ALP4 SoSe 2013, Di. 16-18

Lösung Übungsblatt 8

Christoph van Heteren-Frese (Matr.-Nr.: 4465677)
Sven Wildermann (Matr.-Nr.: 4567553)
Tutor: Alexander Steen, eingereicht am 14. Juni 2013

Aufgabe 1

a)

Der `RWMutex` (Read-Write-Mutex) ist das Standardinstrument, mit dem mehren Prozessen das gleichzeitige Lesen einer Ressource gestattet werden kann, während der schreibende Zugriff nur einem Prozess exklusiv möglich ist. [vgl. 1, S. 185]. Will ein Prozess den Mutex für den Schreibzugriff nutzen, obwohl gerade andere Prozesse lesen, wird dieser blockiert [vgl. ebd.]. Es werden dafür vier Zugriffsfunktionen definiert: `RLock()`, `RUnlock()`, `Lock()` und `Unlock()`. Grundlage der Erläuterung ist folgendes kleines Beispiel:

```
1  package main
2
3  import (
4    "fmt"; "sync"; "time"; "runtime"
5  )
6
7  var (
8    rwm     sync.RWMutex
9    balance int
10 )
11
12 // "reads" the current balance and prints it
13 // to the sdtout
14 func get(n int, balance *int) {
15   rwm.RLock()
16   fmt.Println("Reader",n,*balance)
17   rwm.RUnlock()
18 }
19
20 // "writes" the current balance: increases the
21 // balance by the given amount
22 func put(n int,balance *int, amount int) {
23   rwm.Lock()
24   *balance += amount
25   fmt.Println("Writer",n,"added",amount)
26   rwm.Unlock()
27 }
28
29 func main() {
30   balance = 0     // set initial value
31   runtime.GOMAXPROCS(5)
32   go get(1,&balance)    // start thread 1 of 5...
33   go put(1,&balance, 100)
34   go get(2,&balance)
35   go put(2,&balance, 100)
36   go get(3,&balance)
```

```
37    time.Sleep(10e5)    // wait until threads are supposedly done
38    fmt.Println("balance:",balance) // print balance
39  }
```

Erläuterung: Das Reader-Writer-Mutex `rwm` ist für den gegenseitigen Ausschluss von lesenden und schreibenden Prozessen (im Sinne seiner oben genannten Funktion) zuständig. Die Funktion `get` 'verschließt' `rwm` zunächst mittels `RLock()`, so dass keine 'Schreibzugriff' mehr möglich ist. Nachdem `balance` gelesen und ausgegeben wurde, wird das Schloss mit `RUnlock()` wieder geöffnet.

Die Funktion `put` arbeitet nach dem gleichen Prinzip. Statt `RLock()` und `RUnlock()` kommt hier aber `Lock()` und `Unlock()` zum Einsatz, um andere Prozesse auch den lesenden Zugriff zu verweigern.

b)

Durch die oben genannte Struktur und die erläuterten Funktionen `RLock()`, `RUnlock()`, `Lock()` bzw. `Unlock()` ergibt sich unmittelbar, dass die Invariante eingehalten wird: Wenn ein Prozess als 'Schreiber' auf die Resource `balance` mittels `put` zuzugreifen versucht, das Schloss `rwm` aber bereits durch `RLock()` oder `Lock()` verschlossen ist, wird er blockiert. Will ein 'Leser' auf die Ressource zugreifen während das Schloss verschlossen ist, gelingt ihm das nur, wenn `rwm` mittels `RLock()` verschlossen wurde, also ein 'Leser' gerade auf die Ressource zugreift. Andernfalls (ein Schreiber hat das Schloss mittels `Lock()` versperrt) wird er blockiert.

c)

Es wird das erste 'Leser-Schreiber-Problem' gelöst. **begründung:** Es können immer wieder neue 'Leser' auf die Ressource zugreifen, unabhängig davon ob bereits 'Schreiber' warten, oder nicht.

