Lamport's Fast Mutual Exclusion Algorithm

Vijay K. Garg

1 Introduction

In this handout, I describe an algorithm due to Lamport that allows fast accesses to critical section in absence of contention. The algorithm uses an idea called *splitter* that is of independent interest.

2 Splitter

A *splitter* is a method that splits processes into three disjoint groups: *Left*, *Right*, and *Down*. We can visualize a splitter as a box such that processes enter from the top and either move to the left, the right or go down which explains the names of the groups. The key property a splitter satisfies is that at most one process goes in the down direction and not all processes go in the left or the right direction.

The algorithm for the splitter is shown in Fig. 1.

A splitter consists of two variables: *door* and *last*. The door is initially open and if any process finishes executing splitter the door gets closed. The variable *last* records the last process that executed the statement *last* := *i*.

Each process P_i first records its pid in the variable *last*. It then checks if the door is closed. All processes that find the door closed are put in the group *Left*. We claim

Lemma 1 *There are at most* $n-1$ *processes that return* Left.

Proof: Initially, the door is open. At least one process must find the door to be open because every process checks the door to be open before closing it. Since at least one process finds the door open, it follows that $|Left| \leq n-1$.
■

```
Pi::
 var
      door: open, closed init open
      last : pid initially -1;

 last := i;
 if (door == closed)
      return Left;
 else
      door := closed;
      if (last == i) return Down;
      else return Right;
 end
```

Figure 1: Splitter Algorithm

Process P_i that find the door open checks if the last variable contains its pid. If this is the case, then the process goes in the *Down* direction. Otherwise, it goes in the *Right* direction.

We now have the following claim.

Lemma 2 *At most one process can return* Down*.*

Proof: Suppose that P_i be the first process that finds the door to be open and *last* equal to i (and then later returns *Down*). We have the following order of events: P_i wrote *last* variable, P_i closed the door, P_i read *last* variable as i. During this interval, no process P_j modified the last variable. Any process that modifies *last* after this interval will find the door closed and therefore cannot return *Down*. Consider any process P_j that modifies *last* before this interval. If P_j checks *last* before the interval, then P_i is not the first process then finds last as itself. If P_j checks *last* after P_i has written the variable last, then it cannot find itself as the last process since its pid was overwritten by P_i. ■

Lemma 3 *There are at most $n-1$ processes that return* Right*.*

Proof: Consider the last process that wrote its index in *last*. If it finds the door closed, then that process goes left. If it finds the door open then it goes down. ■

Note that the above code does not use any synchronization. In addition, the code does not have any loop.

3 Lamport's Fast Mutex Algorithm

Lamport's fast mutex algorithm shown in Fig. 2 uses two shared registers X and Y that every process can read and write. A process P_i can acquire the critical section either using a *fast* path when it finds $X = i$ or using a *slow* path when it finds $Y = i$. It also uses n shared single-writer-multiple-reader registers $flag[i]$. The variable $flag[i]$ is set to value *up* if P_i is actively contending for mutual exclusion using the fast path. The shared variable X plays the role of the variable *last* in the splitter algorithm. The variable Y plays the role of door in the splitter code. When Y is -1, the door is open. A process P_i closes the door by updating Y with i.

Processes that are in the group *Left* of the splitter, simply retry. Before retrying, they lower their flag and wait for the door to be open (i.e. Y to be -1). A process that is in the group *Down* of the splitter succeeds in entering the critical section. Note that at most process may succeed using this route. Processes that are in the group *Right* of the splitter first wait for all flags to go down. This can happen only if no process returned *Down*, or if the process that returned *Down* releases the critical section. Now consider the last process in the group *Right* to update Y. That process will find its pid in Y and can enter the critical section. All other processes wait for the door to be open again and then retry.

Theorem 1 *Fast Mutex algorithm in Fig. 2 satisfies Mutex.*

Proof: Suppose P_i is in the critical section. This means that Y is not -1 and P_i exited either with $X = i$ or $Y = i$.

Any process that finds $Y \neq -1$ gets stuck till Y becomes -1 so we can focus on processes that found Y equal to -1.

Case 1: P_i entered with $X = i$
Its flag stays up and thus other processes stay blocked.

Case 2: P_i entered with $Y = i$
Consider any P_j which read $Y == -1$. X is not equal to j otherwise P_j

```
var
    X, Y: int initially -1;
    flag: array[1..n] of {down, up};

acquire(int i)
{
 while (true)
    flag[i] := up;
    X := i;
    if (Y != -1) { // splitter's left
        flag[i] := down;
        waitUntil(Y == -1)
        continue;
    }
    else {
        Y := i;
        if (X == i) // success with splitter
            return; // fast path
        else {// splitter's right
            flag[i] := down;
            forall j:
                waitUntil(flag[j] == down);
            if (Y == i) return; // slow path
            else {
                waitUntil(Y == -1);
                continue;
            }
        }
 }
}

 release(int i)
 {
    Y := -1;
    flag[i] := down;
 }
```

Figure 2: Splitter Algorithm

would have entered CS and P_i would have been blocked

Since $Y = i$, P_j would get blocked waiting for Y to become -1.

■

Theorem 2 *Fast Mutex algorithm in Fig. 2 satisfies deadlock-freedom.*

Proof:

Consider processes that found the door open, i.e., Y to be -1. Let Q be the set of processes that are stuck that found the door open. If any one of them succeeded in "last-to-write-X" we are done; otherwise, the last process that wrote Y can enter the CS. ■