**Trabalho 10**

Exercícios do AMP

Exercise 93. Reimplement the SimpleReadWriteLock class using Java synchronized, wait(), notify(), and notifyAll() constructs in place of explicit locks and conditions.

Hint: you must figure out how methods of the inner read–write lock classes can lock the outer SimpleReadWriteLock object.

https://github.com/fernandocastor/pp/tree/master/10/ecrm_tccn/01/src

Exercise 96. In the shared bathroom problem, there are two classes of threads, called male and female. There is a single bathroom resource that must be used in the following way:

1. Mutual exclusion: persons of opposite sex may not occupy the bathroom simultaneously,
2. Starvation-freedom: everyone who needs to use the bathroom eventually enters.

The protocol is implemented via the following four procedures: enterMale() delays the caller until it is ok for a male to enter the bathroom, leaveMale() is called when a male leaves the bathroom, while enterFemale() and leaveFemale() do the same for females. For example,

```
enterMale();
teeth.brush(toothpaste);
leaveMale();
```

1. Implement this class using locks and condition variables.
2. Implement this class using synchronized, wait(), notify(), and notifyAll().

For each implementation, explain why it satisfies mutual exclusion and starvation-freedom.

```
public interface Bathroom {
   public void enterMale();
   public void enterFemale();
   public void leaveMale();
   public void leaveFemale();
}
```

```
public class BathroomTest {
   public static void main(String[] args) {
      Bathroom bathroom = new BathroomOne();

      (new Thread(new Female(bathroom, "Ana"))).start();
      (new Thread(new Female(bathroom, "Maria"))).start();
      (new Thread(new Male(bathroom, "Mateus"))).start();
      (new Thread(new Male(bathroom, "Marcos"))).start();
      (new Thread(new Male(bathroom, "Lucas"))).start();
   }
}
```

```
import java.util.Random;

public class Male implements Runnable {

   private Bathroom bathroom;
   private String name;

   public Male(Bathroom bathroom, String name) {
      this.bathroom = bathroom;
      this.name = name;
   }

   @Override
   public void run() {
      while (true) {
         this.bathroom.enterMale();
         try {
            System.out.println("Male (" + this.name + ") inside.");

            // person is doing something inside
            Random random = new Random();
            Thread.sleep(random.nextInt(2000));
         } catch (InterruptedException e) {
            // do nothing
         } finally {
            System.out.println("Male (" + this.name + ") leaving.");
            this.bathroom.leaveMale();
         }

         try {
            // person is doing something outside
            Random random = new Random();
```

```
            Thread.sleep(random.nextInt(4000));
        } catch (Exception e) {
            // do nothing
        }
    }
  }
}
```

```
import java.util.Random;

public class Female implements Runnable {

  private Bathroom bathroom;
  private String name;

  public Female(Bathroom bathroom, String name) {
    this.bathroom = bathroom;
    this.name = name;
  }


  @Override
  public void run() {
    while (true) {
      this.bathroom.enterFemale();
      try {
        System.out.println("Female (" + this.name + ") inside.");

        // person is doing something inside
        Random random = new Random();
        Thread.sleep(random.nextInt(2000));
      } catch (InterruptedException e) {
        // do nothing
      } finally {
        System.out.println("Female (" + this.name + ") leaving.");
        this.bathroom.leaveFemale();
      }

      try {
        // person is doing something outside
        Random random = new Random();
        Thread.sleep(random.nextInt(4000));
      } catch (Exception e) {
        // do nothing
      }
    }
```

```
    }
}
```

**Implement this class using locks and condition variables.**

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class BathroomOne implements Bathroom {

    private int maleCount;
    private int femaleCount;
    private Lock lock;
    private Condition condition;

    public BathroomOne() {
        this.lock = new ReentrantLock();
        this.condition = this.lock.newCondition();
        this.maleCount = 0;
        this.femaleCount = 0;
    }

    @Override
    public void enterMale() {
        this.lock.lock();
        try {
            while (this.femaleCount > 0) {
                try {
                    this.condition.await();
                } catch (InterruptedException e) {
                    // do nothing
                }
            }

            this.maleCount++;
            this.verifyBathroom();
        } finally {
            this.lock.unlock();
        }
    }

    @Override
    public void enterFemale() {
        this.lock.lock();
        try {
```

```
            while (this.maleCount > 0) {
                try {
                    this.condition.await();
                } catch (InterruptedException e) {
                    // do nothing
                }
            }

            this.femaleCount++;
            this.verifyBathroom();
        } finally {
            this.lock.unlock();
        }
    }

    @Override
    public void leaveMale() {
        this.lock.lock();
        try {
            this.maleCount--;
            this.condition.signalAll();
        } finally {
            this.lock.unlock();
        }
    }

    @Override
    public void leaveFemale() {
        this.lock.lock();
        try {
            this.femaleCount--;
            this.condition.signalAll();
        } finally {
            this.lock.unlock();
        }
    }

    private void verifyBathroom() {
        if (this.femaleCount > 0 && this.maleCount > 0) {
            throw new RuntimeException("There is a male and a female inside.");
        }
    }
}
```

Na primeira solução a exclusão mútua é atendida pelo uso do mesmo Lock em todas as operações. Já a propriedade de starvation freedom é atendida pela chamada de this.condition.signalAll(), onde as threads em espera retornam à execução para verificar novamente se podem entrar na seção desejada.

**Implement this class using synchronized, wait(), notify(), and notifyAll().**

