ALP IV: Übung 3

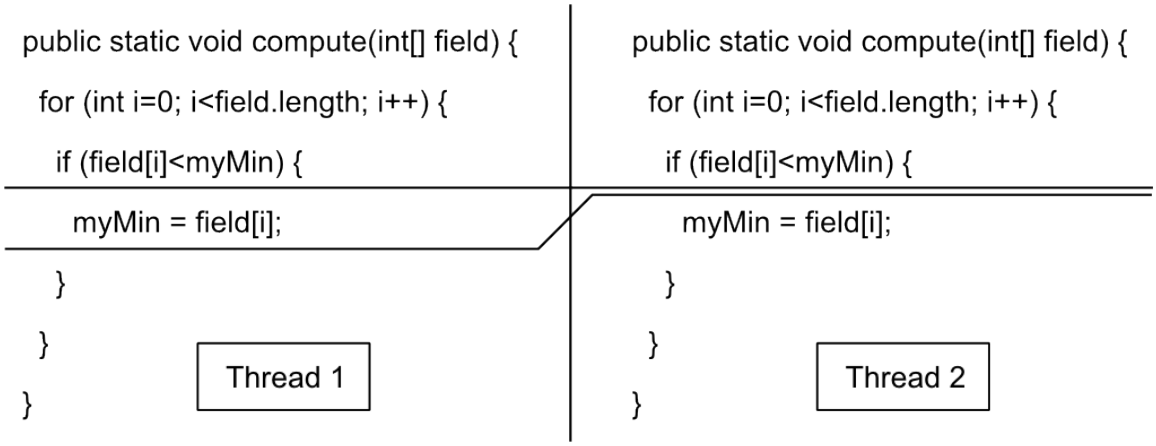
Tutor: Christoph Beuck

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

21. Mai 2008

Aufgabe 1

Bei nichtsequentieller Ausführung können Werte verloren gehen, sodass am Ende ein falsches Minimum berechnet wird.



Zum Beispiel kann myMin gerade 5 sein, Thread1 hat die if-Abfrage passiert und will das Minimum auf 2 setzen, Thread2 hat ebenfalls die if-Abfrage passiert und will es auf 3 setzten. Die 2 (und damit ein potentielles Minimum) geht verloren.
Um sowas zu verhindern, muss man dafür sorgen, dass jeweils nur ein Thread gerade im `if` sein kann. Dazu kann man einen synchronized-Block benutzen:

```
1 public static void compute(int[] field) {
2   for (int i=0; i<field.length; i++) {
3     synchronized("muh") {
4       if (field[i] < myMin) {
5         myMin = field[i];
6       }
7     }
8   }
9 }
```

So kann es nicht mehr dazu kommen, dass ein Thread sein `if` passiert, obwohl der andere Thread gerade dabei ist einen neuen Wert für myMin zu setzen.

Aufgabe 2

```
1 import java.util.*;
2
3 public class leSet {
4     static final String x="muh";
5     static final String y="kuh";
```

```
6       Node head;
7       leSet(Object obj) {
8           head = new Node(obj);
9       }
10      void add(Object obj) {
11          Node leNode =new Node(obj);
12          //wenn wir hier nicht synchronisieren, können elemente
13          //verloren gehen
14          synchronized(x) {
15              leNode.next=head;
16              head=leNode;
17          }
18      }
19      void remove(Object obj) {
20          Node last=head;
21          Node thisOne=head;
22          String lock;
23          while (thisOne.next!=null) {
24           //andere Leute sollten nicht löschen, während wir grade löschen
25           synchronized(y) {
26           //hier andere add-Threads ausschließen, falls wir grade am Anfang
27           //der Liste sind. Während wir kucken soll nicht am head
                 rumgepfuscht
28           //werden
29           synchronized(x) {
30           lock = (thisOne==head || last==head) ? x : y;
31           }
32           synchronized(lock) {
33               if (thisOne.obj.equals(obj)) {
34                  last.next=thisOne.next;
35               }
36               last=thisOne;
37               thisOne=thisOne.next;
38           }
39           }
40          }
41      }
42      //wird immer nur von einem benutzt
43      Iterator iterator() {
44          return new leSetIterator(head);
45      }
46  }
47  class leSetIterator implements Iterator {
48      Node current;
49      public leSetIterator(Node c) {current = c;}
50      public Node next() {
51          current=current.next;
52          return current;
53      }
54      public boolean hasNext() {
55          return (current.next!=null);
56      }
57      public void remove() {}
58  }
59  class Node {
60      Object obj;
61      Node next;
62      Node(Object obj) {
63          this.obj=obj;
64          this.next=null;
65      }
66  }
```

