go get my/vulnerabilities

Green threads are not eco friendly threads

Who

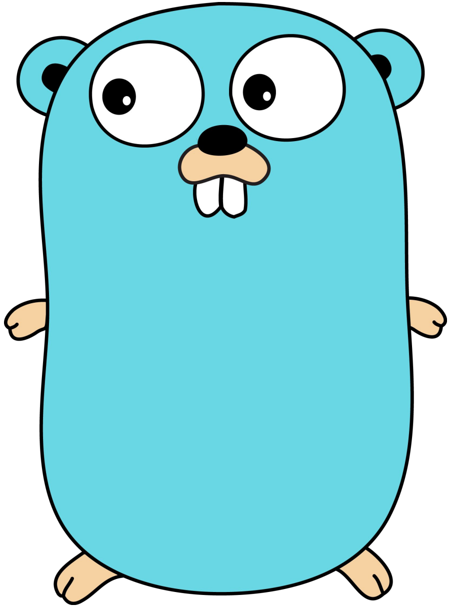
- (Web|Mobile) penetration tester
- Code reviewer
- Programmer

Roberto Clapis

@empijei

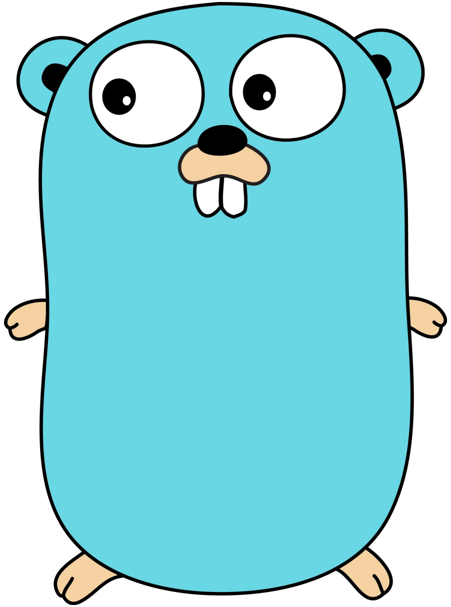
Go

- Google’s language
- Born in 2007 (quite new)
- Widespread



Cool, but how do I break it?

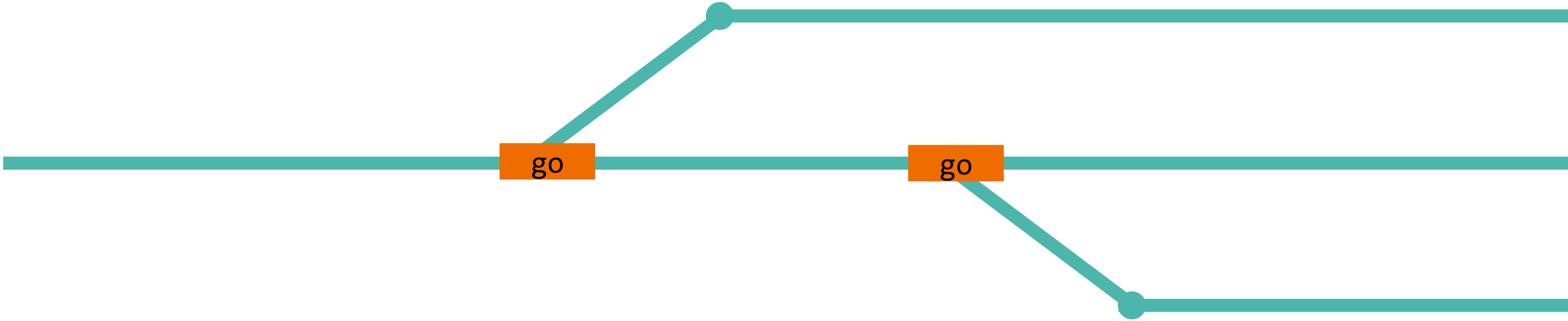
- Memory safety, Garbage Collection
- Anti-XSS/SQLi sanitization
- Built-in thread-safe constructs



Let’s start the digging

- New features usually lead to new vulnerabilities
- Goroutines are one of the main new features introduced by Go

Goroutines are concurrent function calls



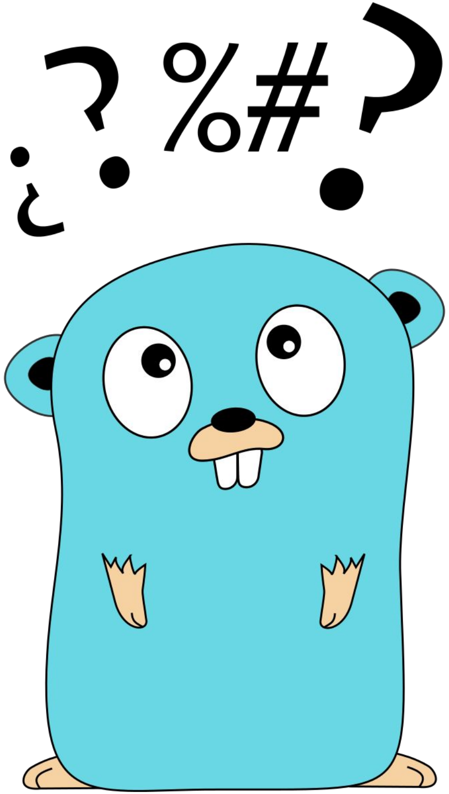
```
go fmt.Println(“Hello goroutines”)
```

Let’s try this

```
for i := 0; i <= 9; i++ {
  go func() {
    fmt.Println(i)
  }()
}
```

Expectation	Reality
1	10
3	10
2	10
...	...
8	10
9	10

Wait...



