick:
      while Qserve < j and not c[Qserve].isQuick:
       Qserve=Qserve+1
      if not c[Qserve].isQuick:
       Nserve=Nserve+1; Serving = Nserve
```

```
        else:
          Serving = Qserve
      else:
        while Nserve < j and c[Nserve].isQuick:
          Nserve=Nserve+1
        if c[Nserve].isQuick:
          Qserve=Qserve+1; Serving = Qserve
        else:
          Serving = Nserve
      mutexQ.signal()
      # "Now serving customer number 'Serving'"
      c[Serving].tn = number                  # flag our number
      c[Serving].call.signal()                # tell the customer
      c[Serving].customer_arrive.wait()       # wait for her
      c[Serving].transaction_complete.wait()# and her transaction
```

The if quick: ... statement does all the queue calculations. We scan forward on the list of customers, looking for customers of our type. If we find one, that is the next customer to be served. If there are none, then the next customer in the opposite type of queue is to be served. Since this is updating shared variables, it must be a mutual exclusion zone.

write bank process (This is really the main program.)

```
Qserve = 0                     # customer number for quick service Q
Nserve = 0                    # customer number for normal service Q

t = n*[0]                                              # list of tellers
for i in range(1,n):                   # create and start all tellers
   if i < k:
      t[i] = teller(1,i)                # create quick service teller
   else:
      t[i] = teller(0,i)               # create normal service teller
   t[j].start()                                          # start teller

j = 0                                                  # customer number
c = []                                                   # customer list
while 1:                       # create and start customers forever
   j = j+1
   x = random()
   if x < QuickRatio:          # QuickRatio is fraction of customers
                                             # wanting quick service
      c.append(customer(1,j))     # create a quick service customer
    else:
      c.append(customer(0,j))    # create a normal service customer
   c[j].start()                                        # start customer
                              # now customer[j] is implicitly on one
                                              # of the service queues
   wait_random_interval()              # for next customer to arrive
```

Add critical sections There is only one: the queue computation to see who is next to be served. Identified above by the mutexQ variable.

5.7 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.

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

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

"x is 10" is printed.

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

```
Instruction                          M(x)    P1-R0   P2-R0
P1: LD   R0, x                       10      10      --
P1: DECR R0                          10      9       --
P1: STO  R0, x                       9       9       --
`
P2: LD   R0, x                       9       9       9
P2: DECR R0                          9       9       8
P2: STO  R0, x                       8       9       8

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

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

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

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

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

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

5.9 Here the solution is simple: enclose the operations on the shared variable within semaphore operations, which will ensure that only one process will operate on x at a time. The only trick here is to realize that we have to enclose the if statement as well since if we do not, erroneous printing can still happen if one process is in the middle of the critical section while another is testing x.

```
s: semaphore;

parbegin
P1: {
        shared int x;
        x = 10;
        for (; ;) {
                semWait(s);
                x = x - 1;
                x = x + 1;
                if (x != 10)
                  printf("x is %d", x);
                semSignal(s);
        }
}

P2: {
        shared int x;
        x = 10;
        for (; ;) {
                semWait(s);
                x = x - 1;
                x = x + 1;
                if(x != 10)
                  printf("x is %d", x);
                semSignal(s);
        }
}
parend
```

5.10 Here the essential point is that without an atomic operation to test and set the semaphore variable, the only way to ensure that the semaphore manipulations will not be interrupted and thus potentially corrupted, is to create a system call which blocks all interrupts. That way, after the spl(highest) we know that nothing will interrupt execution until the priority is set back to the previous value. The sleep and wakeup calls are used to avoid busy waiting in the kernel. A busy waiting solution was declared acceptable since the point of the question was to use spl as the way to ensure atomicity. However, if used, it will not actually work, because the machine will be trapped in an uninterruptible loop waiting for the semaphore to be released. Note that key(s) is meant to symbolize creating a unique integer to represent the semaphore in question.

```
semWait(s, val)

  int old;

  while (1) {
    old = spl(highest);
    if ( s < val ) {
      spl(old);
      /* we could busy wait here, but would block the kernel */
      sleep(key(s));
      continue;
    } else {
      s = s - val;
      spl(old);
    }
  }


semSignal(s, val)

  int old;

  old = spl(highest);
  s = s + val;
  spl(old);
  wakeup(key(s));
```

5.11 To move the statement inside the critical section, but as late as possible, the statement would occur immediately after `n--`. But at this point, n = 0, therefore, `consumer` will not wait on semaphore `delay`. This means that consumer will not issue `semSignalB(s)`. Therefore, `n` remains at 1. The producer therefore cannot issues a `semSignalB(delay)` and can get hung up at its statement `semWaitB(s)`. Thus, both processes are waiting and deadlocked.

