Concurrent Programming

Exercise Booklet 2: Mutual Exclusion

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.

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

```
thread P: {                          thread 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
  }                                    }
}                                    }
```

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

```
global int current = 0, turns = 0;

RequestTurn() {                FreeTurn() {
  turn  = turns;                 current = current + 1;
  turns = turns + 1;             turns = turns - 1;
  return turn;                 }
}
```

We assume that each thread executes the following protocol:

```
// non-critical section
turn = RequestTurn();
await (current==turn);
// critical section
FreeTurn();
// non-critical section
```

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

(b) Assume that both operations `RequestTurn` and `FreeTurn` are atomic.

i. Show that this proposal still does not guarantee mutual exclusion

ii. Show that freedom from starvation fails too.

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

```
global boolean[] flags = replicate(n,false);
```

and the following auxiliary function

```
boolean FlagsOr(id) {
  result = false;
  for (i : range(0,n-1))
    if (i != id)
      result = result || flags[i];
  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
...
```

(a) Explain why this proposal does enjoys mutual exclusion. Hint: reason by contradiction.

(b) Does it enjoy absence of deadlock?

4. Use a state diagram to show that Peterson's algorithm solves the MEP.

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

```
global int np,nq =0;

thread P: {                        thread Q: {
  while (true) {                     while (true) {
   // non-critical section            // non-critical section
   [np = nq + 1];                     [nq = np + 1];
   await nq==0 or np<=nq;             await np==0 or nq<np;
   // CRITICAL SECTION                // CRITICAL SECTION
   np = 0;                            nq = 0;
   // non-critical section            // non-critical section
  }                                  }
}                                  }
```

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

```
global int np,nq =0;

thread P: {                        thread Q: {
  while (true) {                     while (true) {
   // non-critical section            // non-critical section
   temp = nq;                         temp = np;
   np = temp + 1;                     nq = temp + 1;
   await nq==0 or np<=nq;             await np==0 or nq<np;
   // CRITICAL SECTION                // CRITICAL SECTION
   np = 0;                            nq = 0;
   // non-critical section            // non-critical section
  }                                  }
}                                  }
```

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 deadlock.

```
global boolean[] choosing = replicate(N,false);
global int[] ticket = replicate(N,0);

thread {
  // non-critical section
  choosing[threadId] = true;
  ticket[threadId] = 1 + maximum(ticket);
  choosing[threadId] = false;
  for (j : range(0,n-1)) {
    while (choosing[j]);
    while (ticket[j] != 0 &&
          (ticket[j] < ticket[threadId] ||
          (ticket[j] == ticket[threadId])));
  }
  // critical section
  ticket[threadId] = 0;
  // non-critical section
}
```