```
public class BathroomTwo implements Bathroom {

   enum Someone { MALE, FEMALE }

   private int maleCount;
   private int femaleCount;

   public BathroomTwo() {
      this.maleCount = 0;
      this.femaleCount = 0;
   }

   private synchronized void waitSomeone(Someone someone) {
      if (someone == Someone.MALE) {
         while (this.maleCount > 0) {
            try {
               wait();
            } catch (InterruptedException e) {
               // do nothing
            }
         }

         this.femaleCount++;
      } else {
         while (this.femaleCount > 0) {
            try {
               wait();
            } catch (InterruptedException e) {
               // do nothing
            }
         }

         this.maleCount++;
      }
   }

   @Override
   public void enterMale() {
      this.waitSomeone(Someone.FEMALE);
      this.verifyBathroom();
   }
```

```
    @Override
    public void enterFemale() {
        this.waitSomeone(Someone.MALE);
        this.verifyBathroom();
    }

    @Override
    public void leaveMale() {
        this.leaveSomeone(Someone.MALE);
    }

    @Override
    public void leaveFemale() {
        this.leaveSomeone(Someone.FEMALE);
    }

    private synchronized void leaveSomeone(Someone someone) {
        if (someone == Someone.MALE) {
            this.maleCount--;
        } else {
            this.femaleCount--;
        }

        notifyAll();
    }

    private void verifyBathroom() {
        if (this.femaleCount > 0 && this.maleCount > 0) {
            throw new RuntimeException("There is a male and a female inside.");
        }
    }
}
```

Na segunda solução a propriedade de exclusão mútua é atingida através do uso de um método único sincronizado para espera. Eventualmente todos vão poder entrar por conta do notifyAll() chamado quando os contadores são decrementados, o que indica a saída de alguma pessoa.

Exercise 98. Consider an application with distinct sets of active and passive threads, where we want to block the passive threads until all active threads give permission for the passive threads to proceed.

A CountDownLatch encapsulates a counter, initialized to be n, the number of active threads. When an active method is ready for the passive threads to run, it calls countDown(), which decrements the counter. Each passive thread calls await(), which blocks the thread until the counter reaches zero. (See Fig. 8.16.)

```
1  class Driver {
2    void main() {
3      CountDownLatch startSignal = new CountDownLatch(1);
4      CountDownLatch doneSignal = new CountDownLatch(n);
5      for (int i = 0; i < n; ++i) // start threads
6        new Thread(new Worker(startSignal, doneSignal)).start();
7      doSomethingElse();          // get ready for threads
8      startSignal.countDown();    // unleash threads
9      doSomethingElse();          // biding my time ...
10     doneSignal.await();         // wait for threads to finish
11   }
12   class Worker implements Runnable {
13     private final CountDownLatch startSignal, doneSignal;
14     Worker(CountDownLatch myStartSignal, CountDownLatch myDoneSignal) {
15       startSignal = myStartSignal;
16       doneSignal = myDoneSignal;
17     }
18     public void run() {
19       startSignal.await();    // wait for driver's OK to start
20       doWork();
21       doneSignal.countDown(); // notify driver we're done
22     }
23     ...
24   }
25 }
```

**Figure 8.16** The CountDownLatch class: an example usage.

Provide a CountDownLatch implementation. Do not worry about reusing the CountDownLatch object.

```
import java.util.Random;

public class Worker implements Runnable {

   private CountDownLatch startSignal;
   private CountDownLatch doneSignal;

   public Worker(CountDownLatch startSignal2, CountDownLatch doneSignal2) {
      this.startSignal = startSignal2;
      this.doneSignal = doneSignal2;
   }

   @Override
   public void run() {
      try {
         this.startSignal.await();
      } catch (InterruptedException e) {
         System.out.println("Exception on worker.");
      }
      this.doSomething();
      this.doneSignal.countDown();
```

```
    }

    private void doSomething() {
        try {
            System.out.println("Something will be made on worker...");
            Random random = new Random();
            Thread.sleep(random.nextInt(4000));
        } catch (Exception e) {
            // do nothing
        }
    }
}
```

```
import java.util.Random;

public class Driver {

    public static void main(String[] args) {
        int n = 3;
        CountDownLatch startSignal = new CountDownLatch(1);
        CountDownLatch doneSignal = new CountDownLatch(n);
        for (int i = 0; i < n; i++) {
            new Thread(new Worker(startSignal, doneSignal)).start();
        }
        doSomething();
        startSignal.countDown();
        doSomething();
        try {
            doneSignal.await();
        } catch (InterruptedException e) {
            System.out.println("Exception on driver.");
        }
    }

    private static void doSomething() {
        try {
            System.out.println("Something will be made...");
            Random random = new Random();
            Thread.sleep(random.nextInt(4000));
        } catch (Exception e) {
            // do nothing
        }
    }
}
```

```
public class CountDownLatch {

    private int count;

    public CountDownLatch(int count) {
        this.count = count;
    }

    public synchronized void countDown() {
        this.count--;
        notifyAll();
    }

    public synchronized void await() throws InterruptedException {
        while (this.count > 0) {
            wait();
        }
    }
}
```