CHAPTER 6 CONCURRENCY: DEADLOCK AND STARVATION

6.1 P1 can complete, and release, allowing P0 to complete. But neither P2 or P3 or P4 can complete. System is not safe.

6.2 Available: A = 1; B = 2
User 1 stills needs (8 2)
User 2 stills needs (5 2)
User 3 stills needs (2 2)
User 4 stills needs (2 3)

The algorithm would not have allowed these allocations, since none of the processes are guaranteed to be able to complete.

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

b. Need Matrix = Max Matrix – Allocation Matrix

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

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

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

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

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

Then the new need matrix would be

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

And Available is then:

Available			
A	B	C	D
3	1	2	1

Which means I could NOT satisfy ANY process' need.

6.4 a. Concurrency ratings In order from most-concurrent to least, here 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, since 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. Efficiency ratings. In order from most efficient to least, here is a rough partial order on the deadlock-handling algorithms:

1. reserve all resources in advance; resource ordering. These algorithms are most efficient because they involve no runtime overhead. Notice that this is a result of the same static restrictions that made these rank poorly in concurrency.

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

3. detect deadlock and roll back thread's actions. This algorithm performs the same runtime check discussed previously but also entails a logging cost which is O(n) in the total number of memory writes performed.

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

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

6.5 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.

CHAPTER 7 MEMORY MANAGEMENT

7.1 MVT uses the remainder of the hole that is left during allocation, otherwise Worst-fit makes no sense whatsoever. The concept of Worst-fit is that allocation leaves a large enough hole for another process, while best fit leaves a small fragment. Thus, using worst-fit, 20,30, 10 and 100 all go into the 200k partition (i.e., what is left each time) and there is no room for 60k. Using first fit, 20 and 30 go into the 50k partition, 10k into the next 30k, 100 into 200k and 60 into the 100k hole that is left.

7.2 **a.** 1219 + 430 **b.** illegal **c.** 90 + 50 **d.** 2327 + 400 **e.** illegal

7.3 **a.** 100K
500K holds 417K
200K holds 112K
300K holds 212K
600K
350K waits for 500K partition to become free

b. 100K
500K holds 417K
200K holds 112K
300K holds 212K
600K holds 350K

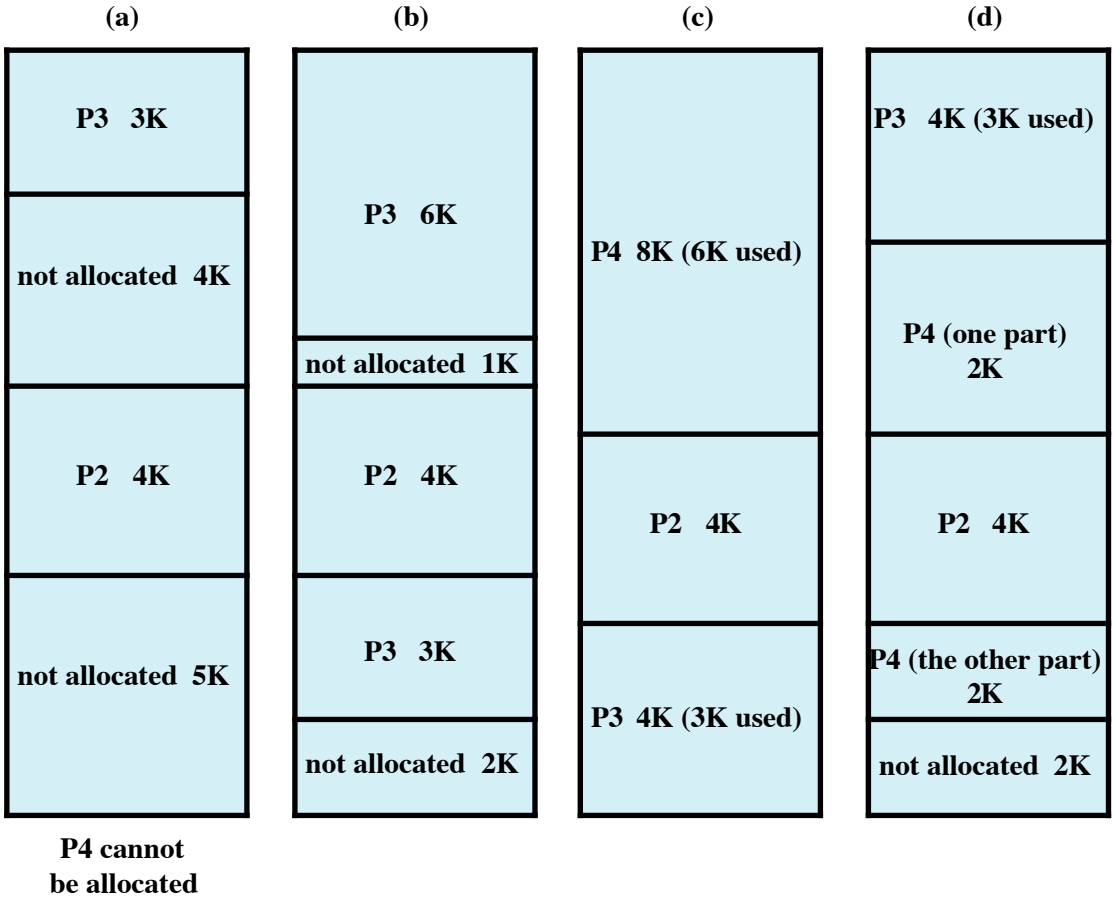
c. internal after a.: (500K – 417K) + (200K – 112K) + (300K – 212K) = 259K
d. external after b.: zero, since no request is pending