Aufgabe 3

Möglichkeiten des Zwei-Phasen-Sperrens:

1. Konservativ und strikt: Sperren am Anfang anfordern und am Schluss freigeben.

```
class Account {
    long balance;

    public long balance() {
      READING(this) {
        return balance;
      }
    }
    public void deposit(long amount) {
      WRITING(this) {
        balance += amount;
      }
    }
    public void transfer(long amount, Account dest) {
      WRITING(this) {
        WRITING(dest) {
             balance -= amount;
              dest.balance += amount;
        }
      }
    }
    public static long total(Customer cust) {
        // sum of balances of customers checking
        // and savings accounts
        READING("check") {
          return cust.checkings.balance + cust.savings.balance;
        }
    }
}
```

2. Konservativ: Sperren am Anfang anfordern und baldmöglichst freigeben.

```
class Account {
    long balance;

    public long balance() {
      READING(this) {
        long temp;
        temp = balance;
      }
      return temp;
    }
    public void deposit(long amount) {
      WRITING(this) {
        balance += amount;
      }
    }
    public void transfer(long amount, Account dest) {
      WRITING(dest) {
        WRITING(this) {
            balance -= amount;
          }
          dest.balance += amount;
        }
    }
    public static long total(Customer cust) {
        // sum of balances of customers checking
        // and savings accounts
        READING("check") {
          READING("save") {
            int sum;
            sum += cust.checkings.balance;
          }
          sum += cust.savings.balance;
        }
        return sum;
```

```
    }
}
```

3. Strikt: Sperren spätmöglichst anfordern und am Schluss freigeben. Bei `deposit()` käme es mit den Deifinitionen mit `READING()` und `WRITING()` aus der Vorlesung zu einem Deadlock, deswegen ändern wir die Semantik so, dass durch die Verschachtelung die Sperre von lesend zu schreibend erweitert wird.

```
class Account {
    long balance;

    public long balance() {
      READING(this) {
          return balance;
        }
    }
    public void deposit(long amount) {
      long temp;
      READING(this) {
        temp = balance + amount;
        WRITING(this) {
            balance = temp;
          }
        }
    }
    public void transfer(long amount, Account dest) {
      WRITING(this) {
          balance -= amount;
          WRITING(dest) {
            dest.balance += amount;
          }
        }
    }
    public static long total(Customer cust) {
        // sum of balances of customers checking
        // and savings accounts
        int sum;
        READING("check") {
          sum += cust.checkings.balance;
          READING("save") {
            sum += cust.savings.balance;
          }
        }
      return sum;
    }
}
```

4. Effizient: Sperren spätmöglichst anfordern und baldmöglichst freigeben.

```
class Account {
    long balance;

    public long balance() {
      long temp;
      READING(this) {
        temp = balance;
      }
      return balance;
    }
    public void deposit(long amount) {
      long temp;
      READING(this) {
        temp = balance + amount;
      }
      WRITING(this) {
         balance = temp;
```

```
18        }
19      }
20      public void transfer(long amount, Account dest) {
21          WRITING(this) {
22              balance -= amount;
23          }
24          WRITING(dest) {
25            dest.balance += amount;
26          }
27      }
28      public static long total(Customer cust) {
29          // sum of balances of customers checking
30          // and savings accounts
31          int sum;
32          READING("check") {
33             sum += cust.checkings.balance;
34          }
35          READING("save") {
36             sum += cust.savings.balance;
37          }
38          return sum;
39      }
40  }
```

Aufgabe 4

Die erste Version arbeitet mit Delegation. Mit einem Objektattribut vom Typ ArrayList lassen sich die Aufgaben an die ArrayList Klasse weitergeben:

```
1  public class MyArrayList1<E> {
2    public ArrayList<E> list;
3
4    public MyArrayList1 () {
5      list = new ArrayList<E>();
6    }
7    synchronized boolean add (E o) {
8      return list.add(o);
9    }
10   synchronized boolean remove (Object o) {
11     return list.remove(o);
12   }
13   synchronized E get (int index) {
14     return list.get(index);
15   }
16 }
```

Die zweite Version erbt von der ArrayList Klasse, sodass man deren Methoden durch das Schlüsselwort `super` aufrufen kann.

```
1  public class MyArrayList2<E> extends ArrayList<E> {
2    public MyArrayList2 () {
3      super();
4    }
5    public synchronized boolean add (E o) {
6      return super.add(o);
7    }
8    public synchronized boolean remove (Object o) {
9      return super.remove(o);
10   }
11   public synchronized E get (int index) {
12     return super.get(index);
13   }
14 }
```