Chapter 2

1. The transition from blocked to running is conceivable. Suppose that a process is blocked on I/O and the I/O finishes. If the CPU is otherwise idle, the process could go directly from blocked to running. The other missing transition, from ready to blocked, is impossible. A ready process cannot do I/O or anything else that might block it. Only a running process can block.

12. A worker thread will block when it has to read a Web page from the disk. If user-level threads are being used, this action will block the entire process, destroying the value of multithreading. Thus it is essential that kernel threads are used to permit some threads to block without affecting the others.

16. User-level threads cannot be preempted by the clock unless the whole process' quantum has been used up (although transparent clock interrupts can happen). Kernel-level threads can be preempted individually. In the latter case, if a thread runs too long, the clock will interrupt the current process and thus the current thread. The kernel is free to pick a different thread from the same process to run next if it so desires.

18. The biggest advantage is the efficiency. No traps to the kernel are needed to switch threads. The biggest disadvantage is that if one thread blocks, the entire process blocks.

27. Each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses, and so on. This is equally true for user-level threads as for kernel-level threads.

34. With kernel threads, a thread can block on a semaphore and the kernel can run some other thread in the same process. Consequently, there is no problem using semaphores. With user-level threads, when one thread blocks on a semaphore, the kernel thinks the entire process is blocked and does not run it ever again. Consequently, the process fails.

39. Three processes are created. After the initial process forks, there are two processes running, a parent and a child. Each of them then forks, creating two additional processes. Then all the processes exit.

45.

1）For round robin, during the first 10 minutes each job gets 1/5 of the CPU. At the end of 10 minutes, *C* finishes. During the next 8 minutes, each job gets 1/4 of the CPU, after which time *D* finishes.

Then each of the three remaining jobs gets 1/3 of the CPU for 6 minutes, until *B* finishes, and so on.

The finishing times for the five jobs are 10, 18, 24, 28, and 30, for an average of 22 minutes.

2）For priority scheduling, *B* is run first. After 6 minutes it is finished. The other jobs finish at 14, 24, 26, and 30, for an average of 20 minutes.

3）If the jobs run in the order *A* through *E*, they finish at 10, 16, 18, 22, and 30, for an average of 19.2 minutes.

4）Shortest job first yields finishing times of 2, 6, 12, 20, and 30, for an average of 14 minutes.

50. The fraction of the CPU used is 35/50 + 20/100 + 10/200 + x/250. To be schedulable, this must be less than 1. Thus x must be less than 12.5 msec.

60. Similar with reader and writers problem.

```
Int w_count = 0, m_count =0 ;
Semaphore    w_mutex =1, m_mutex =1;
Semaphore    bathroom =1;

Woman_wants_to_enter:
    Down(&w_mutex ) ;
    W_count = w_count +1;
    If (w_count ==1) down (&bathroom);
    Up(&w_mutex) ;

woman_leaves:
    Down(&w_mutex ) ;
    W_count = w_count -1;
    If (w_count ==0) up (&bathroom);
    Up(&w_mutex) ;

Man_wants_to_enter:
    Down(&m_mutex ) ;
    m_count = m_count +1;
    If (m_count ==1) down (&bathroom);
    Up(&m_mutex) ;

Man_leaves:
    Down(&m_mutex ) ;
    m_count = m_count -1;
    If (m_count ==0) up (&bathroom);
    Up(&m_mutex) ;
```