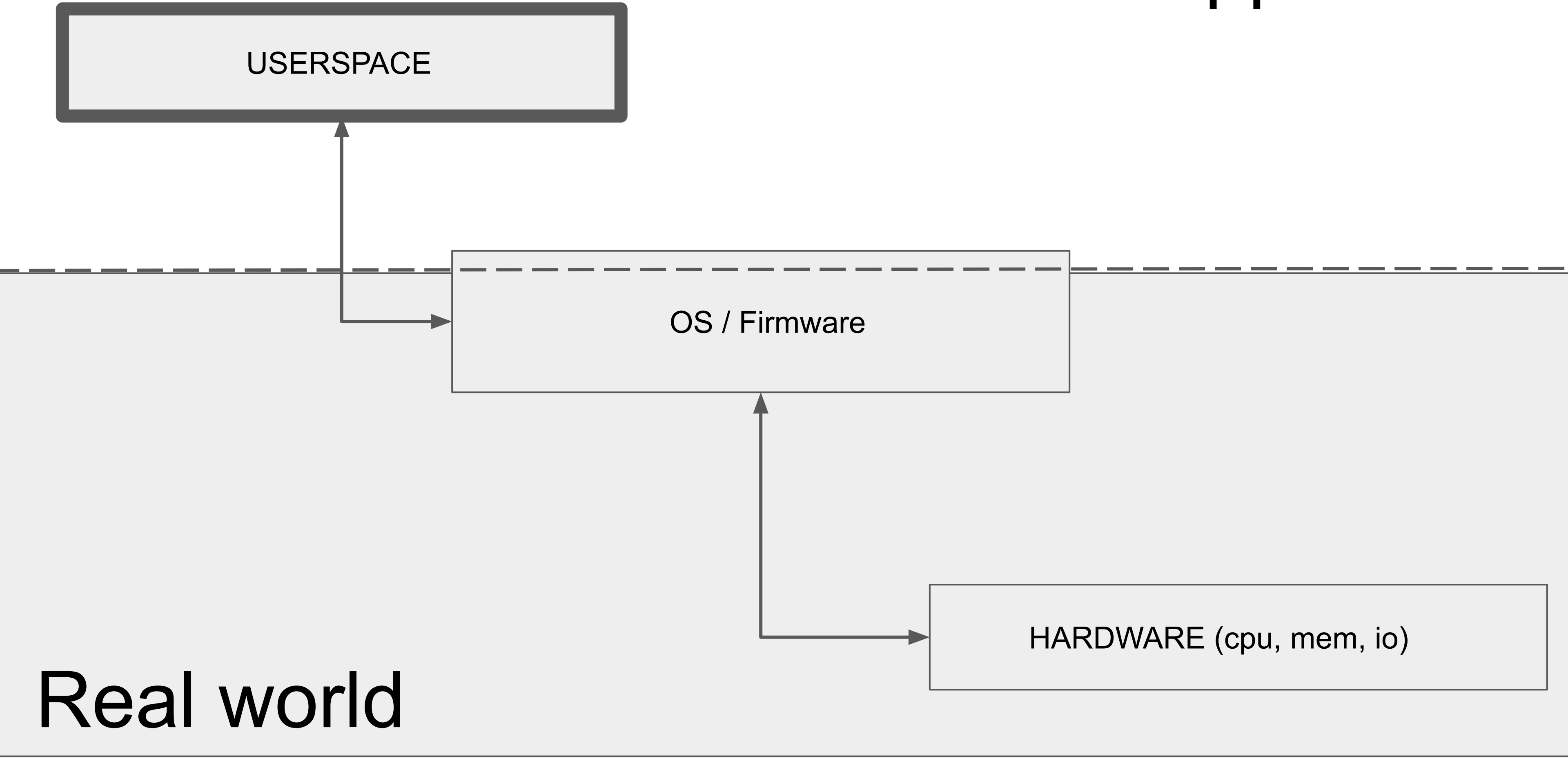
Runtime - Go’s matrix

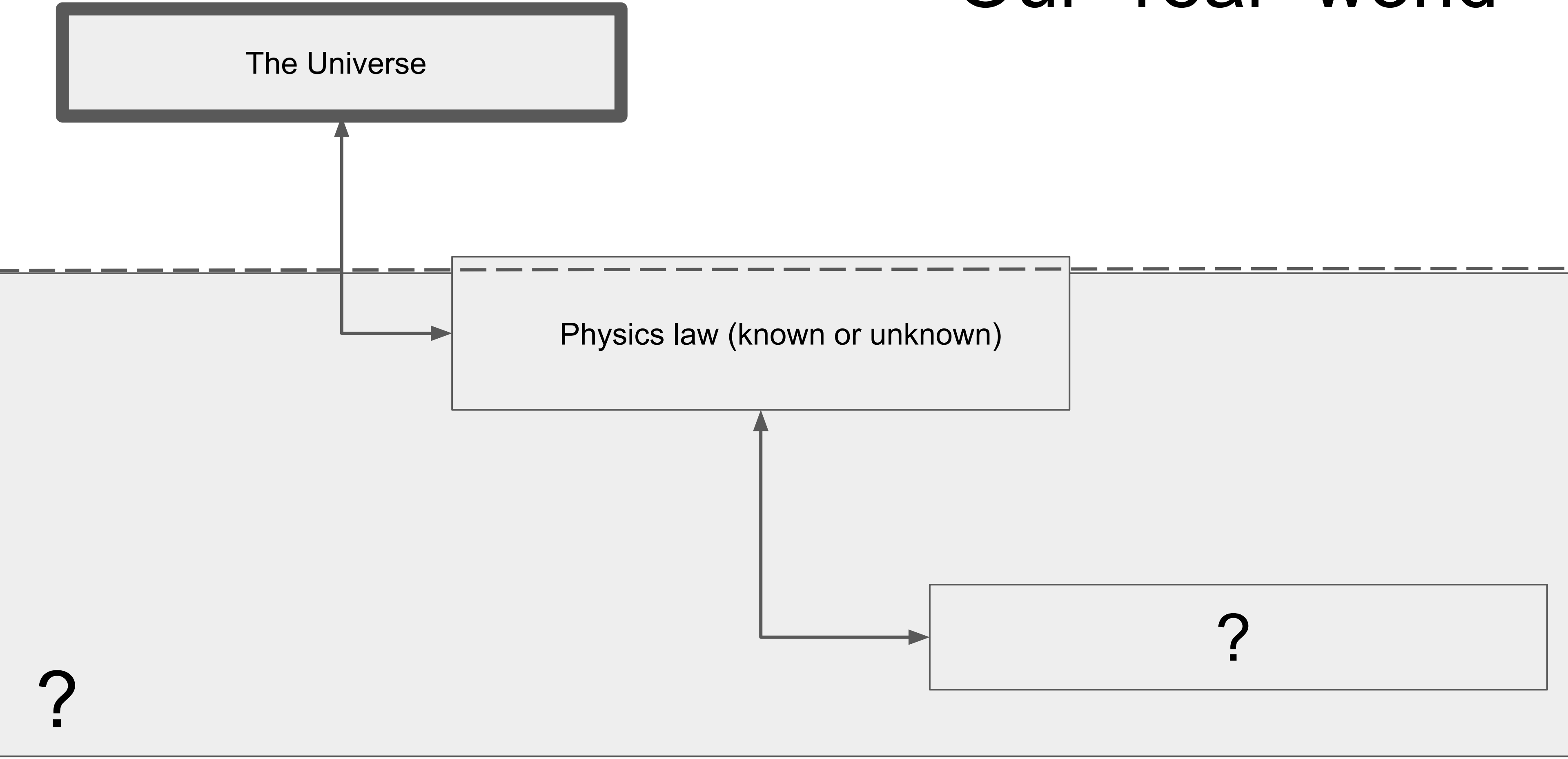
Mateusz Szczyrzyca
https://devopsiarz.pl | https://devopsiarz.pl/yt

