**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;

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. Prove mutual exclusion:

If both P0 and P1 want CS, they still cannot access simultaneously because either P0 reads Turn1 first or P1 reads Turn0 first.

Case 1, P0 reads Turn1 first:

Turn0 = Turn1+10, and Turn1 must be 0 because it needs to read Turn0 before it can set Turn1 to anything else. Then Turn0=10. P1 sets Turn1 to be either 11 or 1 depending on what it reads. Either way, P1 gives P0 priority and waits. P0 enters CS because Turn0=10. Upon P0’s exit, it clears Turn0 so P1 can enter CS.

Case 2, P1 reads Turn0 first:

Turn1 = Turn0 + 1, and Turn0 must be 0 because it needs to read Turn1 before it can set Turn0 to anything else. Then Turn1=1. P0 sets Turn0 to be either 11 or 10. If Turn0 is 10, P0 enters CS and P1 waits. If Turn0 is 11,

P1 sets Turn1 to be either 11 or 1 not sure actually

2. Progress, and 3. Starvation Freedom: