Nicholas White
Nww295

5 0

Concurrent: Assignment 1 02/06/18

Written Questions:

3.) TACC username: nwhite

4.) Show any of the following make Peterson's incorrect:

a.) Process sets a turn variable to itself.

Peterson's algorithm states there are three critical variables for sharing memory. want[i] and want[j] are booleans for processes i and j, and turn is the process id of whose turn it is.

T1	T2
want[i] = true	want[j] = true
turn = i	turn = j
while (want[j] && ~~tu~~turn == i) {}	while(want[j] && turn == j) {}
want[i] = false	want[j] = false

As seen by the example code, if process T1 were to set its turn variable to its own process id, then the while() condition does not hold true. Thus, process T2 is still in the critical section while T1 has entered it. Thus there are two processes in the same critical section, thus incorrect.

b.) Sets turn before setting wantCS:

4) b.) continued:

code:
```
public void RequestCS (int i) {
    int j = 1-i
    turn = j
    wantCS [i] = true
    while (wantCS[j] && turn = j) {}
}
```

Setting turn variable before wantCS variable to break Peterson's algorithm due to a violation of mutual exclusion. Say there are two threads, thread 0 and 1, and thread 0 calls requestCS, but the OS makes a switch and starts executing thread 1 after thread 0 set turn = j. Thread 1 was thus able to ~~enter the~~ exit the while (wantCS[j] && turn = j) before thread 0 set wantCS [i] to true. Because thread 0 has already entered the critical section, and thread 0 has already executed turn = j, then the turn variable results in false and thread 0 is able to enter the critical section. Therefore it violates the principal of mutual exclusion.

Nicholas White

Nww295

02/06/18

Concurrent and Distributed Systems

Assignment 1

5.) Modify Peterson's algorithm to use two variables turn0 and turn1 such that no process writes to the same variable:

```
class PetersonAlgorithm implements Lock {
  boolean wantCS[] = {false, false};
  int[] turn = {0, 0};
  public void requestCS(int i) {
    int j = 1 - i;
    wantCS[i] = true;
    turn[i] = turn[j] + 1;
    while(wantCS[j] && ((turn[i] == turn[j])||(turn[i] > turn[j]))) {
      if(turn[i] == turn[j]) {
        turn[i] = turn[j] + 1;
      }
    }
  }

  public void releaseCS(int i) {
    wantCS[i] = false;
    turn[i] = 0;
  }
}
```

In the example above, we will prove the mutual exclusivity property by contradiction. Suppose that in the algorithm above, two processes are able to enter the critical section at the same time.

In order to enter the critical section, some condition in each of the threads while loop must hold false. Let us assume two threads, thread 0 and thread 1. Suppose thread 0 enters the critical section first. In the while loop of thread 1, wantCS[0] evaluates to true. The only conditions left to be false are turn[1] == turn[0] or turn[1] > turn[0]. Because turn[1] = turn[0] + 1, then turn[1] will always have a higher value than turn[0], therefore never allowing thread 1 to enter the critical section while thread 0 is in the critical section. Thus, this contradicts our assumption that two processes are able to enter the critical section. The algorithm proves the mutually exclusive property of progress because if both thread 0 and thread 1 have turn = 0, then one of the threads' while loop will increment their turn value to be greater than the other thread, allowing only one to go into the critical section. The property of starvation holds because if there is a thread 0 trying to enter the critical section and thread 1 (the thread previously in the critical section) attempts to request the critical section, eventually thread 0 will have a lower turn value than thread 1, giving the thread 0 a higher priority and allowing it to enter the critical section.

6.) Bakery algorithm without choosing variable:

```
// without the choosing variable
  public void requestCS(int i) {
    for(int k = 0; k < N; k++) {
      if(number[k] > number[i]) {
        number[i] = number[k];
      }
    }
    number[i]++;

    for(int k=0; k < N; k++) {
            while((number[k] != 0)&&((number[k] < number[i])||(( number[k]==number[i]) &&
k<i)))){}
      }
  }
```

In the above algorithm, the choosing variable is removed for the bakery algorithm. Suppose thread 0 calls requestCS first. All values in the numbers array are initialized to 0 and after the first for loop, number[0] = 0. Suppose then the processor switched to thread 1 before thread 0 executes number[0]++. Thread 1 will then hold a value of numbers[1]==1 after numbers[1]++. When thread 1 executes the while loop, all conditions hold true and thread 1 is able to enter the critical section. Assume this is when thread 0 resumes execution. Thread 0 increments numbers[0] to 1, which is the same as numbers[1] of thread 1. When thread 0 enters the second for loop, all conditions hold true except j < i. In this case, j = 1 and i = 0, resulting in a false condition. Therefore, thread 0 is allowed to enter the critical section, violating the properties of mutual exclusion. The bakery algorithm breaks when the choosing variable is removed.