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

Lecture 6: September 13

Lecturer: Vijay Garg *Scribe: Sarang Bhadsavle*

Agenda and Announcements

Today's lecture covered the following topics:

- The lower bound on the number of shared locations required for mutual exclusion
- Fischer's Algorithm (A timing-based algorithm)
- Lamport's Fast Mutex Algorithm
  - Splitter Construct

Additionally, remember to watch the video seminar *Toward Extreme-Scale Manycore Architectures* and the video lectures on Semaphores from last week, as they will be covered on the first exam.

6.1 Lower Bound on Number of Shared Locations

Question: Is there any algorithm which, by using just one shared variable, we can achieve mutual exclusion?

Definition 6.1 *Covering State*

The notion of a state in which all shared variables are about to be overwritten by processes and the shared state is consistent with no process in the critical section.

Theorem 6.2 *(Barnes and Lynch) Any mutex algorithm that only uses read-write variables on n processes requires at least n shared locations.*

Proof: (n = 2)

Let P and Q be processes and let A be a shared memory location whose initial value is $\perp$. Let Q run until it is about to write to A, i.e. it is in the Covering State. Now let P run and enter the critical section. Let Q run again. Q writes to A and enters the critical section.

$\therefore$ Two processes are simultaneously in the critical section

$\therefore$ Violation of mutual exclusion ■

This proof can be extended to $n > 2$ by following a similar sequence of events to achieve a Covering State, and then letting processes run and enter the critical section simultaneously, causing a violation of mutual exclusion.

6.2 Fischer's Algorithm

It is possible to achieve mutual exclusion of n processes with a single shared variable, provided that a key assumption is made about the timing of the algorithm. The algorithm along with this assumption is shown below.

Algorithm 1 Fischer's Algorithm (for a process P_i)

```
shared var turn = -1; // door is open

procedure REQUESTCS
    while(true):
        while(turn != -1):
            noOp();
        turn = i;
        wait for △t time units;
        if(turn == i):
            return;

procedure RELEASECS
    turn = -1;
```

The key assumption with Fischer's Algorithm is that $\triangle$t $\geq c$, where c is is the maximum time required to close the door i.e. the time required to set *turn* to *i*.

Without this assumption, the following sequence of events could occur and cause a violation of mutual exclusion for two processes, P_i and P_j:

1. P_i reads turn = -1
2. P_j reads turn = -1
3. P_j sets turn = j
4. P_j reads turn = j, and enters critical section
5. P_i sets turn = i
6. P_i reads turn = i and enters critical section

Fischer's algorithm satisfies mutex by ensuring that the time period that a process P_j waits after setting turn = j and before checking if turn == j is greater than the time period that another process P_i takes after reading turn == -1 and before setting turn = i. This ensures that only one process (in this case, P_i) can enter the critical section.

6.3 Lamport's Fast Mutex Algorithm

Lamport's Fast Mutex Algorithm is used to provide mutual exclusion in a very fast way when there is no contention for the critical section. When there is contention, the regular method of mutual exclusion is followed.

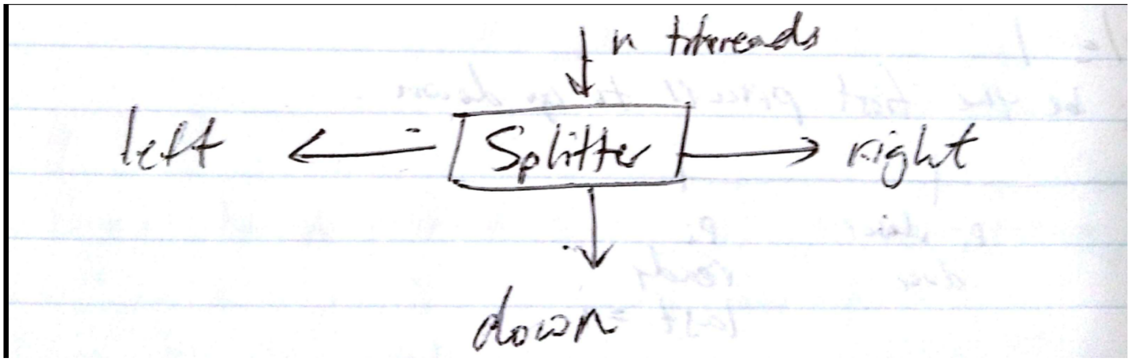

Figure 6.1: A Splitter Construct

Splitter

The construct of a "splitter" is used in Lamport's Fast Mutex algorithm, as seen in Figure 6.1.

