CSC2221: Introduction to Distributed Computing **Fall 2017**

Lecture 9: Mutual Exclusion

Lecturer: Faith Ellen *Scribe: Lily Li*

9.1 Definitions

Mutual exclusion concerns a group of processes that share a critical resource. The program of a process is partitioned into the following sections.

Definition 9.1 *If the process don't care about the resource, it is in the* ***remainder section****. After getting an input event* $entry_i$*, the process enter the* ***entry section****. When the process is granted the resource,* $crit_i$*, an output event occurs and we enter the* ***critical section****. Once the process is done, we have an exit event* $exit_i$ *and enter the* ***exit section****. Upon exiting the protocol we get a* rem_i *to go back to the* ***remainder section****. Each section is actually a set of operations (could be quite simple).*

Consider an admissible execution: assume that all processes initially in the remainder section. Further assume that no process crashes in the critical section (for all p_i if p_i enters the critical section, eventually it will leave). p_i only receives $exit_i$ when it is in the critical section and only receives $entry_i$ if p_i is in the remainder section. Either p_i takes an infinite number of steps or it is in the remainder section after taking its last step (read: only crash in the remainder section).

Mutual exclusion algorithm consists of code for the entry and exit sections; such an algorithm must satisfy:

Mutual Exclusion: two ore more processes are never simultaneously in the critical section.

Bounded Exit: a process finishes the exit protocol in a bounded number of steps.

No Deadlock: if some process is in the entry section and no process is in the critical section then some process will enter the critical section.

9.2 Mutual Exclusion Using Test&Set

(Straight Forward): [Entry Section] test and set the object. [Exit Section] reset the test and set object. It is easy to see that this satisfies the bounded exit property.

Definition 9.2 *A process becomes* ***critical*** *if it is about to enter the critical section (about to perform* $crit_i$*) or just before it enters the exit section (about to perform* $exit_i$*). Here a process is critical if it performs a T&S and gets* 0 *or performs a reset.*

Lemma 9.3 *The above straight forward algorithm satisfies mutual exclusion.*

Proof: Suppose for a contradiction that there is an admissible execution of the algorithm in which two or more processes are critical. Let C be the first such configuration and let p_i and p_j be critical. Let C' be

Algorithm 1 Simple implementation of mutex: code for p_i.

```
1: → entry_i
2: x_i = 1
3: while x_i = 1 do
4:     x_i ← T&S(v)
5: end while
6: crit_i
7:
8: → exit_i
9: reset(v)
10: rem_i
```

the configuration just before C in which one process, say p_j is critical. Then we know that p_i must have performed a T&S and received 0. When p_j entered the critical section it changed the T&S object to 1, thus there must be some other process p_k which changed the T&S object to 0 before p_i performed its T&S. This is a contradiction since p_k and p_j must both be in the critical section, but p_j and p_i was the first just occurrence. ■

Lemma 9.4 *The above straight forward algorithm satisfies no deadlock.*

Proof: Observe that no process is critical (recall what this means) if and only if $v = 0$. This can be proved by induction using the mutual exclusion property. Thus if some process is in the entry section, it will receive 0 from the T&S call and enter the critical section. ■

The problem with the algorithm is starvation. Before we fix this problem, lets introduce the following condition:

Definition 9.5 ***No lockout:*** *(starvation freedom) if some processes is in the entry section then that process will eventually enter the critical section.*

9.3 Mutex with no Lockout

Make use of a queue Q. See pseudo-code 2.

Algorithm 2 No lockout implementation of mutex using a queue Q: code for p_i.

```
1: → entry_i
2: enqueue(Q, i)
3: x_i ← Head(Q)
4: while x_i ≠ i do
5:     x_i ← Head(Q)
6: end while
7: crit_i
8:
9: → exit_i
10: dequeue(Q)
11: rem_i
```

Yes, there are more issues namely: a queue is a bit too much to ask for. Thus we must specify our implementation of a queue. But your job is actually a bit simpler than implementing a general queue. Note: (1) entries of the queue are bounded size (length of a process ID) and (2) the length of the q is at most n. You can use a Fetch&Increment object.

Algorithm 3 No lockout implementation of mutex using Fetch&Increment object F and register R: code for p_i.

```
1: → entry_i
2: x_i ← Fetch&Increment(F)
3: v_i ← R
4: while x_i ≠ v_i do
5:     v_i ← R
6: end while
7: crit_i
8:
9: → exit_i
10: R ← x_i + 1  mod i
11: rem_i
```

9.4 Mutex with Registers

We begin with two processes. The general idea is to determine if the other process is in the critical section or not.

