

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 verschiedenen 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, enthält aber keine Botschaft, da die `for`-Schleife in Zeile 6 nicht durchlaufen wurde.

2. Bei Aufruf der P-Operation wird eine Botschaft 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 Verlassen 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

TODO CHRIS

Aufgabe 3

Der Effekt des asynchronen Nachrichtenaustauschs lässt sich mit synchronem Austausch mit Hilfe eines Puffers erzeugen. So kann dann mindestens einer der Prozesse eine nicht blockierende Sende-oder Empfangsoperationen 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" kennzeichnen. 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

Literatur

- [1] Christian Maurer. *Nichtsequentielle Programmierung mit Go 1 Kompakt*. Springer Vieweg, 2012. ISBN 978-3642299681.
- [2] A Tour of Go. URL <http://tour.golang.org/#63>.