Matthew Zhan, Robert Golshan

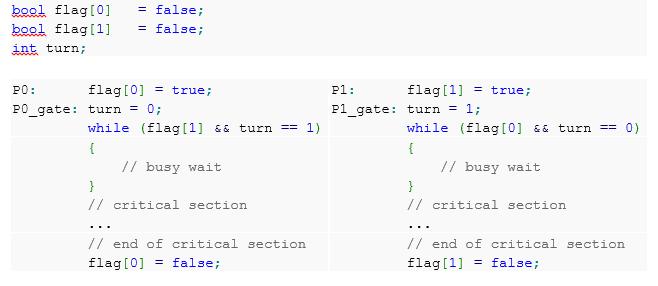
Myz84 rpg499

2/3/2015

HW 1

3. Peterson’s Solution

A)



If the processes set the turn flag for itself instead of setting it for the other process, then mutual exclusion can be violated. For example, suppose initially that P0 is invoked…

P0: flag[0] = true

P0\_gate: turn = 0;

(flag[1] && turn ==1 ) => (false && false) => false, no busy-wait

P0 CS…

P0 is currently in the CS, but consider when P1 is also invoked…

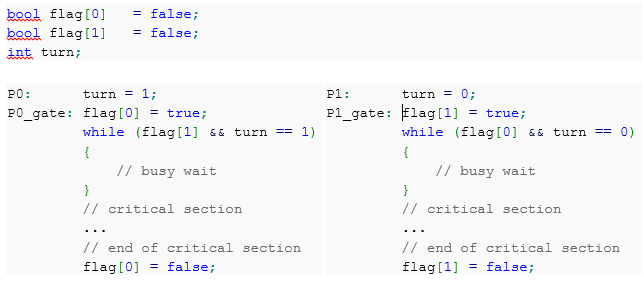
P1 flag[1] = true

P1\_gate: turn = 1;

(flag[0] && turn == 0) => (true && false) => false, no busy-wait

P1 CS… (P0 still in CS)

The condition for P1’s busy-wait is false even though P0 is in the CS, violating mutual exclusion.

B) 

If the turn is set before the wantCS variable, mutual exclusion can be violated. For example, consider the following execution when initially P1 is invoked…

P1 turn = 0;

P0 turn = 1;

P0 flag[0] = true;

P0 (flag[1] && turn == 1) => (false && true) => false

P0 CS…

P1 flag[1] = true;

P1 (flag[0] && turn == 0) => (true && false) => false

P1 CS… (P0 still in CS)

In this example, P1 is requesting the CS and sets the turn for P0. Now consider when P0 is invoked, it sets the turn for P1, and sets flag[0] to true, and is able to enter the CS because

P0 (flag[1] && turn == 1) => (false &&true) => false

So P0 has no busy-wait and enters the CS. The problem occurs when P1 wants to enter the CS, now P1 flag[1] will be set to true, and the busy-wait condition

P1 (flag[0] && turn == 0) => (true && false) => false

Will also be false, so P1 can enter the CS while P0 is already in the CS, violating mutual exclusion.

4. Without choosing variables for the bakery algorithm, there is a possibility that two processors believe they have the CS, which violates mutual exclusion.

For example, consider when two processors P0 and P1 are initially requesting CS at the same time and receive the same ticket number and the following sequence occurs, where all processors start at 0

P1 begins choosing a number [0, 0]

P0 begins choosing a number [0, 0]

P1 chooses 1 [0, 1]

P1 begins checking if it is the smallest number

P1 ignores P0 choosing, so it does not wait [0< , 1]

P0 chooses 1 [1, 1]

P0 begins checking if it is the smallest number [1<, 1]

P1 passes the busy-waits and enters the CS [1, 1<]

P0 passes the busy-waits (P0 and P1 are both 1, but 0 < 1) with the biggest priority in a tie number and enters the CS

The issue is that not having choosing variables allows for processors to skip over some necessary busy waits. If P0 and P1 are both assigned the same ticket, P0 should always go before P1 because it has the higher priority number of 0. No choosing variables can cause the processors to prematurely enter the CS and violate mutual exclusion.

5.

A starvation free algorithm means that any thread that requests the CS will eventually get the CS. Peterson’s algorithm involves two threads, so the scenario where a starved thread occurs is when one of the threads will never enter the CS.

Suppose that Peterson’s algorithm is not starvation free and that say that for threads P0 and P1, P0 is stuck in a busy-wait. The possible actions for P1 include: no action, busy-wait, or constantly entering its CS.

Case 1: P1 no action

P1 flag[1] = false  
P0 (flag[1] && turn == 1) => (false && XXXX) => false

P0 can enter the CS

Case 2: P1 busy-wait

P0 (flag[1] && turn == 1)   
P1 (flag[0] && turn == 0)

P0 and P1 cannot both be in a busy wait because turn is either 0 or 1, but both flag[0] and flag[1] are true so at least one of the threads will always be able to proceed.

Case 3: P1 repeatedly entering its CS

P0 is in a busy-wait, so P0 (flag[1] && turn == 1) => true && true  
As soon as P1 finishes its CS, flag[1] = false  
Then P0 (flag[1] && turn == 1) => (false && true) => false

P0 can enter the CS

Each case has a contradiction where P0 will enter the CS, therefore the original supposition that a thread can be starved is false, and the algorithm is starvation free. Additionally the algorithm is starvation free for N processors because there are only more instances of identical 1 to 1 comparisons of this algorithm.

6.

P0 can write to flag[0] and turn[0]  
P1 can write to flag[1] and turn[1]

|  |  |
| --- | --- |
| bool flag[0] = false;  bool flag[1] = false;  int turn[0] = 0;  int turn[1] = 0;  int local[0] = 0;  int local[1] = 0; | |
| P0: flag[0] = true;  P0\_gate: local[0] = turn[1];  Turn[0] = local[0];  while (flag[1] &&  local[0]==turn[1])  {  // busy wait  }  // critical section  ...  // end of critical section  flag[0] = false; | P1: flag[1] = true;  P1\_gate: local[1] = 1 – turn[0];  Turn[1] = local[1];  while (flag[0] &&  local[1]!=turn[0])  {  // busy wait  }  // critical section  ...  // end of critical section  flag[1] = false; |