Assignment 3: Synchronization algorithms

ETH Zurich

1 Proving Mutual Exclusion in Spin

1.1 Background

In last week's assignment, we introduced the SPIN model checker. In this exercise, you will learn to SPIN use for proving the correctness of a concurrent algorithm. You can refer to the manual for LTL specifications in Spin for performing the following task. This exercise is taken from a tutorial presented on the Fourth Summer School on Formal Techniques [1].

Note: You must specify the LTL property as something that you want the model to satisfy. For example, if you want a model M to satisfy a property $<>[]\varphi$, then the LTL specification should look like:
ltl *property_name* $\{<>[]\varphi\}$

In case M does not satisfy *property_name*, Spin will find a counter example satisfying the negation of *property_name*.

1.2 Task

Prove/disprove the correctness of an implementation of Dijkstra's mutual exclusion algorithm with the help of LTL specifications. The promela model of the algorithm is given below.

- First understand what is happening in the code by refering to the language manual here.
- Generate the automata for the model and run some simulations.
- Next, add appropriate LTL specifications to the model file and select the Verification option from the top menu. In the "Never Claims" section, select "use claim" and the click on Run.
- What is the result? Is the model verified against the LTL specification? Explain your answer based on the simulation run.

```
bool b[2]

active [2] proctype p()
{               pid k, i = _pid, j = 1 − _pid

C0:     b[i] = false
C1:     if
        :: k!= i
C2:       if
          :: !b[j] → goto C2
          :: else → k = i; goto C1
          fi
        :: else
CS:       skip
        fi
        b[i] = true
        skip
        goto C0
}
```

2 Yet Another Lock: Proofs

2.1 Background

This task is taken from *The Art of Multiprocessor Programming* [2]. Consider the following protocol to achieve n-thread mutual exclusion.

```
turn := 0
busy := false
```

P_i

```
    do {
2       do {
            turn := i
4       } while (busy)
        busy := true
6   } while (turn != i)
    critical section
8   busy := false
    non-critical section
```

2.2 Task

For each of the following questions either provide a proof, or display an execution where it fails.

1. Does the protocol satisfy mutual exclusion?
2. Is the protocol starvation-free?
3. Is the protocol deadlock-free?

3 Tree-based mutual exclusion

3.1 Background

This question assumes a "tree-based mutual exclusion" (TBME) algorithm which is based on the following idea: The algorithm can be represented by a binary tree where each internal (non-leaf) node represents a critical section shared by its descendants. The threads are at the leaves of the tree. The root of the tree is the main critical section shared by all the threads.

To enter the main critical section, a thread starts at its leaf in the tree. The thread is required to traverse the path from its leaf up to the root, entering all the critical sections on its path. Upon exiting the critical section, the thread traverses this path in reverse, this time leaving all the critical sections on its path. Figure 1 illustrates this process. If thread 1 wants to enter the main critical section, it must first enter critical section B. After having successfully entered critical section B, thread 1 must enter critical section A, and so on.

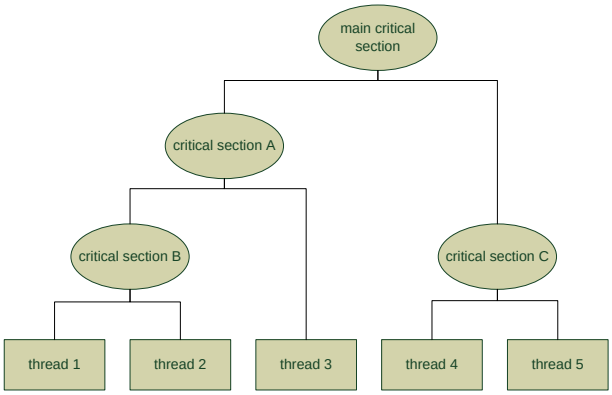

Figure 1: tree-based mutual exclusion algorithm example

At each internal node, there is a maximum of two threads competing against each other to enter the node's critical section. Therefore, a mutual exclusion algorithm for two threads (e.g. Peterson's algorithm for 2 threads) can be used to implement the critical section of an internal node.

3.2 Task

1. What is the main advantage of the TBME algorithm over the Peterson algorithm for n threads?

2. Provide a Java implementation of the TBME algorithm using the Peterson algorithm for 2 threads.

References

[1] http://fm.csl.sri.com/SSFT14/

[2] Maurice Herlihy und Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.