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

Lecture 3: September 1

Lecturer: Vijay Garg *Scribe: Aditi Ranganath*

3.1 Introduction

In this lecture we will discuss various mutual exclusion protocols that work for n threads, where n is greater than 2. The scope of this lecture extends to the following algorithms :

1. Proof of Peterson's algorithm (Review)
2. Filter algorithm (Peterson-n algorithm)
3. Tournament algorithm
4. Bakery algorithm

3.2 Peterson's Algorithm (continued)

So far we have developed an informal and a formal proof (Dijktra's proof) for Peterson's two-thread algorithm and discussed the concepts of *Deadlock Freedom* and *Mutual Exclusion* for the same.

Thread 0 (P_0):	Thread 1 (P_1):
wantCS[0] = true;	wantCS[1] = true;
turn = 1;	turn = 0;
while (wantCS[1] && (turn==1)) ;	while (wantCS[0] && (turn==0)) ;
$<$ *critical section* $>$	$<$ *critical section* $>$
wantCS[0] = false;	wantCS[1] = false;

Table 3.1: Peterson's algorithm for two threads P_0 and P_1.

In case of *Deadlock Freedom*, combining the conditions in which both P_0 and P_1 are waiting leads to contradiction, hence proving that the algorithm is starvation-free or deadlock-free. That is:

$$(wantCS[1] \;\&\&\; (turn == 1)) \;\&\&\; (wantCS[0] \;\&\&\; (turn == 0))$$

On simplifying, we get (turn==1) && (turn==0) which cannot be true.

In case of *Mutual Exclusion*, we used auxiliary variables $trying[0]$ and $trying[1]$ to prove by Dijktra's method that P_1 cannot falsify predicate $H[0]$ set by P_0 and vice versa, where, $H[0]$ is defined as:

$$H[0] = wantCS[0] \;\&\&\; ((turn == 1) || ((turn == 0) \&\& (trying[1])))$$

On simplifying, we again get the contradiction case where (turn==1) && (turn==0) which cannot be true. Thus, Peterson's algorithm for two threads is both *Deadlock Free* and *Mutually Exclusive* [1].

3.3 Filter Algorithm: Peterson-n Algorithm

We now try to extend Peterson's mutual exclusion protocol to work for $n(>2)$ threads. For this, we keep the algorithm to be symmetric and instead of the semantic *turn*, we use the variable *last*. We expect this to work well as it is easier to know which process wrote into the shared variable at the end. Thus, for P_i processes, where $i \in \{0, 1, 2, .., N-1\}$, we have

```
wantCS[i] = true;
last = i;
while( (∃ j: j≠i: wantCS[j]) && (last==i) ) ;
< critical section >
wantCS[i] = false;
```

Now, lets examine if this is *Mutually Exclusive* : Consider three processes P_0, P_1 and P_2. If P_2 was the last to write into the shared variable, only P_2 waits. Both P_0 and P_1 can now enter the critical section. This is not good as there is no *Mutual Exclusion*. If n threads are at the gate at the same instance of time, only the last one is waiting while the remaining $(n-1)$ enter. In order to ensure only one thread enters the *critical section*, we have to repeat this process $(n-1)$ times. Thus, at each of the $(n-1)$ gates, we have one thread waiting which allows only one thread to enter the *critical section* thereby ensuring *Mutual Exclusion*. With this, the algorithm looks as follows:

```
int n;
int [n]gate;
int [n]last init 0;                                   // last[0] will not be used
for (int k=1 ; k<n ; k++) {                           //entry protocol
    gate[i]=k;                                        // P_i is at gate k now
    last[k]=i;                                        // P_i updates variable last for that gate
    for (int j=0 ; j<n ; j++) {
        while( (j≠i) && (gate[j]≥k) && (last[k]==i) ) ;
    }                                                 //inner for-loop
}                                                     //outer for-loop
< critical section >
gate[i]=0;                                            //exit protocol
```

where, i is the process index and k is the gate index. Every process should go through $(n-1)$ gates to enter the *critical section*. Thus, Filter Algorithm can be visualized to be stacking of Peterson's algorithm on one another $(n-1)$ times with the following complexity:
Space complexity : $O(N)$
Time complexity : $O(N^2)$
On further analysis of the above algorithm, it can be shown that if process P_i is pausing at any point, other processes P_j, P_k, etc. can enter *critical section* overtaking P_i arbitrary number of times.

3.4 Tournament Algorithm

Another simple technique to extend the use of a two-thread mutual exclusion algorithm for n threads in using the Tournament Algorithm. In this case, each thread is progressing from the leaf to the root of the tree by participating in a two-thread mutual exclusion algorithm at every step. Thus, a thread has to pass through log_2(N) locks to enter the *critical section*.

