Freie Universität Berlin

Prof. Dr. Margarita Esponda

Nichtsequentielle Programmierung, SoeSe 2017

Übungsblatt 4

TutorIn: Lilli Walter
Tutorium 6

Boyan Hristov, Sergelen Gongor

6. Juni 2017

Link zum Git Repository: https://github.com/BoyanH/FU-Berlin-ALP4/tree/master/Solutions/Homework4

Aufgabe 1

Es ist wichtig, dass das playing state von dem Bank ihr Gehalt entspricht, sonst können gleichzeitig zwei Spieler spielen und ein negatives Gehalt von dem Bank kriegen. Weiter ist es wichtig, dass keiner spielen kann, wenn das Casino nicht mehr spielz bzw. wenn es kein Geld mehr hat.

Wir haben das relevante Teil von CasinoBank::playAGame() synchronisiert, damit das Verlieren/Gewinnen von Geld, die Veränderung des isPlaying und das Erlaubnis dem Spieler ein weiteres Spiel zu spielen alle richtig behandelt werden.

```
1  package fu.alp4;
2
3  import javax.swing.*;
4
5  public class CasinoBank {
6
7      private int money;
8      private boolean playing;
9      private JTextArea textArea;
10     private Object moneyLock;
11
12     public CasinoBank() {
13         this.money = 20;
14         this.textArea = new JTextArea();
15         this.moneyLock = new Object();
16         this.playing = true;
17     }
18
19
20     public boolean getAvailable() {
21         return this.playing;
22     }
23
24     public boolean playAGame() throws Exception {
```

```
26          boolean playerWon;


29          double random = Math.round(Math.random());
30          playerWon = random <= 0.4;

32          synchronized (moneyLock) {

34              if (!this.playing) {
35                  throw new Exception("Can't play further games. Casino closed!");
36              }

38              if (playerWon) {
39                  this.money--;
40                  this.playing = this.money > 0;
41              } else {
42                  this.money++;
43              }
44          }

46          this.textArea.setText(String.format("Casino has %s dollars. Casino playing: %s",
        this.money, this.playing));

48          return playerWon;
49      }

51      JTextArea getTextArea() {

53          return this.textArea;
54      }
55  }


58  package fu.alp4;

60  import javax.swing.*;

62  /**
63   * Created by hristov on 5/8/17.
64   */

66  public class Gambler extends Nap {

68      private CasinoBank playingInCasino;
69      private int budget;
70      private final int initialBudget;
71      private boolean playing;
72      private String name;
73      private JTextArea textArea;

75      public Gambler(String givenName, CasinoBank casino) {

77          this.initialBudget = 5;
78          this.budget = this.initialBudget;
79          this.playing = true;
80          this.name = givenName;
81          this.textArea = new JTextArea();
82          this.playingInCasino = casino;
83      }

85      public void run() {

87          while(this.isPlaying()) {

89              this.think();
90              this.bet();
91          }
```

```
92      }
94      public void think() {
96          randomNap(500, 1500);
97      }
99      public void bet() {
101         boolean justWon;
103         try {
104             justWon = this.playingInCasino.playAGame();
106             if(justWon) {
107                 this.budget++;
109             }
110             else {
111                 this.budget--;
112             }
114             if(this.budget == 0 || this.budget == this.initialBudget*2 || !this.
    playingInCasino.getAvailable()) {
116                 this.playing = false;
117             }
119             this.textArea.setText(this.name + " just " +
120                     (justWon ? "won" : "lost") +
121                     " 1 dollar. Current budget: " + this.budget + " ; still playing: " +
122                     this.isPlaying());
123         }
124         catch (Exception e) {
125             // casino closed
126             this.playing = false;
127             this.textArea.setText(this.name + " is no longer playing, because the casino
    closed!");
128         }
129     }
131     public boolean isPlaying() {
133         return this.playing;
134     }
136     JTextArea getTextArea() {
138         return this.textArea;
139     }
140 }


143 package fu.alp4;
145 import javax.swing.*;
147 public class Main extends JPanel{
149     public static void main(String[] args) {
151         CasinoBank casino = new CasinoBank();
153         Gambler[] gamblers = new Gambler[3];
154         gamblers[0] = new Gambler("Pesho", casino);
155         gamblers[1] = new Gambler("Stamat", casino);
156         gamblers[2] = new Gambler("Gosho", casino);
```

```
158         JFrame f = new JFrame();
159         f.setContentPane(new Main());
160         f.setSize(400,400);
161         f.setVisible(true);

163         f.add(casino.getTextArea());
164         for(int i = 0; i < gamblers.length; i++) {

166             gamblers[i].start();
167             f.add(gamblers[i].getTextArea());
168         }

170     }
171 }
```

Aufgabe 2

Für diese Aufgabe haben wir für die Synchronation ein Semaphor freeBabySlots benutzt. Wenn neue Babies kommen / bzw. Eltern, wird ein acquire auf dem Semaphor gemacht. Eine neue Nurse gibt 5 Slots frei wenn sie zur Arbeit kommt, und acquired dann wieder 5 Slots wenn sie nach Hause geht.

```
1  package fu.alp4;

3  import java.util.LinkedList;
4  import java.util.List;
5  import java.util.concurrent.Semaphore;

7  public class Kita {

9      private int babies;
10     private List<Parent> parents;
11     private List<Nurse> nurses;

13     private Semaphore freeBabySlots;

15     public Kita() {
16         this.parents = new LinkedList<>();
17         this.nurses = new LinkedList<>();
18         this.babies = 0;
19         this.freeBabySlots = new Semaphore(0, true);
20     }

22     public void giveBabyToNursery(Parent parent) {

24         System.out.println("New baby on the horizon!");
25         try {
26             this.freeBabySlots.acquire();
27             this.babies++;
28         } catch (InterruptedException e) {
29             e.printStackTrace();
30         }
31         this.parents.add(parent);
32         System.out.printf("New baby accepted!");
33         this.printState();
34     }

36     public void takeBabyFromNursery(Parent parent) {
37         this.freeBabySlots.release();
38         this.parents.add(parent);
39         this.babies--;
40         System.out.println("Baby taken from nursery!");
41         this.printState();
42     }
```

```
    public void requestNewNurse(Nurse nurse) {
        this.freeBabySlots.release(5);
        nurses.add(nurse);
        System.out.println("A new nurse came. Things are going well.");
        this.printState();
    }

    public void requestSendNurseHome(Nurse nurse) {
        System.out.println("A nurse want to abandon ship!");
        try {
            this.freeBabySlots.acquire(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        nurses.remove(nurse);
        System.out.println("The nurse left the vessel.");
        this.printState();
    }

    public void printState() {
        System.out.printf("#Nurses: %s; #Babies: %s; Babies capacity left: %s\n\n",
                this.nurses.size(), this.babies, this.freeBabySlots.availablePermits());
    }
}



package fu.alp4;

public class Nurse extends Nap {

    private Kita kita;

    public Nurse(String name, Kita kita) {
        this.kita = kita;
    }

    public void run() {

        while(true) {
            Nurse.randomNap(5000, 15000);
            this.goToWork();
            Nurse.randomNap(30000, 70000);
            this.goHome();
        }
    }

    public void goHome() {
        kita.requestSendNurseHome(this);
    }

    public void goToWork() {
        this.kita.requestNewNurse(this);
    }
}
package fu.alp4;




public class Parent extends Nap {

    private Kita kita;

    public Parent(String name, Kita kita) {
        this.kita = kita;
    }
```

```
    public void run() {

        while(true) {
            Parent.randomNap(5000, 15000);
            this.sendBabyToNursery();
            Parent.randomNap(20000, 50000);
            this.takeBabyFromNursery();
        }
    }

    public void sendBabyToNursery() {
        this.kita.giveBabyToNursery(this);
    }

    public void takeBabyFromNursery() {
        this.kita.takeBabyFromNursery(this);
    }
}
```

```
package fu.alp4;

public class Main {

    public static void main(String[] args) {
    // write your code here

        Kita kita = new Kita();

        for (int j = 0; j < 15; j++) {
            String crntParentName = "Parent #" + (j+1);
            Parent crntParent = new Parent(crntParentName, kita);
            crntParent.start();
        }

        for (int i = 0; i < 3; i++) {
            String crntNurseName = "Nurse #" + (i+1);
            Nurse crntNurse = new Nurse(crntNurseName, kita );
            crntNurse.start();
        }
    }
}
```

```
package fu.alp4;

/**
 * Created by hristov on 5/8/17.
 */
public class Nap extends Thread{

    public static void nap(int milliSeconds) {

        try {
            Thread.sleep(milliSeconds);
        }
        catch (InterruptedException e) {
            System.out.println("Sleep was interrupted. " + e.getMessage());
        }
    }

    public static void randomNap(int minMilliSeconds, int maxMilliSeconds) {
```

```
181         int randomMilliSeconds = (int) Math.round(Math.random()*(maxMilliSeconds-
    minMilliSeconds) + minMilliSeconds);
182         nap(randomMilliSeconds);
183     }
184 }
```

Aufgabe 3

a) Done

b) Mit ReentrantLock

```
2  package control;
4  import vehicle.Vehicle;
6  import java.util.concurrent.locks.ReentrantLock;
8  public class OneAtATimeBCReentrantLock implements BridgeControl {
9      double maxLoad;
10     final ReentrantLock lock = new ReentrantLock();
12     @Override
13     public void init(Double maxLoad) {
14         this.maxLoad = maxLoad;
15     }
17     @Override
18     public void requestCrossing(Vehicle v) {
19         if (v.getWeight() <= maxLoad) {
20             lock.lock(); // acquire the lock, no one else can pass in the same time
21         }
22         else {
23             try {
24                 if (v.getWeight() > maxLoad) {
25                     Thread.sleep(Long.MAX_VALUE);
26                 }
27             } catch (InterruptedException e) {
28                 e.printStackTrace();
29             }
30         }
31     }
33     @Override
34     public void leaveBridge(Vehicle v) {
35         lock.unlock(); // release the lock once the auto has passed the bridge
36     }
37 }
```

Mit Monitorkonzept

```
1  package control;
3  import vehicle.Vehicle;
5  public class OneAtATimeBCMonitor implements BridgeControl {
6      double maxLoad;
7      boolean bridgeFree = true;
9      @Override
10     public void init(Double maxLoad) {
11         this.maxLoad = maxLoad;
```

```
    }

    @Override
    public synchronized void requestCrossing(Vehicle v) {
        // this method is synchronized, so only
        try {

            while (!bridgeFree || v.getWeight() > maxLoad) {
                wait();
            }

            bridgeFree = false;
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public synchronized void leaveBridge(Vehicle v) {
        bridgeFree = true;
    }
}
```

Mit Semaphoren

```
package control;

import vehicle.Vehicle;
import vehicle.Vehicle.VOrigin;

import java.util.concurrent.Semaphore;

public class OneAtATimeBCSemaphore implements BridgeControl {

    double maxLoad;
    final Semaphore bridgeCrossingSlot = new Semaphore(1, true); // one car can pass
    at a time, fair queue

    @Override
    public void init(Double maxLoad) {
        this.maxLoad = maxLoad;
    }

    @Override
    public void requestCrossing(Vehicle v) {

        try {
            if (v.getWeight() > maxLoad) {
                Thread.sleep(Long.MAX_VALUE);
            }

            bridgeCrossingSlot.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void leaveBridge(Vehicle v) {
        bridgeCrossingSlot.release();
    }

}
```

c) Hier haben wir eine Ampel simuliert, deswegen haben wir auch dafür gesorgt, dass man dann später noch weitere Straßen (nicht nur West und Öst) leicht einfügen kann und auch dafür gesorgt, dass

wenn 5 Autos vor andere 5 auf je Seite angekommen sind, dann werden diese auch früher über die Brücke fahren.

```java
package control;

import vehicle.Vehicle;
import vehicle.Vehicle.VOrigin;

import java.util.concurrent.Semaphore;

public class FivePerDirectionBC implements BridgeControl {

    static final int autosPerWave = 5;
    static VOrigin currentWaveOrigin;
    static int autosPassedFromCurrentWave;

    Semaphore[] slotsByOrigin;
    Semaphore weight;

    Semaphore leaveBridgeMutex;
    Semaphore initializeMutex;
    Semaphore weightHandlerMutex;

    double maxLoad;

    /**
     * Initialize all variables and semaphores, set currentLoad and autosPassedFromCurrentWave to 0,
     * set currentWave to West and add 5 available slots for cars on the west side
     *
     * @param maxLoad Max load (depending on users input)
     */

    @Override
    public synchronized void init(Double maxLoad) {

        this.maxLoad = maxLoad;
        weight = new Semaphore((int) Math.floor(maxLoad*100), true);
        leaveBridgeMutex = new Semaphore(1, true);
        initializeMutex = new Semaphore(1, true);
        weightHandlerMutex = new Semaphore(1, true);
        autosPassedFromCurrentWave = 0;

        slotsByOrigin = new Semaphore[VOrigin.values().length];
        for (int i = 0; i < slotsByOrigin.length; i++) {
            slotsByOrigin[i] = new Semaphore(0, true);
        }

        try {
            initializeMutex.acquire();
            setWave(VOrigin.WEST);
            initializeMutex.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    /**
     *
     * Each vehicle that wants to pass the bridge acquires from the semaphore for it's side.
     * If it does indeed get a slot, it checks if the bridge can sustain it.
     * If not, the thread goes to sleep for a looong time.
     *
     * Further each vehicle with a slot checks if the bridge can sustain it and all vehicle on the bridge and
     * waits until the condition is met.
     *
```

```
63      * @param v  Vehicle asking for the permission to cross the bridge.
64      */
65
66     @Override
67     public void requestCrossing(Vehicle v) {
68
69         try {
70             getSlotsByOrigin(v.getOrigin()).acquire();
71
72             weightHandlerMutex.acquire();
73
74             if (v.getWeight() > maxLoad) {
75                 getSlotsByOrigin(v.getOrigin()).release();
76                 weightHandlerMutex.release();
77                 Thread.sleep(Long.MAX_VALUE);
78             }
79
80             weightHandlerMutex.release();
81
82         } catch (InterruptedException e) {
83             e.printStackTrace();
84         }
85
86         try {
87             weight.acquire(getWeightFromVehicle(v));
88         } catch (InterruptedException e) {
89             e.printStackTrace();
90         }
91
92     }
93
94
95     /**
96      *
97      * When leaving the bridge each auto adds to the count of total vehicles passed.
        That way, we are able to change
98      * the current wave (west or east at the time) exactly when the last vehicle has
        left the bridge.
99      *
100     * Further we remove its weight from the calculation.
101     *
102     * @param v  Vehicle finished the crossing
103     */
104
105    @Override
106    public void leaveBridge(Vehicle v) {
107
108        try {
109            leaveBridgeMutex.acquire();
110            autosPassedFromCurrentWave++;
111            weight.release(getWeightFromVehicle(v));
112
113            if (autosPassedFromCurrentWave == autosPerWave) {
114                autosPassedFromCurrentWave = 0;
115                setWave(getNextOrigin(currentWaveOrigin));
116            }
117            leaveBridgeMutex.release();
118        } catch (InterruptedException e) {
119            e.printStackTrace();
120        }
121    }
122
123
124    /**
125     *
126     * Gets the next origin from the sorted VOrigin.values() array. We treat this
        array as a circular array.
127     *
```

```
128         * @param origin VOrigin
129         * @return VOrigin
130         */
131        private VOrigin getNextOrigin(VOrigin origin) {
132
133            VOrigin[] vOrigins = VOrigin.values();
134            int currentOriginIndex = java.util.Arrays.binarySearch(vOrigins, origin);
135
136            if (currentOriginIndex != vOrigins.length - 1) {
137                return vOrigins[currentOriginIndex + 1];
138            }
139
140            return vOrigins[0];
141        }
142
143        /**
144         * Gets all the origins but the current one from VOrigin.values() array
145         *
146         * @param origin
147         * @return
148         */
149        private static VOrigin[] getNonCurrentOrigins(VOrigin origin) {
150
151            VOrigin[] vOrigins = VOrigin.values();
152            VOrigin[] nonCurrentOrigins = new VOrigin[vOrigins.length - 1];
153            int ncoCounter = 0;
154
155            for (int i = 0; i < vOrigins.length; i++) {
156
157                if (vOrigins[i] != origin) {
158                    nonCurrentOrigins[ncoCounter] = vOrigins[i];
159                    ncoCounter++;
160                }
161            }
162
163            return nonCurrentOrigins;
164        }
165
166        /**
167         * Maps each item's index in VOrigins.values() to a Semaphore in slotsByOrigin
168         *
169         * @param origin
170         * @return
171         */
172        private Semaphore getSlotsByOrigin(VOrigin origin) {
173
174            VOrigin[] vOrigins = VOrigin.values();
175
176            return slotsByOrigin[java.util.Arrays.binarySearch(vOrigins, origin)];
177        }
178
179        /**
180         * Removes all available slots for other waves (not really used at the moment as
        available slots for current wave
181         * are already 0 but could be useful in the future)
182         *
183         * Adds autosPerWave amount of slots to the current wave
184         *
185         * @param wave
186         */
187        private void setWave(VOrigin wave) {
188
189            currentWaveOrigin = wave;
190            VOrigin[] nonCurrentOrigins = getNonCurrentOrigins(currentWaveOrigin);
191
192            for (int i = 0; i < nonCurrentOrigins.length; i++) {
193                getSlotsByOrigin(nonCurrentOrigins[i]).drainPermits();
194            }
```

```
196             getSlotsByOrigin(currentWaveOrigin).release(autosPerWave);
197         }
199         /**
200          * Used to make acquiring and releasing of weight easier. Weight of vehicle is
            multiplied by 100 and ceiled to
201          * make a precise enough integer way of measuring weight
202          * @param v Vehicle
203          * @return
204          */
205         private int getWeightFromVehicle(Vehicle v) {
206             return (int) Math.ceil(v.getWeight() * 100);
207         }
209 }
```

d) Hier haben wir ganz simpel das Gewicht als ein Semaphor deklariert. Da aber die Semaphoren Integers sind, haben wir das Gewicht mit 100 multipliziert und dann für die Deklaration des Semaphors abgerundet und später bei recquire aufgerundet. So erreichen wir ziemlich nah die Tragbarkeit der Brücke. Wir haben als letzten Parameter zu dem Semaphor true übergeben, dass heißt eine Semaphor mit faire Warteschlange hergestellt. Deswegen müssen wir uns nicht mehr um Fairnisskeit kümmern, da es eine faire Warteschlange ist - sobald die Brücke mehr tragen kann, kommt das nächste Auto, dass seit längstens wartet.

Bei allen Tests haben wir die gewünschte Fairness und Effizienz erreicht, in dem letzten Test sogar viel drüber - Efficiency: 14168 Fairness: 7978

```
1  package control;
3  import vehicle.Vehicle;
4  import vehicle.Vehicle.VOrigin;
6  import java.util.concurrent.Semaphore;
8  public class FairBC implements BridgeControl {
10     double maxLoad;
11     Semaphore weight;
13     @Override
14     public void init(Double maxLoad) {
16         this.maxLoad = maxLoad;
17         weight = new Semaphore(transformWeight(maxLoad, false), true);
18     }
20     @Override
21     public void requestCrossing(Vehicle v) {
23         try {
24             weight.acquire(transformWeight(v.getWeight(), true));
25         } catch (InterruptedException e) {
26             e.printStackTrace();
27         }
28     }
30     @Override
31     public void leaveBridge(Vehicle v) {
33         weight.release(transformWeight(v.getWeight(), true));
34     }
```

```
36      private int transformWeight(double weight, boolean roundUp) {

38          double multipliedWeight = weight * 100;

40          if (roundUp) {
41              return (int) Math.ceil(multipliedWeight);
42          }

44          return (int) Math.floor(multipliedWeight);
45      }

47  }
```

e) Wir haben hier zu der vorigen Aufgaben noch ein lowriderHandlerLock mit Monitorkonzept eingefügt. Dieser sorgt dafür, dass beim requestCrossing und leaveBridge nur ein Thread unsere Logik auf einmal ausführen kann. Wenn es noch Lowriders unterwegs gibt und ihr Origin anders ist, dann wartet das Thread, bis es keine mehr gibt oder das LowriderDirection geändert wurde. Beim Verlassen der Brücke wird überprüft ob es noch Lowriders darauf gibt - falls nein, dann werden alle Threads, die auf dem Lock warten, notified.

```
1   package control;

3   import vehicle.Vehicle;
4   import vehicle.Vehicle.VOrigin;

6   import java.util.concurrent.Semaphore;

8   public class FairBCHandleLowriders implements BridgeControl {

10      static VOrigin lowridersOrigin;
11      static int lowridersInDirection;

13      double maxLoad;
14      Semaphore weight;
15      Object lowriderHandlerLock;
16      Object lowriderReleaserLock;
17      Object changeLowriderDirectionLock;

19      @Override
20      public void init(Double maxLoad) {

22          lowridersInDirection = 0;
23          this.maxLoad = maxLoad;
24          weight = new Semaphore(transformWeight(maxLoad, false), true);
25          lowriderHandlerLock = new Object();
26      }

28      @Override
29      public void requestCrossing(Vehicle v) {

31          try {
32              weight.acquire(transformWeight(v.getWeight(), true));
33          } catch (InterruptedException e) {
34              e.printStackTrace();
35          }

37          if (v.isLowrider()) {

39              synchronized (lowriderHandlerLock) {

41                  try {
42                      if (lowridersOrigin == null) {
43                          lowridersOrigin = v.getOrigin();
44                      } else {
45                          while (lowridersOrigin != v.getOrigin() &&
    lowridersInDirection != 0) {
```

```
                            lowriderHandlerLock.wait();
                        }
                        lowridersOrigin = v.getOrigin();
                    }

                    lowridersInDirection++;

                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Override
    public void leaveBridge(Vehicle v) {

        weight.release(transformWeight(v.getWeight(), true));

        if (v.isLowrider()) {

            synchronized (lowriderHandlerLock) {

                lowridersInDirection--;

                if (lowridersInDirection == 0) {
                    lowriderHandlerLock.notifyAll();
                }
            }

        }
    }

    private int transformWeight(double weight, boolean roundUp) {

        double multipliedWeight = weight * 100;

        if (roundUp) {
            return (int) Math.ceil(multipliedWeight);
        }

        return (int) Math.floor(multipliedWeight);
    }

}
```

f) Da gerade Zahlen langweilig sind, müssen Fahrzeuge mit geraden Nummern zwei mal so viel warten wie diese mit ungeraden. Dafür haben wir nach dem acquire wieder ein release gemacht und nochmal acquire, da es eigentlich nicht so leicht ist, eine Prioritätswarteschlange in dem Semaphore zu ingegrieren.

```
package control;

import vehicle.Vehicle;
import vehicle.Vehicle.VOrigin;

import java.util.concurrent.Semaphore;

/**
 * Simple example. Only light vehicles coming from west will be able to pass the
    Bridge.
 */
public class OddNumbersProBC implements BridgeControl {

    double maxLoad;
    Semaphore weight;
```

```
16      @Override
17      public void init(Double maxLoad) {
18
19          this.maxLoad = maxLoad;
20          weight = new Semaphore(transformWeight(maxLoad, false), true);
21      }
22
23      @Override
24      public void requestCrossing(Vehicle v) {
25
26
27          try {
28              weight.acquire(transformWeight(v.getWeight(), true));
29
30              // If the vehicle has an even id, it has to wait two times more
31              if (v.getVehicleId() % 2 == 0) {
32                  weight.release(transformWeight(v.getWeight(), true));
33                  weight.acquire(transformWeight(v.getWeight(), true));
34              }
35
36          } catch (InterruptedException e) {
37              e.printStackTrace();
38          }
39
40      }
41
42      @Override
43      public void leaveBridge(Vehicle v) {
44
45          weight.release(transformWeight(v.getWeight(), true));
46      }
47
48      private int transformWeight(double weight, boolean roundUp) {
49
50          double multipliedWeight = weight * 100;
51
52          if (roundUp) {
53              return (int) Math.ceil(multipliedWeight);
54          }
55
56          return (int) Math.floor(multipliedWeight);
57      }
58
59  }
```