Chapter 3: 3,4

3. A network of workstations can only communicate with one another by sending and receiving information via the network. If two or more processes are executing on different workstations in the network and they assume the existence of a logical shared memory environment, then the system must provide facilities by which at least some processes exchange messages over the network on memory references. There are several different approaches. For example, assume that processes p_i and p_j are executing on machines M_i and M_j, respectively. Further, assume the variable X has been declared to be in the shared memory, *and is allocated on M_i in p_i 's address space*. Whenever p_i reads X, no special action needs to be taken. Whenever p_j reads X, the system must send a message to system code on M_j to read the value of X and to return it to p_j. (The value could be copied for efficiency, but this introduces new problems not described in this solution.) Whenever p_i writes X, no special action needs to be taken. Whenever p_j writes X, the system must send a message to system code on M_i to cause X to be written with the new value. The logical shared memory is an asset to the application programmer, since s/he need not learn about network messages, using only ordinary memory reads and writes (perhaps using a system call). The logically shared memory is easy to use.

4.Writing to the stack segment register. When the assembly language program writes to the stack segment register, the process's entire stack contents are potentially lost. This follows since the part of memory holding the stack is no longer referenced by the stack segment register. In particular, when the assembly language program returns to the C program (presuming it somehow saved a return address prior to clobbering the stack), the C program's automatic variables and call stack will all be lost.
Writing to the code segment register. Writing to the code segment register immediately causes an unusual form of a branch statement. When the instruction finishes executing, the control unit will add the contents of the PC to the new value of the code segment register, which will now point to a different 64KB block in memory than it did before the instruction was executed. The next instruction will be fetched from the PC displaced address in the new code block.

Chapter 5: 3, 11

3. An input-bound process attempts to read characters at a higher rate than the input device can produce them (and an output-bound process attempts to write characters at a higher rate than the output device can consume them). Consider the input-bound case: The process reads the character from buffer A, then asks for the character from buffer B while the device is still reading; the process blocks until the device completes, consumes the character, and immediately requests the next character from buffer A. The device is still busy. Thus, the effect is the input-bound device achieves little overlap with the device, and still tends to wait on the device. On output, the process writes a character to buffer A, starting the driver. The process then immediately writes a second character to buffer B, then attempts to write a third character to buffer A, but is blocked by the first output operation. At this point, the process will procede no faster than the device.
A compute-bound process rarely requests I/O, thus the device "runs ahead" of the process. On input, it produces a character in buffer A and buffer B while the process is computing. It then goes to sleep. The process eventually requests the character in buffer A, causing the driver to refill buffer A while the process enters its next compute phase, etc. Thus the device always keeps the buffers full, fully overlapping its operation with the process computation.

11. The blocks to be read are 97, 84, 155, 103, 96, and 197.

a. FCFS steps across 13+71+52+7+101 = 244 steps.

b. SCAN visits the tracks in the order 97, 103, 155, 197, 199, 96, 84 incurring 6+52+42+2+103+12 = 217 steps.

c. LOOK visits the tracks in the order 97, 103, 155, 197, 96, 84 incurring 6+52+42+101+12 = 213 steps.

Chapter 6: 6,7

6. The PC always addresses the next instruction to be fetched. Therefore it must not be changed by an instruction execution until the fetch unit has already finished retrieving the last instruction in the save sequence. The following pseudo code fragment illustrates this:

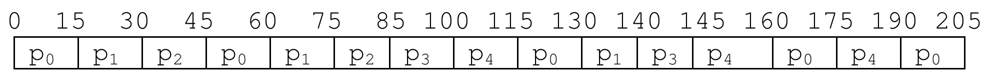
```
; The process's register set is at location "pid",
; With the PC stored at "pid-1"
; Load the arithmetic-logical registers
      load  R0, =0
      incr  R0
      load  R1, pid[R0]
      incr  R0
      load  R2, pid[R0]
      incr  R0
      load  R3, pid[R0]
      incr  R0
; Load the PSR, CC
      load  CC, pid[R0]
      incr  R0
      load  PSR, pid[R0]
      load  R0, pid          ; Restore R0
; Load the PC
      load  PC, pid-1
```

7. Here is a solution. You might want the return from interrupt to make this process be Ready instead of Running.

Chapter 7: 6, 10

6. RR scheduling with a time quantum of 15

a. Gantt chart



b. Turnaround time for $p_3 = 65$

c. Average wait time

W(p_0) = 0
W(p_1) = 5
W(p_2) = 20
W(p_3) = 5
W(p_4) = 15

Average = (0+5+20+5+15)/5 = 9

10. RR converges to FCFS scheduling since every job has its entire need met the first time it receives a time quantum.

Chapter 8:3,10

3.This is Dekker's software solution to the critical section problem, as published in Dijkstra's original semaphore paper. In the paper there is a proof of correctness; the following paraphrases the proof:

First, suppose that turn is set to 1, and process 0 attempts to enter the critical section while process 1 is in its compute section. Then flag[0] = TRUE and flag[1] = FALSE; therefore, process 0 is allowed to enter its critical section since the while test fails immediately.

Suppose that turn is set to 1, and process 0 is in the critical section when process 1 attempts to enter the critical section. Then process 1 sets flag[1] to TRUE and executes the while test; it finds flag[0] is TRUE, so it checks turn. Since turn is equal to 1, process 1 ignores the remaining statements in the range of the while loop and proceeds directly to retest flag[0]. Eventually, process 0 will exit the critical section and set turn to 0 and flag[0] to FALSE. Process 1 will detect the changing value of flag[0] and proceed to the critical section without rechecking the value of turn.

Suppose that turn is set to 1 and processes 0 and 1 simultaneously attempt enter the critical section. Then each will set flag[i] to TRUE and each will evaluate the condition in the while statement to be TRUE. Process 0 will then fail the nested if test and will set its flag to FALSE;

Suppose that turn is set to 1 and processes 0 and 1 simultaneously attempt enter the critical section. Then each will set flag[i] to TRUE and each will evaluate the condition in the while statement to be TRUE. Process 0 will then fail the nested if

test and will set its flag to FALSE; it will then wait for turn to be set to its own value. When this eventually happens (by virtue of process 1 exiting its critical section), process 0 will exit the loop and retry the entire process. Meanwhile, process 1 will behave as in the previous paragraph, waiting for flag[0] to go FALSE (in this case, that happens by the tie-breaking code rather than by process 0 finishing its critical section).

1. 10. tra points out, this is just a restatement of the producer-consumer (bounded buffer) problem.

```
customer() {
  while(TRUE) {
    customer = nextCustomerArrives();
    if(emptyChairs < 0) continue;
    P(chair);
      P(mutex);
        emptyChairs--;
        takeChair(customer);
      V(mutex);
    V(waitingCustomer);
  }
}

barber() {
  while(TRUE) {
    P(waitingCustomer);
      P(mutex);
        emptyChairs++;
        customer = takeCustomer();
      V(mutex);
    V(chair);
  }

semaphore mutex = 1, chair = N, waitingCustomer=0;
int emptyChairs = N;
fork(customer, 0);
fork(barber, 0);
}
```