Algorithm 4 (Peterson's Algorithm): mutex for two processes using registers: code for p_i.

```
1: Let f_0 and f_1 be single writer registers; one for each process. Further let X be a multi-writer register.
2: → entry_i
3: f_i ← 1
4: X ← i
5: v_i ← read(X)
6: while read(f_{1−i}) ≠ 0 and v_i ≠ 1 − i do
7:     v_i ← read(X)
8: end while
9: crit_i
10:
11: → exit_i
12: f_i ← 0
13: rem_i
```

Lemma 9.6 *Peterson's algorithm satisfies mutual exclusion.*

Proof: Suppose there is a configuration C such that p_0 and p_1 are both in the critical section. Further suppose WLOG suppose p_1 wrote to X last before C. The order of operation is: p_0 write 1 to f_0, p_0 wrote 0 to X, p_1 wrote 1 to X, and p_1 reads f_0 and X. Note that p_1 read 1 from f_0 and 1 from X so it will not enter the critical section. This is a contradiction. ■

Lemma 9.7 *Peterson's algorithm has no deadlock.*

Proof: Suppose there is an admissible execution with a configuration C such that at least one process is in the entry section in C, but from C onwards no process enters the critical section. Consider the possible cases:

Case 1: suppose only p_i is in the entry section. After a finite number of steps p_{1-i} must end up in the exit section and set f_{1-i} to be zero. This allows p_i to enter the critical section (p_i must continue to take steps). This is a contradiction.

Case 2: both p_0 and p_1 are in the entry section. Eventually both process have to write to their respective flag registers and write to X. Suppose p_0 was the last to write to X. p_1 (who is still taking steps in the entry section) will enter the critical section. This is a contradiction.

■

Lemma 9.8 *Peterson's algorithm has no lockout.*

Proof: Suppose for a contradiction there exists an infinite admissible execution such that p_i is in the entry section after some configuration C onwards. Since we have no deadlock p_{1-i} must be in the critical section after a finite number of steps. Eventually p_{1-i} will write $1-i$ to X. Once p_i sees this value for X it will enter the critical section. ■

9.5 Tournament Algorithm

We will use a tournament tree. The leaves of the tree is the set of processes. For the entry section: each pair of processes uses the two process mutex to go up one level of the tree. Eventually we get to the top of the tree. The process at the top will enter the critical section. For the exit section: go down the tree and perform the exit section for the two process mutex.

9.6 A Lower Bound

Definition 9.9 *A process p **covers** a register r in configuration C if p writes to r in its next allocated step.*

*Let C be a configuration where $P = \{p_0, ..., p_{r-1}\}$ covers all r registers. A **block write** by $p_0, ..., p_{r-1}$ is a sequence of r consecutive steps by $p_0, ..., p_{r-1}$ where each process performs a write, let this sequence of write be execution β. Consider another set of process Q such that $P \cup Q \neq \emptyset$. Then for operation γ by Q, $C\beta \widetilde{p_i}\ C\gamma\beta$ for all $p_i \notin Q$.*

*A **quiescent configuration** is one where all processes are in the remainder section.*

Theorem 9.10 *Any algorithm for mutex with $n \geq 2$ processes that uses only registers uses at least n registers (size of registers don't matter).*

Proof: (By covering argument). Essentially what we are going to do is iteratively build up a set of distinct registers covered by the processes. The theorem follows from Lemma 9.12 by setting $k = n$. ■

Lemma 9.11 *Suppose C is a configuration in which p_i is in the remainder section. Let α be a finite history by process p_i starting from configuration C such that p_i is in the critical section in configuration $C\alpha$. Let R*

be the set of registers which are covered in the configuration C. Then, during α, process p_i writes to some register that is not in R.

Proof: Suppose that during α process p_i only writes to registers in R. Then $C\beta \stackrel{\sim}{q} C\alpha\beta$ for all processes $q \neq p_i$ where β is a block write to R starting from C. We will find an p_i free execution γ such that there is some process p_j in the critical section in configuration $C\beta\gamma$. The idea is that either there is already a process in the critical section in $C\beta$ (in which case γ is empty) or the adversary can schedule a process so that it eventually ends up in the critical section.

Let the process in the critical section in $C\beta\gamma$ be p_j with $i \neq j$. Observe that $C\beta\gamma \stackrel{\sim}{p_j} C\alpha\beta\gamma$. Thus in the configuration $C\alpha\beta\gamma$ we have two processes p_i and p_j both in the critical section violating mutual exclusion. ■

Lemma 9.12 *From any quiescent configuration, for all $k = 1, ..., n$ there are reachable configurations C and D such that D is quiescent, each register has the same value in C and D, $p_0, ..., p_{k-1}$ cover k different registers in C and $C \stackrel{\sim}{q} D$ for all $q \in \{q_k, ..., q_{n-1}\}$.*

Proof: By induction on k. Suppose Q is a quiescent configuration. Consider some solo execution α by p_0. By Lemma 9.11, p_0 writes to some register. Let α' be the largest prefix of α which contains no writes. Let $C = Q\alpha'$ and $D = Q$. Thus the base case holds. Assume the claim holds for $1 \leq k < n$... ■