

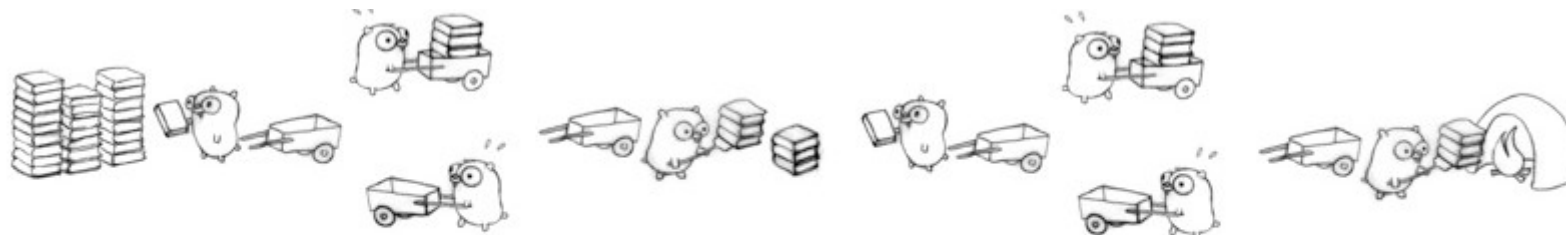
Konkurentné vzory v Go

(gorutina-kanál-mutex)

Peter Borovanský, KAI, I-18,
borovan(a)ii.fmph.uniba.sk



Channels are one of the most popular features of Go and allow for elegant streamlining of data reading/writing and are most often used to prevent data races. They become particularly powerful when used concurrently, as multiple Go routines can write to the same channel.



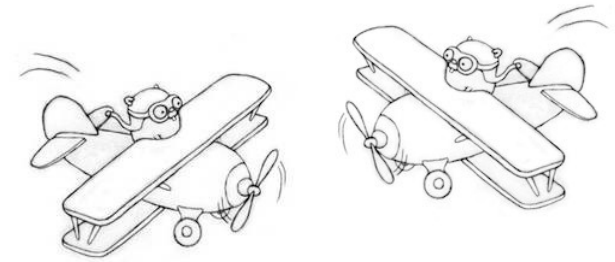
Konkurencia vs. paralelizmus

- kompozícia nezávislých výpočtov
 - spôsob myslenia, ako výpočet (prácu) rozdeliť medzi nezávislých agentov
 - keďže okolitý svet je paralelný, je to spôsob ako lepšie interagovať s ním
 - málo kto z nás má skutočne paralelný HW, možno tak 8-, 16-jadro...
-
- na jednom procesore paralelizmus neurobíte, ale konkurentný výpočet áno ale ...
 - konkurentný výpočet na jednom procesore bude *pravdepodobne* pomalší ako sekvenčný, takže viac ide o konkurentnú paradigmu (myslenie) ako o čas

Go konkurencia založená na Communicating Sequential Processes (T. Hoare, 1978) poskytuje:

- konkurentné procedúry (tzv. gorutiny, 8kB stack)
- poskytujú synchronizáciu a komunikáciu prostredníctvom kanálov, mutexov
- príkaz select

Gorutina - príklad



Gorutina `loopForever` sa vykonáva ako funkcia `loopForever` len sa nečaká na jej výsledok, resp. skončenie

```
package main
```

```
import (    "fmt"    "math/rand"    "time")
```

```
func loopForever(task string) {  
    for i := 1; ; i++ {        // počítame do nekonečna  
        fmt.Printf("%s:%d\n", task, i)  
        time.Sleep(time.Duration(rand.Intn(500)) *  
            time.Millisecond)    }}
```

```
func main() {
```

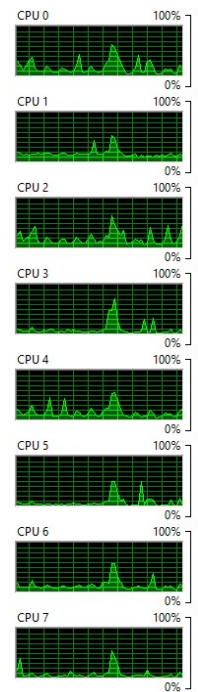
```
→ go loopForever("prvy")        // spustenie 1.gorutiny
```

```
→ go loopForever("druhy")       // spustenie 2.gorutiny
```

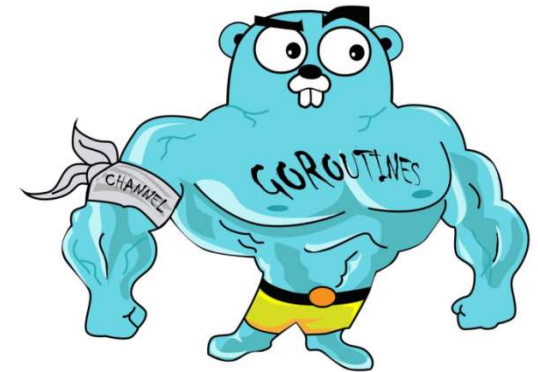
```
var input string // toto čaká na input, v opačnom
```

```
fmt.Scanln(&input) // prípade, keď umrie hlavné vlákno
```

```
fmt.Println("main stop")} // umrie v Go všetko..
```



Gorutina



Gorutina nie je corutina (tá má bližšie generátorom, async/await z Python 3.5)

- je nezávisle vykonávaná funkcia
- má vlastný stack 8kB - rastie sa podľa jej potrieb, GO1.3 ([Contiguous stacks](#))
- môže ich byť veľa, aj veľmi veľa (uvidíme ~ 1.000.000)
- je to menej ako vlákno (thread), ale k nemu to má najbližšie

Anonymná gorutina je de-facto bezmenná funkcia, ktorú aj hneď zavoláme:

```
func main() {  
    go func /*tu chýba meno fcie*/ (task string) {  
        for i := 1; ; i++ { // počítame do nekonečna  
            fmt.Printf("%s:%d\n", task, i)  
            time.Sleep(...)  
        }  
    } ("prvy") // tu hneď voláme anonymnú fciu s argumentom
```

Komunikácia a synchronizácia

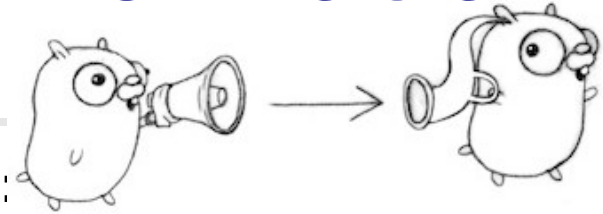
(high-level)

Pomocou kanálov (nebuffrovaná–synchronná verzia):

```
var ch chan int
```

```
resp. ch := make(chan int)
```

```
ch = make(chan int)
```



zápis do kanála je blokujúca operácia, kým hodnotu niekto neprečíta z kanála

```
ch <- 123
```

čítanie z kanála je blokujúca operácia, až kým hodnotu niekto nezapíše do kanála

```
x = <-ch
```

takže ide o komunikáciu (prenos dát), ale aj o synchronizáciu rutín/vláknien.

