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

Aufgabe 3

Aufgabe 4

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`.