Concurrent Programming

Exercise Booklet 5: Semaphores (cont)

Solutions to selected exercises ($\Diamond$) 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. ($\Diamond$) On Fridays the bar is usually full of Jets fans. Since the owners are Patriots fans they would like to implement an access control mechanism in which one Jets fan can enter for every two Patriots fans.

1. Implement such a mechanism, assuming that Jets fans will have to wait indefinitely if no Patriots fans arrive. You may assume, to simplify matters, that once fans go in, they never leave the bar. Here is a stub you can use as guideline:

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

// Declare semaphores here

20.times{
    Thread.start { // Patriots
    // fan goes into bar
    }
}

20.times {
    Thread.start { // Jets
    // fan goes into bar
    }
}
```

2. Modify the solution assuming that, after a certain hour, everybody is allowed to enter (those that are waiting outside and those that yet to arrive). For that there is a thread that will invoke, when the time comes, the operation `itGotLate`. You may assume that the code for this thread is already given for you, the only thing that you must do is define the behavior of `itGotLate` and modify the threads that model the Jets and Patriots fans.

Exercise 2. A farm breeds cats and dogs. It has a common feeding area for both of them. Although the feeding area can be used by both cats and dogs, it cannot be used by both at the same time for obvious reasons. Provide a solution using semaphores. The solution should be free from deadlock but not necessarily from starvation. You should have one thread for cats and one thread for dogs. There could be any number of instances of these threads, of course. Here is a stub you can use as guideline:

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

// Declare semaphores here
```

```
20.times{
    Thread.start { // Cat
       // access feeding area
       // eat
       // exit feeding area
    }
}

20.times {
    Thread.start { // Dog
       // access feeding area
       // eat
       // exit feeding area
    }
}
```

Exercise 3. A common lounge is used by students and faculty. There can be any number of students and faculty in the lounge at any given time. Robot cleaning machines are available to clean the lounge every now and then. In order to do so, there can be no faculty nor students in the lounge at the time. Moreover, since robot cleaning machines interfere with one another, only one robot cleaning machine can be cleaning at any given time. Below is a stub you can use as guideline. You need no additional semaphores. Note: you may ignore fairness.

```
import java.util.concurrent.Semaphore
Semaphore mutexS = new Semaphore(1)
Semaphore mutexF = new Semaphore(1)
Semaphore studentPermit = new Semaphore(1)
Semaphore facultyPermit = new Semaphore(1)

final int S = 10  // Number of students
final int F = 10  // Number of faculty
final int CM = 3  // Number of cleaning machines

S.times {
    int id = it
    Thread.start { // Student
    // complete
    }
}

F.times {
    int id = it
    Thread.start { // Faculty
    // complete
    }
}

CM.times{
    int id = it
    Thread.start { // Clean
      while (true) {
        // complete
      }
    }
}
```

Exercise 4. Model a ferry between two coasts, say the East (0) and the West (1) coasts, using semaphores. The ferry has capacity for N passengers and works in the following way. It waits at one coast until it fills up to capacity and then automatically switches to the other coast. When it arrives at a coast, it waits for all the passengers to get off and then allows new passengers to board. The ferry and each passenger has to be implemented as a thread. There is no cap on the number of passengers at either coast. Also, for the purpose of simplicity, passengers use the service once and then never again. Use the following stub as guideline:

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

// Declare semaphores here

Thread.start { // Ferry
    int coast=0;
    while (true) {

         // allow passengers on
         // move to opposite coast
         coast = 1-coast;
         // wait for all passengers to get off

      }
}

100.times {
  Thread.start { // Passenger on East coast
    // get on
    // get off at opposite coast
    }
}

100.times {
  Thread.start { // Passenger on West coast
    // get on
    // get off at opposite coast
    }
}

return;
```

Exercise 5. In a gym there are four apparatus (numbered 0 to 3 for easy reference), each involving a different muscle group. The apparatus are loaded with weight discs; all weight discs are of the same weight; there are a total of `MAX_WEIGHTS` of them in the gym. Each gym client has a routine. A routine is a list of exercises; each exercise consists of an apparatus and the number of weight discs to be loaded onto the apparatus. The gym requires that each client, when finished using an apparatus, unloads all weight discs and places them in their storage area. Finally, for security reasons, no more than `GYM_CAP` clients may be in the gym at any given time.

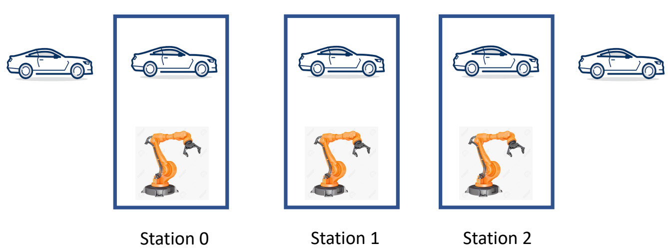
1. Write code that simulates the gym's workings, guaranteeing mutual exclusion in the access of the shared resources and freedom of deadlock. Use the following stub as a guideline.

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