V prípade buffrovaných kanálov `make(chan int, 10)` prideme o synchronizáciu, takže to skúsime neskôr...



Go mantra

“
Do not communicate by
sharing memory; instead,
share memory by communicating.

— *Effective Go*

”

Golang Puzzlers

(čo sa stane, keď...)

- deadlock ch<-1
- syntax error
- deadlock ch<-2
- skončí main

- zapisane 1
- zapisane 2
- idem citat
- precitane 1

```
func main() {  
    ch := make(chan int, 4)
```

```
    go func () {  
        time.Sleep(time.Duration(5 * time.Second))  
        fmt.Println("idem citat")  
        fmt.Printf("precitane %d \n", <- ch )  
    } ()
```

```
    ch <- 1  
    fmt.Println("zapisane 1")  
    ch <- 2  
    fmt.Println("zapisane 2")
```

```
    var input string // toto čaká na input, v opačnom  
    fmt.Scanln(&input) // prípade, keď umrie hlavné  
    fmt.Println("main stop")
```

```
}
```

Timers

http://divan.github.io/posts/go_concurrency_visualize/

```
func timer(d time.Duration) (ch chan int) {  
    ch = make(chan int)  
    go func() {  
        time.Sleep(d)  
        ch <- 1  
    }()  
    return  
}  
  
func main() {  
    for i := 0; i < 24; i++ {  
        c := timer(1 * time.Second)  
        fmt.Println(<-c)  
    }  
}
```



kol'ko gorutin beží zároveň ?

- 1
- 2
- 24
- 25

<http://divan.github.io/demos/timers/>

[timers.go](http://divan.github.io/demos/timers/)



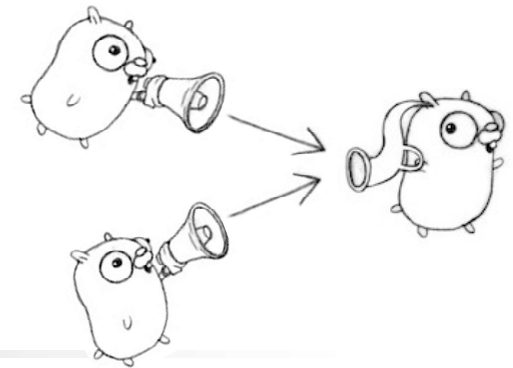
Timers2

```
func timer2(d time.Duration, ch chan int) {  
    go func() {  
        time.Sleep(d)  
        ch <- 1  
    }()  
}  
  
func main() {  
    ch := make(chan int)  
    for i := 0; i < 24; i++ {  
        timer2(time.Duration(i) * time.Second, ch)  
    }  
    for x := range ch {  
        fmt.Println(x)  
    }  
}
```

kol'ko gorutin beží zároveň ?

- 1
- 2
- 24
- 25

Dvaja píšu, jeden číta



```
func loopAndSend(task string, ch chan string) {
    for i := 1; i < 30; i++ {
        ch <- fmt.Sprintf("%s:%d\n", task, i)
        time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
    }
}

func main() {
    ch := make(chan string)
    go loopAndSend("prvy", ch)
    go loopAndSend("druhy", ch)
    for {
        // tu to čítame
        msg := <-ch

        fmt.Print(msg)
    }
    fmt.Println("main stop") }

// dve gorutiny píšu do
// toho istého kanála ch

for msg := range ch {
    // range prebieha obsahom
    // celého kanála
    fmt.Print(msg)
}

// nikdy neskončí, prečo ?
```

Funkcia vráti kanál



chan string je typ kanála stringov, funkcia ho môže vrátiť ako výsledok

```
func loopToChannel(task string) chan string {  
    ch := make(chan string) // vytvor kanál  
    go func() {              // pusti nezávislú gorutinu  
        for i := 1; i < 30; i++ { // ktorá píše do kanála  
            ch <- fmt.Sprintf("%s:%d\n", task, i)  
            time.Sleep(...) }  
    }()  
    return ch }              // vráť kanál ch:chan string  
func main() {  
    ch1 := loopToChannel("prvy")  
    ch2 := loopToChannel("druhy")  
    for {  
        fmt.Print(<-ch1)      // čo dostaneme ???  
        fmt.Print(<-ch2)      // chápeme už synchronizáciu ??
```

Kanálový sútok



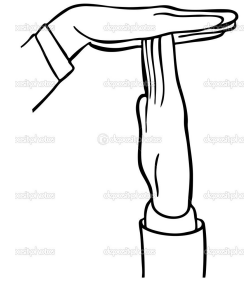
```
func multiplexor(ch1, ch2 chan string) chan string {  
    ch := make(chan string)  
    go func() {                                // prvá gorutina  
        for { ch <- <-ch1 }                    // čítaj z ch1 a píš to do ch  
    }()  
    go func() {                                // druhá gorutina  
        for { ch <- <-ch2 }                    // čítaj z ch2 a píš to do ch  
    }()  
    return ch}  
func main() {  
    ch1 := loopToChannel("prvy")               // tretia gorutina  
    ch2 := loopToChannel("druhy")             // štvrtá gorutina  
    ch := multiplexor(ch1, ch2)  
    for {  
        fmt.Print(<-ch)    }
```



Select

select je príkaz syntaxou podobný switch, à la java

```
func multiplexorSelect(ch1, ch2 chan string) chan string {  
    ch := make(chan string)  
    go func() {  
        // jednu gorutinu sme ušetrili :-)  
        for {  
            → select {  
                // select vykoná niektorý neblokovaný  
                case val := <-ch1: // komunikačný case-príkaz  
                    ch <- val      // ak niekto zapísal do ch1  
                case val := <-ch2: // číta sa z ch1, ak ch2,  
                    ch <- val      // tak z ch2, inak je blokový  
            }  
        }  
        // select odpáli nejaká komunikačná udalosť  
        // (zápis/čítanie z/do kanála) v case príkazoch  
        // alebo timeout...  
    }()  
    return ch }  
}
```



