

# SOLUTIONS TO PRACTICE PROBLEMS

OPERATING SYSTEMS:  
INTERNALS AND DESIGN  
PRINCIPLES  
EIGHTH EDITION

WILLIAM STALLINGS

**Copyright 2014: William Stallings**

## TABLE OF CONTENTS

Chapter 1	Computer System Overview.....	3
Chapter 2	Operating System Overview.....	5
Chapter 3	Process Description and Control .....	6
Chapter 5	Concurrency: Mutual Exclusion and Synchronization...	9
Chapter 6	Concurrency: Deadlock and Starvation .....	16
Chapter 7	Memory Management.....	19
Chapter 8	Virtual Memory .....	22
Chapter 9	Uniprocessor Scheduling.....	30
Chapter 11	I/O Management and Disk Scheduling .....	35
Chapter 12	File Management .....	38

# 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**  $EAT = .9 (10) + .1 [ .8 (10 + 100) + .2 (10 + 100 + 10000) ] = 220\text{ns}$   
OR  
 $EAT = 10 + .1 (100) + .02 (10000) = 220\text{ns}$

- 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 loading multiple copies of utility programs for different users.

# CHAPTER 3 PROCESS DESCRIPTION AND CONTROL

- 3.1 •Creation and deletion of both user and system processes.** The processes in the system can execute concurrently for information sharing, computation speedup, modularity, and convenience. Concurrent execution requires a mechanism for process creation and deletion. The required resources are given to the process when it is created, or allocated to it while it is running. When the process terminates, the OS needs to reclaim any reusable resources.
- Suspension and resumption of processes.** In process scheduling, the OS needs to change the process's state to waiting or ready state when it is waiting for some resources. When the required resources are available, OS needs to change its state to running state to resume its execution.
- Provision of mechanism for process synchronization.** Cooperating processes may share data. Concurrent access to shared data may result in data inconsistency. OS has to provide mechanisms for processes synchronization to ensure the orderly execution of cooperating processes, so that data consistency is maintained.
- Provision of mechanism for process communication.** The processes executing under the OS may be either independent processes or cooperating processes. Cooperating processes must have the means to communicate with each other.
- Provision of mechanisms for deadlock handling.** In a multiprogramming environment, several processes may compete for a finite number of resources. If a deadlock occurs, all waiting processes will never change their waiting state to running state again, resources are wasted and jobs will never be completed.
- 3.2 a.** There is no limit in principle to the number of processes that can be in the Ready and Blocked states. The OS may impose a limit based on resource constraints related to the amount of memory devoted to the process state for each process. At most,  $N$  processes can be in the Running state, one for each processor.
- b.** Zero is the minimum number of processes in each state. It is possible for all processes to be in either the Running state (up to  $N$  processes) or the Ready state, with no process blocked. It is possible for all processes to be blocked waiting for I/O operations, with zero processes in the Ready and Running states.

**3.3** The following example is used in Pinkert, J., and Wear, L. Operating Systems: Concepts, Policies, and Mechanisms. Englewood Cliffs, NJ: Prentice Hall, 1989 to clarify their definition of block and suspend:

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

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

**3.4** ==> 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

Get ptr to next process to execute  
Update pointer  
Execute process

## Dispatcher 2

Get ptr to next process to execute  
Update pointer  
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 /* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
```

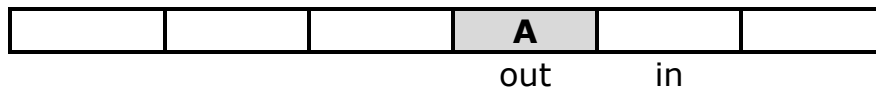
```

{
  while (true) {
    while (!testset (bolt)) /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . . ,P(n));