```
MAX_WEIGHTS = 10;
GYM_CAP = 50;

// Declare semaphores here

def make_routine(int no_exercises) { // returns a random routine
    Random rand = new Random();
    int size = rand.nextInt(no_exercises);
    def routine = [];

    size.times {
      routine.add(new Tuple(rand.nextInt(4),rand.nextInt(MAX_WEIGHTS)));
    }
    return routine;
}

100.times {
    int id = it;

    Thread.start {  // Client
        def routine = make_routine(20); // random routine of 20 exercises
        // enter gym

    routine.size().times {
            // complete exercise on machine
        println "$id is performing:"+routine[it][0] + "--"+ routine[it][1];
    }
    }
}

return ;

return ;
```

2. Indicate whether your solution is free of starvation. If it's not, indicate how you could obtain it.

Exercise 6. We would like to model a *control system* for an automatic car wash. Each car traverses three stations: blast, rinse and dry. Each of these stations is executed by a machine. All vehicles follow these three stations in that exact order.



Some additional considerations:

- A machine can only start working on a car once the car is in place

- A car can only leave a station once it knows the machine has finished its work
- There can be at most one car in each station
- A car cannot advance to the next station if it is occupied by another car

Model the cars and each machine with appropriate threads. Here is a stub you can use as guideline:

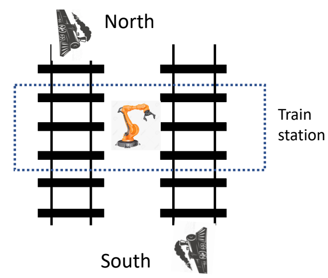
```
import java.util.concurrent.Semaphore;

Semaphore station0 = ??
Semaphore station1 = ??
Semaphore station2 = ??
permToProcess = [??, ??, ??] // list of semaphores for machines
doneProcessing = [??, ??, ??] // list of semaphores for machines

100.times {
    Thread.start { // Car
        // Go to station 0
        // Move on to station 1
        // Move on to station 2
      }
}

3.times {
    int id = it; // iteration variable
    Thread.start { // Machine at station id
        while (true) {
            // Wait for car to arrive
            // Process car when it has arrived
            }
    }
}

return;
```

Exercise 7. (◊) Model a vehicle crossing between two endpoints. We'll denote these endpoints 0 and 1. Since the crossing is narrow, it does not allow for vehicles to travel in opposite directions. Your solution must allow multiple vehicles to use the crossing so long as they are travelling in the same direction. Use the following stub as guideline:

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

// Declare semaphores here
noOfCarsCrossing = [0,0]; // list of ints
r = new Random();

100.times {
    int myEndpoint = r.nextInt(2);  // pick a random direction
    Thread.start {  // Car
      // entry protocol
      // cross crossing
      // exit protocol
    }
}
```

- How would you modify your solution so that at most 3 vehicles are on the crossing at any given time?
- Is your solution fair?

Exercise 8. Trains run in both North-South (0) and South-North (1) direction, each on its own track.



Two kinds of trains ride these tracks: passenger trains and freight trains. You are to model the behavior of a train station for each of these two kinds of trains.

- Passenger trains: A passenger train can only stop at the station if there are no other trains on the same track. It does not matter whether there is a train at the station on the track corresponding to trains travelling in the opposite direction.
- Freight trains: The station has the ability to load freight trains via a loading machine. In order for a freight train to be able to be loaded, there must not be trains at the station (in any of the two tracks). Moreover, if the freight train makes use of the station, it cannot leave until the loading machine is done.

Model the passenger train and the freight train as threads. The loading machine has already been modeled for you below. Additional semaphores may be required.