Matrix?

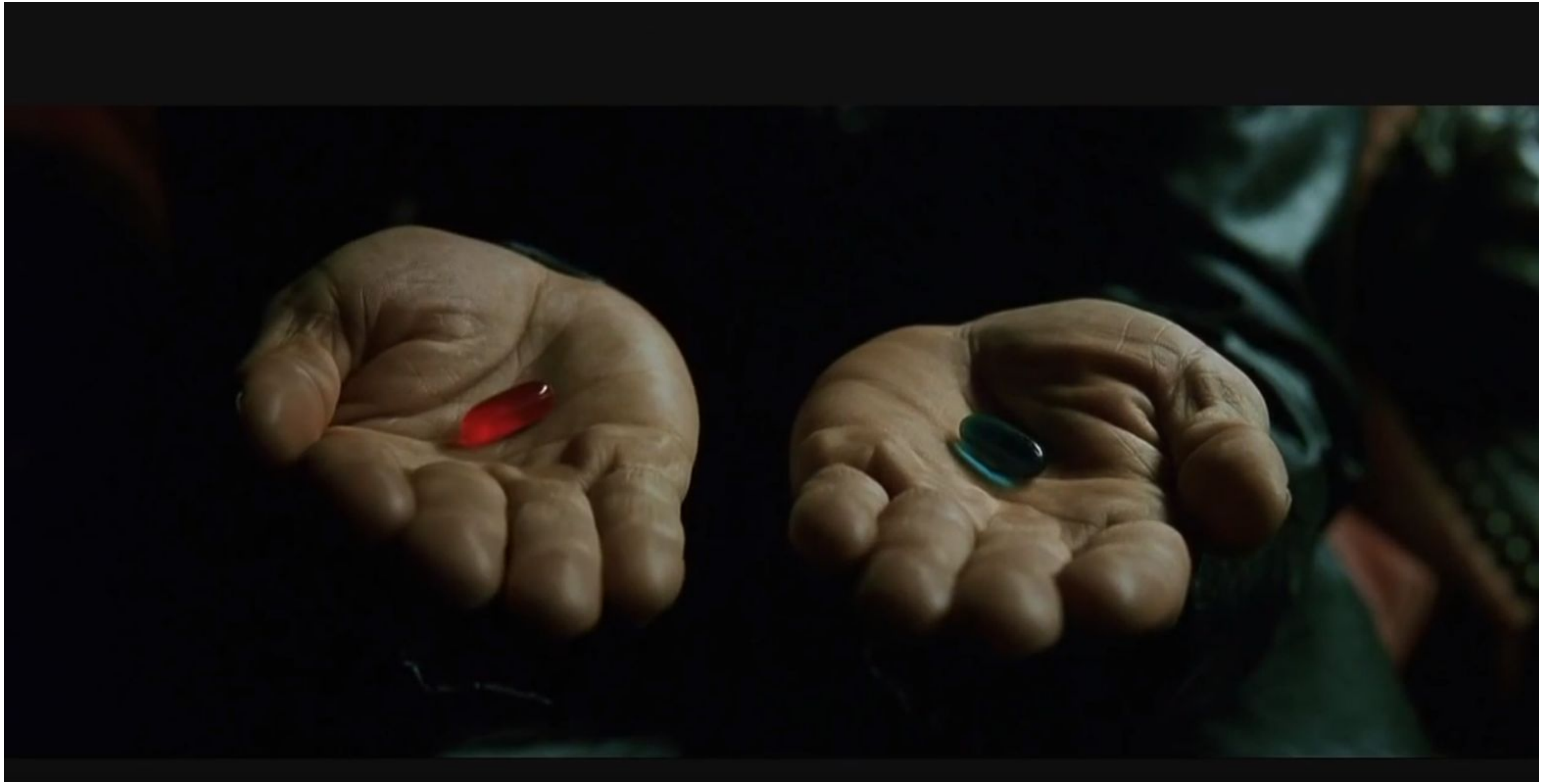
Our app matrix



Our “real” world

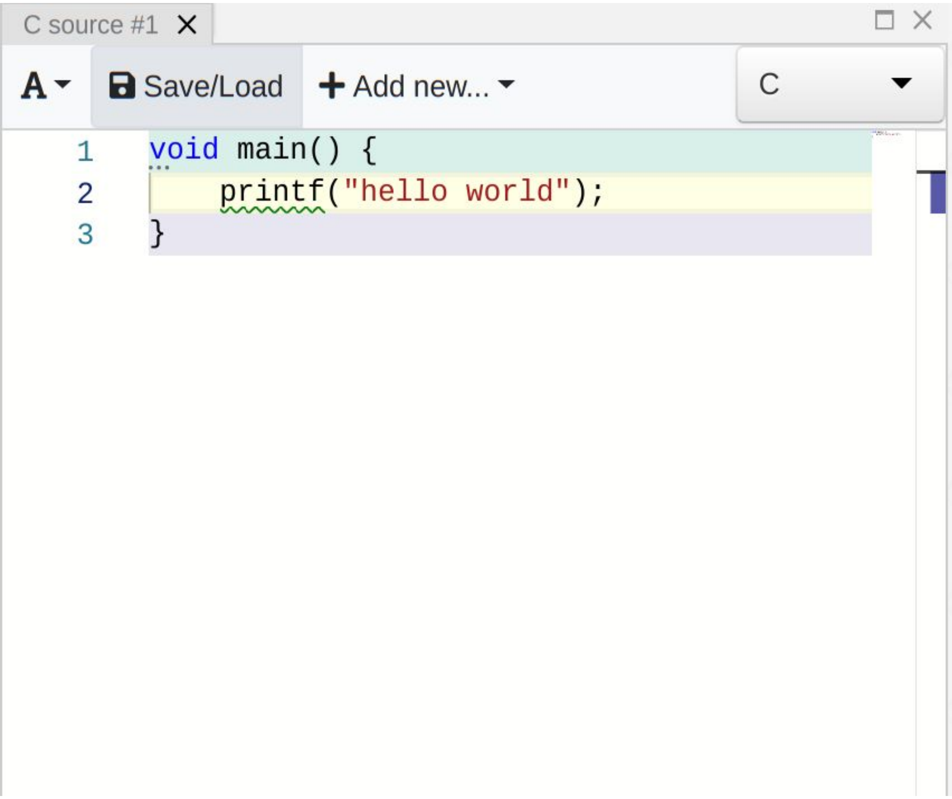


Go has it’s own Matrix