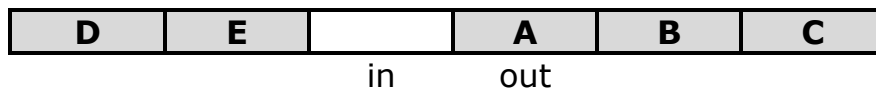
}

```

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



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



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

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

## 5.8 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  $i$ th teller  
 $c[j]$ : the  $j$ th 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.

**5.9** ABCDE; ABDCE; ABDEC; ADBCE; ADBEC; ADEBC;  
DEABC; DAEBC; DABEC; DABCE

**5.10** 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
```

**5.11** 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));
```

**5.12** 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 issue 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** 1.  $W = (2\ 1\ 0\ 0)$   
 2. Mark P3;  $W = (2\ 1\ 0\ 0) + (0\ 1\ 2\ 0) = (2\ 2\ 2\ 0)$   
 3. Mark P2;  $W = (2\ 2\ 2\ 0) + (2\ 0\ 0\ 1) = (4\ 2\ 2\ 1)$   
 4. Mark P1; no deadlock detected

**6.3** 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.

**6.4 a.**  $\begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 6 & 6 & 2 & 2 \\ 2 & 0 & 0 & 2 \\ 0 & 3 & 2 & 0 \end{matrix}$

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

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

**6.5 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  $O(n \cdot m)$  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.

- 6.6 a.** Deadlock occurs if all resource units are reserved while one or more processes are waiting indefinitely for more units. But, if all 4 units are reserved, at least one process has acquired 2 units. Therefore, that process will be able to complete its work and release both units, thus enabling another process to continue.
- b.** Using terminology similar to that used for the banker's algorithm, define  $claim[i]$  = total amount of resource units needed by process  $i$ ;  $allocation[i]$  = current number of resource units allocated to process  $i$ ; and  $deficit[i]$  = amount of resource units still needed by  $i$ . Then we have:

$$\sum_i^N claim[i] = \sum_i^N deficit[i] + \sum_i^N allocation[i] < M + N$$

In a deadlock situation, all resource units are reserved:

$$\sum_i^N allocation[i] = M$$

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

$$\sum_i^N deficit[i] < N$$

