ALP4 SoSe 2013, Di. 16-18

Lösung Übungsblatt 10

Christoph van Heteren-Frese (Matr.-Nr.: 4465677)
Sven Wildermann (Matr.-Nr.: 4567553)
Tutor: Alexander Steen, eingereicht am 28. Juni 2013

Aufgabe 1

Das Krümmelmonsterproblem wird hier mit Hilfe von einer einmalig erzeugen Dose gelöst, die sowohl die Kekse als auch eine Klingel für die Benachrichtungen enthält. Hierauf werden dann die Funktionen essen und backen ausgeführt. Sollen mehr kekse gegessen werden als in der Dose vorhanden sind, so wird über die Funktion Send ein Signal an die Funktion backen geschickt, welche daraufhin die gewünschte Anzahl von Keksen backt. Sobald die Kekse fertig sind, wird über das Klingelsignal die Funktion essen wieder aufgeweckt. Ein Beispielhafter Ablauf wird beim Start der main-Funktion gezeigt.

```
1  package main
2
3  import (
4    "fmt"
5    "time"
6  )
7
8  func main() {
9    x := New()
10   done1 := make(chan bool, 1)
11   done2 := make(chan bool, 1)
12   done3 := make(chan bool, 1)
13   t := time.Duration(3)
14   go x.essen(3, done1)
15   time.Sleep(t * time.Second)
16   go x.essen(2, done2)
17   time.Sleep(t * time.Second)
18   go x.backen(5, done3)
19   <- done1
20   <- done2
21   <- done3
22 }
23
24 type Dose struct {
25   kekse chan int
26   klingel chan string
27 }
28
29 func New() *Dose {
30   x := new(Dose)
31   x.kekse = make(chan int, 1)
32   x.klingel = make(chan string, 1)
33   x.kekse <- 0
34   return x
```

```
35  }
36
37  func (x *Dose) Recv() (n int) {
38    n = <- x.kekse
39    return n
40  }
41
42  func (x *Dose) Send(n int) {
43    x.kekse <- n
44  }
45
46  func (x *Dose) backen(b int, e chan bool) {
47    fmt.Printf("> %d Keks(e) werden gebacken!\n", b)
48    n := x.Recv()
49    fmt.Printf("> Anzahl der Kekse in der Dose: %d\n", n)
50    x.Send(n + b)
51    fmt.Printf("> Anzahl der Kekse nach dem Backen: %d\n", n + b)
52    // wecke wartende Monster auf
53    fmt.Printf("> Wecke wartende Monster: \"Ding!\"\n")
54    x.klingel <- "Ding"
55    e <- true
56  }
57
58  func (x *Dose) essen(b int, e chan bool) {
59    fmt.Printf("%d Keks(e) sollen gegessen werden!\n", b)
60    n := x.Recv()
61    fmt.Printf("Anzahl der Kekse in der Dose: %d\n", n)
62    for n < b {
63      fmt.Printf("Lege die Kekse wieder zurueck :(\n")
64      x.Send(n)
65      fmt.Printf("Warte auf die Klingel...\n")
66      // warte bis jemand wieder Kekse backt
67      <- x.klingel
68      fmt.Printf("Es hat geklingelt!\n")
69      n = x.Recv()
70      fmt.Printf("Anzahl der Kekse in der Dose nach dem Klingeln: %d\n", n)
71    }
72    // jetzt darf ich essen (n >= b)
73    fmt.Printf("Ich darf %d Keks(e) essen! :) \n", b)
74    x.Send(n - b)
75    fmt.Printf("Anzahl der Kekse nach dem Essen: %d\n", n - b)
76    // falls noch andere Monster gewartet hatten
77    x.klingel <- "Dong"
78    e <- true
79  }
```

Aufgabe 2

Das Zigarettenraucherproblem wurde mit Hilfe von Botschaftenaustausch gelöst, in dem der Tisch als Channel entwickelt wurde. Es gibt 3 Zustände für die verschiedenen Kombinationen, die auf dem Tisch liegen können +1 falls garnichts auf dem Tisch liegt. Sobald für einen Raucher das richtige auf dem Tisch liegt, “nimmt” sich dieser das entsprechende Equipment und fängt an, seine Zigarette zu rauchen. Sobald er mit dem Rauchen fertig ist, wird erneut der Wirt gerufen (ebenfalls über einen Channel - der Waiter). Dies wird von der Scheduler-Funktion erkannt und ruft den Wirt und die Raucher (damit diese wieder “lauschen”) auf. Der Zufall entscheidet was der Wirt bringt. Das Spiel beginnt

anschließend von vorn. Weitere Details siehe Implementierung.