Special functions #1: goroutines

- Concurrent
- Lightweight
- Multiplexed on OS Threads

```
go func(){
  //Code here
}()
```

Special functions #2: closures

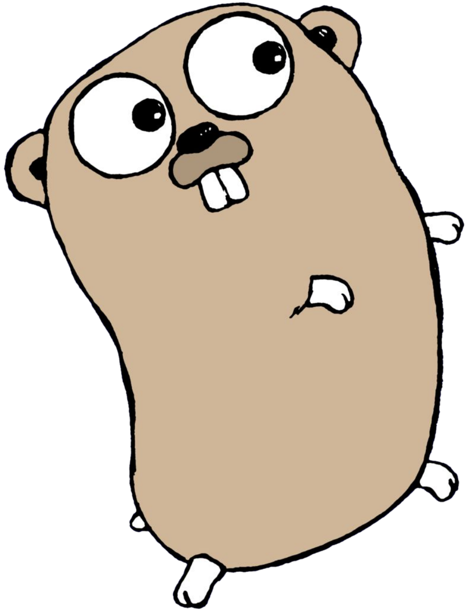
```
freeVar := “Hello ”
f := func(s string){
      fmt.Println(freeVar + s)
      }
f(“Closures”)
// Hello Closures
```

Special functions all together

```
 for i := 0; i <= 9; i++ {
  go func() {
   fmt.Println(i)
  }()
 }
// Here i == 10
```

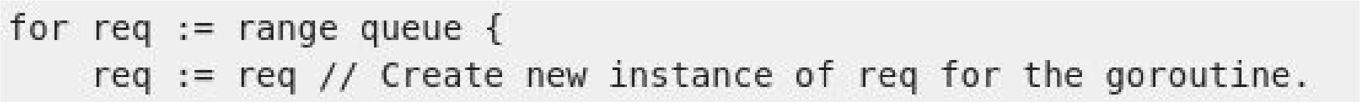
Performance

- Writing to file is slow
- Aware scheduling
- Runtime waits only if necessary



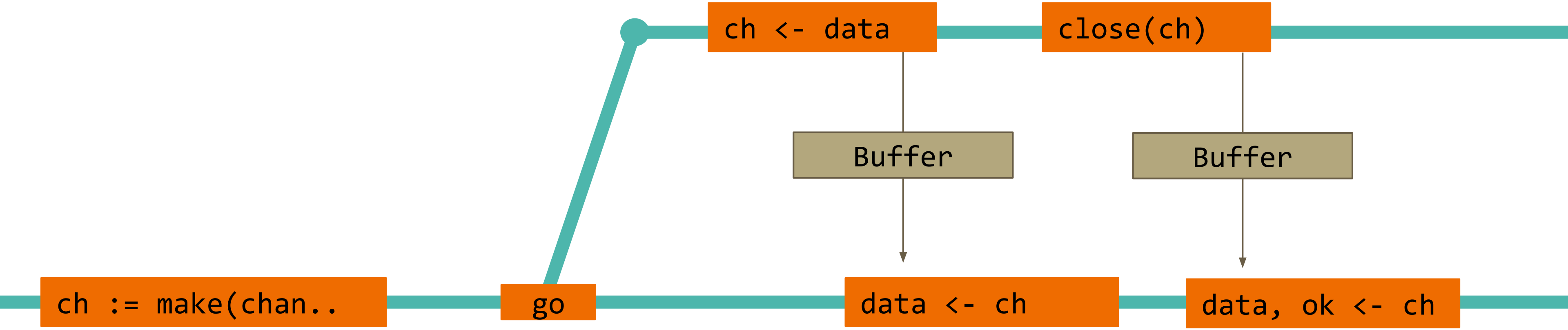
The (odd) fix

```
for i := 0; i <= 9; i++ {
    i := i
    go func() {
        fmt.Println(i)
    }()
```



```
for req := range queue {
    req := req // Create new instance of req for the goroutine.
```

