DD1339 Introduktion till datalogi

Namn: ***Hampus Fristedt***

Uppgift: VT9

Grupp nummer: 1

Övningsledare: Peter Boström

Betyg: ..... **Datum: .....** **Rättad av: .....**

1 Matching.go

1.1 Vad händer om man tar bort go-kommandot från Seek-anropet i main-funktionen?

Eftersom kanalen match har en buffer kommer programmet fortfarande att fungera som förväntat. Det finns dock en liten skillnad. Om vi låter go-kommandot stå kvar finns det ingenting som garanterar ordningen på vem som tar emot vems meddelande eftersom vi inte vet vilken gorutin som körs först. Om vi tar bort go-kommandot kommer ordningen alltid vara Anna till Bob, Cody till Dave och ingen som tar emot Evas.

1.2 Vad händer om man byter deklarationen wg := new(sync.WaitGroup) mot var wg sync.WaitGroup och parametern wg *sync.WaitGroup mot wg sync.WaitGroup?

Då kommer vi skicka en kopia av wg till Seek istället för referensen till vår sync.WaitGroup. Detta gör att wg aldrig blir färdig och vi får ett deadlock vid wg.Wait().

1.3 Vad händer om man tar bort bufferten på kanalen match?

Om vi har ett jämnt antal namn i people kommer allting fungera som vanligt. Däremot kommer det skita sig om vi har ett udda antal namn eftersom den sista gorutinen som kör Seek kommer att vänta på att någon tar emot namnet som skickas till match, och eftersom ingen finns där för att ta emot kommer aldrig den sista wg.Done() kallas, och vi får ett deadlock.

1.4 Vad händer om man tar bort default-fallet från case-satsen i main-funktionen?

Om vi har ett udda antal namn kommer allting att fungera, men om vi har ett jämnt antal kommer select-kommandot i mainmetoden att vänta på att ta emot någonting från match, och eftersom ingenting skickas till match får vi ett deadlock.

2 Julia.go

../julia.go

```
1  // Stefan Nilsson 2013-02-27
2
3  // This program creates pictures of Julia sets
       (en.wikipedia.org/wiki/Julia_set).
4  package main
5
6  import (
7      "sync"
8      "fmt"
9      "image"
10     "image/color"
11     "image/png"
12     "log"
13     "math/cmplx"
14     "os"
15     "strconv"
16     "time"
17     "runtime"
18 )
19
20 type ComplexFunc func(complex128) complex128
```

```
21
22 var Funcs []ComplexFunc = []ComplexFunc{
23     func(z complex128) complex128 { return z*z - 0.61803398875 },
24     func(z complex128) complex128 { return z*z + complex(0, 1) },
25     func(z complex128) complex128 { return z*z + complex(-0.835,
           -0.2321) },
26     func(z complex128) complex128 { return z*z + complex(0.45, 0.1428)
           },
27     func(z complex128) complex128 { return z*z*z + 0.400 },
28     func(z complex128) complex128 { return cmplx.Exp(z*z*z) - 0.621 },
29     func(z complex128) complex128 { return (z*z+z)/cmplx.Log(z) +
           complex(0.268, 0.060) },
30     func(z complex128) complex128 { return cmplx.Sqrt(cmplx.Sinh(z*z))
           + complex(0.065, 0.122) },
31 }
32
33 var numcpu = runtime.NumCPU()
34
35 func init() {
36     runtime.GOMAXPROCS(numcpu) // Try to use all available CPUs.
37 }
38
39 func main() {
40     fmt.Println("Main started")
41     t := time.Now()
42     for n, fn := range Funcs {
43         err := CreatePng("picture-"+strconv.Itoa(n)+".png", fn, 1024)
44         if err != nil {
45             log.Fatal(err)
46         }
47     }
48     fmt.Printf("Time since start: %fs\n", time.Since(t).Seconds())
49 }
50
51 // CreatePng creates a PNG picture file with a Julia image of size n x
       n.
52 func CreatePng(filename string, f ComplexFunc, n int) (err error) {
53     file, err := os.Create(filename)
54     if err != nil {
55         return
56     }
57     defer file.Close()
58     err = png.Encode(file, Julia(f, n))
59     return
60 }
61
62 // Julia returns an image of size n x n of the Julia set for f.
63 func Julia(f ComplexFunc, n int) image.Image {
64     bounds := image.Rect(-n/2, -n/2, n/2, n/2)
65     img := image.NewRGBA(bounds)
66     s := float64(n / 4)
67     wg := new(sync.WaitGroup)
68     wg.Add(numcpu)
69 //  for (routine := 0; routine < numcpu; routines++) {
70     go func() {
71         fmt.Println("Go1 started")
```

```
72          for i := bounds.Min.X; i < (bounds.Max.X / 2); i++ {
73              for j := bounds.Min.Y; j < bounds.Max.Y; j++ {
74                  n := Iterate(f, complex(float64(i)/s, float64(j)/s),
                        256)
75                  r := uint8(0)
76                  g := uint8(0)
77                  b := uint8(n % 32 * 8)
78                  img.Set(i, j, color.RGBA{r, g, b, 255})
79              }
80          }
81          fmt.Println("Go1 done")
82          wg.Done()
83      }()
84      go func() {
85          fmt.Println("Go2 started")
86          for i := bounds.Max.X / 2; i < bounds.Max.X; i++ {
87              for j := bounds.Min.Y; j < bounds.Max.Y; j++ {
88                  n := Iterate(f, complex(float64(i)/s, float64(j)/s),
                        256)
89                  r := uint8(0)
90                  g := uint8(0)
91                  b := uint8(n % 32 * 8)
92                  img.Set(i, j, color.RGBA{r, g, b, 255})
93              }
94          }
95          fmt.Println("Go2 done")
96          wg.Done()
97      }()
98  //  }
99      wg.Wait()
100     return img
101 }
102
103 // Iterate sets z_0 = z, and repeatedly computes z_n = f(z_{n-1}), n =>
        1,
104 // until |z_n| > 2  or n = max and returns this n.
105 func Iterate(f ComplexFunc, z complex128, max int) (n int) {
106     for ; n < max; n++ {
107         if real(z)*real(z)+imag(z)*imag(z) > 4 {
108             break
109         }
110         z = f(z)
111     }
112     return
113 }
```

