Dairon Andrés Benites Aldaz

Exercises for Lecture 2

Exercise 1

Consider the class *VolatileCounters* in figure 1. The "volatile" declaration ensures that variable reads and writes occur in a one-at-a-time sequential order (as one might expect) when accessed by each of the parallel threads. Assume *actor1* and *actor2* are invoked to run in parallel.

1. What are the relevant events?

   Both threads running in parallel go for 10 times each executing the increment() method that increases by 1 the mentioned *volatile variable x.*

   Events:

   START

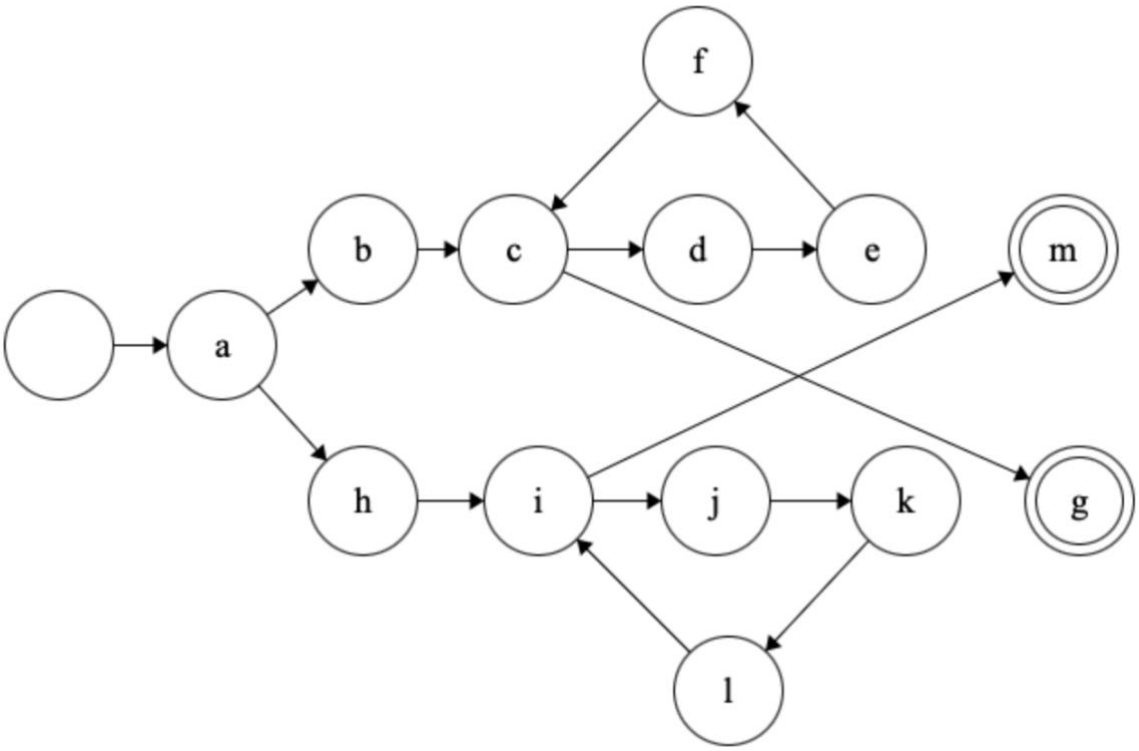
   1. Variable declaration of *volatile int x*

      For each loop: [*actor1* and *actor2*]

      1. Declaration and initialization of *local variable int i*
      2. Check loop condition (i < 10)
         a. If condition is satisfied:
            i. Call method *increment()*
            ii. Increment by 1 *volatile int x*
            iii. Increment by 1 *local variable int i*
            iv. Go to Event 2
         b. If condition is not satisfied:
            END

2. Sketch the state machine graph (only parts, it is quite large). Include all events you have identified in 1.

3. Sketch a few traces of the state machine.



Blank, initial state
a, declaration of volatile x

Thread A {b,c,d,e,j}
Thread B {f,g,h,i,k}

b and h, initialization of *local variable int i_a, i_b*
c and i, check loop condition
d and j, Call method increment
e and k, Increment by 1 volatile x
f and l, Increment by 1 local variable int i_a, i_b
m and g, Accept State [*after loop condition not satisfied*]

4. List the relevant intervals for the program.
5.Which are the possible final values for x? Explain your reasoning very carefully.

Given the volatile keyword declaration it can be used to modify the value of such variable by different threads [actor1 and actor2].