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

## 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

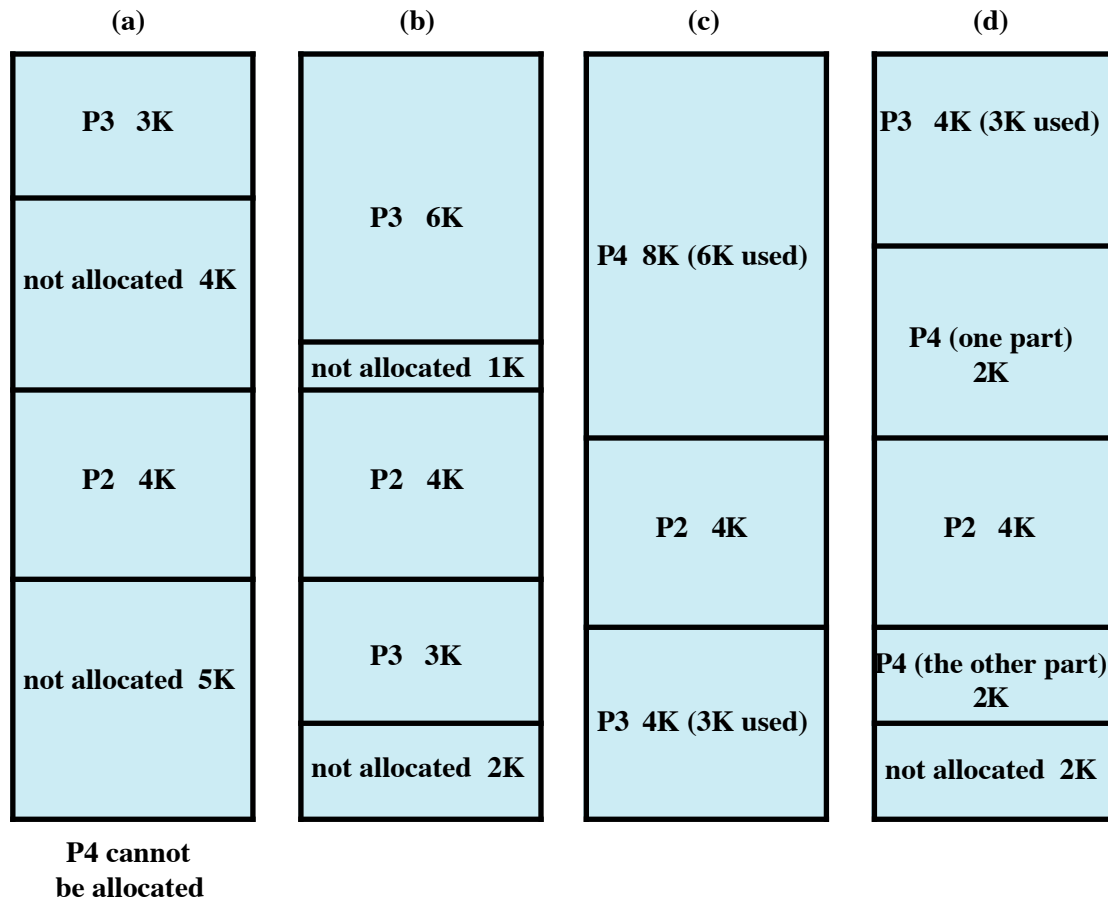
**7.3** a. 100K  
500K holds 417K  
200K holds 112K  
300K holds 212K  
600K  
350K waits for 500K partition to become free  
b. 100K  
500K holds 417K  
200K holds 112K  
300K holds 212K  
600K holds 350K  
c. internal after a.:  $(500K - 417K) + (200K - 112K) + (300K - 212K) = 259K$   
d. external 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

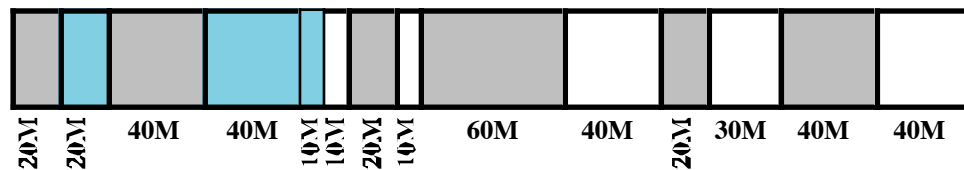
## 7.6



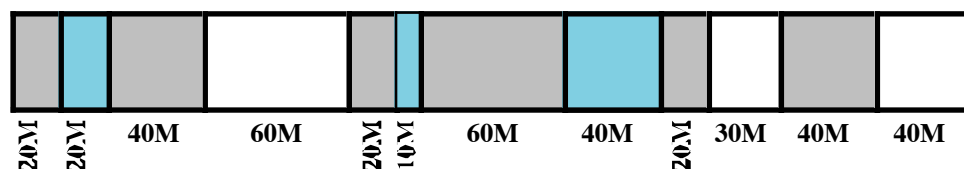
## 7.7 12 bits

## 7.8 64 bits

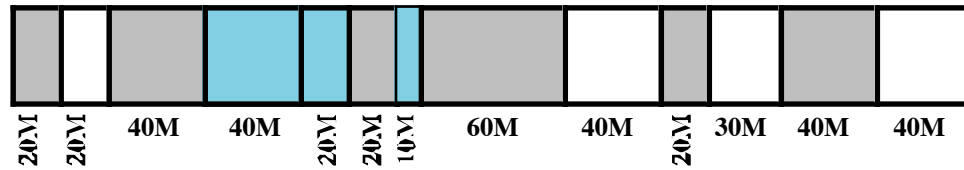
**7.9a.** The 40 M block fits into the second hole, with a starting address of 80M. The 20M block fits into the first hole, with a starting address of 20M. The 10M block is placed at location 120M.



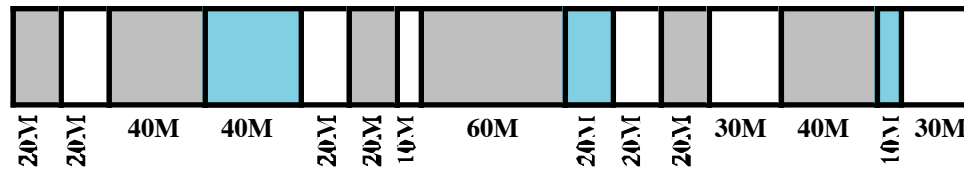
**b.** The three starting addresses are 230M, 20M, and 160M, for the 40M, 20M, and 10M blocks, respectively.



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



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



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

## CHAPTER 8 VIRTUAL MEMORY

- 22-

$$+ 0.5 (1000000) + 75))$$

$$\text{EAT} = 76 + 0.2 (153 + .1 (750170)) = 76 + 15034 = 15110 \text{ ns}$$

- 8.6**
- a.** PFN 3 since loaded longest ago at time 20
  - b.** PFN 1 since referenced longest ago at time 160
  - c.** Clear R in PFN 3 (oldest loaded), clear R in PFN 2 (next oldest loaded), victim PFN is 0 since R=0
  - d.** Replace the page in PFN 3 since VPN 3 (in PFN 3) is used furthest in the future
  - e.** There are 6 faults, indicated by \*

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

**8.7 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	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
		7	7	7	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	1	7	2	2	0	1	3	2	7	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