Select a timeout

```
func multiplexorSelect(ch1, ch2 chan string) chan string {  
    ch := make(chan string)  
    go func() {  
        → gameOver := time.After(10 * time.Second)  
        for {  
            select {  
                case val := <-ch1: ch <- val  
                case ch <- <-ch1:  
                case val := <-ch2: ch <- val  
                case <-gameOver:  
                    ch <- "GAME OVER\n"  
                    close(ch)  
  
                }  
            }  
        }()  
        return ch  
    }  
}
```



Timeout

```
gameOver := time.After(10*time.Second)
```

alebo vlastný kód

```
gameOver := make(chan bool)
go func(seconds int) {
    time.Sleep(seconds*time.Second)
    gameOver <- true // timeout
}(10)
```

```
je kanál už zavretý ?
for { fmt.Print( <-ch) }
zle skončí, ak close(ch)
for {
    val, opened := <-ch
    if !opened {
        break
    }
    fmt.Print(val)
}
```

Ping-Pong

(http://divan.github.io/posts/go_concurrency_visualize/)

```
func main() {
    var Ball int
    table := make(chan int)
    go player(table)
    go player(table)
    go player(table)

    table <- Ball
    time.Sleep(1 * time.Second)
    <-table
}

func player(table chan int) {
    for {
        ball := <-table
        ball++
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```



<http://divan.github.io/demos/pingpong/>



<http://divan.github.io/demos/pingpong3/>

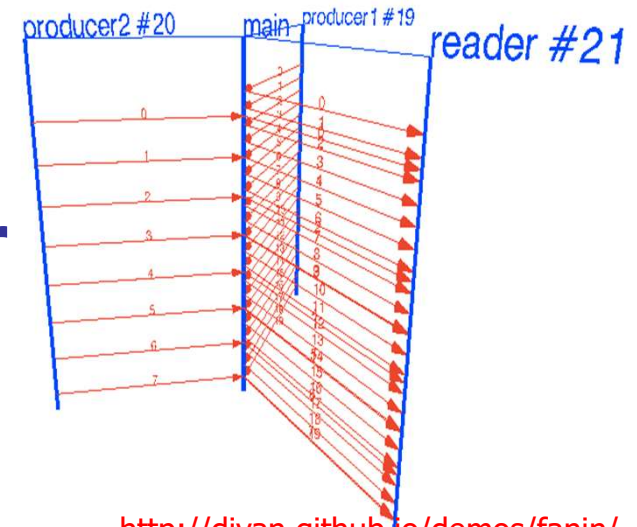
[pingpong.go](http://divan.github.io/demos/pingpong3/)

Producer-Consumer

```
func producer(ch chan int) {
    for i := 1; i <= 30; i++ {
        ch <- i
        fmt.Println("produce: " + strconv.Itoa(i))
        //time.Sleep(time.Second) // lenivá produkcia
    }
}

func consumer(ch chan int) {
    for i := 1; i <= 30; i++ {
        fmt.Println("consume: ", <-ch)
        time.Sleep(time.Second) // lenivá spotreba
    }
}

func main() {
    ch := make(chan int, 5) // buffrovaný kanál veľkosti 5
    go producer(ch) go producer(ch) // 1. a 2. producer
    go consumer(ch)
    time.Sleep(100000000000) // skoro večnosť
}
```



<http://divan.github.io/demos/fanin/>

Čínsky šepkári



```
var number = 1000000
func main() {
    start := time.Now()
    prev := make(chan int)
    first := prev          // ľavé ucho (ľ.u.) nultého šepkára
    go func() { first <- 0 }() // nultému šepneme 0 do ľ.u.
    for i := 0; i < number; i++ { // 40000 číňanov
        next := make(chan int)    // kanál z p.u.i-teho
        go func(from, to chan int) { // do ľ.u. i+1-vého
            for { to <- 1 + <- from } // šepnem ďalej 1+čo
        }(prev, next)              // počujem
        prev = next                // pokračujem, i k i+1
    }
    elapsed := time.Since(start)
    fmt.Println(<-prev) fmt.Println(elapsed)}
```

Prvočísla

(Eratosténovo sito)

```
prev := make(chan int)
first := prev
go func() {
    for i := 2; ; i++ { first <- i } }() // do first sypeme 2,3,4, ...
    for i := 0; i < 10000; i++ {
        prime := <-prev
        fmt.Println(prime)
        next := make(chan int)
        go func(prime int, from, to chan int) { // číta z from, píše do
            for {
                val := <-from
                if val%prime > 0 {
                    to <- val
                }
            }
        }(prime, prev, next)
        prev = next
    }
}
```

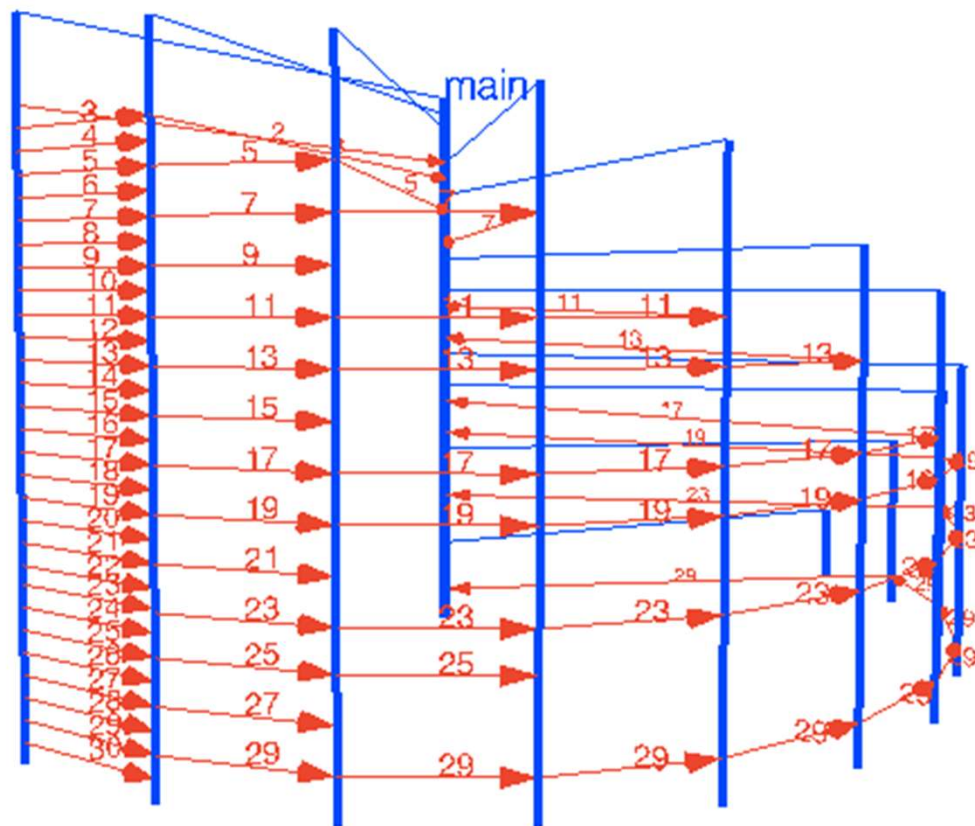
// čínski preosievači prvočísel
 // prvé preosiate musí byť prvočíslo
 // kanál pre ďalšieho preosievača
 // číta z from, píše do
 // do to, vyčiarkne deliteľné prime
 // číta z from - vstupný kanál
 // je deliteľné prime ?
 // ak nie je, píš do to - výstupný
 // spustenie nezávislého preosievača
 // výsledok ide ďalšiemu osievačovi

1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	41	42	43	44	45	46	47	48	49	50

1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	41	42	43	44	45	46	47	48	49	50

Prvočísla

(http://divan.github.io/posts/go_concurrency_visualize/)



<http://divan.github.io/demos/primesieve/>



Quicksort - pivotizácia

Nekonkurentná pivotizácia, nepekné dvojité testy ...

