Concurrent Programming

Exercise Booklet 2: Mutual Exclusion

Exercise 1. Show that Attempt IV 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. Note that the path you have to exhibit is infinite; it suffices to present a prefix of it that is sufficiently descriptive.

```
1  boolean wantP = false;
2  boolean wantQ = false;
```

```
3  Thread.start { //P
4    while (true) {
5      // non-critical section
6      wantP = true;
7      while (wantQ) {
8        wantP = false;
9        wantP = true;
10     }
11     // CRITICAL SECTION
12     wantP = false;
13     // non-critical section
14   }
15 }
```

```
4  Thread.start { //Q
5    while (true) {
6      // non-critical section
7      wantQ = true;
8      while (wantP) {
9        wantQ = false;
10       wantQ = true;
11     }
12     // CRITICAL SECTION
13     wantQ = false;
14     // non-critical section
15   }
16 }
```

(IP_P,IP_Q,wantP,wantQ)
(P6,Q7,false, false) P
(P7,Q7,true,false) Q
(P7,Q8,true,true) Q
(P7,Q9,true,true) Q
(P7,Q10,true,false) P
P critical section
(P12,Q10,true,false) P
(P6,Q10,false,false) P
(P7,Q10,true,flase) Q
(P7,Q8,true,true) P

transition move to line 4, and repeat it
so Q never go into it's critical section

Exercise 2. Consider the following proposal for solving the MEP problem for two threads, that uses the following functions and shared variables:

```
1  current = 0; // global
2  turns = 0; // global
3
4  def requestTurn() {
5    int turn  = turns;
6    turns = turns + 1;
7    return turn;
8  }
9  def freeTurn() {
10   current = current + 1;
11   turns = turns - 1;
12 }
```

We assume that each thread executes the following protocol:

```
1  Thread.start{ //P
2      while (true) {
3       // non-critical section
4       int turn = requestTurn();
5       while (current!=turn) {}; // await (current==turn);
6       // critical section
7       print(Thread.currentThread().getId()+"in the CS");
8       freeTurn();
9       // non-critical section
10     }
11 }
```

1.
both P's and Q's turn are 0
so mutual exclusion fails.
2.(a)
(IP_P, IP_Q, P_turn, Q_turn, current, turns)
(P4, Q4, ?, ?, 0, 0) P
(P5, Q4, 0, ?, 0, 1) P
(P6, Q4, 0, ?, 0, 1) Q
(P6, Q5, 0, 1, 0, 2) P6,P7,P8
(P4, Q5, ?, 1, 1, 1) Q
(P4, Q6, ?, 1, 1, 1) P
(P5, Q6, 1, 1, 1, 1) P
(P6, Q6, 1, 1, 1, 1)

P & Q in their critical section

2.(b)
after the protocol are run third
current = 3 and truns=0
cause turns<=2,
so P and Q will never go into their critical section

1. Show that this proposal does not guarantee mutual exclusion.

2. Assume that both operations `requestTurn` and `freeTurn` are atomic.

   (a) Show that this proposal still does not guarantee mutual exclusion

   (b) Show that even if the operations are atomic, freedom from starvation fails.

Exercise 3. Consider the following extension of Peterson's algorithm for n processes ($n > 2$) that uses the following shared variables:

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

and the following auxiliary function

```
def boolean flagsOr(id) {
  result = false;
  (0..n-1).each {
    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. if two threads are in their critical section simultaneously,
it means their FlagsOr(id) == false, so their flags[id] == false
which is impossible
2. no
if any number (n>=2) thread wants to go in their critical section
any thread will fail
cause their result will always equal true.

1. Explain why this proposal does enjoys mutual exclusion. Hint: reason by contradiction.

2. Does it enjoy absence of livelock?

Exercise 4. Use transition systems to show that Peterson's algorithm solves the MEP.

Exercise 5. 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:

```
1  int np=0;
2  int nq =0;

2  Thread.start  {    //P
3    while (true) {
4     // non-critical section
5     temp = nq;
6     np = temp + 1;
7     while (!(nq==0 || np<=nq)) {}; // await (nq==0 || np<=nq);
8     // CRITICAL SECTION
9     np = 0;
10    // non-critical section
11   }
12 }
13
14 Thread.start { // Q
15   while (true) {
16    // non-critical section
17    temp = np;
18    nq = temp + 1;
19    while (!(np==0 || nq<np)) {} ; // await (np==0 || nq<np);
20    // CRITICAL SECTION
21    nq = 0;
22    // non-critical section
23   }
24 }
```

(IP_P, IP_Q, P_temp, Q_temp, np, nq)
(P5, Q17, ?, ?, 0, 0) P
(P6, Q17, 0, ?, 0, 0) Q
(P6, Q18, 0, 0, 0, 0) Q
(P6, Q19, 0, 0, 0, 1) Q
(P6, Q20, 0, 0, 0, 1) P
(P7, Q20, 0, 0, 1, 1) P
(P8, Q20, 0, 0, 1, 1)

Exercise 6. 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 it may livelock.

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

thread {
  // 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
}
```