Channels



```
for data := range ch {
```

Information Leakage

```
func Serve(queue chan *http.Request) {
    for req := range queue {
        go func() {
            process(req)
        }()
    }
}
```

responses to the wrong requests

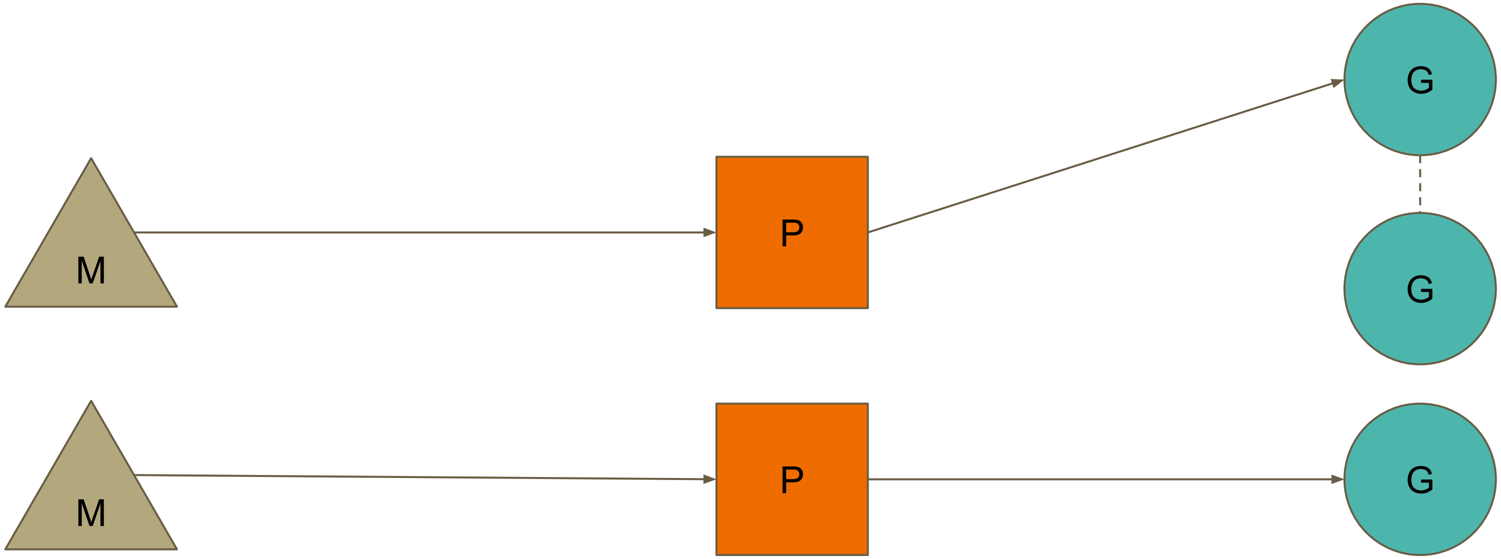
Checkpoint

- **Variable scoping** is a nice point to focus on

- **Aware** scheduling can make it easier to abuse races

how aware is the scheduler?