```
func pivot(pole []int) int {  
    i, j, pivot := 1, len(pole)-1, pole[0]  
    for i <= j {  
        // hľadanie maxiputána medzi liliputánmi  
        for i <= j && pole[i] <= pivot { i++ }  
        // hľadanie liliputána medzi maxiputánmi  
        for j >= i && pole[j] >= pivot { j-- }  
        if i < j { // nájdení kandidáti sa vymenia  
            pole[i], pole[j] = pole[j], pole[i]  
        }  
    } // pivota pichni medzi liliputánov a maxiputánov  
    pole[0], pole[j] = pole[j], pole[0]  
    return i }  

```

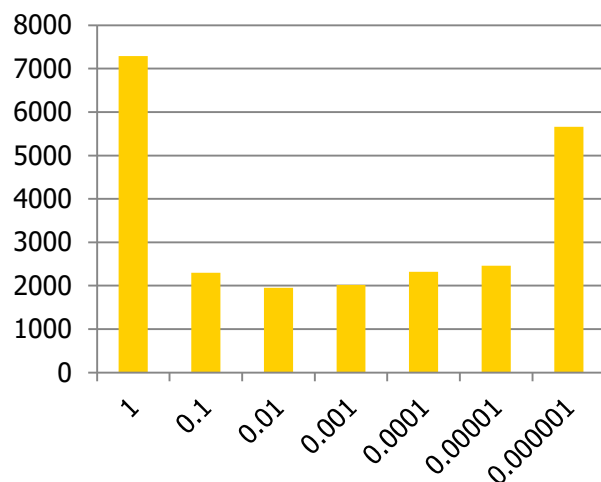


Quicksort

```
func cquickSort(pole []int, done chan bool) {  
    if len(pole) <= 1 {  
        done <- true  
    } else if len(pole) < granularity {  
        squickSort(pole)  
        done <- true  
    } else {  
        index := pivot(pole)  
        left, right := make(chan bool), make(chan bool)  
        go cquickSort(pole[:index-1]), left)  
        go cquickSort(pole[index:], right)  
        done <- (<-left && <-right)  
    }  
}
```

Quicksort výsledky

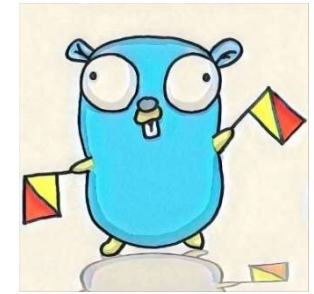
Size	Granularity	time
500.000	500.000	109ms
500.000	50.000	62ms
500.000	5.000	78ms
500.000	500	62ms
500.000	50	171ms
500.000	5	1375ms
500.000	1	niet dost' kanálov...



Size	Granularity	time	time'19
50.000.000	50.000.000	7s 291ms	5s061ms
50.000.000	5.000.000	2s 293ms	1s285ms
50.000.000	500.000	1s 951ms	1s152ms
50.000.000	50.000	2s 015ms	1s142ms
50.000.000	5.000	2s 318ms	1s232ms
50.000.000	500	2s 461ms	1s310ms
50.000.000	50	5s 663ms	2s192ms
50.000.000	5	niet dost' kanálov...	24s635ms

sync.WaitGroup

(mutex)



Semafor, alias mutex, je synchronizácia na nižšej úrovni ako kanál

```
package main
```

```
import ("fmt" "math/rand" "time" "sync")
```

```
func loopForever(task string, goGroup *sync.WaitGroup) {
```

```
    for i := 1; i < 10; i++ {
```

```
        fmt.Printf("%s:%d\n", task, i)
```

```
        time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
```

```
    }
```

```
→ goGroup.Done()
```

```
// dekrementovanie mutexu
```

```
}
```

```
func main() {
```

```
→ goGroup := new(sync.WaitGroup) // vytvorenie mutexu
```

```
→ goGroup.Add(2) // nastavenie mutexu na 2
```

```
go loopForever("prvy", goGroup)
```

```
go loopForever("druhy", goGroup)
```

```
→ goGroup.Wait() // blokuj, kým mutex > 0
```

[concurrent sync.go](https://golang.org/pkg/sync/#WaitGroup)

Semafór

(mutex - <https://gobyexample.com/mutexes>)

“
Do not communicate by
sharing memory; instead,
share memory by communicating.
— Effective Go
”

```
var state = make(map[int] int) //  
state alias HasmMap<Integer, Integer>  
var mutex = &sync.Mutex{}  
var readOps uint64  
var writeOps uint64  
for r := 0; r < 100; r++ {  
    go func() {  
        total := 0  
        for {  
            key := rand.Intn(5)  
            mutex.Lock()  
            total += state[key] state reader  
            mutex.Unlock()  
            atomic.AddUint64(&readOps, 1)  
            time.Sleep(time.Millisecond)  
        }  
    }()  
}
```

```
for w := 0; w < 10; w++ {  
    go func() {  
        for {  
            key := rand.Intn(5)  
            val := rand.Intn(100)  
            mutex.Lock()  
            state[key] = val state writer  
            mutex.Unlock()  
            atomic.AddUint64(&writeOps, 1)  
            time.Sleep(time.Millisecond)  
        }  
    }()  
}
```



Robotníci a lopaty

(workers & worker pool)

