ALP IV: Übung 2

Tutor: Christoph Beuck

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

12. Mai 2008

Aufgabe 1

In unserem Programm haben wir drei arbeitende Runnables. Das erste ist für das Stellen der Zeiger zuständig, das zweite kümmert sich um das Läuten und das dritte wartet auf Eingaben von stdin. Alle drei greifen auf ein Uhr-Objekt zu, dass die Turmuhr kapselt. Das Uhr Objekt stellt drei Methoden zur Verfügung `setTime(int,int,int)` (wie bei Turmuhr), `incTime()` (Zeit um `incAmount` erhöhen) und `klingeln(boolean,int)` (true = Highbell, int = Anzahl der Schläge). Über Synchronisation wird dafür gesorgt, dass immer nur ein Thread gleichzeitig die Zeit setzt und nur ein Thread gleichzeitig läutet. Man kann aber gleichzeitig läuten und die Zeit verändern (wir benutzen zwei Objekte für das synchronized).
`incTime()` überprüft, ob geläutet werden muss und startet bei Bedarf ein `Bell`-Objekt.
In main machen wir

```
1  LeUhr leUhr= new LeUhr(uhr); //kapselungsklasse
2  Thread zeiger =new Thread(new Zeiger(leUhr)); //zeiger bewegen
3  Thread stdInSet = new Thread(new Setter(leUhr)); //auf stdin warten
4  stdInSet.start();
5  zeiger.start();
6  zeiger.join(); //ewig warten
```

Die Kapselungsklasse ist ziemlich lang, deswegen zeigen wir hier nur Ausschnitte:

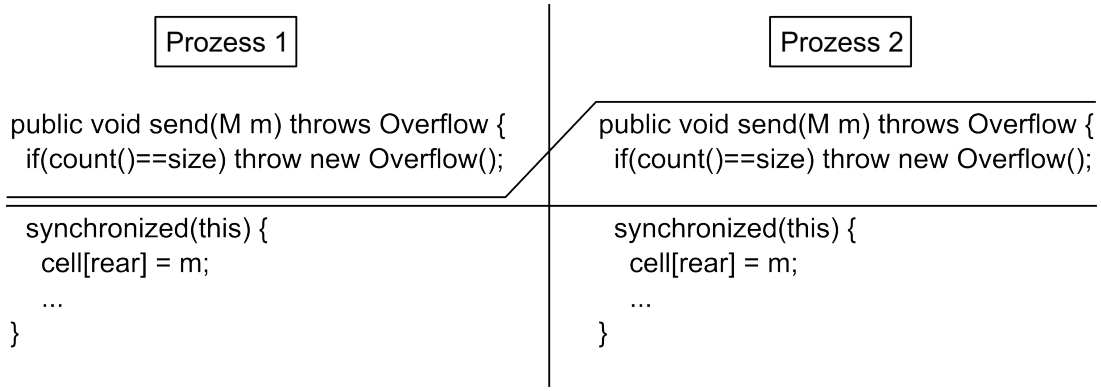
```
1  class LeUhr {
2      private volatile int stunde, minute, sekunde;
3      private final String zeit="muh"; //für synchronized
4      private final incAmount = 1;
5      private Turmuhr uhr;
6  /*...*/
7      //zeit setzen darf immer nur einer
8      public void setTime(int s, int m, int se) {
9         synchronized(zeit) {
10             stunde = s;
11             minute = m;
12             sekunde = se;
13             uhr.setTime(s,m,se);
14         }
15     }
16     public void incTime() {
17         synchronized(zeit) {
18             sekunde+=incAmount; //Sekundenänderungen propagieren sich hoch
19             if (sekunde==60) {
20                 sekunde=0;
21                 minute++;}
22             if (minute==60) { /*...*/
23
24             uhr.setTime(stunde,minute,sekunde);
25             if (sekunde==0) { //wir haben grade eine volle Minute
26             switch (minute) { //womöglich müssen wir läuten
27                               //in neuem Thread
28                 case 15: (new Thread(new Bell(this, 1,0))).start(); break;
```

```
29 /*...*/
30                     case 0: (new Thread(new Bell(this, 4, stunde))).start();
                           break;
31                 }
32                 }
33             }
34     }
35     //läuten darf immer nur einer
36     public synchronized void klingeln(boolean high, int c) {
37         for (int i=0; i<c; i++)
38             if (high) highBell(); else lowBell();
39     }
40     void highBell() {/*...*/} //klingeln und warten
41     void lowBell() {/*...*/}
42 }
```

Die Zeiger-Klasse ruft in einer Endlosschleife `incTime()` auf und legt sich dann eine Sekunde lang schlafen. Die Bell-Klasse ruft einfach `klingeln` auf, das zweite Argument ist die Anzahl hoher Schläge, das dritte Argument die Anzahl tiefer Schläge.

Aufgabe 2

a) Vor dem synchronized Block zu testen, ob der Buffer bereits voll ist, kann leicht zum Datenverlust führen.
Angenommen der Buffer ist bis auf einen Speicherplatz voll und es werden zwei Messages von zwei parallel arbeitenden Prozessen versendet. Beide Prozesse können in beliebiger Reihenfolge die if-Abfrage passieren, da ja noch ein ein Platz frei ist. Erst dann tritt Prozess 1 in den synchronized Block ein und speichert die Message im Buffer. Nun gibt er den Block wieder frei und Prozess 2 kann ihn betreten, obwohl der Buffer inzwischen voll ist und eigentlich eine Exception ausgelöst werden sollte. Prozess 2 überschreibt nun das erste Element im Buffer.



b) Schon der Compiler wird bei diesem Code einen Fehler melden, da die Variable `m` außerhalb des synchronized Blocks ihre Gültigkeit verloren hat und daher unbekannt ist. Dieser Fehler kann in der Tat behoben werden, indem `M m;` vor das synchronized geschrieben wird.

Wenn aber dann mehrere Receive-Threads gleichzeitig etwas lesen wollen, lesen sie alle das selbe `cell[front]`, anstatt nacheinander die Elemente aus dem Buffer zu nehmen.

c) Bei dieser Methode muss sowohl ein paralleler Empfänger als auch ein paralleler Sender ausgeschlossen werden. Der Empfänger könnte `front` verschieben, bevor die Message von `urgent()` gespeichert wurde und der Sender könnte das dringende Element direkt wieder überschreiben, falls im Buffer nur noch ein Platz frei war und er schon die if-Abfrage passiert hat, bevor `urgent()` die Message in den letzten freien Platz schreiben konnte.

```
1 public void urgent(M m) throws Overflow {
```

```
2     synchronized(cell) { // receiver exclusion
3       synchronized(this) { // sender exclusion
4         if(count()==size) throw new Overflow();
5         front = (front-1)%(size+1);
6         cell[front] = m;
7       }
8     }
9   }
```

Aufgabe 3

Da `myColors` ein Array ist, wird die Deklaration als `volatile` nicht auf die Feldelemente angewendet (das geht in Java auch nicht) sondern nur auf die Adresse des Arrays. Die Feldelemente werden also nicht direkt zurückgeschrieben und ein anderer Prozess arbeitet daraufhin eventuell mit veralteten Werten.
Dadurch kann es zu verschiedenen Effekten kommen. Zum einen gibt es die offensichtlichen Probleme, wenn ein Thread nachkuckt, ob eine Farbe im Array ist und einen veralteten Wert liest. Interessanter ist aber die Tatsache, dass die Reihenfolge des Zurückschreibens nicht definiert ist. Es können also alte Veränderungen neuere im Nachhinein überschreiben, so dass die neuen Werte effektiv verloren gehen.