This homework is due in class on Friday, October 28. The problems are mostly from Modern Operating Systems, by Andrew Tannenbaum (1992). Questions basically cover part 2 of our textbook: synchronization, deadlock, and scheduling. You can discuss the questions with other people in the class, but you should write up your own answers. Some of the questions do not have definite, unambiguous answers. In all cases, justify and explain your answers!

1. (From Tannenbaum, p. 71.) Synchronization with condition variables uses the operations WAIT and SIGNAL. A more general form of synchronization would be to have a single primitive WAITUNTIL that takes an arbitrary boolean predicate as parameter. Thus, one could say, for example

```
WAITUNTIL x < 0 or y+z < n
```

The SIGNAL primitive would no longer be needed. This scheme is clearly more general, but it is not used. Why not?

2. (From Tannenbaum, p. 72.) Round-robin schedulers normally maintain a list of all runnable threads, with each thread occurring exactly once in the list. What would happen if a thread occurred twice in the list? Can you think of any reason for allowing this?

3. (From Tannenbaum, p. 73.) Most round-robin schedulers use a fixed-size quantum [time-slice]. Give an argument in favor of a small quantum. Now give an argument in favor of a large quantum.

4. Can you see any problem with combining MFLQ scheduling with "busy waiting" of the form "while (! condition) thread_yield();" ? (Explain.)

5. According to Tannenbaum, the Ostrich Algorithm for dealing with deadlock is, "stick your head in the sand and pretend there is no problem at all." Discuss this "algotithm." Is a good solution? Always? In some circumstances? (Keep in mind that avoiding, detecting, and breaking deadlocks can be difficult.)

6. (From Tannenbaum, p. 263.) In an electronic funds transfer system, there are hundreds of identical processes that work as follows: Each process reads a line specifying an amount of money, the account to be credited, and the account to be debited. Then it locks both accounts and transfers the money, releasing the locks when done. With many processes, there is a very real possibility of deadlock. How could you avoid deadlocks in this system? Does your solution avoid starvation? (A good answer will discuss several possibilities.)

7. (From Tannenbaum, p. 73.) A classic problem is Lamport's (1974) bakery problem. In this problem, a bakery has a fixed number of salespeople. Every entering customer takes a number. Until the number is called, the customer waits. Whenever a salesperson is free, the next number is called. Give pseudocode for a procedure for the salespersons to execute and one for the customers to executed. Assume that there is a function *serve*(*customer*) that a server can call to handle the interaction with a customer. Be clear about what synchronization mechanisms you are using!