Aufgabe 2

a)

Modellierung siehe Aufgabenteil b). Begründung der Korrektheit: Für jedes Gleis an einem Bahnhof gibt es als Semaphor so genannte “Wächter”. Diese geben inital die Gleise frei. Sobald nun ein Zug das Gleis befahren möchte, wird ein Wächter für diesen Vorgang reserviert und das Gleis von diesem Wächter im Anschluss gesperrt. Nachdem der Zug dann durchgefahren ist, wird erst das Gleis durch den Wächter wieder freigegeben und anschließend der Wächter wieder für andere Vorgänge verfügbar. Die Zugriffe auf die Variablen sind durch mutexe geschützt.

b)

```
/* bahnhof.go */
package main

import (
  ."sync"
  "fmt"
  "time"
)

type Imp struct {
  free []int
  val int
  cs, mutex Mutex
}

type Bahnhof struct {
  guard *Imp
  gleis []Mutex
}

func NewImp(n int) *Imp {
  x := new(Imp)
  x.val = n
  x.free = make([]int, n)
  for i := 0; i < n; i++ {
    x.free[i] = i
  }
  if n == 0 {
    x.cs.Lock()
  }
  return x
}

func (x *Imp) P() (i int) {
  x.cs.Lock()
  x.mutex.Lock()
  x.val--
  if x.val > 0 {
    x.cs.Unlock()
  }
  i = x.free[i]
  x.free = x.free[1:]
  x.mutex.Unlock()
  return i
}

func (x *Imp) V(i int) {
  x.mutex.Lock()
  x.val++
  if x.val == 1 {
    x.cs.Unlock()
  }
  x.free = append(x.free, i)
  x.mutex.Unlock()
}

func NewBahnhof(n int) *Bahnhof {
  b := new(Bahnhof)
  b.guard = NewImp(n)
  b.gleis = make([]Mutex, n)
  return b
}
```

```
63
64 func (b *Bahnhof) durchfahren(e chan bool) {
65   i := b.guard.P()
66   b.gleis[i].Lock()
67   fmt.Printf("ich fahre hier auf Gleis %d!\n", i)
68   fmt.Printf("Ich lasse mir jetzt 10 Sekunden Zeit..\n")
69   x := time.Duration(10)
70   time.Sleep(x * time.Second)
71   b.gleis[i].Unlock()
72   b.guard.V(i)
73   e <- true
74 }
75
76 func main() {
77   done1 := make(chan bool, 1)
78   done2 := make(chan bool, 1)
79   done3 := make(chan bool, 1)
80   done4 := make(chan bool, 1)
81   b := NewBahnhof(2)
82   go b.durchfahren(done1)
83   go b.durchfahren(done2)
84   go b.durchfahren(done3)
85   go b.durchfahren(done4)
86   <- done1
87   <- done2
88   <- done3
89   <- done4
90 }
```

c)

Ja, bei der Implementierung kann es zu Kollisionen zweier Züge an zusammenführenden Weichen kommen, da keine Weichen sondern nur vollständig getrennte Gleise gesichert werden. Durch weitere Anpassungen (zusätzliche Wächter für Weichen) könnte dies jedoch verhindert werden.

Aufgabe 3

Algorithmus in Pseudocode.

```
1   // Variabeldeklarationen
2   type objects struct {
3       Enum groesse = {gross, mittel, klein}
4       bool richtung
5       time timestamp_incoming}
6   mainqueue (queue)objects
7   leftqueue (queue)objects
8   rightqueue (queue)objects
9   islocked int
10
11  // Algorithmus
12  func add_object(new objects){
13       // 0 = von links nach rechts
14       // 1 = von rechts nach links
15      if new.richtung=0{
16       leftqueue.add(new)
17       }else{
18       rightqueue.add(new)
19       }
```

```
20        if mainqueue.isEmpty{
21         // wenn es das erste Element ist,
22         // wird die Aktualisierung angestossen
23               mainqueue.add(new)
24               objekte_aktualisieren()
25        }else{
26        //sonst nicht
27               mainqueue.add(new)
28        }
29    }
30   func objekte_aktualisieren(){
31        current objects;
32        // sortiert die Elemente nach Einfuegezeit
33        mainqueue.sort(timestamp)
34        leftqueue.sort(timestamp)
35        rightqueue.sort(timestamp)
36        //ueberpruefen ob der Weg frei ist und ob ein
37        // Objekt den Weg passieren will
38        if islocked == 0 && (not mainqueue.isEmpty){
39        current = mainqueue.get
40        if current.richtung==0{
41        // pruefe nach der Reihe ob es in lefftqueue
42        // weitere Elemente gibt, die mit current
43        // auf den Weg gehen koennen.
44        // Sende diese inkl. current los und loesche Sie aus
45        // den queues, erhoehe jeweils islocked um die Anzahl
46        // der losgeschickten Elemente
47        }else if current.richtung==1{
48        // pruefe nach der Reihe ob es in rightqueue
49        // weitere Elemente gibt, die mit current
50        // auf den Weg gehen koennen.
51        // Sende diese inkl. current los und loesche Sie aus
52        // den queues, erhoehe jeweils islocked um die Anzahl
53        // der losgeschickten Elemente
54             }
55
56
57        }
58
59   }
60
61   func objekt_ist_angekommen(){
62        // wird vom Objekt aufgerufen
63        // sobald ein Objekt angekommen ist
64        islocked.lock();
65        islocked--;
66        islocked.unlock();
67        objekte_aktualisieren();
68   }
```