```
var WORKERS    = runtime.NumCPU(); // pocet jadier
var TASKS = 100;
type Task struct { a, b int }      // vynasob tieto dve cisla

func worker(id int, ch <-chan Task, wg *sync.WaitGroup) {
    defer wg.Done()
    for {
        task, ok := <-ch
        if !ok { return }           // ak došla robota
        result := task.a*task.b;
        time.Sleep(time.Duration(math.Log2(float64(result))) * time.Millisecond)
    }
}

func pool(wg *sync.WaitGroup) {
    ch := make(chan Task)
    for i:=0; i<WORKERS; i++ { go worker(i, ch, wg) }
    for i:=0; i<TASKS; i++ { for j:=0; j<TASKS; j++ { ch<-Task{i, j} } }
    close(ch)                       // násobíme i*j, i,j in [1..TASKS]
}
```

Robotníci a lopaty

(workers & worker pool)

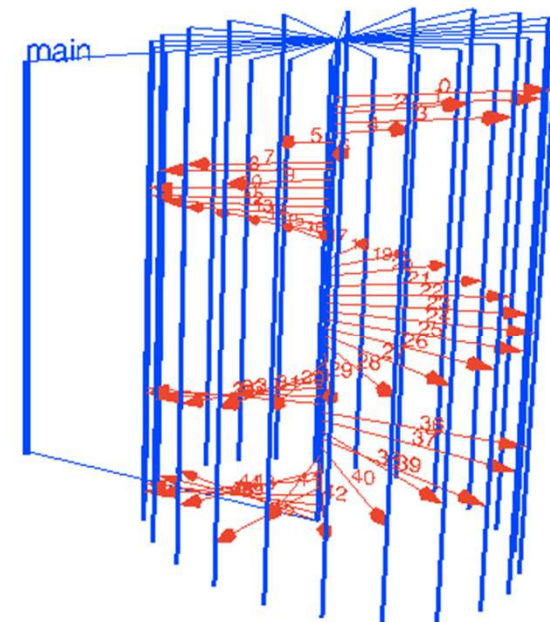
```
var WORKERS    = runtime.NumCPU(); // pocet jadier
var TASKS = 100;
type Task struct { a, b int }      // vynasob tieto dve cisla

func worker(id int, ch <-chan Task, wg *sync.WaitGroup) { ... }
func pool(wg *sync.WaitGroup) { ... }
func main() {
    var wg sync.WaitGroup
    wg.Add(WORKERS)
    go pool(&wg)
    wg.Wait()
}
```

Výplata:

\$1243	Bits 12205
\$1248	Bits 12237
\$1253	Bits 12233
\$1259	Bits 12241
\$1245	Bits 12195
\$1248	Bits 12213
\$1245	Bits 12184
\$1259	Bits 12200

100*100





Klobúky ako predjedlo

(čo to má s programovaním pochopíte dnes)



- 3 biele a 2 čierne
- A, B, C si navzájom vidia farby klobúkov
- nesmú komunikovať, ale (aj tak) sú inteligentní 😊
- vyhrávajú, ak **všetci** uhádnu farbu svojho klobúka
- resp. ak sa jeden pomýli, prehrali všetci.

Hint: A,B,C sú spoluhráči, preto predpokladaj,
že sú chytrí a myslí aj za nich

Hint: úloha nie je o šťastí=hádaní správneho riešenia

Algoritmus

1. Vidíš dve zelené hovoríš biela (výchlo)
2. Vidíš jednu bielu a jednu zelenú, čakáš či niekto niečo povie (biela) ak nie, si biela
3. Vidíš dve biele a nikto nič nehovorí si biela ďalší povie biela (lebo vie, že ten, ktorý povedal, že je biela vidí dve biele) tretí človek vie, že keď druhý človek odpovedal výchlo je zelená ak pomaly je biela.

1. Vidíš dve zelené kričíš (hneď) biela. hneď = po 1 sekunde.
2. Vidíš jednu bielu a jednu zelenú, čakáš či niekto niečo povie, ak nie, si biela
ak áno, si zelená
3. Vidíš dve biele a nikto nič dlho nehovorí, si biela. dlho = 30 sekúnd
ak hovorí, si zelená



Do 10 sekúnd



ak vidím dva čierne, *určite mám biely*, a preto sa
hned' ozvem, že "**mám biely**".

ak sa niekto do 10s ozval, že má biely, *musí vidieť*
dva čierne, preto ja mám čierny,
tak hned' kričím "**mám čierny**".

inak čakám 10s, *nikto neozval, že „mám biely“*,
preto určite nie sú v hre 2 čierne, ale najviac
jeden čierny !!!



10 až 20 sekúnd



v hre je najviac jeden čierny

ak teda vidím čierny, *ja musím mať biely*, tak sa ozvem
hneď, že mám "**mám biely**".

inak, ak sa ozvú dvaja (do 10 s), že biely, *ja mám čierny*,
tak kričím "**mám čierny**".



inak, *nevidím čierny a nikto sa neozval*, čakám ďalších 10s,



po 20 sekundách



v hre nie je žiaden čierny

*keďže sa nikto neozval, tak nie je žiaden čierny, tak kričím "**mám biely**" a ostatní tiež*



Celý algoritmus

(bez vysvetlenia, už pre cvičenú opicu)

Hneď:

ak vidím dva čierne, hneď ozvem, že "**mám biely**".

ak sa niekto do 10 s ozval, tak kričím "**mám čierny**".

inak čakám 10s.

Po 10 sek:

ak vidím čierny, tak hneď kričím "**mám biely**".

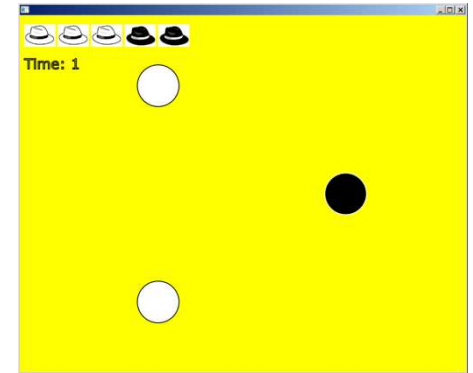
inak, ak sa ozvú dvaja do 10 s, tak kričím "**mám čierny**".

inak čakám ďalších 10s,

Po 20 sek:

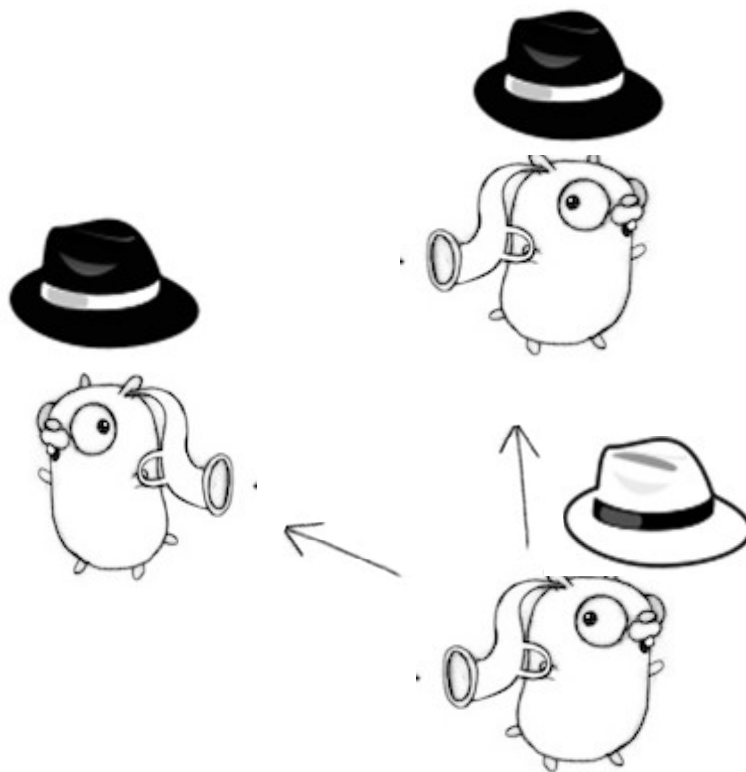
kričím "**mám biely**"

Na zamyslenie



- je podstatné, či kričím *mám biely/mám čierne*, nestačí len **už viem** ?!
- Dalo by sa to pre 3 biele, 3 čierne, 3 ľudia ?
- Dalo by sa to pre 2 biele, 1 čierne, 2 ľudia ?
- Dalo by sa to pre N biele, $(N-1)$ čierne, N ľudia ? (napr. 6,5,6)
- Dalo by sa to pre $>N$ biele, $(N-1)$ čierne, N ľudia ? (napr. 8,5,6)
- Dalo by sa to pre N biele, $<(N-1)$ čierne, N ľudia ? (napr. 6,4,6)

Komunikácia – každý s každým





Správa, kanály, agenti