**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	3	3	3	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	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	>3/1	1/1	1/1	>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	>0/1	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	3/1	>3/1	3/1

**8.8** 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  $300 + 0.5 \times 300$ , or  $450 \mu\text{S}$ . 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

**8.9 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.

- 8.10 a.** Virtual memory can hold  $(2^{32} \text{ bytes of main memory}) / (2^{10} \text{ bytes/page}) = 2^{22}$  pages, so 22 bits are needed to specify a page in virtual memory. Each page table contains  $(2^{10} \text{ bytes per page table}) / (4 \text{ bytes/entry}) = 2^8$  entries. Thus, each page table can handle 8 of the required 22 bits. Therefore, 3 levels of page tables are needed.
- b.** Tables at two of the levels have  $2^8$  entries; tables at one level have  $2^6$  entries.  $(8 + 8 + 6 = 22)$ .
- c.** Less space is consumed if the top level has  $2^6$  entries. In that case, the second level has  $2^6$  pages with  $2^8$  entries each, and the bottom level has  $2^{14}$  pages with  $2^8$  entries each, for a total of  $1 + 2^6 + 2^{14}$  pages = 16,449 pages. If the middle level has 26 entries, then the number of pages is  $1 + 2^8 + 2^{14}$  pages = 16,641 pages. If the bottom level has  $2^6$  entries, then the number of tables is  $1 + 2^8 + 2^{16}$  pages = 65,973 pages.
- 8.11** 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  $(\text{SIZE} + 100 - 900)$  page faults each time through -- 200 for the small case, 9200 for the large one.

The outer loops just repeat the inner loop  $\text{SIZE} \times \text{SIZE} / 100$  times, yielding the overall estimate of:

$\text{SIZE} = 1000 \rightarrow 200 \times 1000 \times 10 \rightarrow 2 \times 10^6$  page faults

$\text{SIZE} = 10000 \rightarrow 9200 \times 10000 \times 100 \rightarrow 9 \times 10^9$  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,  $1000 * (2 \times 10^6)$  is better than  $9 \times 10^9$ . By improving the 1000 case as above, by chopping it in half, this could be improved to under  $1000 \times 10000$ . 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 ( $SIZE \times SIZE / 1000$ ) into the 800 not being used by A and C. Hence:

$SIZE = 1000 \rightarrow (1000 - 800) \times 10 \rightarrow 2000$  page faults (for B)  
 $+ 3 \times 1000$  mandatory page faults = 5000  