7.4 **a.** $8 \times 1024 = 8192 = 2^{13}$
13 bits needed for the logical address
b. $32 \times 1024 = 32{,}768 = 2^{15}$
15 bits needed for the physical address

7.5

Strategy	Base Address	Length
First fit	0	15
Best fit	30	15
Worst fit	64	15

7.6



7.7 12 bits

7.8 64 bits

CHAPTER 8 VIRTUAL MEMORY

8.1 (d) The system is obviously thrashing. You do not want to increase the number of processes competing for page frames. Installing a faster processor will not help; processor is underutilized as is. There is no indication that the current paging disk is inadequate. (A faster paging disk might be helpful.)

8.2 (e) Processes being unable to establish their working set of pages. A local page replacement algorithm may increase the probability of one process thrashing, but cannot be considered the cause of it. If a FIFO page replacement algorithm is ineffective, it will increase the probability of page thrashing, but (e) is the best answer.

8.3 **a.** 200 nsec + 200 nsec = 400 nsec
b. 75 (10 nsec + 200 nsec) + .25 (10 nsec + 200 nsec + 200 nsec) = about 250 nsec
TLB page table page
c. Add 0.7 (10 nsec + 200 nsec) + 0.2 (200 nsec + 200 nsec)
+ 0.1(10 nsec+ 200 nsec + 100 msec +10 nsec + 200 nsec + 200 nsec)
TLB memory disk TLB memory disk

8.4 **a.** logical address space = 32 bits
page number = 20 bits; offset = 12 bits
b. In TLB, need page number (20 bits), frame number (12)
==> 32 bits for each entry
==> 4 bytes for each entry
TLB = 2^6 bytes or 64 bytes
==> 16 entries
c. 2^{20} page table entries are needed for the 2^{20} pages
d. PROBLEM: page table is very large ... it won't fit on one page and the OS won't want to keep the whole table in memory at all times
SOLUTION: page the page table
==> leads to page faults for page table pages
==> more I/O swapping

8.5 **a.** EAT = 0.8 (TLB + MEM) + 0.2 (TLB + MEM + MEM)
EAT = 0.8 (20 + 75) + 0.2 (20 + 75 + 75)
EAT = 76 + 34 = 110 ns
b. EAT = 0.8 (TLB + MEM) + 0.2 (0.9(TLB + MEM + MEM) + 0.1 (TLB + MEM + 0.5 (DISK) + 0.5 (2 DISK) + MEM))
EAT = 0.8 (20 + 75) + 0.2 (0.9 (20 + 75 + 75) + 0.1 (20 + 75 + 0.5 (500000) + 0.5 (1000000) + 75))
EAT = 76 + 0.2 (153 + .1 (750170)) = 76 + 15034 = 15110 ns

8.6 **a.** OPT replacement (three frames are allocated to the process) - 7 page faults

	*	*	*			*				*		*		*				
	0	1	7	0	1	2	0	1	2	3	2	7	1	0	3	1	0	3
0	0	0	0	0	0	0	0	0	0	3	2	3	3	3	3	3	3	3
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		7	7	7	2	2	2	2	2	2	2	7	7	0	0	0	0	0

b. FIFO replacement (three frames are allocated to the process) - 12 page faults

	*	*	*			*	*	*		*	*	*	*	*	*			
	0	1	7	0	1	2	0	1	2	3	2	7	1	0	3	1	0	3
0	0	0	0	0	0	1	7	2	2	0	1	3	2	7	1	1	1	1
	1	1	1	1	1	7	2	0	0	1	3	2	7	1	0	0	0	0
		7	7	7	2	0	1	1	3	2	7	1	0	3	3	3	3	3

c. Pure LRU replacement (three frames are allocated to the process) - 9 page faults