This menas that multiple threads can use the method **increment()** without any problem.
I'd assume given a classical declaration of two threads such as:

Thread actor1 = new Thread(new Runnable() {…}
Thread actor2 = new Thread(new Runnable() {…}

actor1.start(); actor2.start()

Final value of x = 20.

Given that both threads have acces to the variable and each one of them increments 10 times the variabel x in a parallel but (somewhat) controlled manner. Of course each thread has their own local variables (loop counters and such).

Exercise 2 HSLS 2.3 (Flaky Computer Corporation) Use the method presented in class and in the textbook to solve this.

```
1  class Flaky implements Lock {
2    private int turn;
3    private boolean busy = false;
4    public void lock() {
5      int me = ThreadID.get();
6      do {
7        do {
8          turn = me;
9        } while (busy);
10       busy = true;
11     } while (turn != me);
12   }
13   public void unlock() {
14     busy = false;
15   }
16 }
```

FIGURE 2.16

The Flaky lock used in Exercise 2.3.

Exercise 2.3. Programmers at the Flaky Computer Corporation designed the protocol shown in Fig. 2.16 to achieve n-thread mutual exclusion. For each question, either sketch a proof, or display an execution where it fails.

- **Does this protocol satisfy mutual exclusion?**

  Critical regions of multiple threads don't collide.
  For a given Thread A and Thread B let's suppose there is a set of integers such that Thread A [i]-> Thread B [j] OR Thread B [j]-> Thread A[i]

  (Turn = A) -> rA(busy == false) -> A (Busy = true) -> rA (Turn == A)
  (Turn = b) -> rb(busy == false) -> B(Busy = true) -> rB(Turn == B)
  A (Turn == A) -> B (Turn == B)

- Is this protocol starvation-free?

No, it is not. Because we can end in a situation where both threads are stuck in an infinite loop where neither gets the lock.

Example A

If two threads A and B. Run lock() and progress until (busy = true), If we asume B went after so turn is set to B. Then, A receives turn != me and goes to the loop (do while) at line 8. But then, B is also going to get a turn!= me.

Example A

- Is this protocol deadlock-free?
  No, it is not.

Given **Example A** situtation of deadlock already explained.

- Is this protocol livelock-free?

No, it can happen with the turn variable.

Exercise 3

HSLS 2.8 (Fast path lock) Use the method presented in class and in the textbook to solve this.

In practice, almost all lock acquisitions are uncontended, so the most practical measure of a lock's performance is the number of steps needed for a thread to acquire a lock when no other thread is concurrently trying to acquire the lock.
Scientists at Cantaloupe-Melon University have devised the following "wrapper" for an arbitrary lock, shown in Fig. 2.18. They claim that if the base Lock class pro- vides mutual exclusion and is starvation-free, so does the FastPath lock, but it can be acquired in a constant number of steps in the absence of contention. Sketch an argument why they are right, or give a counterexample.

FIGURE 2.18

Fast-path mutual exclusion algorithm used in Exercise 2.8.

```
1  class Bouncer {
2    public static final int DOWN = 0;
3    public static final int RIGHT = 1;
4    public static final int STOP = 2;
5    private boolean goRight = false;
6    private int last = -1;
7    int visit() {
8      int i = ThreadID.get();
9      last = i;
10     if (goRight)
11       return RIGHT;
12     goRight = true;
13     if (last == i)
14       return STOP;
15     else
16       return DOWN;
17   }
18 }
```

Exercise 4
HSLS 2.2 (First-come-first-served needs a doorway)

Exercise 2.2. Why must we define a *doorway* section, rather than defining first- come-first-served in a mutual exclusion algorithm based on the order in which the first instruction in the lock() method was executed? Argue your answer in a case-by- case manner based on the nature of the first instruction executed by the lock(): a read or a write, to separate locations or the same location.

By Definition 2.6.1 of HSLS:

> ***"A lock is first-come-first-served if its lock() method can be split into a bounded wait-free doorway section followed by a waiting section so that whenever thread A finishes its doorway before thread B starts its doorway, A cannot***
>
> $$\text{if } D_A^j \rightarrow D_B^k \text{ then } CS_A^j \rightarrow CS_B^k$$
>
> ***be overtaken by B. That is,***
> **for any threads A and B and integers j and k, where *D*j and *CS*j are the intervals AA during which A executes the doorway section of its j th call to the lock() method and its jth critical section, respectively.**

So the doorway section as well as the waiting section (unbounded number of steps) may help determine fairness in a situation in which **we cannot determine** which Thread called lock() first.

For example a specific order while reading an initial variable (not global).

Writing a certain value after a method of an increasing number of steps(with loops). Access to global variables, etc.