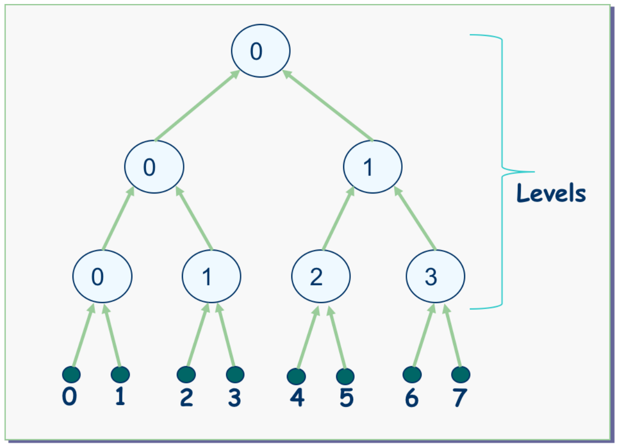

Figure 3.1: Tournament tree with multiple nodes and levels

3.5 Variable read-write Atomicity

A programming language can have three types of variables:

- SRSW : single-reader, single-writer variable
- MRSW : multi-reader, single-writer variable
- MRMW : multi-reader, multi-writer variable

Intuitively, in both SRSW and MRSW, the write order is deterministic. SRSW does not have any concurrency issues. MRSW also possesses sequential consistency but needs some checks for certain scenarios where concurrent reads can possibly return different values. However, in case of MRMW, the writes depend on the time stamps at which the request was made, and hence it requires more checks and locks.

In *Filter algorithm*, we use a MRMW variable *last* which requires some sort of atomicity and mutual exclusion for writes. Thus *Filter algorithm* assumes mutual exclusion for the multi-writer variable. This scenario can be avoided using the Lamport's Bakery Algorithm.

3.6 Bakery Algorithm

The algorithm was developed by Leslie Lamport with an analogy of a bakery with a numbering machine at its entrance so each customer is given a unique number. Numbers increase by one as customers enter the store. A global counter displays the number of the customer that is currently being served. All other customers must wait in a queue until the baker finishes serving the current customer and the next number is displayed. When the customer is done shopping and has disposed the number, the clerk increments the number, allowing the next customer to be served. That customer must draw another number from the numbering machine in order to shop again [2]. Extending this analogy, we have a scenario where every thread has a notion of its own number that can be written on only by itself but multiple threads can read from it. Thus, the algorithm uses a MRSW variable and eliminates the need for MRMW variable. The algorithm works as follows:

1. Every thread enters through a 'doorway' where it takes a number. This thread reads other threads'

numbers to ensure it gets the biggest number.

2. Ideally, only one thread should be at the doorway at a given time to get its number. However, it is possible that two threads enter the bakery at the same time and get the same number. To avoid concurrency issues, a unique ID is issued sequentially from 0 to $(n-1)$ to n threads that enter simultaneously.
3. Once a thread is inside the bakery, it waits till its number is the lowest in the bakery. The lowest thread enters the critical section.
4. If multiple threads have the same number, the one with the lowest unique ID enters the critical section.

```
boolean [ ]choosing;                                   //init false
int [ ]number;                                         //init 0
choosing[i]=true;                                      //Step 1
int t=max(number[0],.....,number[n-1]);
number[i]=t+1;
choosing[i]=false;
for(int j=0; j<n; j++) {                               //Step 2
  while(choosing[j]) ;                                 //avoid conflict
  while((number[j]≠0)&&(number[j]<number[i])||(number[j]==number[i]&&(j<i)));
}                                                      //end of for loop
< critical section >
number[i]=0;
```

Proof:
If P_i is in critical section and P_k (k≠i) has already chosen its number, then, $(number[i], i) < (number[k], k)$
Let t be the time whrn P_i checked choosing [k] and found it false. We have the following two cases:

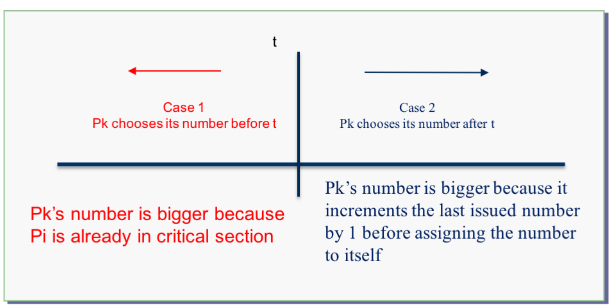

Figure 3.2: Two cases in Bakery algorithm

Thus, no matter when the number is chosen, the thread in the *critical section* is the one with the smallest number.

References

[1] V.K. Garg Introduction to Multicore Computing

[2] https://en.wikipedia.org/wiki/Lamport