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:

Semaphore se=new Semaphore(0);
Semaphore se2=new Semaphore(0);

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

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

```
//P
print("A");
se.release();
print("B");
se2.acquire();
print("C");
```

```
//Q
print("E");
se.acquire();
print("F");
se2.release();
print("G");
```

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

Exercise 2. Given

Semaphore se=new Semaphore(0);
Semaphore se2=new Semaphore(0);

```
Thread.start { // P
  se.acquire()
  print("A");
  print("C");
} se2.release()
```

```
Thread.start { // Q
  print("R"); <- se.release()
  print("E");  se2.acquire()
  print("S");
}
```

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

Exercise 3. Consider the following three threads:

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

```
Thread.start { // Q
  se.acquire()
  print("I");
  se2.release()
  print("OK");
}
```

```
Thread.start { // R
  se2.acquire()
  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;
  se.acquire()
  x = y + z;
}
```

```
Thread.start { // Q
  y = 1;
  se.realse();
  z = 2;
}
```

1. What are the possible final values for x? [0, 1, 3]
2. Is it possible to use semaphores to restrict the set of possible values of x to be just two possible values?

Exercise 5. Given

```
Thread.start {
  while (true) {
    print("A");
    print("B");
    print("C");
    print("D");
  }
}
```

```
Thread.start {
  while (true) {
    print("E");
    print("F");
    print("G");
  }
}
```

```
Thread.start {
  while (true) {
    print("H");
    print("I");
  }
}
```

Add semaphores to guarantee that:

1. the number of F ≤ the number of A

2. the number of H $\leq$ the number of E
3. the number of C $\leq$ the number of G

Exercise 6. 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 semaphores?

se = new Semaphore(0); se2 = new Semaphore(0);

```
Thread.start { // A
    opA;
}   se.release()
```

```
Thread.start { // B
    opB;
}   se2.release();
```

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

Exercise 7. (◊) Consider the following two threads:

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

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

1.semaphore:
permA 1
permB 1

2.semaphore:
permA 1
permB 0

```
//P
permA.acquire()
print("A")
permB.release()
```

```
// Q
permB.acquire()
print("B")
permA.release()
```

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 8. The following threads should cooperate to calculate the value $n2$ which is the sum of the first n 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. You may assume assignment is atomic.

```
int n2=0;        Semaphore permP = new Semaphore(0);
int n=50;        Semaphore permQ = new Semaphore(1);
```

```
Thread.start {
  while (n > 0) {
    permP.acquire()
    n = n-1;
    permQ.release()
  }
  permP.acquire()
  print(n2);
}
```

```
Thread.start {
  while (true) { permQ.acquire()
    n2 = n2 + 2*n + 1;
    permP.release()
  }
}
```

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

Exercise 9. The following code attempts to ensure that the value of counter is only printed after both turnstile threads finish. It relies on negative permissions. The code does not do what it purports to do. It could, for example, print 10. Can you explain what goes wrong?

```
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 of exercise 7

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.