MPG model

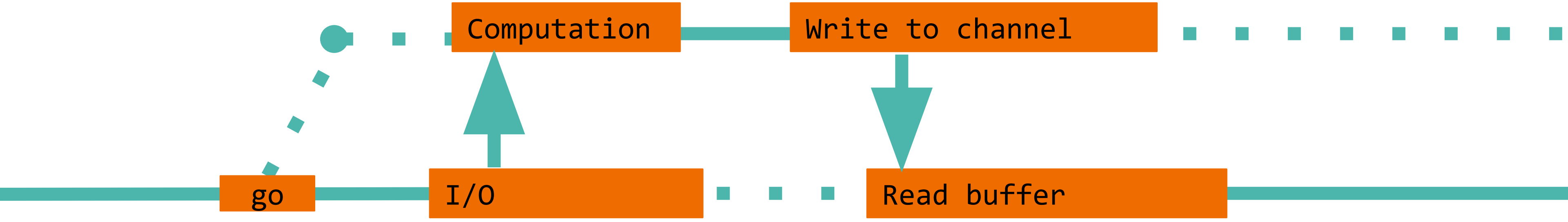


OS Threads

GOMAXPROCS
Logical processors

Goroutines

Schedule me please

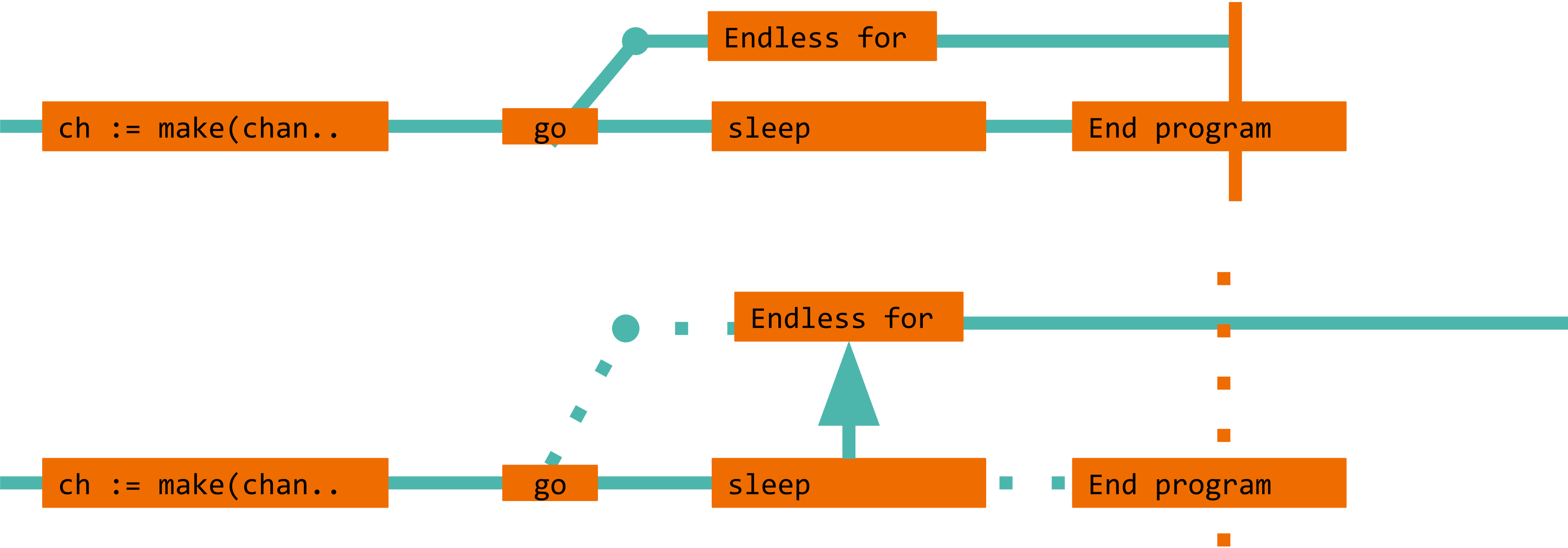


Scheduler calls are emitted at **compile time**

Consequences are weird

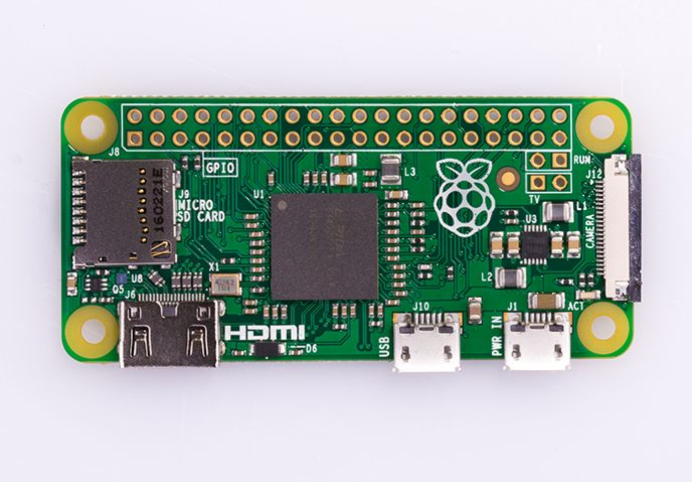
```
go func() {
  for i := 0; true ; i++ {
  }
}()
time.Sleep(2 * time.Second)
fmt.Println("Done")
```

Cores amount matter



Runs the same way everywhere...