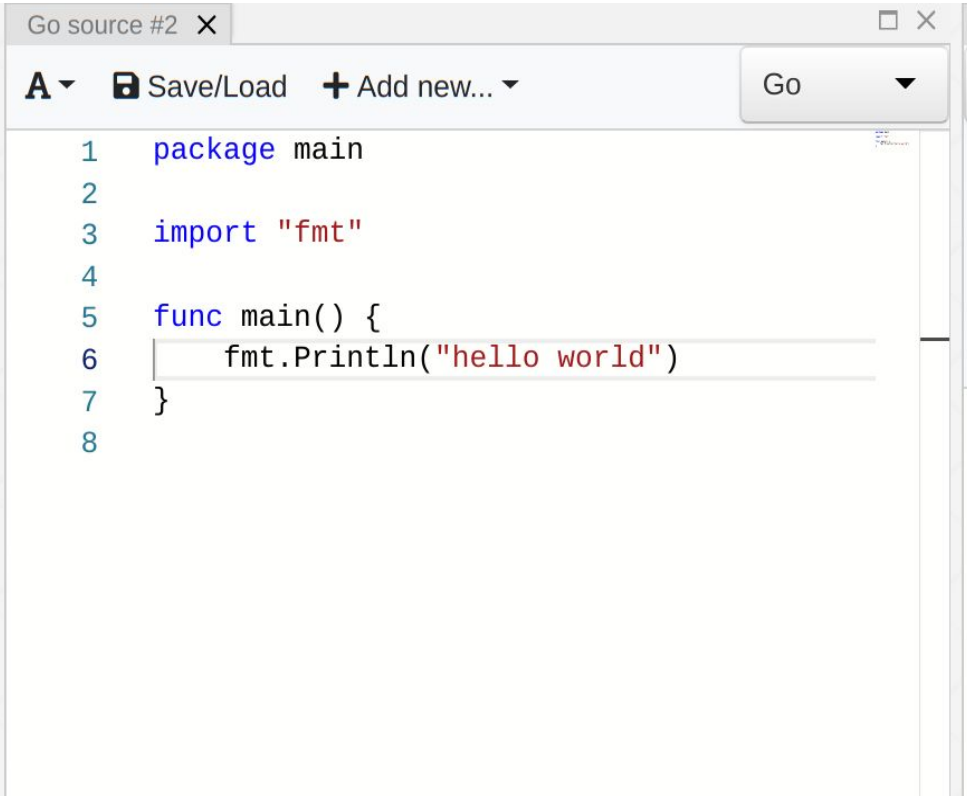


Go: VM or native?

VM or native?



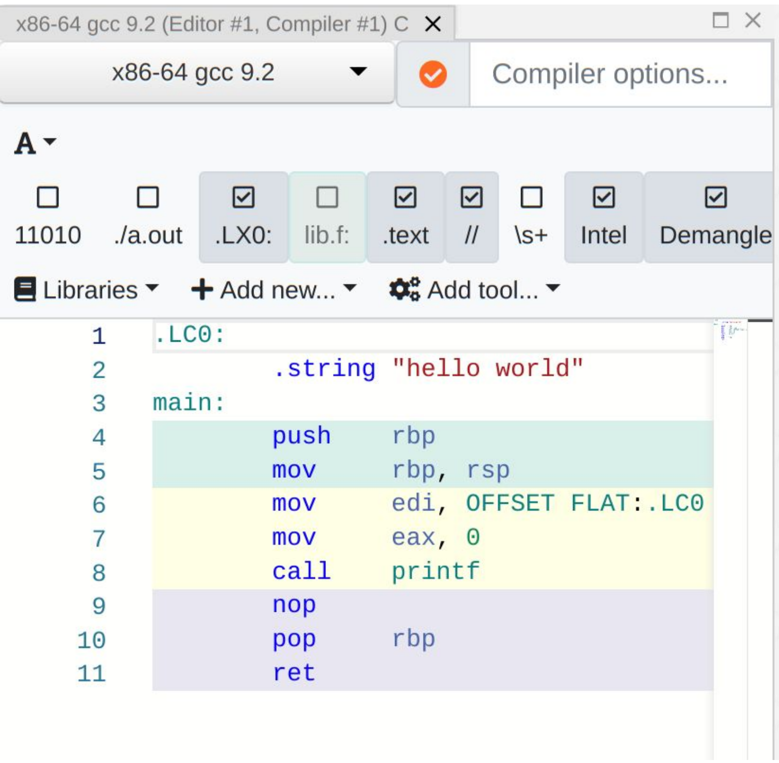
```c
void main() {
    printf("hello world");
}
```



