ALP IV: Übung 6

Tutor: Christoph Beuck

Lisa Dohrmann (4130066), Adrian Neumann (4140810), Naja v. Schumde (4127652)

11. Juni 2008

Aufgabe 1

```
1  import java.util.*;
2
3  class RWlockImpl {
4      private LinkedList<Thread> LeseSperren; //viele Leser
5      private Thread Schreibsperre;  //ein Schreiber
6      synchronized void Rlock() throws LockErrorException,
           InterruptedException {
7          //der Aufrufer ist schon in der Liste
8          if (LeseSperren.contains(Thread.currentThread())) throw new
9              LockErrorException("Hast␣schon␣ne␣Sperre␣altah");
10         while(Schreibsperre!=null) { //solange es noch nen Schreiber gibt
11             LeseSperren.wait(); //warten
12         }
13         //Sperre greifen
14         LeseSperren.add(Thread.currentThread());
15         }
16     synchronized void Wlock() throws LockErrorException,
           InterruptedException{
17         //der Aufrufer schreibt schon
18         if (Schreibsperre.equals(Thread.currentThread())) throw new
19             LockErrorException("Hast␣schon␣ne␣Sperre␣altah");
20         while (Schreibsperre!=null && LeseSperren.size()!=0) {
21             //solange einer schreibt, oder einer liest: warten
22             Schreibsperre.wait();
23         }
24         //Sperre greifen
25         Schreibsperre=Thread.currentThread();
26     }
27     synchronized void unlock() throws LockErrorException {
28         //wir hatten eine schreibsperre:
29         if (Schreibsperre.equals(Thread.currentThread())) {
30             Schreibsperre=null;
31             //hierdurch werden schreiber bevorzugt
32             //leseleute könnten ja durchaus auch
33             //system ist unfair
34             if (LeseSperren.size()==0) {
35                 Schreibsperre.notifyAll();
36                 //wenn es niemanden gibt, der aufs Schreiben wartet, aber
37                 //jemanden, der aufs Lesen wartet würde das Signal
38                 //verloren gehen, deswegen sagen wir den Lesern
39                 //bescheid:
40                 LeseSperren.notifyAll()
41             }
42             else LeseSperren.notifyAll();
43             return;
44         }
45         //gibt ein Leser sein Lock auf, überprüfen wir, ob noch andere
46         //Leser da sind. falls nicht, sagen wir den Schreibern bescheid:
47         if (LeseSperren.remove(Thread.currentThread())) {
48             if (LeseSperren.size()==0)
```

```
49                  Schreibsperre.notifyAll();
50              return;
51          }
52          throw new LockErrorException("Gar kein Lock gesetzt!");
53      }
54  }
55
56  class LockErrorException extends RuntimeException {
57      LockErrorException(String s) {super(s);}
58  }
```

Bei `unlock()` werden die Schreiber bevorzugt, da sie zuerst benachrichtigt werden, wenn eine Schreibsperre freigeben wird und keine Lesesperren vorhanden sind. Theoretisch könnte dann ein armer Leser, der auch darauf wartet, dass die Schreibsperre freigeben wird, „verhungern“, da sich neue Schreiber immer vordrängeln könnten. Die Implementierung ist daher nicht fair. Würde man sie so abwandeln, dass an dieser Stelle zufällig gewählt wird, ob zuerst die Schreiber oder zuerst die Leser benachrichtigt werden, dann wäre sie zumindest schwach fair.

Aufgabe 2

```
1  package alpiv.zettel6;
2
3  public class BooleanSemaphore {
4     private boolean flag; // true = vergeben, false = frei;
5
6     BooleanSemaphore(int init) {
7         //init grade: schon vergeben, ungerade: noch frei
8        flag = (init % 2 == 0);
9     }
10
11    synchronized void P() throws InterruptedException {
12       while(flag) { // belegt -> warten
13          wait();
14       }
15       flag = true; // selber belegen
16    }
17
18    synchronized void V() {
19       if(!flag) // ist schon frei
20          throw new RuntimeException();
21       flag = false; // freigeben und alle informieren
22       notifyAll(); //eigentlich würde hier ein notify reichen
23                    //wenn einer wartet, nimmt er auf jeden fall die flagge
24                    //weil wir aber nacher (extended) die flagge manchmal
                         nicht nehmen
25    }
26 }
```

```
1  package alpiv.zettel6;
2
3  public class ExtendedSemaphore {
4
5     private int anzahl;
6     private BooleanSemaphore flag;
7      private BooleanSemaphore sync;
8
9     ExtendedSemaphore(int init) {
10       anzahl = init; // Anzahl der Tickets
11         // wir initialisieren ein Semaphor, das schon belegt ist,
12         // es ersetzt uns wait/notify
13       flag = new BooleanSemaphore(0);
14         //und noch ein freies, um synchronized zu ersetzen
15         sync = new BooleanSemaphore(1);
16    }
```

```
17
18      void P(int n) throws InterruptedException {
19            sync.P();
20          while(anzahl-n <= 0) // wir haben nicht genug Tickets, also warten
                wir
21             flag.P(); // ist schon belegt, also wait()
22            anzahl -= n;
23            sync.V();
24      }
25
26      void V(int n) { //weil += teilbar ist müssen wir auch synchronisieren
27            sync.P();
28          anzahl += n; //tickets zurückgeben
29          flag.V();
30            sync.V();
31      }
32
33  }
```