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

Semaphore permToLoad = ??;
Semaphore doneLoading = ??;
// Additional semaphores


100.times {
    int dir = (new Random()).nextInt(2);
    Thread.start { // PassengerTrain travelling in direction dir
    // complete
    }
}

100.times {
    int dir = (new Random()).nextInt(2);
    Thread.start { // Freight Train travelling in direction dir
    // complete
    }
}

Thread.start { // Loading Machine
    while (true) {
    permToLoad.acquire();
    // load freight train
    doneLoading.release();
    }
}
```

Exercise 9. Consider a concurrent system with `N` threads. Implement barrier synchronization: threads must suspend upon reaching a barrier until all the other threads arrive, after which they all continue execution. If such a barrier is used just once, we call it a one-time use barrier. If this synchronization pattern is repeated inside a loop, we call it a cyclic barrier. You are asked to implement a one-time use barrier by completing the following stub:

```
import java.util.concurrent.Semaphore
// One-time use barrier
// Barrier size = N
// Total number of threads in the system = N

final int N=3
// Declare semaphores and other variables here



N.times {
    int id = it
    Thread.start {
          // complete barrier arrival protocol

          println id+" got to barrier. Waiting for the other threads"
          // complete suspend at barrier


          println id+" went through."
    }
}
```

1 Solutions to Selected Exercises

Answer to exercise 1

Groovy

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

Semaphore ticket = new Semaphore(0);
Semaphore mutex = new Semaphore(1);

20.times{
    Thread.start { // Patriots
     ticket.release();
    }
}

20.times {
Thread.start { // Jets
    mutex.acquire();
    ticket.acquire();
    ticket.acquire();
    mutex.release();
   }
}
```

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

Semaphore ticket = new Semaphore(0);
Semaphore mutex = new Semaphore(1);
boolean itGotLate = false;

Thread.start { // Jets
    mutex.acquire();
    if (!itGotLate) {
       ticket.acquire();
       ticket.acquire();
    }
    mutex.release();
}

Thread.start { // Patriots
    ticket.release();
}

Thread.start {// ItGotLate
    sleep(10000)
    itGotLate = true
    ticket.release()
    ticket.release()
}

return
```

Explain why the following is incorrect:

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

```
Semaphore ticket = new Semaphore(0);
Semaphore mutex = new Semaphore(1);
boolean itGotLate = false;

Thread.start { // Jets
    if (!itGotLate) {
       mutex.acquire();
       ticket.acquire();
       ticket.acquire();
       mutex.release();
    }
}

Thread.start { // Patriots
    ticket.release();
}

Thread.start {// ItGotLate
    sleep(10000)
    itGotLate = true
    ticket.release()
    ticket.release()
}

return
```

Java

```
package basics;
import java.util.concurrent.Semaphore;
import javax.swing.plaf.multi.MultiTextUI;

public class Bar {

    static Semaphore ticket = new Semaphore(0);
    static Semaphore counters = new Semaphore(1);
    static int jets=0;
    static int patriots=0;

    public static class Jet implements Runnable {

        static Semaphore mutex = new Semaphore(1);

        public void run() {

            try {
                mutex.acquire();
                ticket.acquire();
                ticket.acquire();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            try {
                counters.acquire();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
```

```
            }
            jets++;
            assert jets*2<=patriots;
            System.out.println("J");
            counters.release();
            mutex.release();
        }
    }

    public static class Patriot implements Runnable {

        public void run() {
            try {
                counters.acquire();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            ticket.release();
            patriots++;
            System.out.println("P");
            counters.release();
        }
    }

    public static void main(String[] args) {
        for (int i=0; i<20; i++) {
            new Thread(new Jet()).start();
        }

        for (int i=0; i<20; i++) {
            new Thread(new Patriot()).start();
        }
    }
}
```

Answer to exercise 7

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

Semaphore useCrossing = new Semaphore(1); //mutex
endpointMutexList = [new Semaphore(1, true), new Semaphore(1, true)]; // Strong sem.
noOfCarsCrossing = [0,0]; // list of ints
r = new Random();

100.times { // spawn 100 cars
    int myEndpoint = r.nextInt(2);  // pick a random direction
    Thread.start {
      endpointMutexList[myEndpoint].acquire();
      if (noOfCarsCrossing[myEndpoint] == 0)
        useCrossing.acquire();
      noOfCarsCrossing[myEndpoint]++;
      endpointMutexList[myEndpoint].release();

      //Cross crossing
      println ("car $it crossing in direction "+myEndpoint + " current totals "+noOfCarsCrossing);

      endpointMutexList[myEndpoint].acquire();
```

```
      noOfCarsCrossing[myEndpoint]--;
      if (noOfCarsCrossing[myEndpoint] == 0)
        useCrossing.release();
      endpointMutexList[myEndpoint].release();
    }
}
```