**EE 360P HW1 Questions**

4.(15 points) Show that any of the following modifications to Peterson’s algorithm makes it incorrect:

a) A process in Peterson’s algorithm sets the turn variable to itself instead of setting it to the other process. The remaining algorithm stays the same.

After implementing this change the algorithm no longer satisfies mutual exclusion. At line 3 (the while loop) in both P0 and P1, it is possible for such a scenario:

P0 does not want the critical section, so WantCS[0] = false. P1 is able to access the critical section. As it does, P0 does want the critical section. It sets Turn to 0, so it is able to access critical section before P1 can exit.

P0: WantCS[0] = TRUE;

Turn = 0;

while (WantCS[1] && Turn == 1) wait();

P1: WantCS[1] = TRUE;

Turn = 1;

while (WantCS[0] && Turn == 0) wait();

b) A process sets the turn variable before setting the wantCS variable.

This also does not satisfy mutual exclusion. It is possible for this scenario:

P0 sets Turn = 1;

P1 sets Turn = 0 and WantCS[1] = TRUE;

P1 accesses CS because WantCS[0] has not been set to TRUE yet.

P0 sets WantCS[0] = TRUE;

P0 accesses CS because Turn = 0 (reset by P1) before P1 can exit.

P0: Turn = 1;

WantCS[0] = TRUE;

while (WantCS[1] && Turn == 1) wait();

P1: Turn = 0;

WantCS[1] = TRUE;

while (WantCS[0] && Turn == 0) wait();

5.(15 points) Prove that Peterson’s algorithm is free from starvation.

Suppose by contradiction that Peterson’s algorithm is not free from starvation. Assume without loss of generality that P0 is starved; it cannot access CS although it wants to (WantCS[0]=TRUE). Then it must be trapped in the while loop because that is the only instruction that is conditional to accessing CS. Then WantCS[1]=TRUE and Turn=1, otherwise P0 will exit the while loop. However, P1 sets WantCS[1]=FALSE after exiting CS. If P1 wants to re-access CS, it sets Turn=0 and is forced to wait at its own while loop (since WantCS[0]=TRUE, and WantCS[0] can only be accessed by P0). Then we have that both P1 and P0 are at a deadlock. That is a contradiction because Peterson’s Algorithm does not have deadlocks; for a deadlock to occur, it must be that WantCS[1]=TRUE, Turn=1, WantCS[0]=FALSE, Turn=0, which cannot happen because Turn cannot be both 1 and 0.

6.(15 points) Peterson’s algorithm uses a multi-write variable turn. Modify the algorithm to use two variables turn0 and turn1 instead of turn such that P0 does not write to turn1 and P1 does not write to turn0.

The algorithm remains the same except lines 2 and 3 below.

P0: requestCS(){

WantCS[0] = TRUE;

while(Turn1 > 1){}

Turn0 = Turn1+10;

while (WantCS[1] && Turn1 == 1) {

}

}

releaseCS(){

WantCS[0] = False;

Turn0 = 0;

}

P1: requestCS(){

WantCS[1] = TRUE;

Turn1= Turn0+1;

while (WantCS[0] && Turn0 >= 10){

if(Turn0 >= 10 && Turn1 == 1)

Turn1 = 11;

}

//P1 waits if Turn1 reads 11

}

releaseCS(){

WantCS[1] = FALSE:

Turn1= 0;

}

1. Mutual Exclusion:

Suppose both P0 and P1 have entered CS. Then both while loops were passed, so

(!WantCS[1] || Turn1 != 1) && (!WantCS[0] || Turn0 < 10)

For P0 and P1 to both enter CS, their respective WantCS[] were set to TRUE, so it must be that

(Turn1 != 1 && Turn0 < 10)

This is a contradiction because for P0 to reach its while loop, Turn0 must have been set to either 10 or 11. Then it is not possible for both P0 and P1 to enter CS.

2. No Deadlocks:

Suppose there is a deadlock, that P0 and P1 are stuck at while loop.

Case 1:

P0 is stuck at: while(Turn1 > 1){} and P1is stuck at its only loop: while (WantCS[0] && Turn0 >= 10). This is not possible because for P0 to reach that stage its own Turn0 variable is reset to 0. P1 would be able to enter CS.

Case 2:

P0 is stuck at: while (WantCS[1] && Turn1 == 1) and P1is stuck at its only loop: while (WantCS[0] && Turn0 >= 10). This means that the following is satisfied:

(WantCS[1] && Turn1 == 1&& WantCS[0] && Turn0 >= 10)

That is a contradiction because for above to be true, it must be that (Turn1 == 1 && Turn0 >= 10) through every loop. However, within P1’s requestCS() we have that:

if(Turn0 >= 10 && Turn1 == 1)

Turn1 = 11;

Then it can’t be that Turn1 == 1 for every loop.

Since for both cases there cannot be a deadlock, the modified algorithm satisfies the progress property.

3. Starvation Freedom:

Suppose that the algorithm does not satisfy starvation freedom. There are two cases: P0 cannot access CS, or P1 cannot access CS.

Case 1, P0 faces starvation:

We have already established that there are no deadlocks. Then for P0 to be “starved” it must be that P1 has accessed and re-accessed CS without giving P0 an opportunity to enter. However, by the time P1 wants to re-access CS, Turn0 is either 10 or 11, so P1 has to wait. Since there are no deadlocks, P0 must be able to go.

Case 2, P1 faces starvation:

Suppose that P0 can access and re-access CS without allowing P1 to enter. Turn1 is set to 11 if both parties want to access CS (from the if statement within the while loop). When P0 wants to re-access CS it is stopped at while(Turn1 > 1). Because there are no deadlocks, P1 can enter CS.

In both cases, P0 and P1 cannot access and re-access CS if the opposite party is “waiting”. Then the modified algorithm is starvation free.