ALP4 SoSe 2013, Di. 16-18

Lösung Übungsblatt 4

Christoph van Heteren-Frese (Matr.-Nr.: 4465677),
Sven Wildermann (Matr.-Nr.: 4567553)
Tutor: Alexander Steen, eingereicht am 10. Mai 2013

Aufgabe 1

Das (2te) Leser-Schreiber-Problem wurde in [1] auf den Seiten 85 bis 91 ausführlich besprochen und mit universellen kritischen Abschnitten gelöst. Die hier gemachten Anmerkungen zu einer faireren Lösung beziehen sich auf diesen Lösungsanssatz und die damit verbundene Implmentierung.

Für eine fairere Variante des gegebenen Alogrithmus müssen lediglich die Protokolle der Prozessklassen angepasst werden. Mit den Bedingungen

$$nR == 0 \;\&\&\; (bW == 0 \;||\; !zR) \tag{1}$$

$$nW == 0 \;\&\&\; (bW == 0 \;||\; !zR) \tag{2}$$

findet ein Wechsel zwischen Lesern und Schreibern genau dann statt, wenn jeweils andere warten. Dabei stellt zR eine boolsche Variable dar, die protokolliert, ob Leser oder Schreiber zuletzt im kritischenn Abschnitt waren. Sie wird beim Eintritt von Lesern gesetzt und von Schreibern gelöscht. Die Variablen nR bzw. nW stehen für die Anzahl der Leser bzw. Schhreiber, bR bzw. bW für die Anzahl der blockierten Leser bzw Schreiber [vgl. 1, S. 84].

Somit muss sowohl die Funktion c, die die Synchronisationstabelle übersetzt, als auch die Funktionen i wie folgt angepasst werden:

```
1  func  c(k uint) bool {
2    if k == r {
3      return nW = 0 && (bW == 0 || !zR
4    }
5    return nR == 0 && nW == 0 && (bR == 0 || zR
6  }
7
8  func i(x Any, k uint) {
9    if k == r {
10     nR++
11     zR = true
12   } else {
13     nW = 1
14     nR = false
15   }
16 }
```

Aufgabe 2

Annahme: Jeder der Philosophen hat nur endlich lange Hunger. Grundlage der Lösung ist die auf Seite 89 im Buch dargestellte Struktur für universelle kritische Abschnitte. Jeder der 5 Philosophen wird durch eine Prozessklasse p_i in Form eines unsigned int repräsentiert (im Uhrzeigersinn von 1 bis 5 durchnumeriert). Der Konstruktor der Struktur **type imp struct** erhält somit für die Anzahl der Prozessklassen den Parameter $nK = 5$. Weiterhin wird ein Array der Größe 5 von booleschen Werten $nP[5]$ benötigt, wobei $nP[k]$ angibt ob der Philosoph der zur Prozessklasse k gehört gerade isst, oder nicht. Das Spektrum der Eintrittsbedinungen **c(k uint)** gibt für den Philosophen der Prozessklasse k genau dann true zurück, wenn die Philosophen rechts und links von ihm (also Prozessklasse $k+1 \mod 5$ und $k-1 \mod 5$) sich nicht im kritischen k A befindet (also nicht essen). **in(x Any, k uint)** setzt nP[k] = true und **out(x Any, k uint)** setzt nP[k] = false. Die Funktionen **Enter()**, **Leave()**, **vall()** und **Blocked()** bleiben wie im Buch Seite 90 implementiert.

```
1  const (p_i = i ; nK = 5)
2  var (nP[5] bool; x *Imp)
3
4  func c(k uint) bool{
5    return nP[k+1 mod 5] == false && nP[k-1 mod 5]
6  }
7
8  func in(x Any, k uint){
9    nP[k] = true
10 }
11
12 func out(x Any, k uint){
13   nP[k] = false
14 }
15 func (x *ImpCS) PhilosophEat() { x.Enter(p_i, nil) }
16
17 func main() { x = New(nK, c, in, out) }
```

Listing 1: Problem der speisenden Philosophen mit universellen kritischen Abschnitten in Go.

Aufgabe 3

Für jeden Philosophen wird ein 2-faches Semaphor verwendet, wobei der erste Wert des Tupels für seine linke Gabel und der zweite Wert für seine rechte Gabel steht. Da sich je 2 Philosophen eine Gabel teilen, teilen sie sich auch die entsprechenden Variablen innerhalb der Semaphore. Nach Definition wird der kritische Abschnitt von einem Philosophen betreten, sobald sowohl seine linke als auch seine rechte Gabel verfügbar ist (und er eintrittswillig ist). Daher gilt: $S.P :< await : linke{-}Gabel{-}und{-}rechte{-}Gabel{-}frei >$ und $S.V :< linkeGabel.freigeben; rechteGabel.freigeben >$

Aufgabe 4

a)

Ein Semaphor benötigt als Datenstrukturen einen Zähler vom Typ Integer, der mit einer nicht-negativen Zahl initialisiert wird und eine Warteschalange, die leer initialisiert wird.

```
1  int count;
2  BlockingQueue<Thread> queue;
3
4  public Semaphor(int initialcount){
5    count = initialcount;
6    queue = new LinkedBlockingQueue<Thread>();
7  }
```

Der Zählerstand gibt gib an, wie viele Prozesse noch den kritischen Abschnitt betreten dürfen, bevor eine Blockade eintritt. Die Warteschlange nimmt die blockierten Prozesse auf.