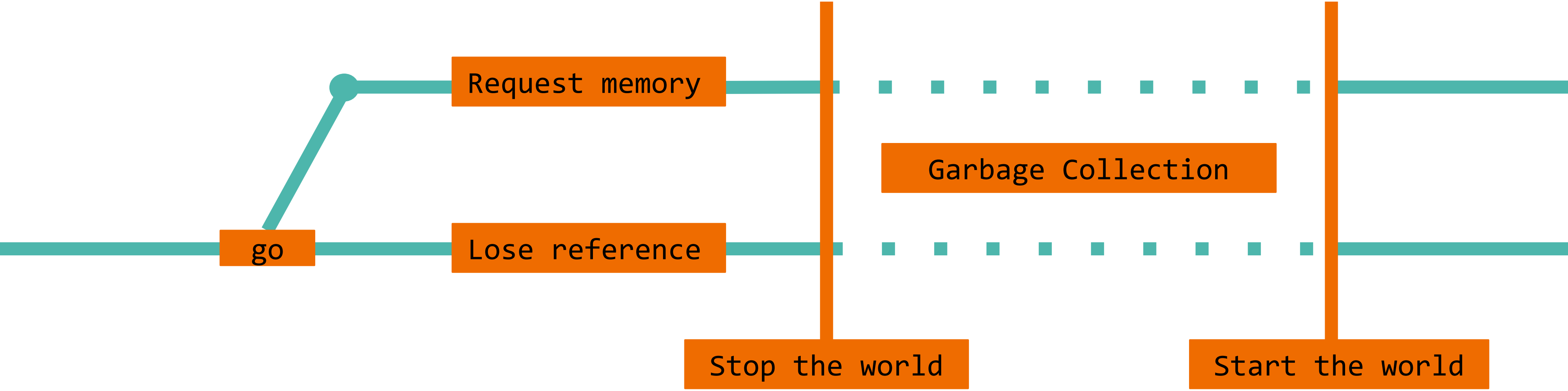
```
runtime.GOMAXPROCS(1)
```



Statically Strongly Typed

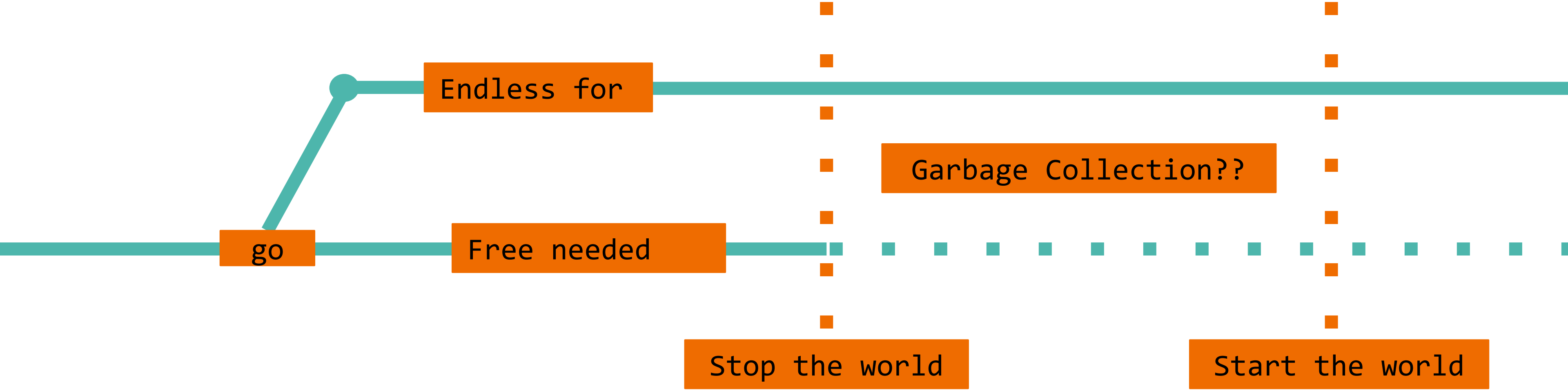
```
go func() {
 for i := range lst {
  for ; i <= 255 ; i++ {
   // Computation
  }
 }
}()
```

Hidden problem: Garbage Collector



Garbage collector needs to stop goroutines

Garbage Collection?



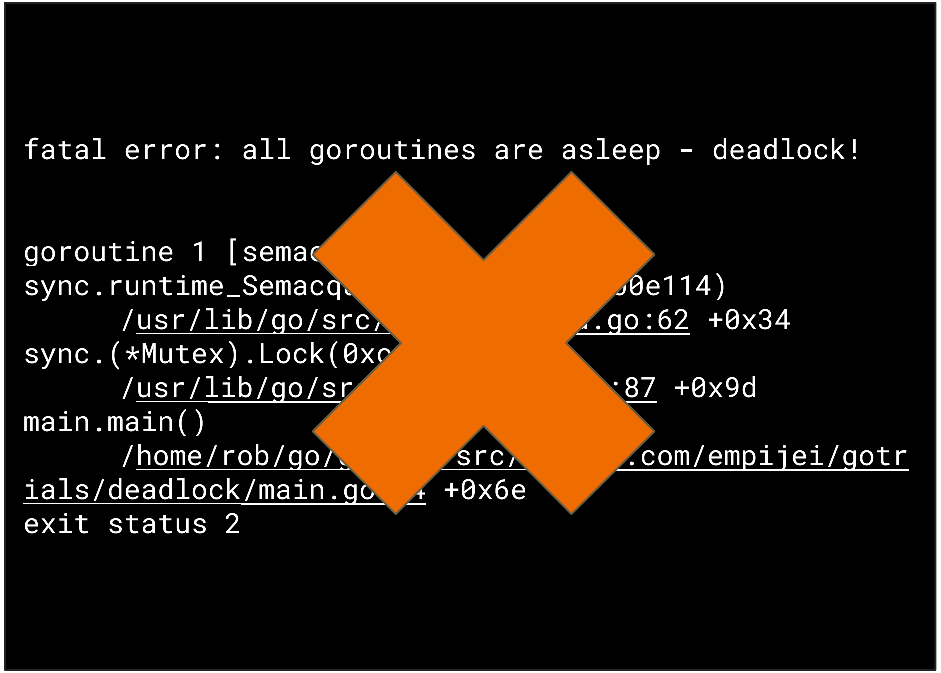
GC politely asks goroutines to stop

Consequences are bad

```
go func() {
    var i byte
    for i = 0; i <= 255; i++ {
    }
}()
runtime.Gosched() //yield execution
runtime.GC()
fmt.Println("Done")
```

Note to make it worse

Golang internal deadlock detector does not detect this deadlocks. Do not expect it to perform magic.



```
fatal error: all goroutines are asleep - deadlock!


goroutine 1 [semac
sync.runtime_Semacq                   0e114)
      /usr/lib/go/src/               .go:62 +0x34
sync.(*Mutex).Lock(0xc
      /usr/lib/go/sr                  :87 +0x9d
main.main()
      /home/rob/go/      src/         .com/empijei/gotr
ials/deadlock/main.go      +0x6e
exit status 2
```

Here is the solution

Weird, less efficient solution: use **non-inlinable function calls** in loops

The correct one: use ***channels***

Checkpoint

- **Scheduling** must be taken into account

- **Goroutines** that don’t yield have potential for DoS

how do goroutines die?

