UT EID: **jad5348**, **apb973**

2 February 2017

1. **jdomingo.** Created account using UT system credentials.
2. Show that any of the following modifications to Peterson’s algorithm makes it incorrect:
   1. **Violates mutual exclusion**. The process will never busy-wait for the other process due to the “turn == j”condition, potentially resulting in both processes entering and executing the critical section code simultaneously. The process that arrives at the doorway latest goes in first, violating the “courteous” principle of Peterson’s algorithm.
   2. **Violates mutual exclusion**. Consider the following counterexample:
      1. Process 0: turn = 1;
      2. Process 1: turn = 0;
      3. Process 1: wantCS[1] = true;
      4. Process 1: while (wantCS[0] && turn == 0) evaluates to false.
      5. Process 1 enters the critical section.
      6. Process 0: wantCS[0] = true;
      7. Process 0: while (wantCS[1] && turn == 1) evaluates to false.
      8. Process 0 enters the critical section.
      9. Both processes in critical section, race conditions can occur.
3. Show that the bakery algorithm does not work in the absence of choosing variables:

* **Violates mutual exclusion.** The priority order for the processes won’t be correct. Consider two processes that both start choosing their number such that they would both be assigned the same number where process **p1** with a higher process ID happens to be in the process of choosing its number slightly after process **p2** with a lower process ID. Since **p1**, in the process of choosing the max number, sees **p2** is unassigned, **p1** will enter the critical section. Similarly, once **p2** has finished choosing the max number, it sees that **p1** is also the same max number. However, since **p1** has a lower process ID, it enters the critical section resulting in the mutual exclusion problem.

1. Prove that Peterson’s algorithm is free from starvation:

* Consider two processes **p1** and **p2**.
  1. Case where **p1** is in critical section and **p2** requests critical section and vice versa:
     1. Whenever **p1** releases the critical section, it sets its corresponding **wantCS** to false, causing **p2**’s busy-wait to end and allowing **p2** to enter the critical section exclusively.
     2. Whenever **p2** releases the critical section, it sets its corresponding **wantCS** to false, causing **p1**’s busy-wait to end and allowing **p1** to enter the critical section exclusively.
  2. Case where **p1** and **p2** request the CS simultaneously:
     1. Whichever process requested CS slightly before the other process will enter the CS first. A set number of steps are executed in the CS. Then, the same reasoning for **a.** applies (the process in CS releases CS, enabling the other process to enter the CS).
  3. Case where a process doesn’t need to quest the CS:

P1: If **p1** doesn’t require the CS, then there is no starvation to begin with.

P2: If **p2** doesn’t require the CS, then there is no starvation to begin with.

* In all of these cases, either process will eventually enter the critical section. Therefore, the algorithm is free from starvation.

1. Modify Peterson’s so it uses two variables **turn0** and **turn1** instead of a multi-write **turn**:

Class PetersonAlgorithm implements Lock {

Boolean wantCS[] = {false, false};

int turn0 = 1;

int turn1 = 0;

public void requestCS(int i) {

int j = 1 – i;

if (i == 0) {

turn0 = turn1;

}

else {

turn1 = turn0;

}

wantCS[i] = true;

if (i == 0) {

while (wantCS[j] && turn0 == 1) {

turn0 = turn1;

}

}

else {

while (wantCS[j] && turn1 == 0) {

turn1 = turn0;

}

}

}

public void releaseCS(int i) { }

if (i == 0) {

turn0 = 1;

}

else {

turn1 = 0;

}

wantCS[i] = false;

}

}