ALP4 SoSe 2013, Di. 16-18

Lösung Übungsblatt 6

Christoph van Heteren-Frese (Matr.-Nr.: 4465677)
Sven Wildermann (Matr.-Nr.: 4567553)
Tutor: Alexander Steen, eingereicht am 31. Mai 2013

Aufgabe 1

Gleich vorweg: Das Beispielprogramm

```
1  package main
2
3  import (
4    "fmt"
5    "test/sem" // Implementierung von S. 171
6  )
7
8  func main() {
9    s := sem.New(0)
10   s.P()
11   fmt.Println("Hello World!\n")
12   s.V()
13 }
```

erzeugt folgende Ausgabe:

```
chris@swan-station ~/go/src/test/hello $ vim hello.go
chris@swan-station ~/go/src/test/hello $ go install
chris@swan-station ~/go/src/test/hello $ hello
throw: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
  /home/chris/go/src/test/hello/hello.go:10 +0x42

goroutine 2 [syscall]:
created by runtime.main
  /usr/local/go/src/pkg/runtime/proc.c:221
```

Dagegen führten verschieden Versuche mit `n > 0` zum Erfolg. Die Implementierung ist also für `n==0` entgegen der im Buch gemachten Behauptung scheinbar nicht korrekt!

Begründung: Gilt `n == 0`, wird mit `x.c = make(chan bool, n)` ein Kanal für *synchronen* Borschaftenaustausch realisiert [vgl. 1, S. 168]. Somit kann der Kanal keine Botschaften puffern: Sobald eine Botschaft gesendet wurde, ist der Sender so lange blockiert, bis ein anderer Prozess empfangsbereit ist und die Botschaft abgenommen hat [vgl. 1, S. 166]. Des Weiteren blockiert eine Empfangsanweisung (hier: `<-x.c` in Zeile 9) einen Prozess solange, bis eine Botschaft empfangen wurde [vgl. 2]. Für das zu betrachtende Beispiel bedeutet das konkret:

1. Nach der Initialisierung eines Semaphor ist der Kanal `c` zwar durch `make` initialisiert, entält aber keine Botschaft, da die for-Schleife in Zeile 6 nicht durchlaufen wurde.

2. Bei Aufruf der P-Operation wird eine Botaschaft empfangen, sofern der Kanal eine enthält. Andernfalls wird der aufrufende Prozess blockiert, bis eine Botschaft verfügbar ist. Letzteres ist beim erstmaligen Aufruf von P() der Fall. Da bei einem regulären Gebrauch der Semaphoroperationen (erst P(), dann V()) die 'erlösende' Botschaft, die zum Betreten des kritischen Bereiches nötig ist, erst beim Velassen erzeugt wird, entsteht der von Go erkannte Deadlock.

Anmerkung: Möglicherweise lässt sich die Implementierung durch geringfügige Modifikationen 'retten' (z.B. manuelles Einfügen einer Botschaft, Vertauschen von P() und V(), etc.). Diese zu benennen scheint uns aber nicht das Ziel der Aufgabe zu sein.

Aufgabe 2

Auf den ersten Blick fällt auf, dass die Version des veralteten Go-Tutorials zwei Kanäle (`ch`, `ch1`) für die erste 100 (prinzipiel: beliebig viele) Primzahlen benutzt, die Version im Buch dagen 315 für eine begrenzte Anzahl von Primzahlen (hier: 313). Der 'Grad der Nebenläufigkeit' schein somit in der Buchversion höher.

Die Funktion `Start` im Buch entspricht der Gunktion `generate` des veralteten Go-Tutorials: In `start` werden zunächst (mit Ausnahme der 2) nur ungerade Zahlen erzeugt, während `generate` jede natürliche Zahl beginnend mit der 2 generiert. Somit ist `start` etwas effizienter.

Die Funktion `sieve` im Buch entspricht der Funktion `filter` des veralteten Go-Tutorials: `sieve` übernimmt drei Parameter vom Typ `chan uint` (`in`, `out`, `off`) `filter` dagegen nur zwei dieser Art (`in`, `out`) und zusätzlich eine positive Ganzzahl (hier: Primzahl) `prime`. Prinzipiell prüfen beide Funktionen die Zahlen, die über den Eingabekanal `in` empfangen werden. Nur wenn diese keine Vielfachen einer bereits gefundenen Primzahl sind, werden sie über den Ausgabekanal `out` weitergeleitet (also nicht herausgesiebt). Der Unterschied liegt hier in der Art, wie die gefundenen Primzahlen prozessiert werden: In der Go-Tutorial-Version werden gefundene Primzahlen direkt ausgegeben, in der Buchversion ist dafür die Funktion `write` zuständig.

Aufgabe 3

Der Effekt des asynchronen Nachrichtenaustauchs lässt sich mit synchronem Austausch mit Hilfe eines Puffers erzeugen. So kann dann mindestens einer der Prozesse eine nicht blockierende Sende-oder Empfangsoperations nutzen. Der Pufferspeicher blockiert nur dann, wenn dieser voll ist. Der Empfänger kann daher jederzeit bereit für eingehende Verbindungen sein, da er diese erst abarbeitet, wenn es für diesen günstig ist. Der Sender hingegen kann damit unabhängig vom Empfänger seine Nachrichten verschicken, da er davon ausgehen kann, dass diese im Puffer des Empfängers landen. Statt der direkten Verbindung: “Sender - Empfänger” gibt es jetzt genau genommen die Verbindung “Sender - Puffer - Empfänger”.
Dies funktioniert natürlich auch umgekehrt. Will der Sender zu einem Zeitpunkt mehrere

Nachrichten verschicken, kann er diese in seinen eigenen Puffer legen und den Sendevorgang als “abgeschlossen” kennzeichen. Der Puffer versendet dann letztlich (aus technischer Schicht) die Nachrichten.

Aufgabe 4

1. direkte Übersetzung des Codes nach go

```
1  package channel
2
3  import "fmt"
4  import "sync"
5
6  type Kanal struct {
7    botschaft byte
8    s,e,mutex Mutex
9    }
10
11 func send(x Kanal, c byte){
12   x.mutex.wait()
13   x.botschaft = c
14   x.s.signal()
15   x.e.wait()
16 }
17
18 func init(x Kanal){
19   x.s = 0
20   x.e = 0
21   x.mutex = 1
22 }
23
24 func recv(x Kanal, c byte){
25   x.s.wait()
26   c=x.botschaft
27   x.e.signal()
28   s.mutex.signal()
29 }
```

2. Ablaufreihenfolge angeben, die Verklemmung erzeugt
   In Anlehnung an das Programm auf S. 169 im Buch wird folgt hier eine Reihenfolge der Befehle, die zu einer Verklemmung führen.
   a) in “order”: $c < -1$
   b) in “add”: $tmp =< -c$
   c) in “add”: $c < -1$
   d) in “add”: $c < -(< -c + temp$ hier wird nun beim $< -c$auf die Botschaft gewartet
   e) in “order” $< -c$ hier wird nun ebenfalls beim $< -c$ auf die Botschaft gewartet
   f) Da nun doppelt gewartet und gleichzeitig nicht mehr gesendet wird, kommt es hier zur Verklemmung.

Literatur

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

[2] A Tour of Go. URL `http://tour.golang.org/\#63`.