3 Väderstation

../client.go

```
1  package main
2
3  import (
4      "fmt"
5      "io/ioutil"
6      "log"
```

```
7       "net/http"
8       "time"
9   )
10
11  func main() {
12      server := []string{
13          "http://localhost:8080",
14          "http://localhost:8081",
15          "http://localhost:8082",
16      }
17      for {
18          before := time.Now()
19          //res := Get(server[0])
20          //res := Read(server[0], 0)
21          res := MultiRead(server, time.Second)
22          after := time.Now()
23          fmt.Println("Response:", *res)
24          fmt.Println("Time:", after.Sub(before))
25          fmt.Println()
26          time.Sleep(500 * time.Millisecond)
27      }
28  }
29
30  type Response struct {
31      Body       string
32      StatusCode int
33  }
34
35  // Get makes an HTTP Get request and returns an abbreviated response.
36  // Status code 200 means that the request was successful.
37  // The function returns &Response{"", 0} if the request fails
38  // and it blocks forever if the server doesn't respond.
39  func Get(url string) *Response {
40      res, err := http.Get(url)
41      if err != nil {
42          return &Response{}
43      }
44      // res.Body != nil when err == nil
45      defer res.Body.Close()
46      body, err := ioutil.ReadAll(res.Body)
47      if err != nil {
48          log.Fatalf("ReadAll: %v", err)
49      }
50      return &Response{string(body), res.StatusCode}
51  }
52
53  // FIXME
54  // I've found two insidious bugs in this function; both of them are
        unlikely
55  // to show up in testing. Please fix them right away - and don't forget
        to
56  // write a doc comment this time.
57
58  // Bug 1: Both the main Go routine and the one calling Get() have read
        access
```

```
59 // to res, which could result in a datarace. Could be fixed by changing
       it so
60 // they do not access the same memory.
61
62 // Bug 2: If both cases of select come true at the same time, one will
       be chosen
63 // at random. This might not be wanted.
64
65 // Read makes a HTTP Get request to the url and returns the response.
       Status
66 // code 200 means success. If the server does not answer within the
       suppied
67 // timeout Read returns a Response stating timeout with status code 504.
68 func Read(url string, timeout time.Duration) *Response {
69     done := make(chan bool)
70     res := new(Response)
71     go func() {
72         res = Get(url)
73         done <- true
74     }()
75     select {
76     case <-done:
77     case <-time.After(timeout):
78         return &Response{"Gateway timeout\n", 504}
79     }
80     return res
81 }
82
83 // MultiRead makes an HTTP Get request to each url and returns
84 // the response of the first server to answer with status code 200.
85 // If none of the servers answer before timeout, the response is
86 // 503 - Service unavailable.
87     func MultiRead(urls []string, timeout time.Duration) (res
          *Response) {
88     var responses = make([]*Response, len(urls))
89     ch := make(chan int)
90     for i, url := range urls {
91         go func() {
92             responses[i] = Get(url)
93             ch <- i
94         }()
95     }
96     select {
97     case i := <-ch:
98         res = responses[i]
99     case <-time.After(timeout):
100        res = &Response{"Service unavailable\n", 503}
101    }
102    return
103 }
```