	*	*	*			*				*		*	*	*	*			
	0	1	7	0	1	2	0	1	2	3	2	7	1	0	3	1	0	3
0	0	0	0	0	0	0	0	0	0	3	3	3	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	1	1	1	7	7	7	3	3	3	3
		7	7	7	2	2	2	2	2	2	2	2	2	0	0	0	0	0

d. Clock Policy (three frames are allocated to the process) - 12 page faults

	*	*	*			*	*	*		*	*	*	*	*	*			
	0	1	7	0	1	2	0	1	2	3	2	7	1	0	3	1	0	3
	>0/1	0/1	0/1	0/1	0/1	2/1	2/1	>2/1	2/1	3/1	3/1	>3/1	1/1	1/1	>1/1			
		1/1	1/1	1/1	1/1	>1/0	0/1	0/1	0/1	>0/0	2/1	2/1	>2/0	0/1	0/1			
			7/1	7/1	7/1	7/0	>7/0	1/1	1/1	1/0	>1/0	7/1	7/0	>7/0	3/1			

8.7 Assume that whenever a fault occurs, there are other processes to run, so that the processor never needs to wait for paging disk traffic. In that case, in each second of time, we will have N instructions executed, and P page faults, so that

$$N \times 1 \times 10^{-9} + P \times 20 \times 10^{-6} = 1$$

But we want the number of instructions per page fault, so we need to compute N/P:

$$N/P = (1/P - 20 \times 10^{-6}) \times 10^{9}$$

What is P? We know that each page fault on average causes 1 disk read and 0.5 disk writes, so the time taken by the disk to handle each "average" fault will be 300 + 0.5 x 300, or 450 μS. Hence

$$450 \times 10^{-6} \times P = 1$$
$$1/P = 450 \times 10^{-6}$$

Substituting in the equation for N/P above, we get

$$N/P = (450 \times 10^{-6} - 20 \times 10^{-6}) \times 10^{9} = (450 - 20) \times 10^{3} = 430{,}000$$

Answer: 1 page fault every 430,000 instructions

8.8 **a.** FIFO with 3 page frames: 10 page faults

*	*	*		*		*	*	*	*			*	*
1	2	3	2	6	3	4	1	5	6	1	6	4	2
1	1	1	1	2	2	3	6	4	1	1	1	5	6
	2	2	2	3	3	6	4	1	5	5	5	6	4
		3	3	6	6	4	1	5	6	6	6	4	2

b. FIFO with 4 page frames: 8 page faults

*	*	*		*		*	*	*					*
1	2	3	2	6	3	4	1	5	6	1	6	4	2
1	1	1	1	1	1	2	3	6	6	6	6	6	4
	2	2	2	2	2	3	6	4	4	4	4	4	1
		3	3	3	3	6	4	1	1	1	1	1	5
				6	6	4	1	5	5	5	5	5	2

c. LRU with 3 page frames: 10 page faults

*	*	*		*		*	*	*	*			*	*
1	2	3	2	6	3	4	1	5	6	1	6	4	2
1	1	1	1	2	2	3	3	4	1	1	1	1	6
	2	2	2	3	3	6	4	1	5	5	5	6	4
		3	3	6	6	4	1	5	6	6	6	4	2

d. LRU with 4 page frames: 9 page faults

*	*	*		*		*	*	*	*				*
1	2	3	2	6	3	4	1	5	6	1	6	4	2
1	1	1	1	1	1	2	3	3	4	4	4	4	4
	2	2	2	2	2	3	6	4	1	1	1	1	1
		3	3	3	3	6	4	1	5	5	5	5	6
				6	6	4	1	5	6	6	6	6	2

No, Belady's Anomaly does not occur, but we note that FIFO gives better performance with 4 frames than LRU, which is counter intuitive.

8.9 **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.

8.10 Estimating the number of page faults is most easily done by considering the loops from the inside out, and analyzing the cumulative effects of each layer. With this method, this solution will consider both sizes of matrices together.

In the innermost loop (k), each processor is accessing a row of Array A, which is only 1 page in the small case, and 10 pages in the large case. At any time, however, only 1 is needed for each Array B, however, during that loop refers to a whole slew of pages, since it is accessing a column, and every element in that column is on a different page. This will therefore generate many page faults. If 100 pages are reserved for A, then there would be 900 left for B to perform this loop. Once it was done, there would similarly be 100 page faults to store the values into each row of C. The need to access SIZE pages for B and 100 for C with only 900 available would result in (SIZE+100 - 900) page faults each time through -- 200 for the small case, 9200 for the large one.