Goroutines end

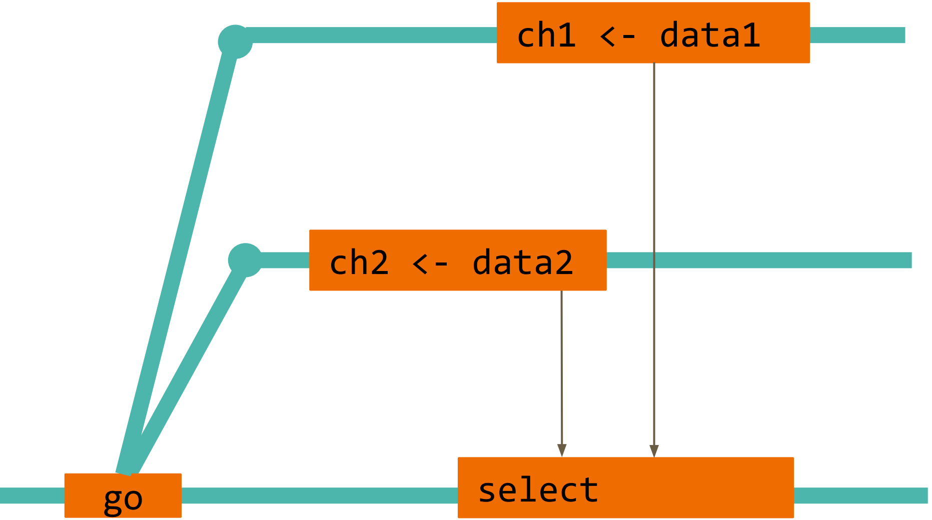
The only way for a goroutine to terminate is for it to **return**, or for **the program to end**.

Goroutines are not Garbage Collected

They **must be signalled to end** or they

constitute an insidious opening for DoS

select the right solution?

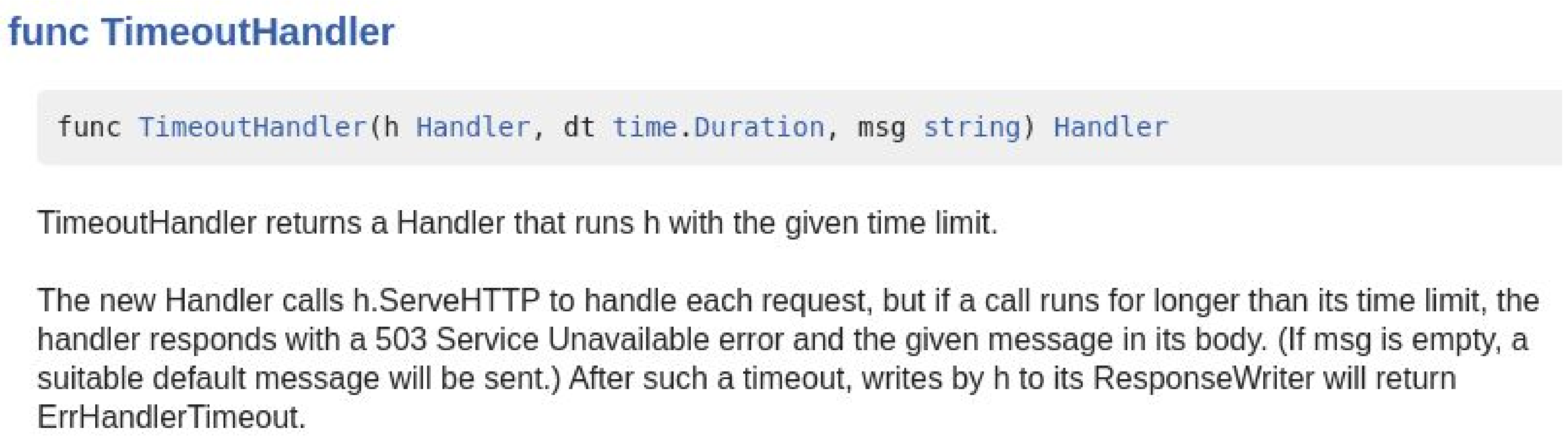


```
select {
   case d1 <- ch1:
   case d2, ok <- ch2:
   default:
}
```

Max execution time in PHP

```
<?php
   set_time_limit(2);
   for($i=0;;$i++){
   }
?>
// Maximum execution time of
// 2 seconds exceeded
```

Max execution time in go



func TimeoutHandler

```
func TimeoutHandler(h Handler, dt time.Duration, msg string) Handler
```

TimeoutHandler returns a Handler that runs h with the given time limit.

