CSI2120: PROGRAMMING PARADIGMS

Tutorial 3: channels and competition in Go

Exercise 1: synchronization

Design a program in which the numberGen function generates integers between start and (start + count) , and the printNumbers function prints the generated integers to the screen (use synchronization and closure of channels).

```
package main

import "fmt"

func numberGen(start, count int, out chan<- int) {
for i:=0; i<count; i++ {
    out<-(start+i)

}
close(out)
}

func printNumbers(in <-chan int, done chan<- bool) {

for {
    value, ok := <- in // ok is false if channel has been closed
    if !ok {
        done<-true
        break;
    }
    fmt.Println(value)
}
}

func main() {

    channel := make(chan int)
    done := make(chan bool)
    go numberGen(7,7,channel)
    go printNumbers(channel, done)
    <-done
}
```

Exercice 2 : Select the time after method

Design a program that reads data entering channels for a certain amount of time.

```
package main

import (
	"fmt"
	"time"
)

func tr(c chan int, w time.Duration) {
	i := 0
	for {
		time.Sleep(w)
		c <- i
		i++
	}
}

func main() {
	m1 := make(chan int)
	m2 := make(chan int)

	go tr(m1, 1*time.Second)
	go tr(m2, 1*time.Second)
	a := time.After(5 * time.Second)
P1:
	for {
		select {
		case msg1 := <-m1:
			fmt.Println("received from m1 message", msg1)
		case msg2 := <-m2:
			fmt.Println("received from m2 message", msg2)
		case <-a:
			fmt.Println("Time out")
			break P1
		}
		time.Sleep(time.Second)
	}

	close(m1)
	close(m2)
	time.Sleep(3 * time.Second)
}
```

However, this program throws a panic when the channels are closed. How to get the program out of this state?

```
func tr(c chan int, w time. Duration)
       { defer func() {
           if r := recover();
           r != nil {
           fmt. Printf("Channel data sending stopped")
             }
      }()

        i:=0
        for {
           time.Sleep(w)
           c <- i
           i++
        }
}
```

A better approach would be to give the producer function the responsibility of creating the write channel and closing it when required. It is then necessary to go through a secondary synchronization channel.

```
package main

import (
     "fmt"
     "time"
)

func tr(done chan bool, w time.Duration) <-chan int {
     intChan := make(chan int)
     go func() {
          defer close(intChan)
          defer fmt.Println("Channel data sending stopped")

          i := 0
          for {
               time.Sleep(w)
               select {
               case intChan <- i:
                    i++
               case <-done: // we can always read from a closed
channel return
               }
          }
     }()

}
```

```
     return intChan
}
func main() {

     done := make(chan bool)

     m1 := tr(done, 1*time.Second)
     m2 := tr(done, 3*time.Second)
     a := time.After(5 * time.Second)
P1:
     for {
          select {
          case msg1 := <-m1:
               fmt.Println("received from m1 message", msg1)
          case msg2 := <-m2:
               fmt.Println("received from m2 message", msg2)
          case <-a:
               fmt.Println("Time out")
               break P1
          }
          time.Sleep(time.Second)
     }

     close(done)
     time.Sleep(3 * time.Second)
}
```

Exercice 3 : Filters

What does the following program do?

```
package main
import "fmt"
func ping(pings chan <- string, msg string)
    { pings <- msg
}
func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}
func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)




}
```

The following program finds 20 prime numbers using a chain of filters. Explain how it works.

```
package main

// Send the sequence 2, 3, 4, ... to channel 'ch'.
func Generate(ch chan <- int) {
      for i := 2; ; i++ {
            ch <- i // Send i to channel 'ch'.
      }
}

// Copy values from channel 'in' to channel 'out' removing those
divisible by 'p'
func Filter(in <- chan int, out chan <- int, p int)
      { for {
            i := <-in // Get the values from 'in'.
            if i%p != 0 {
                  out <- i // Send i to channel 'ch'.
            }
      }
}

func main() {
  ch:= make(chan int) // Create a new channel.
  go Generate(ch)      // We run the Generate goroutine
  for i:= 0; i < 20; i++ {
      p := <-ch
      print(p, "\n")
      ch1 := make(chan int)
      go Filter(ch, ch1, p) // all filters are chained
      ch = ch1
  }
}
```