The outer loops just repeat the inner loop SIZE*SIZE/100 times, yielding the overall estimate of:

SIZE = 1000 --> 200 × 1000 × 10 -> 2×10^6 page faults
SIZE = 10000 -> 9200 × 10000 * 100 -> 9×10^9 page faults

The mandatory page faults for getting data into memory in the first place (and for referring to array A) are negligible in comparison. To see the big picture, see what effect this has on the RAM miss rate. A non-clairvoyant scheme will generate a page fault on every single iteration of the innermost loop. Counting all the other memory requests (arrays A and C, the variables I,J,K, and the instructions), this probably means the RAM miss rate is greater than 1%. It doesn't matter if the instructions are cached, or if the manipulations for C are optimized -- the regular page fault will slow this program down a great deal. Try to minimize the number of page faults by modifying the program to account for them (but without changing the page replacement policy). Then estimate how many page faults the new program has. This program suffers from a lack of locality of reference. Many pages are replaced after very little use, only to be reloaded later. One option to consider limiting how far the innermost loop counts at any moment, to try to improve the likelihood of something not getting kicked out. This would require an additional loop, to determine which "chunk" of rows to do next. The J loop might be divided into two, one for 'small K' and one for larger K, which just looks at alternate halves of the array (page faulting the whole thing only for iterations of I). This would allow the smaller array multiplication to only need 8000 page faults for array K in the entire picture (+1000 for the others). One could extend this divide-and-conquer approach in both dimensions, and imagine that the 10,000x10,000 array is just 100 1000x1000 arrays. However, it should be noticed that the number of instructions of the form

```
c[i][j] = a[i][k] + b[k][j]
```

is $SIZE^3$ -- so there would have to be 1000 'smaller' multiplications (not just 100). Even so, using the numbers above, 1000 * (2×10^6) is better than 9×10^9. By improving the 1000 case as above, by chopping it in half, this could be improved to under 1000 × 10000. Examining the innermost loop, one can see that that loop must access many pages of B in sequence. Although all the processors may access the

same page of B at roughly the same time, they must load a new one at each iteration. One way to force spatial locality on array B is simply to change the layers of the loops, swapping j and k:

```
            for (k=0; k<SIZE; k++)
                for (j=0; j<SIZE; j++)
                    {
                    if (k==0)  c[i][j] = 0;
                    c[i][j] += a[i][k] * b[k][j];
                    }
```

```
{ a[0][0], a[10][0], a[20][0], .... a[990][0] }  one access
{ b[0][0], b[ 0][0], b[ 0][0], .... b[  0][0] }   for each
{ a[0][0], a[10][0], a[20][0], .... a[990][0] }   processor
{ b[0][1], b[ 0][1], b[ 0][1], .... b[  0][1] }
```
etc.

Now, the innermost loop is J, which just uses one row each of each array. The K loop will then try to fit the entirety of array B into the 800 pages available. One could get a similar effect by transposing B. This shows a definite improvement in the use of array B, since now it accesses only a single row for the entire duration of the innermost loop, fetching a new row as J iterates. Each iteration of J, then uses 100 pages each of A and C (repeatedly) and a new row for B. The J loop itself would then try to fit all of the pages of B (SIZE*SIZE/1000) into the 800 not being used by A and C. Hence:

SIZE = 1000 --> (1000–800) × 10 -> 2000 page faults (for B)
  + 3×1000 mandatory page faults = 5000
SIZE = 10000 -> (100000-800) × 100 -> 1×10^7 page faults

One could alternatively notice that array A has poor locality of reference in light of the parallel processors. Perhaps if they all referred to elements of the same page instead of all different ones, things could be improved. Let each processor do a column instead, and compute:

```
          for (i=0; i<SIZE; i++)
              for (j=ProcN ; j<SIZE; j += mult)
                  {
                  temp = 0;
                  for (k=0; k<SIZE; k++)
                      temp += a[i][k] * b[k][j];
                  c[i][j] = temp;
                  }
          }
```

```
{ a[0][0], a[0][ 0], a[0][ 0], .... a[0][  0] }  one access
{ b[0][0], b[0]l10], b[0][20], .... b[0][990] }   for each
{ a[0][1], a[0][ 1], a[0][ 1], .... a[0][  1] }    processor
{ b[1][0], b[1][10], b[1][20], .... b[1][990] }
```
etc.

In this form, only one row is needed for arrays A and C through all of the iterations of J and K. In this form, The K loop cycles through

SIZE pages of B, with 998 available, yielding:
SIZE = 1000 --> (1000–998) × 1000 × 10 -> 2×10^4 page faults
SIZE = 10000 -> (10000–998) × 10000 * 100 -> 9×10^9 page faults

Transposing B in this picture would access 100 rows, instead of so many pages. 100 pages needed for B for the k loop, the entirety of B through the J loop (still with only 2 pages needed for A and C)...

