EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 3: September 1

Lecturer: Vijay Garg *Scribe: Travis Brannen*

3.1 Introduction

This class centered around analysis of various methods to achieve mutual exclusion in a concurrent environment. First, we review the proof of Peterson's Algorithm from last time. Next, we extend Peterson's so that it can be used for more than two processes ultimately arriving a tthe Filter Algorithm. We then briefly cover another extension of Peterson's algorithm, the so called Tournament. We briefly discuss the issue of multiwriter variables. Finally, we discuss another algorithm for concurrent mutual exclusion, the Bakery algorithm.

3.2 Peterson's Algorithm

Previously we developed Peterson's algorithm, and java code for the algorithm can be found in [1]. We also added, for the purpose of our proof, the boolean array *trying*. Here is PetersonAlgorithm.java with the variable *trying*:

```
class PetersonAlgorithm implements Lock {
    boolean wantCS[] = {false, false};
    boolean trying[] = {false, false};
    int turn = 1;
    public void requestCS(int i) {
        int j = 1 - i;
        trying[i] = true;
        wantCS[i] = true;
        turn = j;
        while (wantCS[j] && (turn == j)) ;
        trying[i] = false;
        // process i enters critical section after returning from this method
    }
    public void releaseCS(int i) {
        // process i leaves critical section upon setting wantCS[i] to false
        wantCS[i] = false;
    }
}
```

We review the proof for mutual exclusion that takes advantage of *trying*:

Consider predicate H(0), where:

$$H(0) \equiv wantCS[0] \wedge [(turn == 1) \vee ((turn == 0) \wedge trying[1])]$$

$$H(1) \equiv wantCS[1] \wedge [(turn == 0) \vee ((turn == 1) \wedge trying[0])]$$

P_0 makes $H(0)$ true just before entering the while loop by setting $turn$=1. P_1 makes $H(1)$ true just before entering the while loop by setting $turn$=0.

P_1 cannot falsify $H(0)$.

- Only P_0 can change the value of $wantCS[0]$.
- Additionally,
$$H(0) \Rightarrow turn == 1 \vee (turn == 0 \wedge (trying[1]))$$
Moving line by line through $requestCS(1)$, one can see that although P_1 can change $trying[1]$ and $turn$, $turn$ is only equal to 0 while $trying[1]$ is true. From symmetry, P_1 also cannot falsify $H(0)$.

Now, we will use the above in our proof of mutual exclusion by contradiction. First we suppose that both P_0 and P_1 are in their critical section, a violation of mutual exclusion. For this to be true, $trying$=$\{false, false\}$ since both processes have returned from $requestCS$. Also, $H(0)$ and $H(1)$ are true since both processes made their corresponding predicate true before entering the while loop and the predicates were not made false by any process since then. Therefore,

$$BothInCriticalSection \equiv \neg trying[0] \wedge H(0) \wedge \neg trying[1] \wedge H(1) \Rightarrow turn == 0 \wedge turn == 1 \Rightarrow false$$

/sectionFilter Algorithms Now we try to extend Peterson's Algorithm so that it can be used for N concurrent processes. To do so, we get rid of the *turn* variable and replace it with a variable called *last*. This is to eliminate the need for processes to explicitly reference other processes in their code.

Pseudocode for PetersonN base algorithm:

```
wantCS[i] = true
last = i
while( (exists j such that j!=i and wantCS[j]) and last==i )
***CRITICAL SECTION***
wantCS[i] = false
```

This is very similar to Peterson's algorithm, but close analysis reveals that it will allow N-1 processes to enter the critical section. However, running N-1 of these "gates" in series allows us to go from N processes, to N-1, to N-2, and so on until only one process exits the filter to enter the critical section at a time. This gives us an algorithm to enforce mutual exclusion among N processes such that only 1 process enters the critical section at a time. See PetersonN.java for an implementation of this algorithm[1].