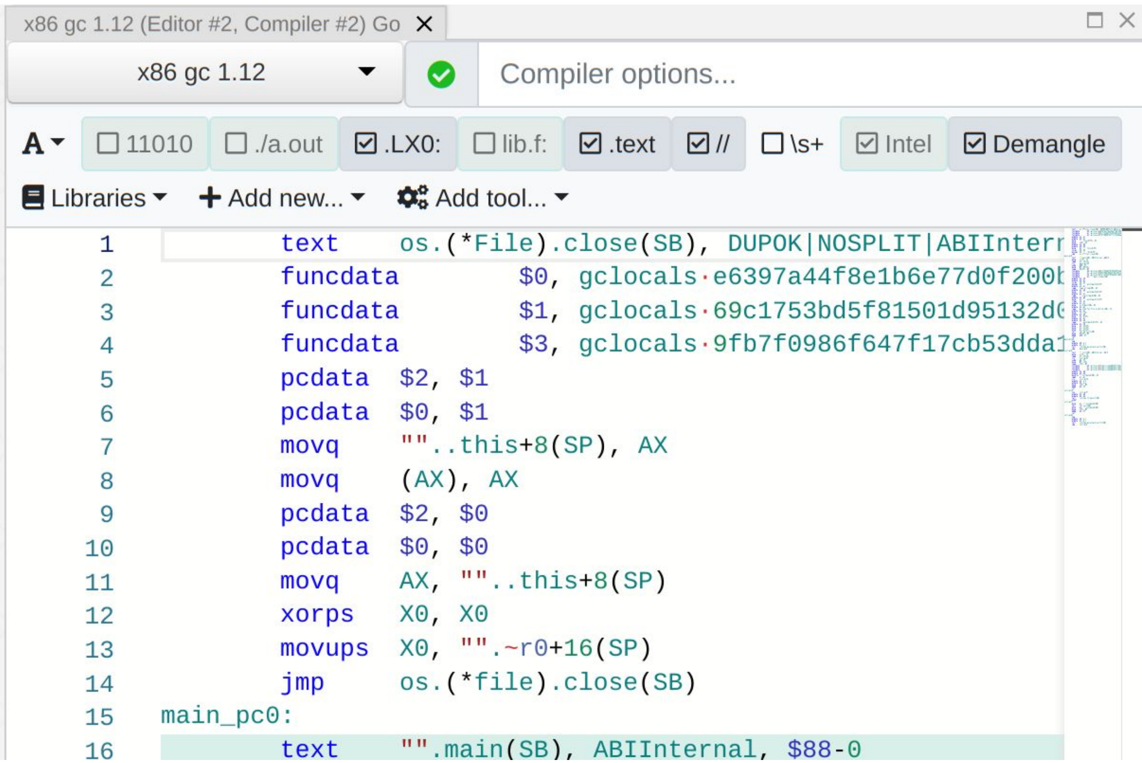
```go
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

https://godbolt.org

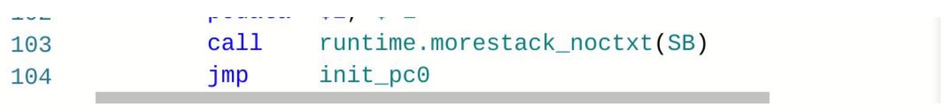
VM or native?



```
.LC0:
        .string "hello world"
main:
        push    rbp
        mov     rbp, rsp
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        nop
        pop     rbp
        ret
```

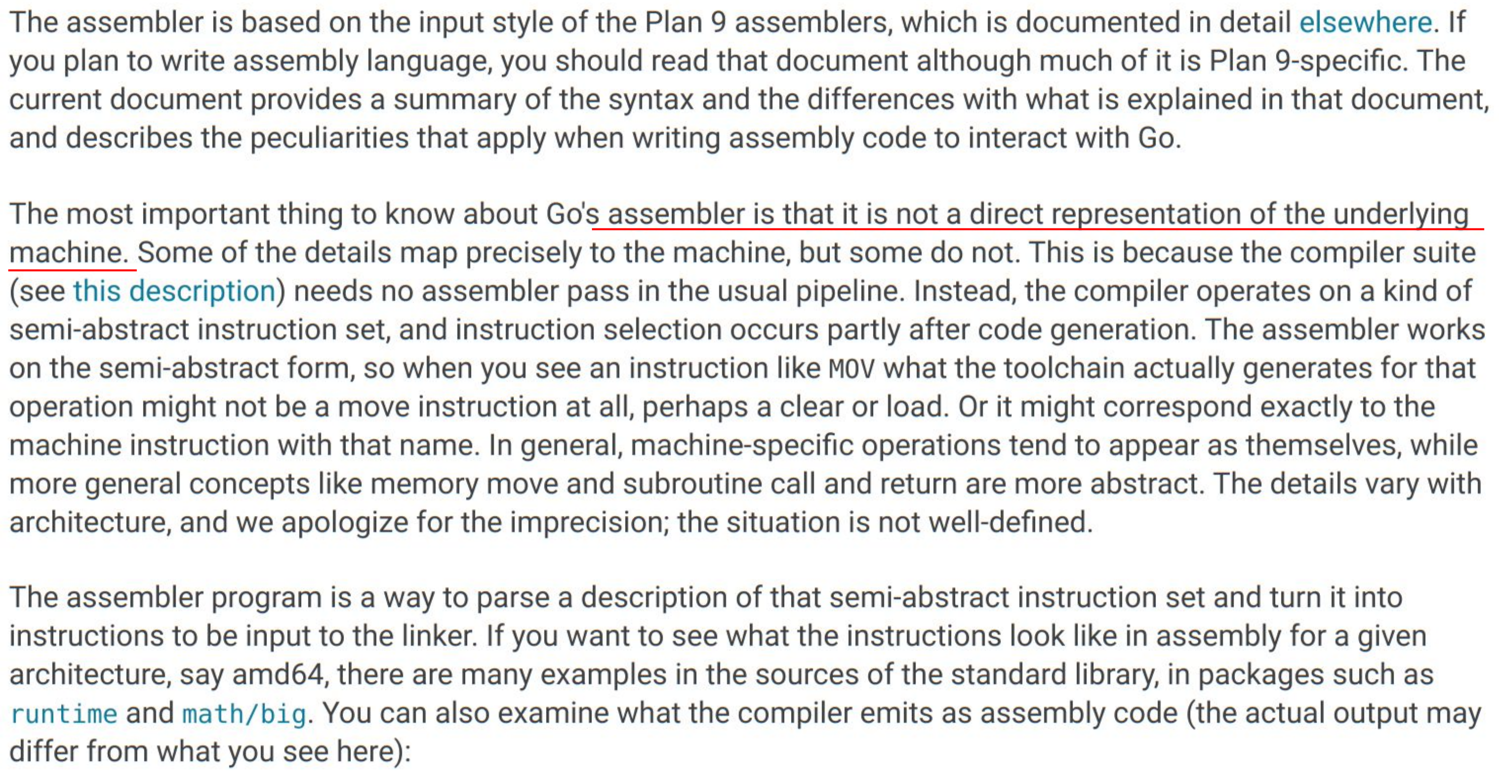