SIZE = 1000 --> (1000–998) × 10 -> 20 page faults (for conflict in B)
+ 3×1000 mandatory page faults = 3000
SIZE = 10000 -> (100000–998) × 100 -> 1×10^7 page faults

At this point, it might be feasible to reconsider the subdivision of the array that was mentioned before, where it was estimated that the large case should 'only' require 1000 times as many page faults as the smaller. If so, it should be possible to design a program that has fewer than 3 * 10^6 page faults. This would probably combine several of the above effects. It is natural to question the validity of using Optimal Page Replacement as a measurement for this exercise. Indeed, it may simplify the calculations, but it probably does not indicate what a real system is likely to do. One could certainly analyze this in terms of a FIFO replacement scheme, which might produce a 'worst case' kind of result. But this certainly shows that an applications programmer can still exercise some control over the use of memory.

8.11 **a.**

0	0	1	1	0	3	1	2	2	4	4	3

b. Page fault rate = 50%

F		F			F		F		F		F

c. Page fault rate = 58%

F		F			F	F	F		F		F

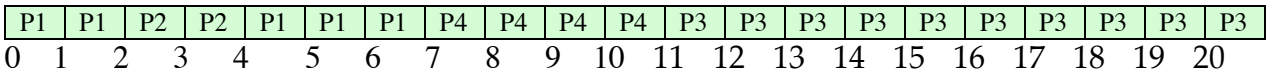
d. Page fault rate = 42%

F		F			F		F		F		

8.12 **a.** $2^{24} / 2^8 = 2^{16} = 65536$ pages
b. $2^{18} / 2^8 = 2^{10} = 1024$ pages
c. virtual: 7FA4F1 physical: 3B2F1
d. virtual: 7FA314 physical: none (page not loaded in PM)
e. virtual: none (invalid address) physical: none

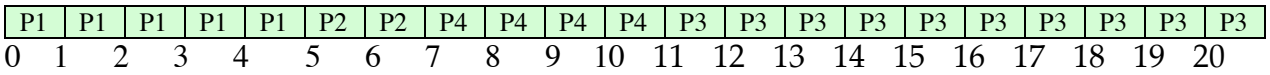
CHAPTER 9 UNIPROCESSOR SCHEDULING

9.1 a. Shortest Remaining Time:



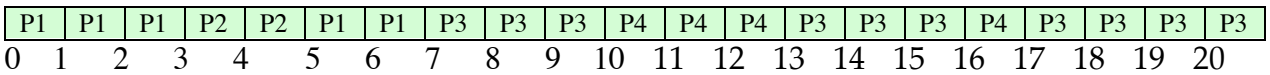
Explanation: P1 starts but is preempted after 20ms when P2 arrives and has shorter burst time (20ms) than the remaining burst time of P1 (30 ms) . So, P1 is preempted. P2 runs to completion. At 40ms P3 arrives, but it has a longer burst time than P1, so P1 will run. At 60ms P4 arrives. At this point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



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:

