Concurrent Programming

Exercise Booklet 6: Monitors

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**.

Strategy: Signal and continue ($E = W < S$)

Exercise 1. Consider the fan bar exercise from Exercise Booklet 5. Here is the statement. 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 a solution using monitors. Use the following stub as guideline:

```
class Bar {
    // your code here



}

Bar b = new Bar();
100.times {
    Thread.start {// jets
        b.jets();
    }
}

100.times {
    Thread.start {// patriots
        b.patriots();
    }
}
```

2. Consider the scenario from Exercise Booklet 5 where it may get late. Add a method `Bar.itGotLate()` to model that situation.

Exercise 2. (◊) We wish to implement a `three-way sequencer` using monitors in order to coordinate N threads. A three-way sequencer provides the following operations `first`, `second`, `third`. The idea is that each of the threads can invoke any of these operations. The sequencer will alternate cyclically the execution of `first`, then `second`, and finally `third`.

Exercise 3. We wish to implement a barrier using monitors in order to coordinate N threads. A barrier provides a unique operation called `waitAtBarrier`. The idea is that each of the N threads invokes the operation `waitAtBarrier` and its effect is that the thread will block, and cannot continue, until all remaining threads invoke the `waitAtBarrier` operation. For example, if `b` is a barrier for coordinating 3 threads, the following use of `b` in each thread

```
class Barrier {
   // complete

}

Barrier b = new Barrier(3);

Thread.start { //T1
    print("a");
    b.waitAtBarrier();
    print("1");
}

Thread.start { //T2
    print("b");
    b.waitAtBarrier();
    print("2");
}

Thread.start {// T3
    print("c");
    b.waitAtBarrier();
    print("3");
}
```

guarantees that the letters will be displayed before the numbers. Supply an implementation for the barrier monitor. You may assume that once all N processes reach the barrier, they will be allowed to continue and so will all subsequent threads that call `waitAtBarrier` (non-cyclic barrier or count down latch).

Exercise 4. This exercise is a based on the Train exercise from Exercise Booklet 5. Trains run in both North-South and South-North direction, each on its own track. Two kinds of trains ride these tracks: passenger trains and freight 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 (which shall not be modeled). In order for a freight train to stop at the station, no other trains can be at the station in any of the two tracks.

Complete the following monitor operations:

```
import java.util.concurrent.locks.*;

class TrainStation {

    void acquireNorthTrackP() {

    }

    void releaseNorthTrackP() {

    }
```

```

    void acquireSouthTrackP() {

    }

    void releaseSouthTrackP() {

    }

    void acquireTracksF() {

    }

    void releaseTracksF() {

    }
}

TrainStation s = new TrainStation();

100.times{
  Thread.start { // Passenger Train going North
    s.acquireNorthTrackP();
    print "NPT"+Thread.currentThread().getId();
    s.releaseNorthTrackP();
  }
}

100.times{
    Thread.start { // Passenger Train going South
      s.acquireSouthTrackP();
      print "SPT"+ Thread.currentThread().getId();
      s.releaseSouthTrackP()
    }
}

10.times {
  Thread.start { // Freight Train
    s.acquireTracksF();
    print "FT "+ Thread.currentThread().getId();
    s.releaseTracksF();
  }
}
```

Exercise 5. In a smart energy grid connected users can either behave as consumers or producers of energy. For example, the following code fragment shows a user that behaves as a consumer for an hour and then as a producer for two hours:

```
   grid.startConsuming();
   sleep(1h);
   grid.stopConsuming();
   grid.startProducing();
   sleep(2h);
   grid.stopProducing();
```

You may use the following stub:

```java
class Grid {

    void synchronized startConsuming() {

    }

    void synchronized stopConsuming() {

    }

    void synchronized startProducing() {

    }

    void synchronized stopProducing() {

    }
}
```

Address the following scenarios using monitors:

1. Users can always behave as producers, but can only behave as consumers if there is an equal or greater number of producers. Moreover, you must guarantee that a producer cannot stop producing if, in doing so, it leaves some consumer without a supply.
2. Modify your solution so that users cannot behave as producers (i.e. they block) if the grid has more than N producers.
3. Extend the previous solution so that the exit of producers is given priority over the entry of new consumers.

Exercise 6. In a local pizza shop two types of pizzas are produced: small and large. The cook places the pizzas on a counter so that the clients can help themselves and then pay at the register. Clients may either buy a small pizza or a large pizza. In the case of large pizzas, if there are no large pizzas, they reluctantly accept two small pizzas. Complete the following stub:

```java
class Pizza {
    // Variables declared here

    void purchaseSmallPizza() {
      // complete
    }

    void purchaseLargePizza() {
      // complete
    }

    void bakeSmallPizza() {
      // complete
    }

    void bakeLargePizza() {
      // complete
    }
}
```

Modify your solution assuming that the counter holds at most N pizzas at any given time.

1 Solutions to Selected Exercises

Answer to exercise 2

```java
class TWS {
  int state = 1;
  static final Lock lock = new ReentrantLock();
  static final Condition first = lock.newCondition();
  static final Condition second = lock.newCondition();
  static final Condition third = lock.newCondition();


  void first() {
    lock.lock();
    try {
      while (state != 1)
         first.wait();
      state = 2;
      second.signal();
    } finally {
      lock.unlock();
    }

  }

  void second() {
    lock.lock();
    try {
      while (state != 2)
      second.wait();
         state = 3;
      third.signal();
    } finally {
      lock.unlock();
    }
  }

  void third() {
    lock.lock();
    try {
      while (state != 3)
         third.wait();
      state = 1;
      first.signal();
    } finally {
      lock.unlock();
    }
  }
}
```