CS4231 Problem Set 1

Tan Yee Jian

January 17, 2021

Contents

Question 1

Devise a mutual exclusion algorithm for `n` processes by using **Peterson's 2-mutual-exclusion algorithm** as a black-box.

Solution

Intuitively, we create a "tournament bracket" for the processes, and for each pair in the "bracket", the processes are promoted to the next level using **Peterson's Algorithm**.

Formally, we have the processes as leaves in a binary tree, and for each non-leaf node in the tree, its value is determined by the "winning" process among its two children, chosen (recursively) by **Peterson's Algorithm**.

Correctness Proof

1. Mutual Exclusion

Lemma 0.1. *Every node can have at most one process at any time.*

Proof. We do by induction from the leaf layer to the root layer. We initialized the leaf layer with exactly one process per node, which is the base case. From one layer to the next, Peterson's Algorithm is run, which ensures the next layer has at most one process at any time. This completes the induction. □

Suppose that more than one process enters the critical section, this means at the root position, more than 2 processes could enter the critical section. Two cases can occur:

1. There are more than 2 processes running Peterson's Algorithm for the root node
2. There are exactly 2 processes running Peterson's Algorithm for the root node.

Case 2 guarantees mutual exclusion by Peterson's Algorithm. Case 1 is impossible by the lemma.

2. Progress

By the nature of a binary tree, the root node is recursively the parent of all leaves, and by the setup of the algorithm, the processes will compete for the root node, which is the ability to enter the critical section.

3. Starvation-freedom

This is guaranteed by Peterson's Algorithm.

Textbook Problems

Problem 2.1(a)

Setting `turn` to itself causes starvation.

process 0	process 1
`wantCS[0] = true`	
`turn = 0`	
	`wantCS[1] = true`
	`turn = 1`
	Enter CS
	Exit CS
	`wantCS[1] = false`
	`wantCS[1] = true`
	`turn = 1`
	Enter CS
	...

Problem 2.1(b)

It violates mutual exclusion as per below. The idea is that one process takes advantage of the gracefulness of the other process, but the other process have not declared that they want CS. That flag is hence false and in a while loop, the AND operator allows entry.

process 0	process 1	turn, wantCS
`turn = 1`		`1, {false, false}`
	`turn = 0`	`0, {false, false}`
	`wantCS[1] = true`	`0, {false, true}`
	Enter CS	
`wantCS[0] = true`		`0, {true, true}`
Enter CS		

Problem 2.3

When a process is still choosing a number (ie. value in `choosing[i]` is `true`), its `number[i]` is constantly changing. Another process might read its intermediate number while still choosing, and wrongly decides to enter the CS. Usually this is not a problem, but consider only two numbers but without the `choosing` array:

Process 0	Process 1	number
`number[0] = 0`		{0,0}
	`number[1] = 0`	
	`number[1]++`	{0,1}
	Enter CS (number[0] = 0)	
`number[0]++`		{1,1}
Enter CS (Smaller ID)		

Problem 2.4

Dekker's Algorithm is correct. Proof is as follows.

Mutual Exclusion

We shall give a direct proof. Without loss of generality, let P0 be in the CS. We show that P1 cannot be in the CS at the same time.

1. Case 1: When P0 first read `wantCS[1]`, value is false. Then P0 either
   (a) haven't started requesting CS,
   (b) is in the outer while loop (which allows it to set `wantCS[1]` to false), or
   (c) has just released CS.

   The middle case is of interest, as the other cases do not violate mutual exclusion. We have two further cases:

   (a) Case 1.1: `turn` is initialized to 1 In which case, P1 will not get the chance to hit the line that sets `wantCS[1]` to false, since it cannot enter the if clause. That leaves us with the two scenarios in Case 1 that do not violate ME.
   (b) Case 1.2 `turn` is initialized to 0 After setting `wantCS[1]` to false, it busy waits in the inner while loop, until P0 exits CS and toggles `turn`. Hence it cannot enter CS before P0 has finished.
2. Case 2: When P0 first read `wantCS[1]`, value is true.
   (a) Case 2.1: `turn` is initialized to 1 This is an impossible case because then P0 will be stuck in the inner while loop due to `turn =` 1= and not enter CS, violating the assumption.
   (b) Case 2.2 `turn` is initialized to 0 P0 patiently waits for `wantCS[1]` to be set false after P1 exits the CS.

Progress

If a process skips the while loop, then we are done. Suppose not, then both processes enter while loop. Suppose `turn =` 0=, then P1 is forced to set its `wantCS[1]` to false and busy wait. This guarantees P0 will pick up this change and exit the while loop and enter the CS. The case for `turn =` 1= is symmetric.

No Starvation

The only place able to switch `turn` is after exiting CS. This ensures each process have a chance to run. Suppose the process correponding to the `turn` does not want to run, the other process can run by skipping the entire while loop.