Operating Systems: Homework #3

Due on February 19, 2016 at 11:59pm

Professor Qu
Monday & Wednesday 3:30pm — 5:17pm

Nicholas Land

Problem 1

- Can there be a thread blocked on a semaphore with non-negative value?
- Can a semaphore have a negative value without having any threads blocked on it?

Solution

Yes, because a thread is blocked on 0, and 0 is non-negative. If the value is negative, there are no more resources allocate to use the semaphore.

Yes, because if there are no threads are requesting it than it can have a negative value.

Problem 2

In the following code, four processes produce output using the routine 'printf' and synchronize using three semaphores 'R', 'S' and 'T.' We assume function 'printf' wont cause context switch.

```
Semaphore R=1, S=3, T=0; /* Initialization */

/* Process 1 */
while(true) {
  P(S);
  printf('A');
}

/* Process 2 */
while(true) {
  P(T);
  printf('B');
  printf('C');
  V(T);
}

/* Process 3 */
while(true) {
  P(T);
  printf('D');
  V(R);
}

/* Process 4 */
while(true) {
  P(R);
  printf('E');
  V(T);
}
```

a) How many **A**'s and **B**'s are printed when this set of processes runs?

b) What is the smallest number of **D**'s that might be printed when this set of processes runs?

c) Is **AEBCBCDAA** a possible output sequence when this set of processes runs? Clarify your answer.

Solution

a) Three **A**'s are printed because S is decremented, but it is never incremented.

B can be printed 0, 1, or {B}* times. It is possible that be could be infinite. However, It is also possible that process 3 and process 4 could be in an infinite loop, and in that case B would not be printed. It could also be that B could get printed only one time, and then process 3 and 4 are in an infinite loop. B could be printed {B}* times if process 2 was in an infinite loop.

b) 0 times. If process 4 is run, it is possible that process 2 could run, and then go between process 2 and 4 infinitely so long as there is no waiting queue.

If there is a waiting queue then D can be printed infinitely many times.

c) Yes, because processes 1 could run, then process 4, then process 2, then process 2, then process 2, then process 2, then process 2, then process 3, then process 1, and finally process 1 if there is a waiting queue.

If there is a waiting queue then this would not be a possible output, because then process 2 is not able to be in the waiting queue when it is being executed. So the output BCBC would be unable to complete.

Problem 3

Consider the following two processes $P[i]$ and $P[j]$. Initially, flag[i] = flag [j] = false.

```
Do {
  flag[i]=true;
  While(flag[j]);

  critical section

  flag[i] = false;

  remainder section
} while(1);
```

```
Do {
  flag[j]=true;
  While(flag[i]);

  critical section

  flag[j] = false;

  remainder section
} while(1);
```

a) Does the above program satisfy the 'progress' requirement? Justify your answer with an informal proof or counterexample. [Simple 'Yes' or 'No' without explanation]

b) Is mutual exclusion assured? Justify your answer with an informal proof or counterexample.

Solution

a) Yes, this follows the 'progress' requirement. According to the 'progress' requirement, if no process is executing in it's critical section, and some process wishes to enter their critical section, then so long as that process is not in the remainder section it will enter the critical section. In the example above once $p[i]$ starts then it sets $p[j]$ to true. Once $p[j]$ begins then it will enter the critical section, but while that happens, $p[i]$ is also set to true. Being that $p[j]$ is in the critical section, that means that $p[i]$ will also be able to enter the critical section being that it is not currently in the remainder section. Therefore, satisfying the 'progress' requirement.

b) Mutual exclusion is assured. In either case each process $p[i]$ & $p[j]$ can only do one thing at a time. Even if one process starts a little bit before another one, we'll assume that the while() loop will still be executed before the other process finishes the critical section and turns flag[] to false. Because of this, we can be assured that mutual exclusion is achieved.