Dadurch dass die Queues nach der Einfügezeit der Objekte sortiert werden führt auch ein nebenläufiges einfügen zur korrekten Reihenfolge der Elemente in der Queue. Sobald entweder ein Objekt angekommen ist oder ein Objekt in die leere haupt-queue eingefügt wird, wird “objekte-aktualisieren” ausgeführt. Diese Funktion überprüft erst, welches das am längsten wartende Objekt ist und dann welche Elemente aus der Queue der selben Richtung mit diesem Element möglicherweise zusammen losgeschickt werden könnten. Diese werden dann alle zusammen losgelassen (Barriere wird geöffnet). Um einen Überblick über die Anzahl der zur Zeit reisenden Objekte zu erhalten, wird der Counter islocked um den entsprechenden Wert erhöht. Sobald die einzelnen Elemente am

Ziel angekommen sind, führen sie objekt-ist-angekommen() aus, der den Counter schrittweise wieder verringert. So wird in der Funktion objekte-aktualisieren sicher gestellt, dass keine neuen Objekte frei gelassen werden wenn noch nicht alle Objekte wieder angekommen sind.

Aufgabe 4

Die Implementierung in GO compiliert leider nicht erfolgreich. In Anbetracht einer pünktlichen Abgabe habe ich es nicht mehr geschafft, dies noch zu fixen.

```
1  package main
2  import "fmt"
3
4  /* Leider habe ich es nicht puenktlich
5  geschafft den Code erfolgreich zum compilieren zu bringen*/
6
7  func mult(matrix1 [][]int, matrix2 [][]int, e chan bool){
8
9      newMa := [][]int{{0,0,0},{0,0,0},{0,0,0}}
10     for i := 0 ; i < len(matrix1) ;i++{
11   w :=  make (chan bool)
12         for j := 0 ; j < len(matrix2[0]) ;j++ {
13             go rowCol(i,j,matrix1,matrix2,newMa,w)
14       <-w
15   }
16
17     }
18     fmt.Println(newMa)
19     e<-true
20 }
21
22 func rowCol( r int , c int, matrix1 [][]int, matrix2 [][]int, newMa [][]int,  e chan bool){
23     var sum int = 0
24     for i := 0 ; i < len(matrix1) ; i++ {
25         sum += matrix1 [r][i] * matrix1 [i][c]
26     }
27      newMa[r][c] = sum
28     e<-true
29
30 }
31
32 func main(){
33     matrix1 := [][]int { {1,2,3} ,
34                          {4,5,6} ,
35                          {7,8,9} }
36     matrix2 := [][]int {{0,0,1} ,
37                          {0,1,0} ,
38                          {1,0,0} }
39
40  q := make (chan bool)
41
42  mult(matrix1 , matrix2, q)
43
44  <-q
45 }
```

Literatur

[1] Rainer Feike and Steffen Blass. *Programmierung in Google Go: Einstieg, Beispiele und professionelle Anwendung.* Addison-Wesley Verlag; Auflage: 1, 2010. ISBN 3827330092. URL `http://www.amazon.de/Programmierung-Google-Beispiele-professionelle-Anwendung/dp/3827330092`.