Concurrent Programming

Exercise Booklet 2: Mutual Exclusion

Exercise 1. Implement Peterson's solution in Groovy by completing the following stub and verifying that the result is always 20.

```
public interface Lock {
    void lock(int id);
    void unlock(int id);
}

public class Peterson implements Lock {
    private volatile int last;
    private volatile boolean[] want;

    Peterson() {
       last=0;
       want = new boolean[2];
       want[0]=false;
       want[1]=false;
    }
    void lock(int id) {
    // complete
    }
    void unlock(int id) {
    // complete
    }
  }

// Sample use
Lock l = new Peterson()
int c=0

P = Thread.start  { // P
    10.times{
       l.lock(0)
       c++
       l.unlock(0)
    }
}

Q = Thread.start  { // Q
    10.times{
       l.lock(1)
       c++
       l.unlock(1)
    }
}

P.join()
Q.join()
println c
```

Note: the `volatile` keyword ensures that when `P` and `Q` write to `last` and `want`, these changes will be seen immediately (rather than being cached). Also, it avoids reordering of instructions that involve read and writes to these variables (see Exercise 4).

Exercise 2. Show that Attempt IV (naive back-out) at solving the MEP, as seen in class and depicted below, does not enjoy freedom from starvation. Reminder: in order to do so you must exhibit a path in which one of the threads is trying to get into its CS but is never able to do so. Since the path you have to exhibit is infinite, it suffices to present a prefix of it that is sufficiently descriptive.

```
boolean wantP = false
boolean wantQ = false
Thread.start { //P                    Thread.start { //Q
  while (true) {                        while (true) {
   // non-critical section               // non-critical section
   wantP = true                          wantQ = true
   while (wantQ) {                       while (wantP) {
     wantP = false                         wantQ = false
     wantP = true                          wantQ = true
   }                                     }
   // CRITICAL SECTION                   // CRITICAL SECTION
   wantP = false                         wantQ = false
   // non-critical section               // non-critical section
  }                                     }
}                                     }
```

Exercise 3. Use transition systems to show that Peterson's algorithm solves the MEP.

Exercise 4. Peterson's solution to the MEP is given below. Show that if one swaps lines 8 and 9 in `P` and `Q`, then mutex fails.

```
   int last = 1
 2 boolean wantP = false
   boolean wantQ = false
 4
   Thread.start  { // P                  Thread.start  { // Q
 6   while (true) {                        while (true) {
      // non-critical section               // non-critical section
 8    wantP = true;                         wantQ = true;
      last = 1;                             last = 2;
10    await !wantQ or last==2;              await !wantP or last==1;
      // CRITICAL SECTION                   // CRITICAL SECTION
12    wantP = false;                        wantQ = false;
      // non-critical section               // non-critical section
14   }                                     }
   }                                     }
```

Exercise 5. Consider the following extension of Peterson's algorithm for n processes ($n > 2$) that uses the following shared variables:

```
flags = [false] * n; // initialize list with n copies of false
```

and the following auxiliary function

```
def boolean flagsOr(id) {
  result = false;
  n.times {
    if (it != id)
      result = result || flags[it];
  }
  return result;
}
```

Moreover, each thread is identified by the value of the local variable `threadId` (which takes values between 0 and $n-1$). Each thread uses the following protocol.

```
  ...
  // non-critical section
  flags[threadId] = true;
  while (FlagsOr(threadId)) {};
  // critical section
  flags[threadid] = false;
  // non-critical section
  ...
```

1. Explain why this proposal does enjoy mutual exclusion. Hint: reason by contradiction.
2. Does it enjoy absence of livelock?

Exercise 6. Consider the simplified presentation of Bakery's Algorithm for two processes seen in class:

```
int np = 0;
int nq = 0;
Thread.start {   //P
  while (true) {
   // non-critical section
   [np = nq + 1];
   while (!(nq==0 || np<=nq)) {}; // await (nq==0 || np<=nq);
   // CRITICAL SECTION
   np = 0;
   // non-critical section
  }
}

Thread.start { //Q
  while (true) {
   // non-critical section
   [nq = np + 1];
   while (!(np==0 || nq<np)) {} ; // await (np==0 || nq<np);
   // CRITICAL SECTION
   nq = 0;
   // non-critical section
  }
}
```

Show that if we do not assume that assignment is atomic (indicated with the square brackets), then mutual exclusion is not guaranteed. For that, provide an offending path for the following program:

```
int np = 0;
int nq = 0;
Thread.start  {   //P
  while (true) {
   // non-critical section
   temp = nq;
   np = temp + 1;
   while (!(nq==0 || np<=nq)) {}; // await (nq==0 || np<=nq);
   // CRITICAL SECTION
   np = 0;
   // non-critical section
  }
}

Thread.start { // Q
  while (true) {
   // non-critical section
   temp = np;
   nq = temp + 1;
   while (!(np==0 || nq<np)) {} ; // await (np==0 || nq<np);
   // CRITICAL SECTION
   nq = 0;
   // non-critical section
  }
}
```

Exercise 7. Given Bakery's Algorithm, show that the condition $j <$ `threadId` in the second while is necessary. In other words, show that the algorithm that is obtained by removing this condition (depicted below) fails to solve the MEP. Indeed, show that mutex may fail. You must assume that assignment is not atomic.

```
choosing = [false] * N; // list of N false
ticket = [0] * N // list of N 0

Thread.start {
  // non-critical section
  choosing[threadId] = true;
  ticket[threadId] = 1 + maximum(ticket);
  choosing[threadId] = false;
  (0..n-1).each {
    await (!choosing[it]);
    await (ticket[it] == 0 ||
            (ticket[it] < ticket[threadId] ||
            (ticket[it] == ticket[threadId]))
          );
  }
  // critical section
  ticket[threadId] = 0;
  // non-critical section
}
```