```
        text    os.(*File).close(SB), DUPOK|NOSPLIT|ABIInterr
        funcdata        $0, gclocals·e6397a44f8e1b6e77d0f200b
        funcdata        $1, gclocals·69c1753bd5f81501d95132d
        funcdata        $3, gclocals·9fb7f0986f647f17cb53dda1
        pcdata  $2, $1
        pcdata  $0, $1
        movq    "".this+8(SP), AX
        movq    (AX), AX
        pcdata  $2, $0
        pcdata  $0, $0
        movq    AX, "".this+8(SP)
        xorps   X0, X0
        movups  X0, "".~r0+16(SP)
        jmp     os.(*file).close(SB)
main_pc0:
        text    "".main(SB), ABIInternal, $88-0
```



```
        call    runtime.morestack_noctxt(SB)
        jmp     init_pc0
```

Answer #2



The assembler is based on the input style of the Plan 9 assemblers, which is documented in detail elsewhere. If you plan to write assembly language, you should read that document although much of it is Plan 9-specific. The current document provides a summary of the syntax and the differences with what is explained in that document, and describes the peculiarities that apply when writing assembly code to interact with Go.

The most important thing to know about Go's assembler is that it is not a direct representation of the underlying machine. Some of the details map precisely to the machine, but some do not. This is because the compiler suite (see this description) needs no assembler pass in the usual pipeline. Instead, the compiler operates on a kind of semi-abstract instruction set, and instruction selection occurs partly after code generation. The assembler works on the semi-abstract form, so when you see an instruction like `MOV` what the toolchain actually generates for that operation might not be a move instruction at all, perhaps a clear or load. Or it might correspond exactly to the machine instruction with that name. In general, machine-specific operations tend to appear as themselves, while more general concepts like memory move and subroutine call and return are more abstract. The details vary with architecture, and we apologize for the imprecision; the situation is not well-defined.

The assembler program is a way to parse a description of that semi-abstract instruction set and turn it into instructions to be input to the linker. If you want to see what the instructions look like in assembly for a given architecture, say amd64, there are many examples in the sources of the standard library, in packages such as `runtime` and `math/big`. You can also examine what the compiler emits as assembly code (the actual output may differ from what you see here):

https://golang.org/doc/asm

2 x WTH

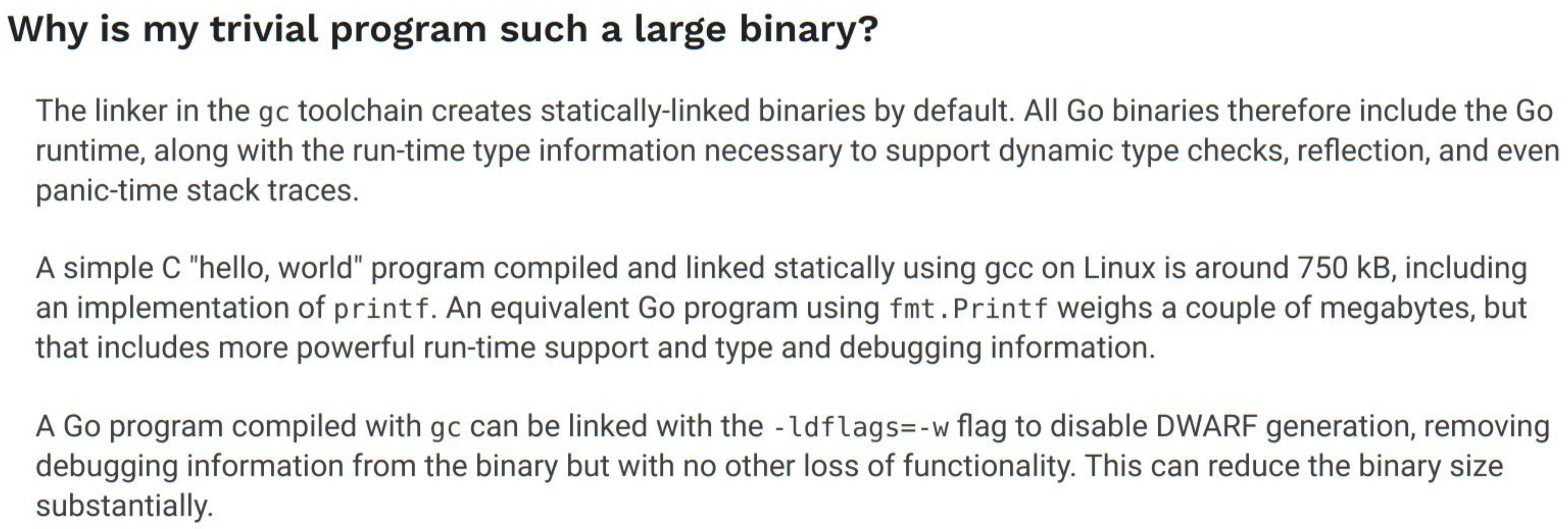
1. Why so long assembler code in comparison with C?

2. Is this an assembler at all?

2 x WTH

Answer #1: Runtime

Answer #1 - cd



Why is my trivial program such a large binary?

The linker in the `gc` toolchain creates statically-linked binaries by default. All Go binaries therefore include the Go runtime, along with the run-time type information necessary to support dynamic type checks, reflection, and even panic-time stack traces.

A simple C "hello, world" program compiled and linked statically using gcc on Linux is around 750 kB, including an implementation of `printf`. An equivalent Go program using `fmt.Printf` weighs a couple of megabytes, but that includes more powerful run-time support and type and debugging information.

A Go program compiled with `gc` can be linked with the `-ldflags=-w` flag to disable DWARF generation, removing debugging information from the binary but with no other loss of functionality. This can reduce the binary size substantially.

`https://golang.org/doc/faq#runtime`

What’s Go Runtime?

Runtime Responsibility

- ★ interacting with OS
- ★ managing goroutines
- ★ bound checking
- ★ GC
- ★ monitoring

Interacting with OS

simple problem: **open a file**

Interacting with OS

easy-peasy: **just use a syscall**

Interacting with OS

… maybe less easy-peasy:

which OS and which syscall?

Interacting with OS