The Splitter guarantees that:

- The number of processes sent right $\leq$ n - 1
- The number of processes sent left $\leq$ n - 1
- The number of processes sent down $\leq$ 1

Algorithm 2 Splitter Construct (for a process P_i)

shared var *door* : $\{open, closed\}$ *init* open;
shared var *last* : pid *init* -1;

last $=$ *i;*
if(door $==$ *closed):*
    return *left;*
else:
    door $=$ *closed;*
    if(last $==$ *i):* return *down;*
    else: return *right;*

Claim 6.3 $\|$*left*$\|$ $\leq$ *n - 1*

Proof: Some process *must* have closed the door. ■

Claim 6.4 $\|$*right*$\|$ $\leq$ *n - 1*

Proof: Consider the process P_i such that textitlast = textiti. Then P_i must be part of left or down. ■

Claim 6.5 $\|$*down*$\|$ $\leq$ *1*

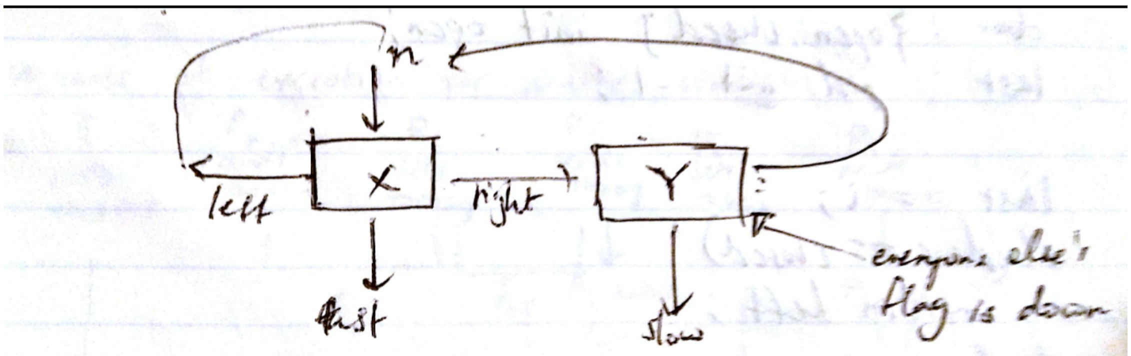

Figure 6.2: Design of Lamport's Fast Mutex Algorithm using Splitter

Proof: Let a process P_i be the first process to go down. In order to go down, P_i must execute 3 steps:

1. P_i writes *last*
2. P_i closes *door*
3. P_i reads $last == i$

Suppose there is some other process P_j also in the splitter. P_i must write *last* at some point. There are 3 cases:

- Suppose P_j writes *last* before 1. Then P_j would be the first process to go down, which violates the initial supposition that P_i went down first
- Suppose P_j writes *last* after 1 but before 3. However, this is not possible because P_i read $last == i$.
- Suppose P_j writes *last* after 3. However, this is not possible because P_j could not have found *door* to be open in the first place.

$\therefore$ Since all 3 cases are shown to be impossible, there can only be at most one process that goes down in the Splitter. ■

Lamport's Fast Mutex algorithm can be constructed using Splitter, as seen in Figure 6.2.

Proposition 6.6 *Note that Lamport's Fast Mutex Algorithm does not guarantee starvation-freedom*

Algorithm 3 Lamport's Fast Mutex Algorithm (for a process P_i)

```
procedure ACQUIRE
    while(true):
        flag[i] = up;
        x = i;
        if(y != i): // splitter's left
            flag[i] = down;
            waitUntil(y == -1):
            continue;
        else:
            y = i;
            if(x == i): return; // fast path
            else: // splitter's right
                flag[i] = down;
                waitUntil(∀j : flag[j] == down);
                if(y == i): return; // slow path
                else:
                    waitUntil(y == -1);
                    continue;

procedure RELEASE
    y = -1;
    flag[i] = down;
```

References

[1] Vijay Garg. EE382C, Multicore Computing, The University of Texas at Austin. September 2016.