```
type Message struct {  
    who int    // od koho, odosielateľ  
    what int } // čo, obsah správy  
func makeChannels(n int) []chan Message {  
    chArray := make([]chan Message, n)  
    for i:= 0; i < n; i++ { // kanál, na ktorom počúva i-ty agent  
        chArray[i] = make(chan Message)  
    }  
    return chArray  
}  
func main() {  
    chArray := makeChannels(numb)  
    for a:= 0; a<numb; a++ {  
        runAgent(a, chArray)  
    }  
}
```



Agenti napriamo

```
func runAgent(agent int, channels []chan Message) {
    go func() {          // ID agenta, kanaly na vsetkych agentov
        i := 1           // iniciálny stav agenta
        for {            // loop forever
            timeout := time.After(...)
            select {
                case msg := <- channels[agent]: // agent počúva len svoj
                    fmt.Printf("agentovi %d: prišla správa:%s", agent, msg)
                case <-timeout: // prešiel timeout, vyrobíme správu msg
                    msg := Message{who:agent, what:i++} //zmeníme svoj stav
                    for index, ch := range channels { // povedz každému
                        if index != agent {           // okrem seba
                            go func(ch chan Message) { // !!!!!!!!!!!!!!!!!!!!!
                                cha <- msg              // správu msg
                            }(ch)
                        }
                    }
                }
            }
        }
    }()
```



Agenti napriamo

```
func runAgent(agent int, channels []chan Message) {
    go func() {          // ID agenta, kanaly na vsetkych agentov
        i := 1           // iniciálny stav agenta
        for {            // loop forever
            timeout := time.After(...)
            select {
                case msg := <- channels[agent]: // agent počúva len svoj
                    fmt.Printf("agentovi %d: prišla správa:%s", agent, msg)
                case <-timeout: // prešiel timeout, vyrobíme správu msg
                    msg := Message{who:agent, what:i++} //zmeníme svoj stav
                    for index, ch := range channels { // povedz každému
                        if index != agent {           // okrem seba
                            go func() { // !!!!! ZLE !!!!!
                                ch <- msg           // správu msg
                            }()
                        }
                    }
                }
            }
        }
    }()
```



Príklad komunikácie 3 agentov

1: povedal 1

agentovi 2: prisla sprava:"1: povedal 1"

agentovi 0: prisla sprava:"1: povedal 1"

0: povedal 1

agentovi 2: prisla sprava:"0: povedal 1"

agentovi 1: prisla sprava:"0: povedal 1"

1: povedal 2

agentovi 2: prisla sprava:"1: povedal 2"

0: povedal 2

1: povedal 3

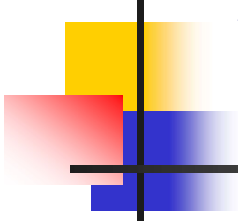
agentovi 0: prisla sprava:"1: povedal 2"

agentovi 0: prisla sprava:"1: povedal 3"

agentovi 1: prisla sprava:"0: povedal 2"

agentovi 2: prisla sprava:"0: povedal 2"

agentovi 2: prisla sprava:"1: povedal 3"



Aplikácia na klobúky

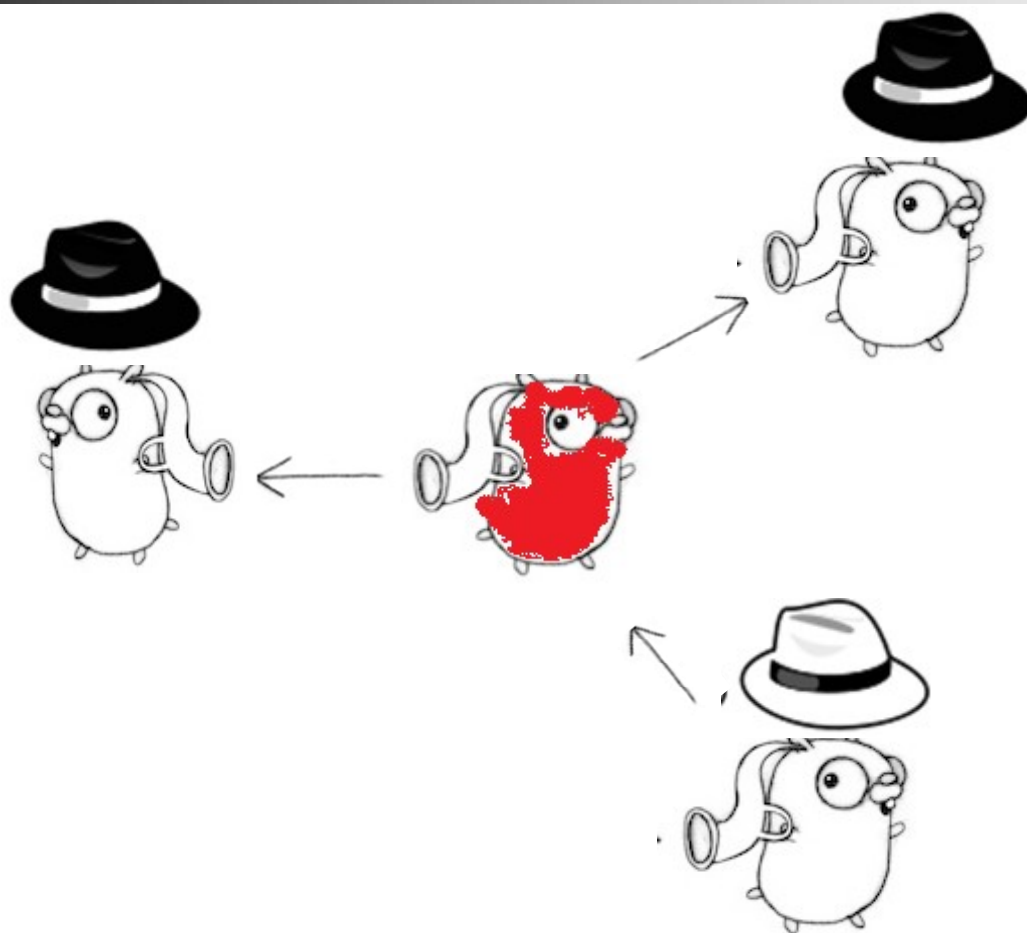
(domáca úloha)