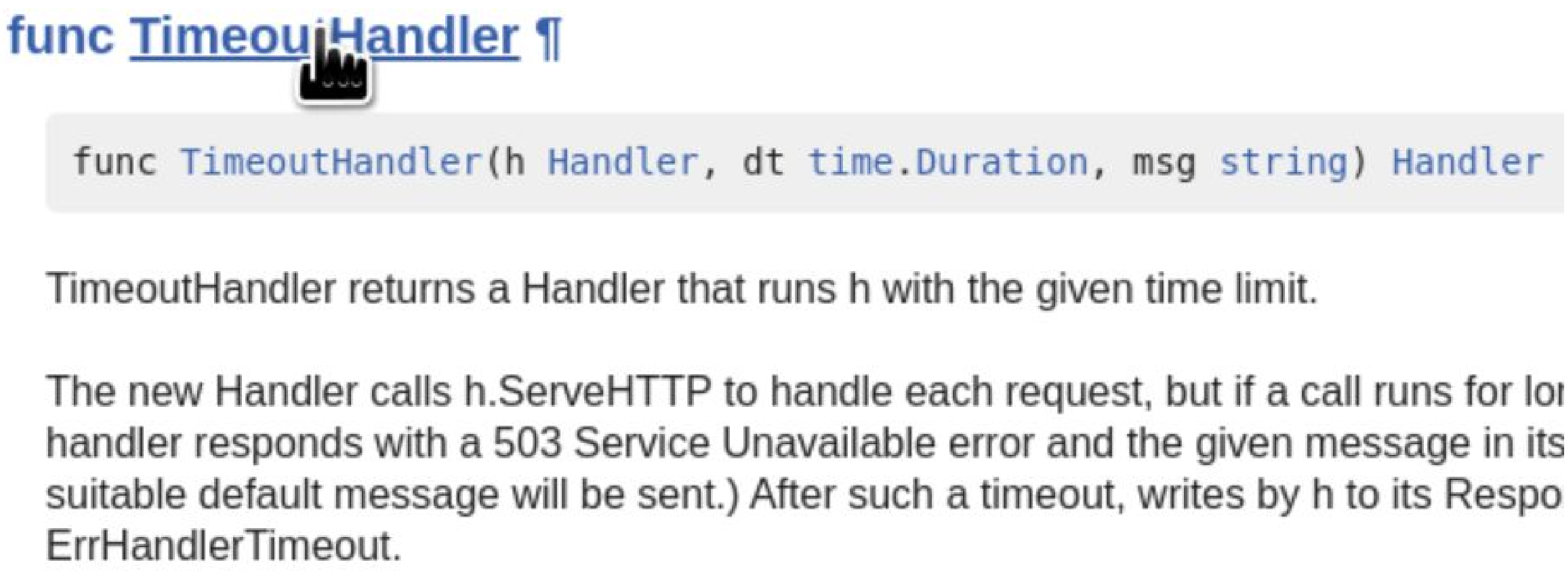
The new Handler calls h.ServeHTTP to handle each request, but if a call runs for longer than its time limit, the handler responds with a 503 Service Unavailable error and the given message in its body. (If msg is empty, a suitable default message will be sent.) After such a timeout, writes by h to its ResponseWriter will return ErrHandlerTimeout.

So is this magic?

This is NOT PHP

```
type simpleHandler struct {
}
func (t *simpleHandler) ServeHTTP(w http.ResponseWriter,
        r *http.Request) {
    time.Sleep(10 * time.Second)
    fmt.Println("Got here")
}
func main() {
    sh := &simpleHandler{}
    tsh := http.TimeoutHandler(sh,
        time.Second*2,
        "Timeout!")
    http.ListenAndServe(":8080", tsh)
```

Just a click away



func TimeoutHandler ¶

```
func TimeoutHandler(h Handler, dt time.Duration, msg string) Handler
```

TimeoutHandler returns a Handler that runs h with the given time limit.

The new Handler calls h.ServeHTTP to handle each request, but if a call runs for lo
handler responds with a 503 Service Unavailable error and the given message in its
suitable default message will be sent.) After such a timeout, writes by h to its Respo
ErrHandlerTimeout.

Dive into sources

```
// Create timer
go func() {
    h.handler.ServeHTTP(tw, r)
    // Signal done channel
}()
select {
case <-done:
    // Handle HTTP stuff
case <-timeout:
    // Write error
}
```

Mind the gap

The standard library isn’t more powerful than you are, if you can’t kill a goroutine, neither can the standard library.

Some more problems with signals

```
// The worker goroutine
for {
    select{
        case job <- jobs:
            process(job)
        case <-done:
            return
    }
}
```

```
// The main goroutine:
go worker()
// Work needs to end:
done <- true
```

