Assignment 2: Synchronization algorithms

ETH Zurich

1 Mutual Exclusion

1.1 Background

Consider the following algorithm for a process P_i in a set of processes $P_1, \ldots, P_n$:

```
turn := 0
∀ i ∈ {1,...,n}: claimed[i] := false
```

P_i

```
    claimed[i] := true
2   while ∃ j ∈ {1,...,n} \ {i} : claimed[j] = true loop
      claimed[i] := false
4     await (turn = 0 or turn = i)
      turn := i
6     claimed[i] := true
    end
8   critical section
    claimed[i] := false
10  turn := 0
    non-critical section
```

1.2 Task

Answer the following questions:

1. Does the algorithm enforce mutual exclusion? If so, justify your answer with an informal proof. If not, provide a sequence of actions to illustrate how mutual exclusion could be violated.

2. Does the algorithm guarantee the absence of deadlocks? If so, justify your answer with an informal proof. If not, provide a sequence of actions to illustrate how a deadlock could occur.

3. Does the algorithm guarantee the absence of starvation? If so, justify your answer with an informal proof. If not, provide a sequence of actions to illustrate how starvation could occur.

1.3 Solution

The algorithm is described in [2].

1. The algorithm satisfies mutual exclusion. The proof works by deriving a contradiction. We assume that there is more than one process in the critical section. One of these processes P_x must have executed $claimed[x]$:= **true** first and then checked that the loop condition is false before entering the critical section. Another process P_y must have set $claimed[y]$:= **true** after P_x checked that the loop condition is false. This means, that P_y must have checked that the loop condition is false, after P_x set $claimed[x]$:= **true**. This is a contradiction, because the loop condition could not have been false. Hence, the algorithm satisfies mutual exclusion.

2. The algorithm guarantees the absence of deadlocks. The proof works by deriving a contradiction. We assume that there is a deadlock and all processes are trapped in the loop. As each process P_i was entering the loop, it must have executed $claimed[i]$:= **false**. If the process was quick enough, *turn* was still set to its initial value. In this case it did not have to wait and it went on with $turn = i$. If the process was not quick enough it had to wait. We put all the quick processes in a set s and we use P_x to denote the last process who set $turn = x$. Each of the processes P_j in s went on with $claimed[j]$:= **true** and then went through another loop iteration, thus setting $claimed[j]$:= **false**. Every process except P_x will have waited at this point and for every process Pi except for P_x we know that $claimed[i]$ = **false**. This means that process P_x went on with $turn := x$ and $claimed[x]$:= **true** and then P_x left the loop. This is a contradiction. Hence, the algorithm guarantees the absence of deadlocks.

3. The algorithm does not guarantee the absence of starvation. The following trace with two processes P_x and P_y shows the problem:

   (a) Process P_x: $claimed[x]$:= **true**
   (b) Process P_y: $claimed[y]$:= **true**
   (c) Process P_x: Check loop condition and enter loop.
   (d) Process P_x: $claimed[x]$:= **false**
   (e) Process P_y: Check loop condition and enter critical section.
   (f) Process P_y: Execute critical section.
   (g) Process P_y: $claimed[y]$:= **false**
   (h) Process P_y: $turn := 0$
   (i) Process P_y: Execute non-critical section.
   (j) Process P_y: $claimed[y]$:= **true**
   (k) Process P_x: Checks the wait condition and continues.
   (l) Process P_x: $turn := x$
   (m) Process P_x: $claimed[x]$:= **true**
   (n) Process P_x: Check loop condition and enter loop.
   (o) Process P_x: $claimed[x]$:= **false**
   (p) Process P_y: Check loop condition and enter critical section.
   (q) ...

2 Yet Another Lock: Proofs

2.1 Background

This task is taken from *The Art of Multiprocessor Programming* [1]. 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 prooved 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 i.

(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.

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.

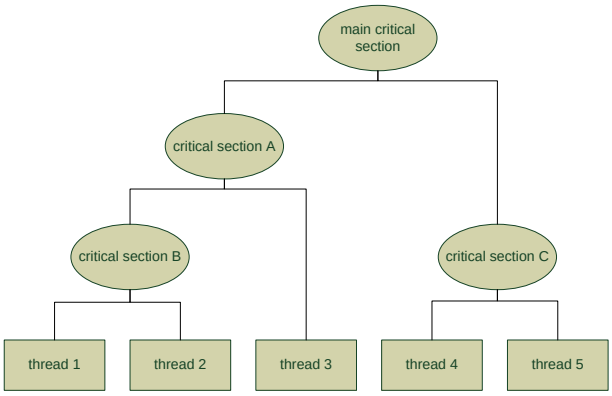

Figure 1: tree-based mutual exclusion algorithm example

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

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