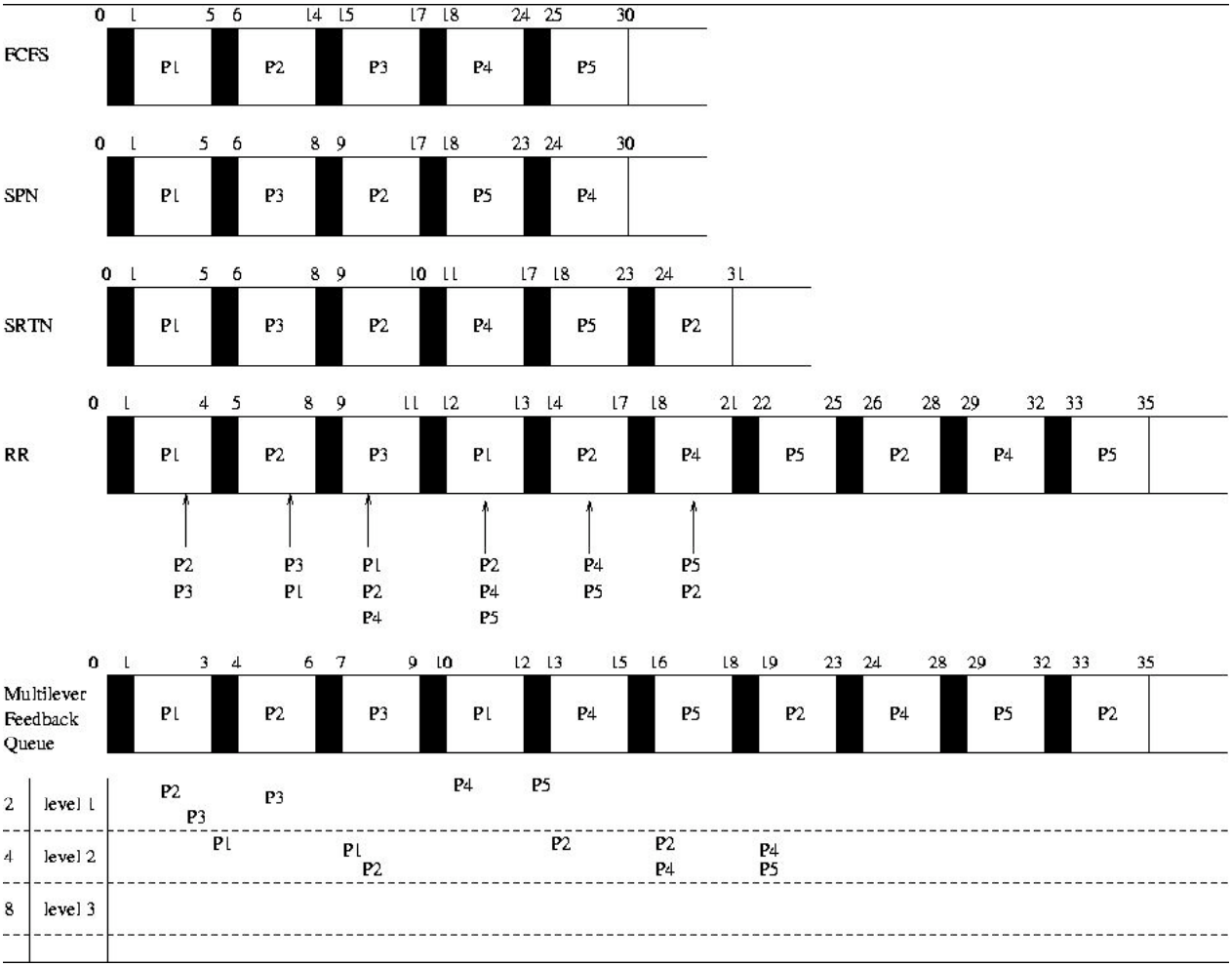


Explanation: P1 arrives first, so it will get the 30ms quantum. After that, P2 is in the ready queue, so P1 will be preempted and P2 is scheduled for 20ms. While P2 is running, P3 arrives. Note that P3 will be queued after P1 in the FIFO ready queue. So when P2 is done, P1 will be scheduled for the next 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 = 25ms.
Explanation: P2 does not wait, but P1 waits 20ms, P3 waits 70ms and P4 waits 10ms.
Non-preemptive Priority: (0+30+10+70)/4 = 27.5ms
Explanation: P1 does not wait, P2 waits 30ms until P1 finishes, P4 waits only 10ms since it arrived at 60ms and it is scheduled at 70ms. P3 waits 70ms.
Round-Robin: (20+10+70+70)/4 = 42.5ms

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

9.3 The following figure shows the timing charts for each scheduling method.



a. FCFS

turnaround = [(5-0) + (14-1) + (17-3) + (24-10) + (30-12)] / 5 = 64 / 5 = 12.8

normalized turnaround:

P1: 5 / 4 = 1.25

P2: 13 / 8 = 1.625

P3: 14 / 2 = 7 <--- FCFS can be unfair to short processes

P4: 14 / 6 = 2.333

P5: 18 / 5 = 3.6

processor efficiency = 25 / 30 = 83.33%

b. SPN

turnaround = [(5-0) + (17-1) + (8-3) + (30-10) + (23-12)] / 5 = 57 / 5 = 11.4

NOTE: SPN has the smallest turnaround, compared to other algorithms

normalized turnaround:

P1: 5 / 4 = 1.25

P2: 16 / 8 = 2
P3: 5 / 2 = 2.5 <--- SPN much more fair to short processes (compared to FCFS)
P4: 20 / 6 = 3.333
P5: 11 / 5 = 2.2

processor efficiency = 25 / 30 = 83.33%

c. SRTN

turnaround = [(5-0) + (31-1) + (8-3) + (17-10) + (23-12)] / 5 = 58 / 5 = 11.6

normalized turnaround:

P1: 5 / 4 = 1.25
P2: 30 / 8 = 3.75 <--- largest normalized turnaround; also largest process
P3: 5 / 2 = 2.5
P4: 7 / 6 = 1.166
P5: 11 / 5 = 2.2

processor efficiency = 25 / 31 = 80.6%

d. RR quantum = 3

turnaround = [(13-0) + (28-1) + (11-3) + (32-10) + (35-12)] / 5 = 93 / 5 = 18.6

normalized turnaround:

