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.

```
boolean wantP = false;
boolean wantQ = false;
```

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

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

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

```
current = 0; // global
turns = 0; // global

def requestTurn() {
  int turn  = turns;
  turns = turns + 1;
  return turn;
}
def freeTurn() {
  current = current + 1;
}
```

We assume that each thread executes the following protocol:

```
Thread.start{ //P
    while (true) {
     // non-critical section
     int turn = requestTurn();
     while (current!=turn) {}; // await (current==turn);
     // CRITICAL SECTION
     print(Thread.currentThread().getId()+"in the CS");
     freeTurn();
     // non-critical section
    }
}
```

1. Show that this proposal does not guarantee mutual exclusion by exhibiting a path that leads to both accessing the CS.

2. Show that this proposal does not enjoy absence of livelock (and hence also 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:

```
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. Explain why this proposal does enjoy 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:

```
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 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 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
}
```