 $SIZE = 10000 \rightarrow (100000 - 800) \times 100 \rightarrow 1 \times 10^7$  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][10], 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:

$SIZE = 1000 \rightarrow (1000 - 998) \times 1000 \times 10 \rightarrow 2 \times 10^4$  page faults  
 $SIZE = 10000 \rightarrow (10000 - 998) \times 10000 \times 100 \rightarrow 9 \times 10^9$  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)...

$SIZE = 1000 \rightarrow (1000 - 998) \times 10 \rightarrow 20$  page faults (for conflict in B)  
 $+ 3 \times 1000$  mandatory page faults = 3000

SIZE = 10000  $\rightarrow (100000-998) \times 100 \rightarrow 1 \times 10^7$  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  $3 \times 10^6$  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.

**8.12 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		
---	--	---	--	--	---	--	---	--	---	--	--

**8.13 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

FCFS

RR,  $q = 1$

RR,  $q = 4$

SPN

SRT

HRRN

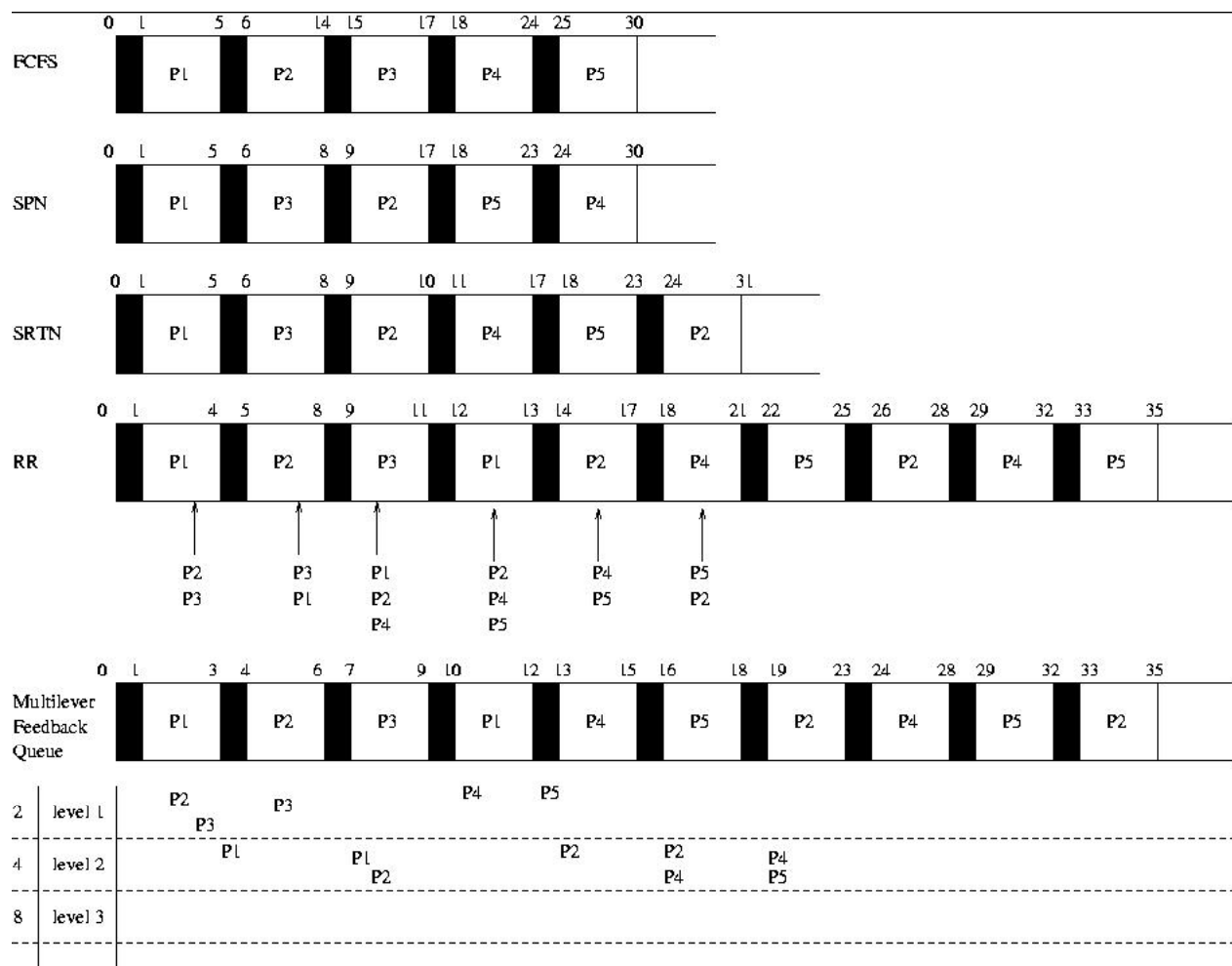
Feedback,  $q = 1$

Feedback,  $q = 2^i$

A	B	B	B	B	B	B	B	B	B	C	D	D	D	D	D	D	D	D	D
A	B	C	B	D	B	D	B	D	B	D	B	D	B	D	B	D	B	D	D
A	B	B	B	B	C	B	B	B	B	D	D	D	D	B	D	D	D	D	D
A	B	B	B	B	B	B	B	B	B	C	D	D	D	D	D	D	D	D	D
A	B	C	B	B	B	B	B	B	B	B	D	D	D	D	D	D	D	D	D
A	B	B	B	B	B	B	B	B	B	C	D	D	D	D	D	D	D	D	D
A	B	C	D	B	D	B	D	B	D	B	D	B	D	B	D	B	D	B	D
A	B	C	D	B	B	D	D	B	B	B	B	D	D	D	D	B	B	D	D

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

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



**a. FCFS**

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

normalized turnaround:

$$P1: 5 / 4 = 1.25$$

$$P2: 13 / 8 = 1.625$$

$$P3: 14 / 2 = 7$$

<--- FCFS can be unfair to short

processes

$$P4: 14 / 6 = 2.333$$

$$P5: 18 / 5 = 3.6$$

$$\text{processor efficiency} = 25 / 30 = 83.33\%$$

**b. SPN**

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

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 =  $25 / 35 = 71.4\%$

**9.3 An argument in favor of a small quantum:** Using a small quantum will enhance responsiveness by frequently running all processes for a short time slice. When the ready queue has many processes that are interactive, responsiveness is very important e.g. general-purpose computer system.

**An argument in favor of a large quantum:** Using a large quantum will enhance the throughput, and the CPU utilization measured with

respect to real work, because there is less context switching and therefore less overhead. e.g. batch jobs.