```
if Linux {
LinuxSyscall
}
```

```
if Windows {
WindowsSyscall
}
```

```
if macOS {
macOSSyscall
}
```

?

Interacting with OS

```
 import "os/exec"

func main() {

     app := "cat"

     arg0 := "file.go”

     cmd := exec.Command(app, arg0)

     stdout, err := cmd.Output()

     if err != nil {

          println(err.Error()) return

     }

     print(string(stdout))

}
```

Interacting with OS

that was a joke of course… :-)

Interacting with OS

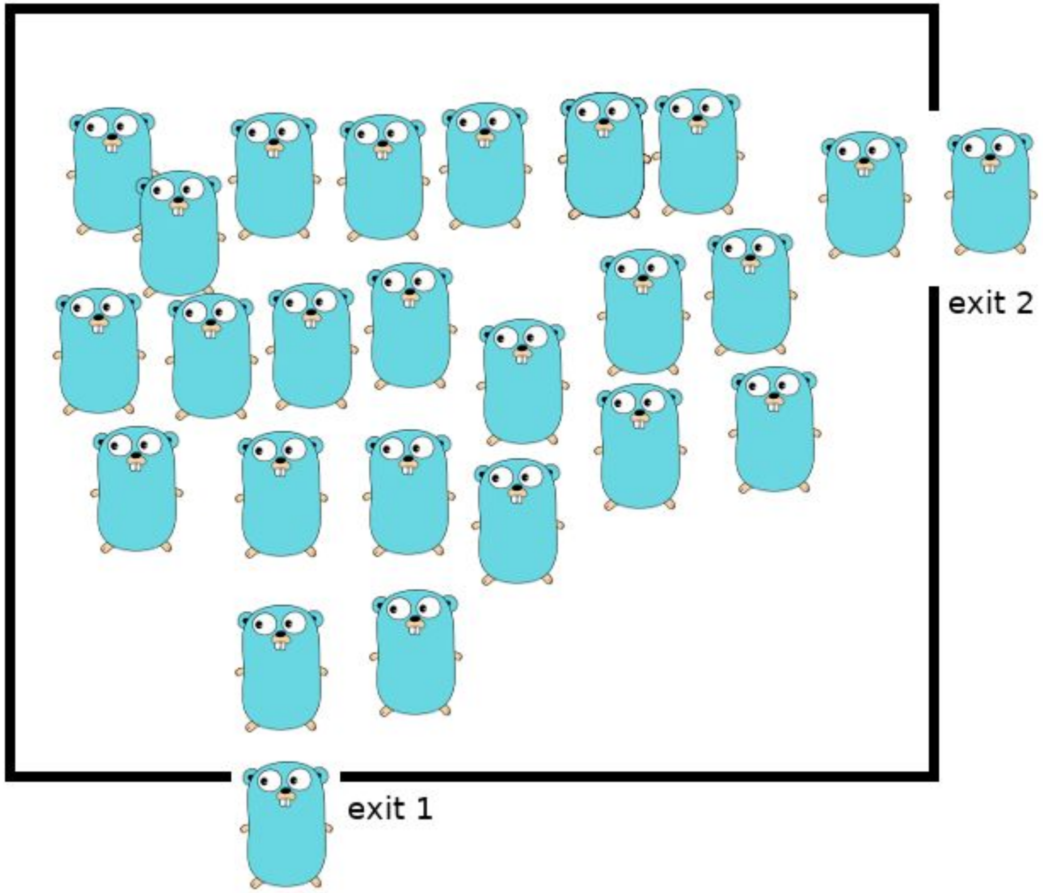
```
file, err := os.Open("file.go") // For read access.

if err != nil {

    log.Fatal(err)