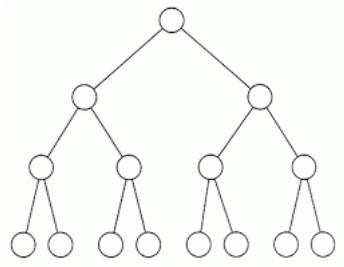
There are two new variables used in this algorithm. *gate* is an array with N integers, with the ith position representing the gate P_i is currently waiting behind. *last* is an array of integers with the value for last for each gate. Since we only need N-1 gates, the 0 position of *last* will be unused. Also, $gate$[i]=0 indicates that P_i is not currently attempting to access the critical section.

This algorithm may also be modified to allow any number of processes between N-1 and 1 into the critical section, for resources that may be shared between a limited number of users.

Space Complexity: O(N) Time Complexity: O(N^2)

3.3 Tournament

The nested for loops in PetersonN.java mean that it is not very temporally efficient. Using a tournament structure as shown below would take less time. Each process would enter a Peterson's Algorithm lock competing with one other process at the bottom. The winner would compete with the winner of an adjacent contest and so on. After passing through log_2(N) such locks, one process would enter the critical section at a time.



3.4 Multwriter Variables

SRSW - Single reader single writer
MRSW - Multireader single writer
MRMW - Multireader Multiwriter

SRSW does not pose a concurrency problem. MRSW allows for sequential consistency but locks are required for strict consistency. MRMW is the most difficult to implement since without locks even sequential consistency cannot be relied on.

3.5 Bakery

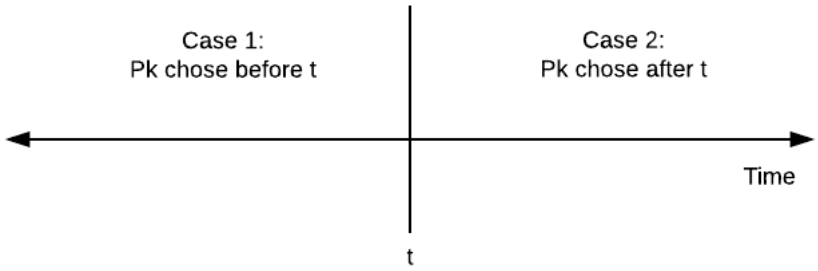
The Bakery Algorithm is another method of locking a shared resource so that only 1 of N processes may use it at once. The basic idea is that there are two steps to acquiring the lock. First, you come in through the "doorway" of the bakery and take a number. This number should be one higher than everyone who is already inside the bakery. Second, you wait until your number is the lowest number of anyone inside the bakery. Because of concurrency, it is possible that two processes will enter the bakery simultaneously and get the same number. Because of this, sequential, persistent process IDs from 0 to N-1 are issued to each process and in the case where two processes have the same number the one with the lower process ID enters the critical section first.

A full java implementation is given in Bakery.java[1]. There are two variables of interest *choosing* and *number*. P_i sets $choosing[i]$ to true when it enters the doorway and to false once it has a *number*. The *choosing* variable allows a process to ensure that any other processes that were choosing while it was choosing have finished by the time it attempts to enter the critical section. $number[i]$ is set by P_i in the doorway to be equal to $max(number) + 1$ and used to determine who should get in to the critical section. *number* is initialized to an array of all zeros because zero is used as a flag indicating that a process is not attempting to enter the critical section. Also for this reason, $number[i]$ is set to 0 after P_i exits the critical section.

Note that in the for loop checking if a process has the smallest number you need not explicitly take care of the case where j=i since the while condition will always return false in this case.

Check for Mutual Exclusion
If P_i is in the critical section and P_k (s.t. k!=i) has already chosen its number, then $(number[i], i) < (number[k], k)$. Then, assume P_i and P_k are in the critical section. This is a contradiction by the following:

t=time when P_i checked $choosing[k]$ and found it false.



In both Case 1 and Case 2, $(number[i], i) < (number[k], k)$.

3.6 Final Thoughts

In Java, even with locking algorithms you can still have mutual exclusion. This is because by default sequential consistency is not enforced. Multiple copies of variables are stored by main memory and the caches for each processor. You can use the volatile keyword for variables and the synchronized keyword for functions to enforce sequential consistency. The atomic keyword for variables can also be used to implement test and set.

References

[1] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/tree/master/chapter1-threads