**A system for which both might be appropriate:** There are some systems for which both small and large quanta are reasonable. e.g. a long job that requires just a few user interactions. Although this type of job can be considered as a batch job, in some sense, it still has to interact with the user. Therefore, during the times when there is no user interaction, the quantum might be increased to optimize the throughput and during interactive time, the quantum might be lowered to provide better responsiveness.

- 9.4 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.**  $T = \text{transfer time} = b/rN$ , where  $b$  = number of bytes to transfer,  $r$  = rotation speed, and  $N$  = number of bytes on a track  $\rightarrow b = 1\text{MByte} = 1,048,576 \text{ Bytes}$ ,  $r = 15,000\text{rpm}$ ,  $N = 512 \times 400 = 204800 \text{ bytes per track} \rightarrow 1,048,576 / (15,000/60,000 \times 204800) = 20.48\text{ms}$

Here are the units:

$$T[\text{ms}] = b[\text{bytes}] / (r[\text{rotations/ms}] \times N[\text{bytes/rotation}])$$

$$15,000[\text{rotations/min}] / 60,000[\text{ms/min}] = 0.25[\text{rotations/ms}]$$

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.**  $T_a = \text{average access time of the whole file} = T_s + 1/2r + b/rN = 4 + 2 + 20.508 \text{ ms} = 26.508\text{ms}$
- c.** Rotational delay = (average 180 degree wait) =  $1/2r = 2\text{ms}$
- d.** Total time to read 1 sector (512 Bytes) = seek time + rotational delay + transfer time =  $4\text{ms} + 2\text{ms} + 512 / (15000/60000 \times 204,500) = 4 + 2 + 0.01\text{ms} = 6.01\text{ms}$ .
- 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 =  $4\text{ms} + 2\text{ms} + 4\text{ms} = 10\text{ms} = (T_{\text{seek}} + 3/(2r))$

**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 =  $8 \times (\text{Seek time} + \text{Rotational delay} + \text{Transfer time})$   
=  $8(15 \text{ msec} + 10 \text{ msec} + 2.5 \text{ msec}) = 220 \text{ msec}$

