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
}
```

1.3 Solution

Refer to the solution code. The LTL specification is a global property stating that both processes cannot simultaneously be in the critical section, labeled as "CS". The verification of the model fails. Use the trail to re-run the simulation. It gives a counter example which violates the LTL specification at depth 44. It can be seen that both the processes enter the critical state at the same time which violates mutual exclusion.

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?

2.3 Solution

1. The protocol satisfies mutual exclusion. The property can be proved by deriving a contradiction starting from the assumption that more than one thread is in the critical section. In such a case every thread i must have gone through the following sequence of actions.
   (a) Set $turn = i$.
   (b) Verify that *busy* is **false**.
   (c) Set *busy* = **true**.
   (d) Verify that *turn* is i.

   One of the threads in the critical section must have started the sequence first. This thread is denoted by i. While thread i was going through the sequence, no other thread could have set *turn*. Otherwise thread i could not have completed the sequence before entering the critical section. Therefore no other thread could have started its sequence, because setting *turn* is at the start of every thread's sequence. Therefore every other thread must have started its sequence after thread i was done with its sequence. This means that all the other threads must have seen *busy* set to **true** before starting their sequence. Based on this, no other thread could have completed its sequence. This is a contradiction.
2. The protocol is not free of starvation as can be shown with the following execution.
   (a) Thread i attempts to enter the critical section. It enters the inner loop and sets *turn* to i.
   (b) Thread j attempts to enter the critical section. It enters the inner loop and sets *turn* to j.
   (c) Thread j leaves the inner loop, sets *busy* to **true**, leaves the outer loop and enters the critical section.
   (d) Thread i continues to execute the inner loop.
   (e) Thread j leaves the critical section. It sets *busy* to **false** and enters the non-critical section.
   (f) Thread j attempts to enter the critical section again. It enters the inner loop and sets *turn* to j.
   (g) The initial situation is restored and therefore thread j can once more overrun thread i.
3. The protocol is not free of deadlocks as can be shown with the following execution.
   (a) Thread i attempts to enter the critical section. It enters the inner loop and sets *turn* to i.
   (b) Thread j attempts to enter the critical section. It enters the inner loop and sets *turn* to j.
   (c) Thread j leaves the inner loop and sets *busy* to **true**.
   (d) Thread i continues to execute the inner loop and sets *turn* to i.

(e) Thread j continues to execute the outer loop. Thread j cannot leave the outer loop as *turn* is set to i. Therefore thread j enters the inner loop again while *busy* remains **true**.

(f) Both threads continuously execute the inner loop, because *busy* will remain **true** forever.

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.

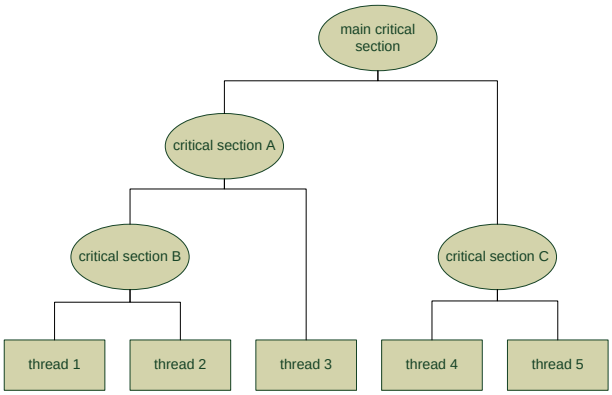

Figure 1: tree-based mutual exclusion algorithm example

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.

3.3 Solution

The main advantages of the TBME algorithm over the Peterson algorithm for n threads are:

- In the TBME algorithm for n threads, a thread only needs to go through $O(log(n))$ steps in order to enter the critical section. In the Peterson algorithm for n threads, a thread needs to go through $O(n)$ steps.
- The TBME algorithm has $O(log(n))$-bounded waiting.
- The TBME algorithm can be used with any mutual exclusion algorithm for 2 threads.

An implementation is given by the source code of this solution.

References

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

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

[3] Alain J. Martin. A New Generalization of Dekker's Algorithm for Mutual Exclusion. Technical Report 1985.5195-tr-85. California Institute of Technology, 1985.