}

// do something with file
```

Go Scheduler

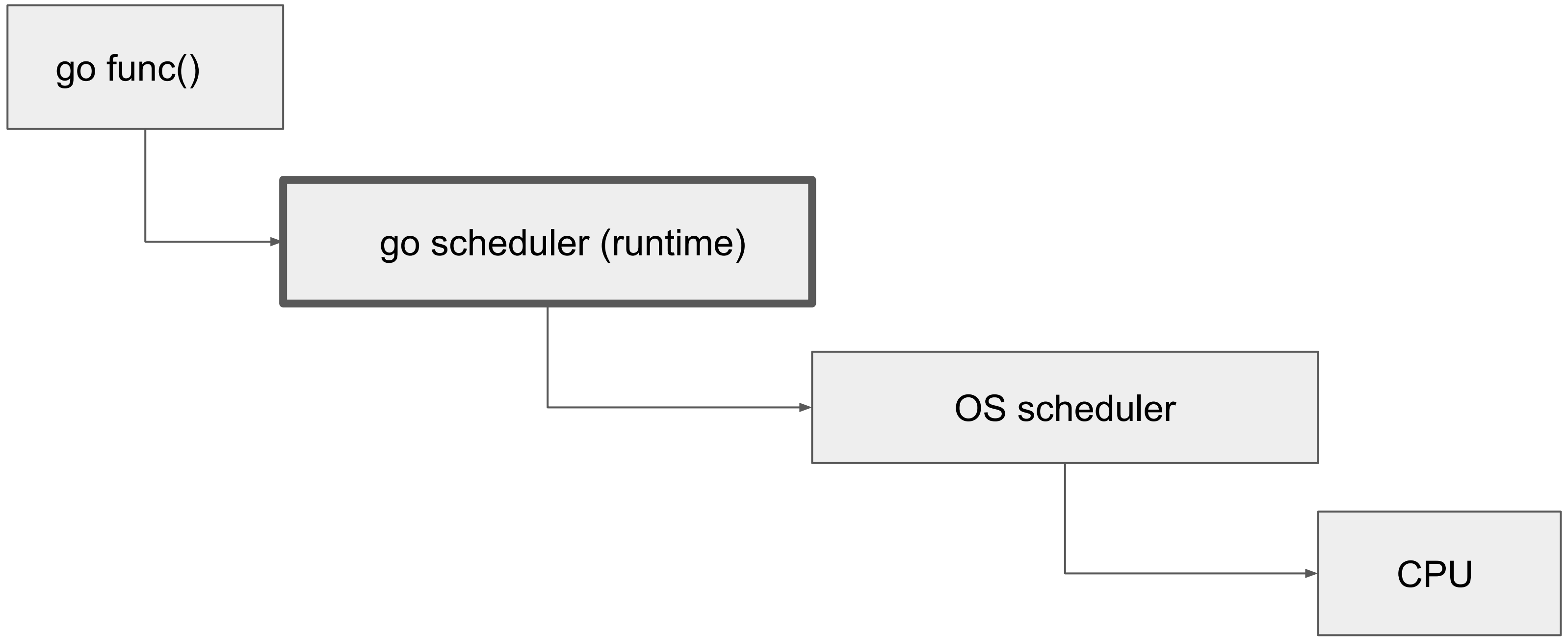
Go Scheduler

Decides how many goroutines will be run, parked, resumed, stopped, when, and on which logical processor

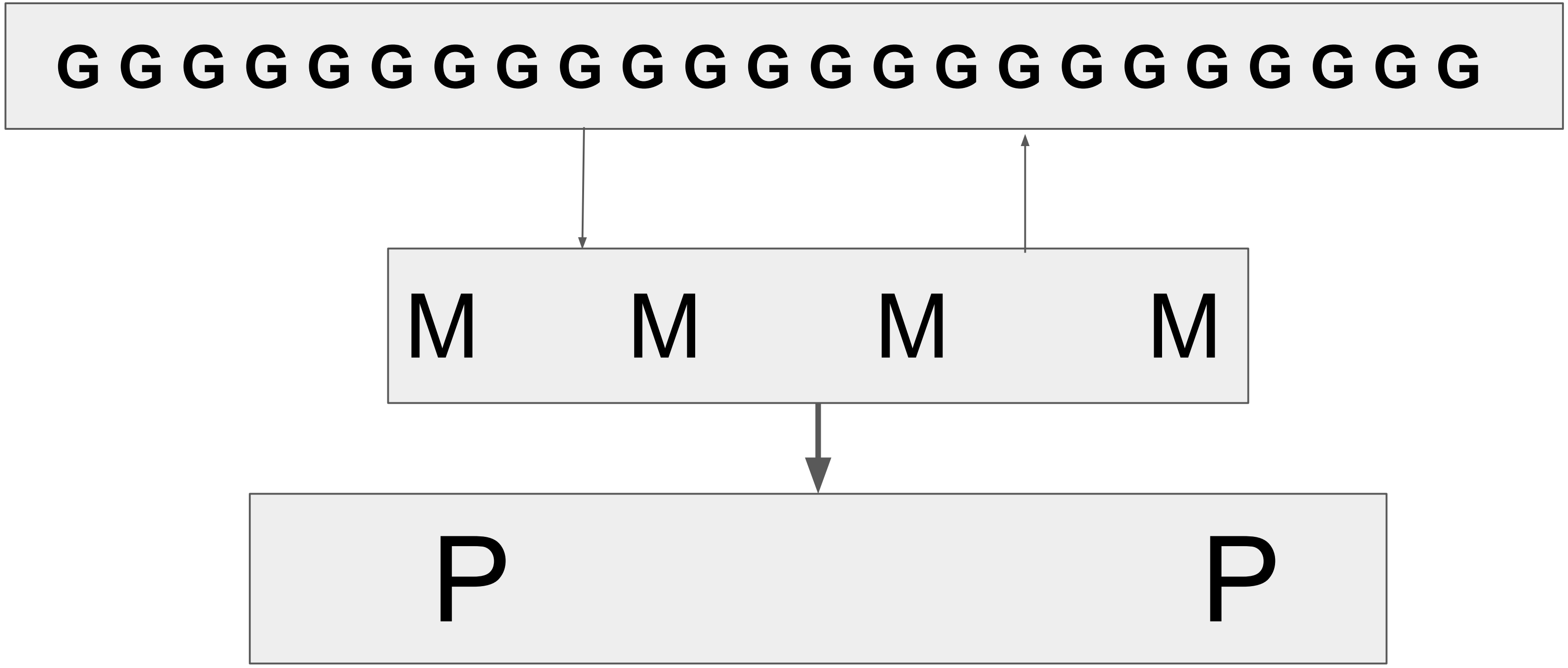
It interacts with OS scheduler and reacts to it’s decisions


exit 2
exit 1

Goroutine road to host CPU



Go Scheduler



Go Scheduler

It’s a work stealing scheduler - it uses an approach, which is used to distribute a number of concurrent tasks on limited resources (OS threads or vCPU), when the number of tasks is greater than available resources (usually).

Such kind of schedulers are non deterministic - you cannot predict their behavior at any given time

Go Scheduler

To invoke context-switch scheduler is invoked by specific actions. These actions are: (channel send/receive, go statement, blocking syscalls and GC)

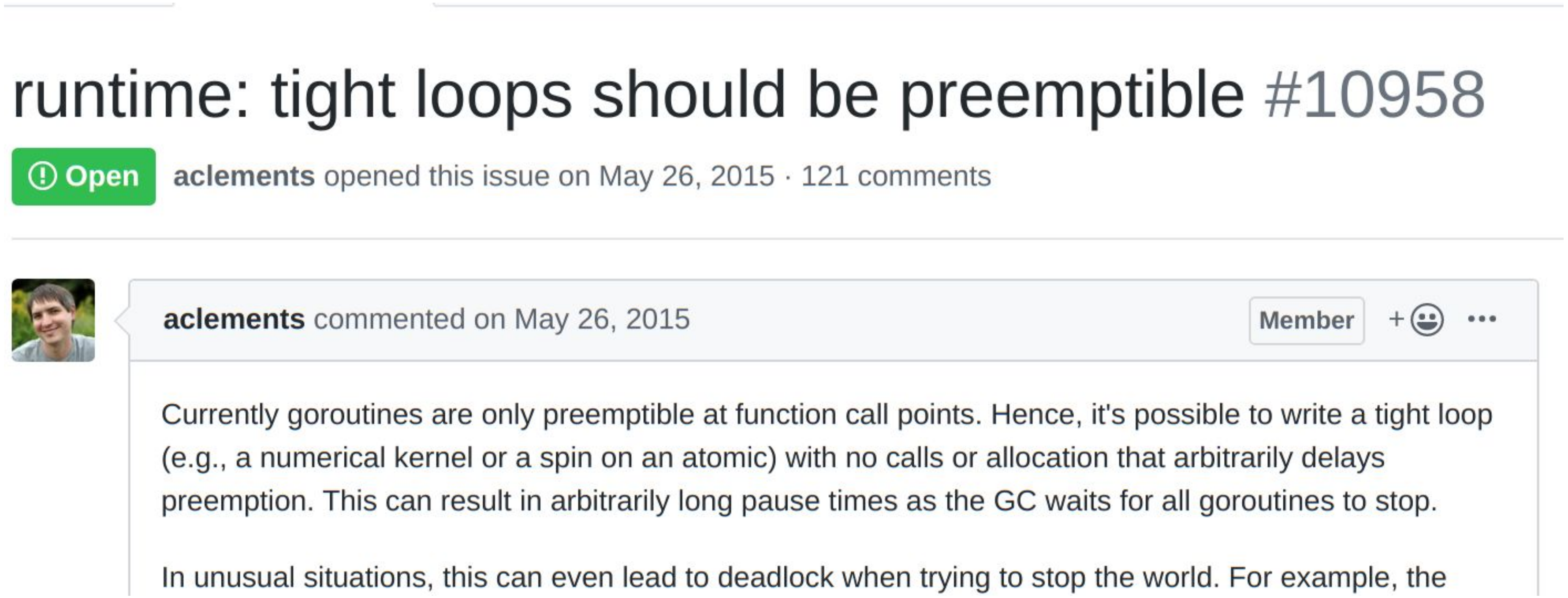
Preemption points (function calls) are inserted by the compiler during compilation.

Go Scheduler

Go Scheduler has been developing heavily, it’s good enough for most cases, but still has some serious and rare problems.

`Full algorithm described: src/runtime/proc.go`