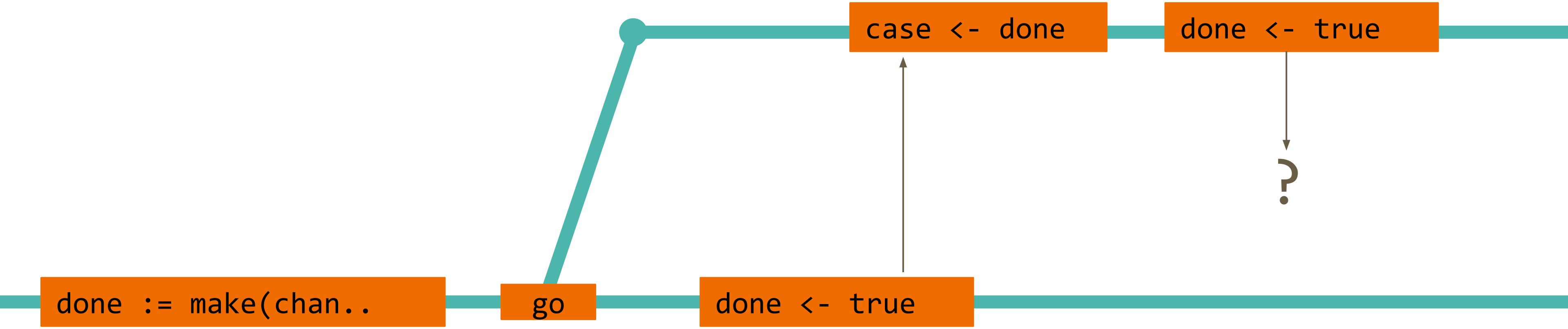
Other (still not) correct fixes

```
go worker()
go worker()
go worker()
done <- true
done <- true
done <- true
```

```
case <-done:
    done <- true
    return

go worker()
done <- true
```

Even worse



case <- done

done <- true

?

done := make(chan..

go

done <- true

Other (still not) correct fixes

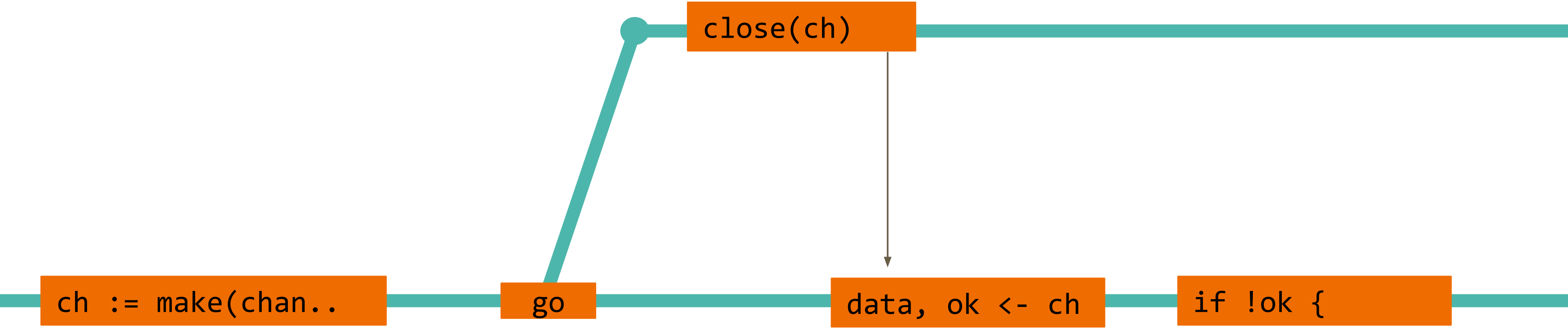
```
case <-done:
   done <- true
   return

go worker()
done <- true
<- done
```

Just close it

```
go worker()
go worker()
go worker()
close(done)
```

Close channels



```
close(ch)
```

```
ch := make(chan..
```

```
go
```

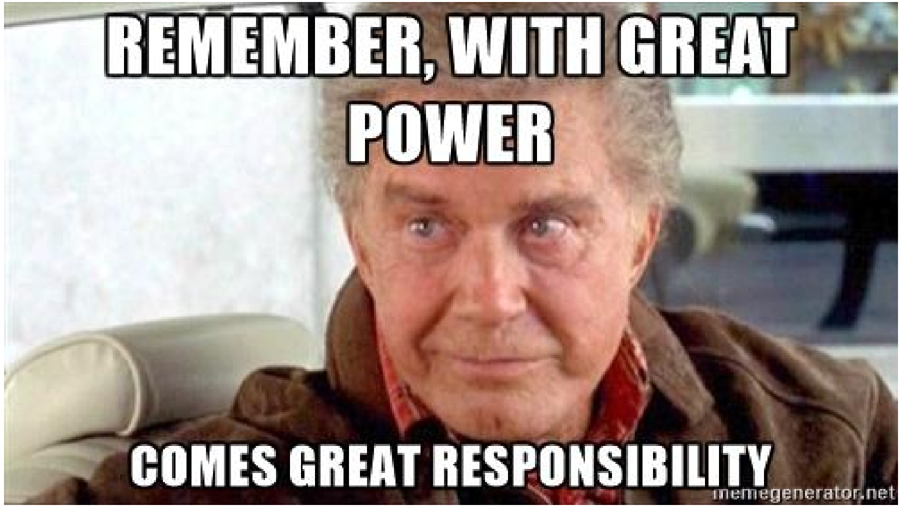
```
data, ok <- ch
```

```
if !ok {
```

```
for data := range ch {
```

Conclusions

- Mind race conditions
- Dive into sources
- Follow signals
- Check for yielding calls



Thanks

Roberto Clapis

@empijei