# Concurrent Programming

## Exercise Booklet 4: Semaphores

Solutions to selected exercises (◊) are provided at the end of this document. Important: You should first try solving them before looking at the solutions. You will otherwise learn **nothing**.

**Exercise 1.** Given the following threads:

```
Thread.start { // P
  print("A");
  print("B");
  print("C");
}
```

```
Thread.start { // Q
  print("E");
  print("F");
  print("G");
}
```

use semaphores in order to guarantee that A is printed before F and F is printed before C.

**Exercise 2.** Given

```
Thread.start { // P
  print("A");
  print("S");
}
```

```
Thread.start { // Q
  print("L");
  print("E");
  print("R");
}
```

use semaphores to guarantee that the only possible output is LASER.

**Exercise 3.** Consider the following three threads:

```
Thread.start { // P
  print("R");
  print("OK");
}
```

```
Thread.start { // Q
  print("I");
  print("OK");
}
```

```
Thread.start { // R
  print("O");
  print("OK");
}
```

Use semaphores to guarantee that the output is R I O OK OK OK (we assume that print is atomic).

**Exercise 4.** Consider the following threads that share the variables `y` and `z`. Assume assignment is atomic.

```
int y = 0, z = 0;
Thread.start { // P
  int x;
  x = y + z;
}
```

```
Thread.start { // Q
  y = 1;
  z = 2;
}
```

1. What are the possible final values for `x`?
2. Is it possible to use semaphores to restrict the set of possible values of `x` to be just two possible values?

**Exercise 5.** Consider the following code. Set the number of initial permits for the semaphores declared below and add **only** acquire and release operations so that the expression `aaabaaabaaab...` is output.

```
import java.util.concurrent.Semaphore

Semaphore okA = new Semaphore(??)
Semaphore okB = new Semaphore(??)

Thread.start { // P
  while (true) {
    print "a"
  }
}

Thread.start { // Q
  while (true) {
    print "b"
  }
}
```

**Exercise 6.** Consider the following code. Set the number of initial permits for the semaphores declared below and add **only** acquire and release operations so that every expression in

`aaa(b+c)aaa(b+c)aaa(b+c)...`

has an interleaving that can print it. Note: `b+c` means `b` or `c`:

```
import java.util.concurrent.Semaphore

Semaphore okA = new Semaphore(??)
Semaphore okBorC = new Semaphore(??)
Semaphore mutex = new Semaphore(??)

Thread.start { // P
  while (true) {
     print "a"
  }
}

Thread.start { // Q
  while (true) {
     print "b"
  }
}

Thread.start { // R
  while (true) {
     print "c"
  }
}
```

**Exercise 7.** We have three threads $A$, $B$, $C$. We wish operation $op_C$ of $C$ be executed only

after $A$ has executed $op_A$ and $B$ has executed $op_B$. How can we synchronize these processes using just one semaphore?

```
Thread.start { // A
    opA;
}
```

```
Thread.start { // B
    opB;
}
```

```
Thread.start { // C
    opC;
}
```

**Exercise 8. ($\Diamond$)** Consider the following two threads:

```
Thread.start { // P
  while (true) {
    print("A");
  }
}
```

```
Thread.start { // Q
  while (true) {
    print("B");
  }
}
```

1. Use semaphores to guaranteee that at all times the number of A's and B's differs at most in 1.
2. Modify the solution so that the only possible output is ABABABABAB...

**Exercise 9.** The following threads should cooperate to calculate the value $n2$ which is the sum of the first $n+1$ odd numbers. The processes share the variables $n$ and $n2$ which are initialized as follows: $n = 50$ and $n2 = 0$. The expected result is 2601 since the sum of the first 51 odd numbers is 2601. You may assume assignment is atomic.

```
int n2=0
int n=50
P = Thread.start {
  while (n > 0) {
    n = n-1
  }
}

Thread.start {
  while (true) {
    n2 = n2 + 2*n + 1
  }
}
P.join()
 // if your code prints 2600 you might need an extra line of code here...
print(n2)
```

Provide a solution using semaphores that guarantees that the correct value of $n2$ is printed.

**Exercise 10.** Consider the pseudocode for `release` given in class:

```
  atomic release() {
    if (processes.empty() || permissions<0) {
      permissions++;
    } else {
```

```
      wakingThread = processes.removeAny();
      wakingThread.state = READY;
    }
  }
```

Show that if we remove the condition `permissions<0`, then we cannot allow negative permits. An example of what can go wrong follows. This code attempts to ensure that the value of `counter` is only printed after both turnstile threads finish. It relies on negative permissions and the modified pseudocode for `release`. It is possible for the code to print the value of counter before both turnstile have finished. Why?

```
import java.util.concurrent.Semaphore;

int counter = 0
Semaphore mutex = new Semaphore(1)
Semaphore s = new Semaphore(-1)

Thread.start { // turnstile
   10.times {
      mutex.acquire()
      counter ++
      mutex.release()
   }
   s.release()
}

Thread.start { // turnstile
   10.times {
      mutex.acquire()
      counter ++
      mutex.release()
   }
   s.release()
}

s.acquire()
print counter
```

# 1 Solutions to Selected Exercises

**Answer to exercise 8**

Item 1.

```
Semaphore allowA = new Semaphore (1);
Semaphore allowB = new Semaphore (1);

Thread.start { // P
  while (true) {
  allowA.acquire();
  print("A");
  allowB.release();
  }
}

Thread.start { // Q
  while (true) {
    allowB.acquire();
    print("B");
    allowA.release();
  }
}
```

What happens if we initialize both semaphores any $k > 0$?

Item 2. Have `allowB` be initialized with 0 permits.