```
func vidim(name String) (int, int) {
```

- [0s] A: vidim 1 biele a 1 cierne
- [0s] A: cakam 10 sek
- [0s] C: vidim 1 biele a 1 cierne
- [0s] C: cakam 10 sek
- [0s] B: vidim 2 biele a 0 cierne
- [0s] B: cakam 10 sek

- [10s] B: cakam dalsich 10 sek
- [11s] A: mam biely !!! true
- [11s] C: mam biely !!! true
- [11s] B:: prisla sprava, ze [11s] A: mam biely !!! true
- [12s] B: mam cierny !!! true
- finito

Komunikácia s dispečerom





Dispatcher

čo počujete to prepošle

```
func runDispatcher(channels []chan Message) chan Message {
    dispch := make(chan Message)
    // kanál na komunikáciu s dispatcherom
    go func() {
        for {
            msg := <- dispch // ak prišla správa
            fmt.Println("dispecer sa dozvedel: " + msg.toString())
            for _,ch := range channels {
                go func(x chan Message) {
                    x <- msg
                }(ch)
            }
        }
    }()
    return dispch }
```



Agenti cez dispečera

```
func runAgentCommunicatingWithDispatcher(agent int,
    dispch chan Message, input chan Message) {
    go func() {
        i := 1 // stav agenta
        for {
            timeout := time.After(...) // náhodny delay
            select {
                case msg := <- input: // ak prišla správa agentovi,
                    fmt.Printf("agentovi %d: prisla sprava:%s", agent, msg)
                case <-timeout: // po timeout, vytvoríme správu
                    msg := Message{who:agent, what:i}
                    dispch <- msg // pošleme dispecerovi
                    i++ // agent si zvýši stav
            }
        }
    }()
}
```



Agenti cez dispečera

```
func runAgentCommunicatingWithDispatcher(agent int,
    dispch chan Message, input chan Message) {
    go func() {
        i := 0 // stav agenta
        for {
            timeout := time.After(...) // náhodny delay
            select {
                case msg := <- input: // ak prišla správa agentovi,
                    fmt.Printf("agentovi %d: prisla sprava:%s", agent, msg)
                case <-timeout: // po timeout, vytvoríme správu
                    msg := Message{who:agent, what:i}
                    go func() { dispch <- msg }() // pošleme dispecerovi
                    i++ // agent si zvýši stav
            }
        }
    }()
}
```



Príklad komunikácie 3 agentov

1: povedal 0

dispecer sa dozvedel: 1: povedal 0

agentovi 2: prisla sprava:"1: povedal 0"

agentovi 0: prisla sprava:"1: povedal 0"

agentovi 1: prisla sprava:"1: povedal 0"

0: povedal 0

dispecer sa dozvedel: 0: povedal 0

agentovi 2: prisla sprava:"0: povedal 0"

agentovi 0: prisla sprava:"0: povedal 0"

agentovi 1: prisla sprava:"0: povedal 0"

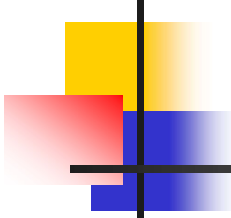
0: povedal 1

dispecer sa dozvedel: 0: povedal 1

agentovi 2: prisla sprava:"0: povedal 1"

agentovi 0: prisla sprava:"0: povedal 1"

agentovi 1: prisla sprava:"0: povedal 1"



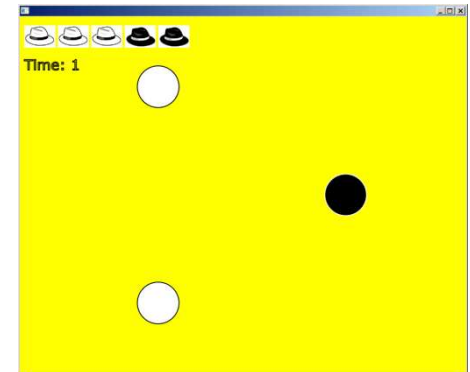
Aplikácia na klobúky

(domáca úloha)