Statt eine Funktion zu haben, die für alle Raucher verantwortlich ist, könnte man auch jeden Raucher einzeln modellieren und diese den Channel überprüfen lassen. Hier wäre der wichtigste Unterschied, dass nach der Überprüfung des Wertes im Channel dieser im Falle einer **Nicht-Übereinstimmung** wieder zurück in den Channel geschrieben wird, damit das Equipment nicht verloren geht. Im Sinne der Übersichtlichkeit halte ich diese Version allerdings für angebrachter.

```
1  package main
2
3  import (
4    "fmt"
5    "time"
6    "math/rand"
7  )
8
9  const (
10 //declare the states - what the host can bring
11   PapersTobacco = iota
12   PapersLucifers = iota
13   TobaccoLucifers = iota
14   Empty = iota
15 )
16
17 func main() {
18 // set the values and start the schedular,
19 //who starts the game
20   var waiter = make(chan int,1)
21   var table = make(chan int,1)
22   waiter <-1
23   scheduler(waiter,table)
24 }
25
26 func scheduler(waiter chan int, table chan int){
27 // check always for waiting guests, which  need
28 // to be served
29   for {
30     <-waiter
31     go givesomething(table)
32     go getsomething(table,waiter)
33   }
34
35 }
36
37 func givesomething(cs chan int){
38 //the host gives something to the table
39 //which is randomly choosen
40   rand.Seed(time.Now().UnixNano())
41   cs<-rand.Intn(Empty)
42   fmt.Println("=== HOST IS SERVING ===")
43   time.Sleep(2*1e9)
44 }
45
46
47 func getsomething(cs chan int, waits chan int){
48 //the smokers need to check who is able to smoke
49 //so they check whether there is something on the table
50 // and check what it is  and decide because of that
51   number := <-cs
```

```
52    time.Sleep(2*1e9)
53    if number==PapersTobacco {
54      fmt.Println("Thats what PETER  was looking for - lets begin to smoke")
55    }else if number==PapersLucifers {
56      fmt.Println("Thats what JAN was looking for - lets begin to smoke")
57    }else if number==TobaccoLucifers{
58      fmt.Println("Thats what ADRIAN was looking for - lets begin to smoke")
59    }else{
60      // this state should never be reached
61      fmt.Println("Waiting for something that fits")
62    }
63    fmt.Println("===SMOOKING===");
64    time.Sleep(2*1e9)
65    fmt.Println("===END OF SMOOKING===")
66    time.Sleep(2*1e9)
67    //having smoked, they call the host again and wait
68    waits <-1
69  }
```

Aufgabe 4

Das hier gegebene Implementierung entspricht weitestgehend dem Ansatz im Buch. Lediglich die Idee, einen zusätzlichen Kanal für das Signalisieren der Absicht zu empfangen wurde hinzugefügt.

```
1   package async
2
3   type ImpAsync struct {cS, cR chan byte; cA chan bool; in int}
4
5   func New (n uint) *ImpAsync {
6     p:= new (ImpAsync)
7     p.cS, p.cR, p.cA = make (chan byte), make (chan byte), make (chan bool)
8     go func() {
9       buffer := make([]byte, n)
10      var count, in, out uint
11      for {
12        if count == 0 {
13          buffer[in] = <-p.cS
14          in = (in+1) % n; count = 1
15        } else if count == n {
16          p.cR <- buffer[out]
17          out = (out+1) % n; count = n-1
18        } else {
19          select {
20          case  buffer[in] = <- p.cS:
21            in = (in + 1) % n; count++
22          case <- p.cA:
23            p.cR <- buffer[out]
24            out = (out+1) % n; count--
25          }
26        }
27      }
28    }()
29    return p
30  }
31
32  func (p *ImpAsync) Send (b byte) { p.cS <- b }
33
34  func (p *ImpAsync) Recieve() (b byte) {
```

```
35    p.cA <- true // Absicht senden, etwas entnehmen zu wollen
36    return <- p.cR
37  }
```

Ein neuer asynchroner Kanal mit einer Buffergröße von `n` kann somit mit `asyncChannel:=async.New(n)` erzeugt werden. Mit `asyncChannel.Send` kann eine Botschaft versendet werden, mit `asyncChannel.Recieve` kann eine Botschaft empfangen werden.