P1: 13/ 4 = 3.25
P2: 27 / 8 = 3.375
P3: 8 / 2 = 4.0
P4: 22 / 6 = 3.666
P5: 23 / 5 = 4.6

In RR, all processes are considered equally important

processor efficiency = 25 / 35 = 71.4%

e. Multilevel Feedback Queue

turnaround = [(12-0) + (35-1) + (9-3) + (28-10) + (32-12)] / 5 = 90 / 5 = 18

normalized turnaround:

P1: 12/ 4 = 3.0
P2: 34 / 8 = 4.25 <--- largest job
P3: 6 / 2 = 3.0 <--- smallest job
P4: 18 / 6 = 3.0
P5: 20 / 5 = 4.0

processor efficiency = 25 / 35 = 71.4%

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

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

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

9.5 **a.** Lower Bound: n

b. Upper Bound: p

CHAPTER 11 I/O MANAGEMENT AND DISK SCHEDULING

11.1 **a.** 11 + 63 + 86 +64 + 55 + 66 + 87 = 432
b. 7 + 4 + 23 + 2 + 76 + 11 + 1 = 124
c. 7 + 4 + 51 + 11 + 1 + 86 + 2 = 162
d. 7 + 4 + 51 + 11 + 1 + 88 +2 + 164

11.2 The first factor could be the limiting speed of the I/O device; Second factor could be the speed of bus, the third factor could be no internal buffering on the disk controller or too small internal buffering space. Fourth factor could be erroneous disk or transfer of block.

11.3 **a.** T=transfer time = b/rN, where b=number of bytes to transfer, r=rotation speed, and N = number of bytes on a track → b = 1MByte = 1,048,576 Bytes, r=15,000rpm, N=512 × 400=204800 bytes per track → 1,048,576/(15,000/60,000 × 204800)=20.48ms

Here are the units:
T[ms] = b[bytes]/(r[rotations/ms] × N[bytes/rotation])
15,000[rotations/min]/60,000[ms/min] = 0.25[rotations/ms]

Comment: This calculation is a simplified estimation but sufficient for basic disk transfer models: A more sophisticated estimate might notice that the file spans several tracks, thus a worst-case estimate is to include additional seek (but assume the track starts are staggered so there is no significant rotational delay).

b. Ta=average access time of the whole file = Ts+ 1/2r+b/rN = 4+2+20.508 ms = 26.508ms
c. Rotational delay = (average 180 degree wait) = 1/2r=2ms
d. Total time to read 1 sector (512 Bytes) = seek time + rotational delay + transfer time = 4ms+2ms+ 512/(15000/60000 * 204,500) = 4+2+0.01ms = 6.01ms.
e. If the disk uses sequential organization, then the total time to read 1 track (400 sectors per track) = seek_time + rotational_delay + additional_time_to_go_around = 4ms + 2ms + 4ms = 10ms = (Tseek+3/(2r)

11.4 Prefetching is a user-based activity, while spooling is a system-based activity. Comparatively, spooling is a much more effective way of overlapping I/O and processor operations. Another way of looking at it is to say that prefetching is based upon what the processor might do in the future, while spooling is based upon what the processor has done in the past.

11.5 **a.** Total time = Seek time + Rotational delay + Transfer time
= 15 msec + 10 msec + 20 msec = 45 msec

b. Total time = 8 × (Seek time + Rotational delay + Transfer time)
= 8(15 msec + 10 msec + 2.5msec) = 220 msec

CHAPTER 12 FILE MANAGEMENT

12.1 13 (1 block) + 1 (2k pointers) (1 block) + 1 (2k)(2k) (1 block) + (2k)(2k)(2k) (1 block) = 13 (8kB) + 2k (8kB) + 4M (8kB) + 8G (8kB) =
104KB + 16MB + 32GB + 64 TB = 64032016024000Bytes
It is actually greater than that since kB = 1024 Bytes, but I have written it like that to indicate the size of the file that each portion can handle.

12.2 Record 5 is preceded by 4 records, each of 150 bytes. The fifth record will start at byte (4 × 15o) + 1 = 601.

12.3 The answer is the same as for the preceding question, 601. The logical layout of the records is the same for both access methods.

12.4 Eight 60-byte records can be allocated to each block, leaving 20 bytes wasted.

12.5 The first block can contain the first two records, leaving 30 bytes wasted. The second block can contain the third record, wasting 65 bytes. The third block can contain the fourth record, wasting 30 bytes. The fourth block can contain the last two record, wasting 40 bytes. Of the 400 bytes in the four blocks, 30 + 65 + 30 + 40 = 165 bytes, or 41.25%, are wasted.