```
func vidim(name String) (int, int) {
```

- [0s] A: vidim 1 biele a 1 cierne
- [0s] A: cakam 10 sek
- [0s] B: vidim 2 biele a 0 cierne
- [0s] B: cakam 10 sek
- [0s] C: vidim 1 biele a 1 cierne
- [0s] C: cakam 10 sek
- [10s] B: cakam dalsich 10 sek
- [11s] A: mam biely !!! true
- od A prisla sprava, ze [11s] A: mam biely !!! true
- [11s] C: mam biely !!! true
- [11s] B:: prisla sprava, ze [11s] A: mam biely !!! True
- od C prisla sprava, ze [11s] A: mam biely !!! true
- od B prisla sprava, ze [11s] A: mam biely !!! true
- od B prisla sprava, ze [11s] C: mam biely !!! true
- [12s] B: mam cierny !!! True
- finito

Riešenia



- je podstatné, či kričím mám biely/mám čierne, stačí len „už viem“ ?!
áno
- Dalo by sa to pre 3 biele, 3 čierne, 3 ľudia ?
nie
- Dalo by sa to pre 2 biele, 1 čierne, 2 ľudia ?
áno
- Dalo by sa to pre N biele, $(N-1)$ čierne, N ľudia ? (napr. 6,5,6)
áno
- Dalo by sa to pre $>N$ biele, $(N-1)$ čierne, N ľudia ? (napr. 8,5,6)
áno
- Dalo by sa to pre N biele, $<(N-1)$ čierne, N ľudia ? (napr. 6,4,6)
áno



Fibonacciho agenti

(cvičenie)

Vyrobíme niekoľko nezávislých agentov, ktorí

- `zipf(ch1, ch2 chan int, f func(int, int) int) chan int`
spája dvojice prvkov z kanála ch1 a ch2 pomocou funkcie f (u nás +)
- `tail(ch1 chan int) chan int`
číta z kanála ch1, priamo píše do výstupu, akurát prvý prvok z ch1 zabudne
- `func fib1() chan int`
podivným spôsobom generuje fibonacciho čísla...
aj to len trochu...
- `splitter(ch chan int) (ch1 chan int, ch2 chan int)`
číta z ch, a výsledky konkurentne kopíruje do ch1 aj ch2



Agent zip

(cvičenie)

```
func zipf(ch1, ch2 chan int, f func(int, int) int) chan int {  
    ch := make(chan int)  
    zipCount++  
    go func() {  
        for {  
            f1 := <-ch1    // číta dvojice f1  
            f2 := <-ch2    // f2 z ch1 a ch2  
            ch <- f(f1, f2) // píše f(f1, f2), alias f1+f2  
        }  
    }()  
    return ch  
}
```



Agent tail

(cvičenie)

```
func tail(ch1 chan int) chan int {  
    ch := make(chan int)  
    tailCount++  
    <-ch1                                // prvý prvok zabudne  
    go func() {  
        for {  
            ch <- <-ch1  
        }  
    }()  
    return ch  
}
```



Agent fib1

(katastrofické výsledky)

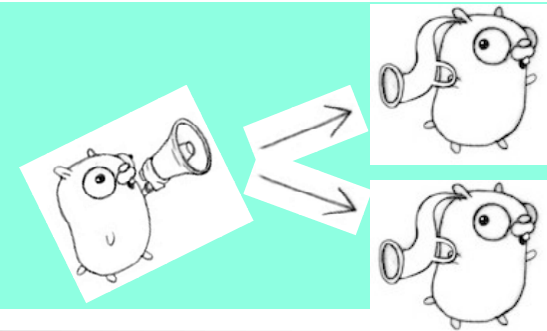
```
func fib1() chan int {
    ch := make(chan int)
    fibCount++
    go func() {
        ch <- 1
        ch <- 1
        for val := range zipf(fib1(), tail(fib1())),
            func(x, y int) int { return x + y }) {
            ch <- val
        }
    }()
    return ch
}
```

fib (fibCount, zipCount, tailCount)

1	(1,0,0)
1	(1,0,0)
2	(7,1,3)
3	(23,7,11)
5	(63,31,31)
8	(255,71,127)
13	(1023,255,511)
21	(2111,1023,1055)
34	(8191,4095,4095)

Agent splitter

(cvičenie)



```
func spliter(ch chan int) (ch1 chan int, ch2 chan int) {  
    ch1 = make(chan int)  
    ch2 = make(chan int)  
    spliterCount++  
    go func() {  
        for {  
            val := <-ch  
            // ch1 <- val deadlock! why ?  
            // ch2 <- val  
            go func() { ch1 <- val }()  
            go func() { ch2 <- val }()  
        }  
    }()  
    return ch1, ch2  
}
```



Agent fib

(prijateľné výsledky ?)

```
func fib() chan int {
    ch := make(chan int)
    fibCount++
    go func() {
        ch <- 1
        ch <- 1

        ch1, ch2 := splitter(fib()) // použitie splittera
        for val := range zipf(ch1, tail(ch2),
            func(x, y int) int { return x + y }) {
            ch <- val
        }
    }()
    return ch
}
```

```
1 (1,0,0, 0)
1 (1,0,0, 0)
2 (5,2,4, 4)
3 (8,6,7, 7)
5 (12,9,11, 11)
8 (15,13,14, 14)
13 (19,16,18, 18)
21 (22,20,21, 21)
.....
40.Fibonacciho číslo
165580141 (138,135,137, 137)
Success: process exited with
code 0.
```

Input:

F_{41}

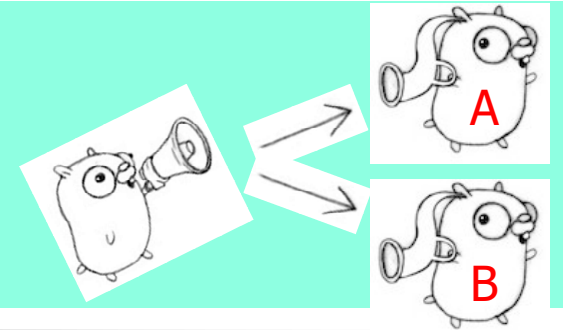
Result:

165580141

fibStream.go

Agent splitter

(cvičenie)



```
func splitter(ch chan int) (ch1 chan int, ch2 chan int) {  
    ch1 = make(chan int)  
    ch2 = make(chan int)  
    splitterCount++  
    go func() {  
        for {  
            val := <-ch  
            select {  
                case ch1 <-val:  
                case ch2 <-val:  
            }  
        }  
    }()  
    return ch1, ch2  
}
```

B:2
A:1
B:3
B:4
B:5
B:6
B:7
B:8
B:9
B:10
B:11
A:12
B:13
B:14
B:15