**11.6** The system requires at least one buffer the size of a sector, usually more than one. For systems with small memory, a smaller sector size reduces the memory needed for I/O buffers. The disk system requires a gap between sectors to allow for rotational and timing tolerances. A smaller sector size means more sectors and more gaps

**11.7** A variable sector size complicates the disk hardware and disk controller design. In early systems it had the advantage of allowing for small block sizes with small memory systems, and larger block sizes for larger systems. Also, it allows the programmer to choose the optimal size for the data.

**11.8 a.**  $T = \text{transfer time} = b/rN$ , where  $b$  = number of bytes to transfer,  $r$  = rotation speed, and  $N$  = number of bytes on a track  $\rightarrow b = 1 \text{ MByte} = 1,048,576 \text{ Bytes}$ ,  $r = 15,000 \text{ rpm}$ ,  $N = 512 \times 400 = 204,800 \text{ bytes per track}$   
 $\rightarrow 1,048,576 / (15,000 / 60,000 \times 204,800) = 20.48 \text{ ms}$

Here are the units:

$$T[\text{ms}] = b[\text{bytes}] / (r[\text{rotations/ms}] \times N[\text{bytes/rotation}])$$
$$15,000[\text{rotations/min}] / 60,000[\text{ms/min}] = 0.25[\text{rotations/ms}]$$

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.**  $T_a = \text{average access time of the whole file} = T_s + 1/2r + b/rN = 4 + 2 + 20.508 \text{ ms} = 26.508 \text{ ms}$

**c.** Rotational delay = (average 180 degree wait) =  $1/2r = 2 \text{ ms}$

**d.** Total time to read 1 sector (512 Bytes) = seek time + rotational delay  
+ transfer time =  $4 \text{ ms} + 2 \text{ ms} + 512 / (15000 / 60000 \times 204,500) =$

$$4+2+0.01\text{ms} = 6.01\text{ms}.$$

- 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 =  $(T_{\text{seek}} + 3/(2r))$

**11.9** 512 bytes  $\times$  1024 sectors = 0.5 MB/track. Multiplying by 2048 tracks/platter gives 1 GB/platter, or 5 GG capacity in the drive.

## CHAPTER 12 FILE MANAGEMENT

**12.1**  $13 (1 \text{ block}) + 1 (2k \text{ pointers}) (1 \text{ block}) + 1 (2k)(2k) (1 \text{ block}) + (2k)(2k)(2k) (1 \text{ block}) = 13 (8kB) + 2k (8kB) + 4M (8kB) + 8G (8kB)$   
 $= 104KB + 16MB + 32GB + 64 \text{ TB} = 64032016024000\text{Bytes}$

It is actually greater than that since  $1 \text{ kB} = 1024 \text{ Bytes}$ , 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  $(4 \times 150) + 1 = 601$ .

**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,  $30 + 65 + 30 + 40 = 165 \text{ bytes}$ , or 41.25%, are wasted.