P-Operation

Die P-Operation dekrementiert den Zähler und blockiert den aufrufenden Prozess genau dann, falls das durch den Zähler repräsentierte Potential erschöpft ist. Dabei wird der Prozess in die Warteschalage der wartenden Prozesse eingereiht.

```
1  public void P(){
2    synchronized(this){
3      if(count==0){
4        try{
5          queue.put(Thread.currentThread());
6          wait();
7        }catch(InterruptedException ie){
8          System.err.println("caught InterruptedException in wait()");
9        }
10     }
11     count--;
12   }
13 }
```

V-Operation

Die V-Operation inkrementiert den Zähler und gibt einen ggf. blokierten Prozess aus der Warteschalnge frei, falls wieder Potential dazu vorhanden ist.

```
1  public void V(){
2    synchronized(this){
3      count++;
4      if(count==1) {
5        queue.poll().notify();
6      }
7    }
8  }
```

Erläuterung

Durch die Verwendung einer FIFO-Warteschlange wird das Deblockieren der Prozesse in der zeitlichen Reihenfolge, in der der Zugang zum kritischen Abschnitt beantragt wurde, umgesetzt. Darüber hinaus wird gewährleistet, das jeder blockierte Prozess irgendwann auch wieder deblockiert wird.

Es ist wichtig, dass die logische Unteilbarkeit der Listenmanipulationen sichergestellt wird. Andernfalls könnte u.a. die Integrität der Datenstrukturen korrumpiert werden, wenn nebenläufige Prozesse auf sie zurückgreifen. Dazu sollten die P- und V-Operationen mit geeigneten Methoden für exclusiven Zugriff gekapselt werden. Das in [1] erläuterte Prozessmodel unterscheidet fünf Zustände: *nicht existent*, *bereit*, *aktiv*, *blockiert* und *beendet*. Ein Semaphor steuert die Übergänge *aktiv* → *blockiert* und *blockiert* → *bereit*. Der erste Übergang kann durch den Aufruf einer P-Operation ausgelöst werden, der zweite durch eine V-Operation.

b)

Das Konvoi-Problem kann dadurch umgangen werden, dass bei einer V-Operation nicht der älteste Prozess in die Bereitliste übernommen wird, sondern ein zufällig gewählter.

```
1  public void V(){
2    synchronized(this){
3      count++;
4      if(count==1) {
5        Thread[] prozesse;
6        prozesse = (Thread[]) queue.toArray();
7        int i = (int) (Math.random()*prozesse.length+1);
8        prozesse[i].notify();
9      }
10   }
11 }
```

Weitere Möglichkeiten sind in [1, S. 106] geschildert.

c)

Semaphore können in Java wie folgt implementiert werden:

```
1  import java.util.concurrent.BlockingQueue;
2  import java.util.concurrent.LinkedBlockingQueue;
3
4  public class Semaphor {
5    int count;
6    BlockingQueue<Thread> queue;
7
8    public Semaphor(int initialcount){
9      count = initialcount;
10     queue = new LinkedBlockingQueue<Thread>();
11   }
12
13   public void P(){
14     synchronized(this){
15       if(count==0){
16         try{
17           queue.put(Thread.currentThread());
```

```
18            wait();
19          }catch(InterruptedException ie){
20            //this should never happen
21            System.err.println("caught InterruptedException in wait()");
22          }
23        }
24        count--;
25      }
26    }
27
28    public void V(){
29      synchronized(this){
30        count++;
31        if(count==1) {
32          queue.poll().notify();
33        }
34      }
35    }
36  }
```

Listing 2: Implemntierung eines Semaphor in Java.

Anmerkung

Einige der hier zur Synchronisation verwendeten sprachlichen Mittel nutzen intern das Monitorkonzept. Es scheint vieleicht etwas seltsam Semaphore auf diese Art zu implementieren; die Funktionalität und Klarheit stand hier aber im Vordergrund.

Literatur

[1] Christian Maurer. *Nichtsequentille Programmierung mit Go 1 Kompakt.* Springer Vieweg, 2012. ISBN 978-3642299681.