CS471: Operating System Concepts
Fall 2007
(Lecture: MW 9:15-10:30AM)
Homework #3
Points: 20
Due: September 21, 2007

Question 1 [Points 5] Exercise 6.1: Provide an argument to show that the algorithm satisfies all three requirements for the critical section problem.

Solution: General structure of critical section is
1) Mutual exclusion
2) Progress
3) Bounded waiting

To prove property (1) we note that each pi enters the critical section only if the *flag*[j] = *false.* Also note that if both processes can be executing in their critical sections at the same time then *flag*[0]=*flag*[1]=true. These two observations imply that p0 and p1 could not have successfully executed their while statements about the same time, since the value of turn can be either 0 or 1 but cannot be both. The while loop repeats here till the *flag*[i] becomes false or *flag*[i]=true. Also in the while loop while turn=j then *flag*[i]= false. While turn=j it does noting and makes the *flag*[i]=true. So when one process is ready for enter into the critical section not other process is allowed into the critical section thus preserving the Mutual Exclusion property.

To prove properties (2) and (3) we note that a process pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition *flag*[j]=true and if turn=j then it makes the *flag*[i]=false. This loop is the only one possible. If pj is not ready to enter the critical section then *flag*[j]=false and pi can enter its critical section. If it has set *flag*[j] to true and is also executing in its while loop then either turn=i or turn =j. If turn=i then pi will enter the critical section, if turn=j then pj will enter the critical section. However once pi exits its critical section it will resets *flag*[i]=false and turn=j thus allowing pj to enter the critical section. If pj resets *flag*[j] to true it must also set turn to i. Thus, since pi does not change the value of the variable turn while executing the while statement pi will enter the critical section (progress) after at most one enters by pj (bounded waiting).

Question 2 [Points 5] Exercise 6.9: Show that, if the wait and signal operations are not executed atomically, then mutual exclusion may be violated.

Solution: Consider the wait(S) semaphore operation defined in page 202 of the textbook. Suppose wait(S) is not executed atomically. Then there will be several problems. As an example consider the statement S->value--. This statement actually consists of 3 CPU instructions.

```
LOAD S->value
SUB 1
STO S-> value
```

If two processes Pi and Pj execute the wait statement and if the S->value is 1 before Pi and Pj execute the wait statement. We know how the final value of S->value may be either -1 or 0 depending on how the CPU instructions of Pi and Pj are interleaved. Only -1 value is correct. If the value was 0, then there would be 1 process in the list but the value would show that there are now. This creates a problem.

Similarly, if signal(S) is not implemented atomically, due to similar problems (e.g., S->value++), mutual exclusion condition could be violated due to incorrect value.

Question 3[Points 5] Exercise 6.12: Demonstrate that monitors and semaphores are equivalent in so far as they can be used to implement the same types of synchronization problems.

Solution: If semaphores can be implemented using monitors and monitors implement semaphores, then we can show that they resolve the same class of synchronization problems.

<u>Semaphore using monitor</u>

```
monitor{
        int count;
        condition waiting;
                function init(initial_count) {
                                        count=initial_count;
                                        }
         function p(){
                  while(count<0) waiting.wait();
                  count=count-1;
                 }
                function v(){
                            count=count+1;
                            waiting.release();
                           }
        }
```

<u>Monitor using semaphores</u>
semaphore mutex :=1, waiting :=0;
int count :=0;

For each invocation of a monitor function do {mutex.p(); function; mutex.v()}. For each use of a condition variable.wait() do {waiting.p(), count=count+1}. For each use of a condition variable.signal() do {waiting.v(), count=count-1}. For each use of a condition variable.broadcast() do {while (count 0) {waiting.v(); count=count-1 ;}}

In both the cases semaphores implemented monitor and vice versa so we can say that they resolve the same class of synchronization problems.

Question 4 [Points 5] Exercise 6.16: How does the signal() operation associated with monitors differs from the corresponding operation defined for semaphores?

Solution: In monitor x, signal() operation resumes exactly one suspended process. If no process is suspended then the signal() operation has no effect. That is the state of x is the same as if the operation had never been executed contrast this operation with the signal() operation associated with semaphore, which always affects the state of the semaphore.