Go Scheduler



runtime: tight loops should be preemptible #10958

Open aclements opened this issue on May 26, 2015 · 121 comments

aclements commented on May 26, 2015 Member

Currently goroutines are only preemptible at function call points. Hence, it's possible to write a tight loop (e.g., a numerical kernel or a spin on an atomic) with no calls or allocation that arbitrarily delays preemption. This can result in arbitrarily long pause times as the GC waits for all goroutines to stop.

In unusual situations, this can even lead to deadlock when trying to stop the world. For example, the

Garbage Collector

Mechanism which is responsible for automatic memory management (allocating, keeping, freeing, etc)

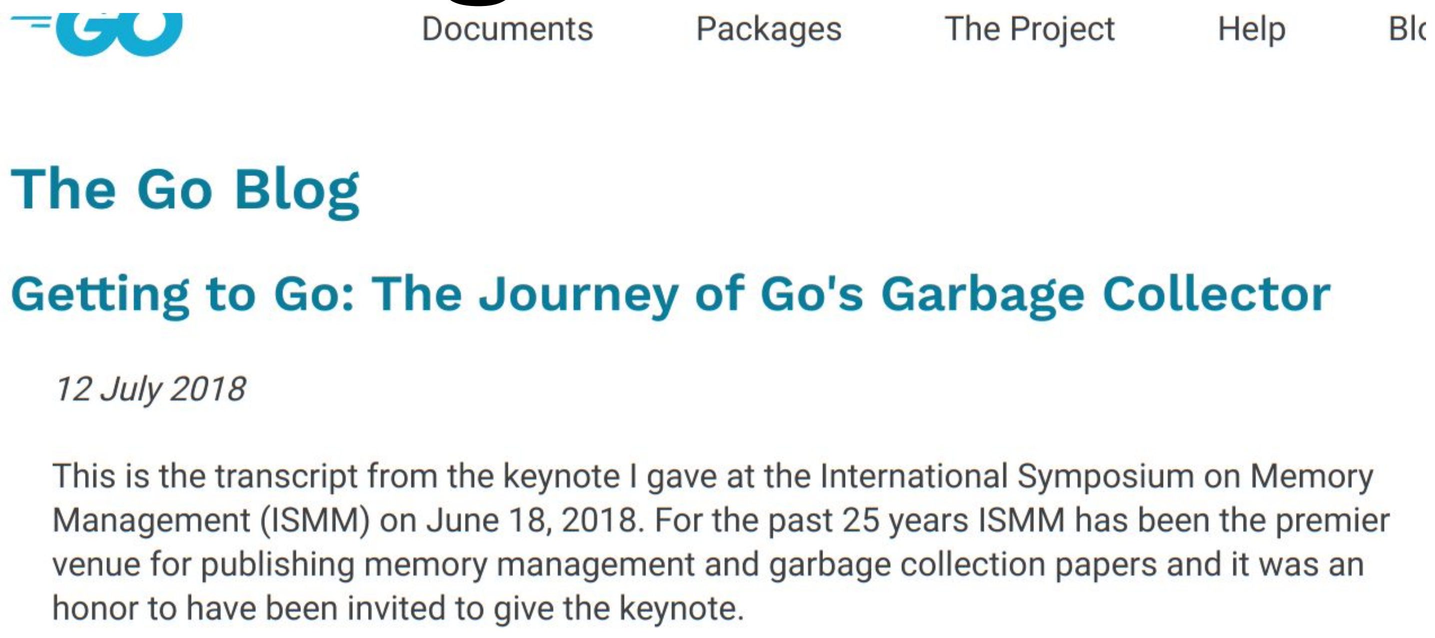
It can pause a program as there is a phase which requires all goroutines to be stop/paused to “collect” objects

It should use up to 25% of CPU resource during it’s work

Garbage Collector

Because of nature of Go Scheduler (current) it can pause very long or even block the entire program (in rare and specific cases)

Garbage Collector



The Go Blog

Getting to Go: The Journey of Go's Garbage Collector

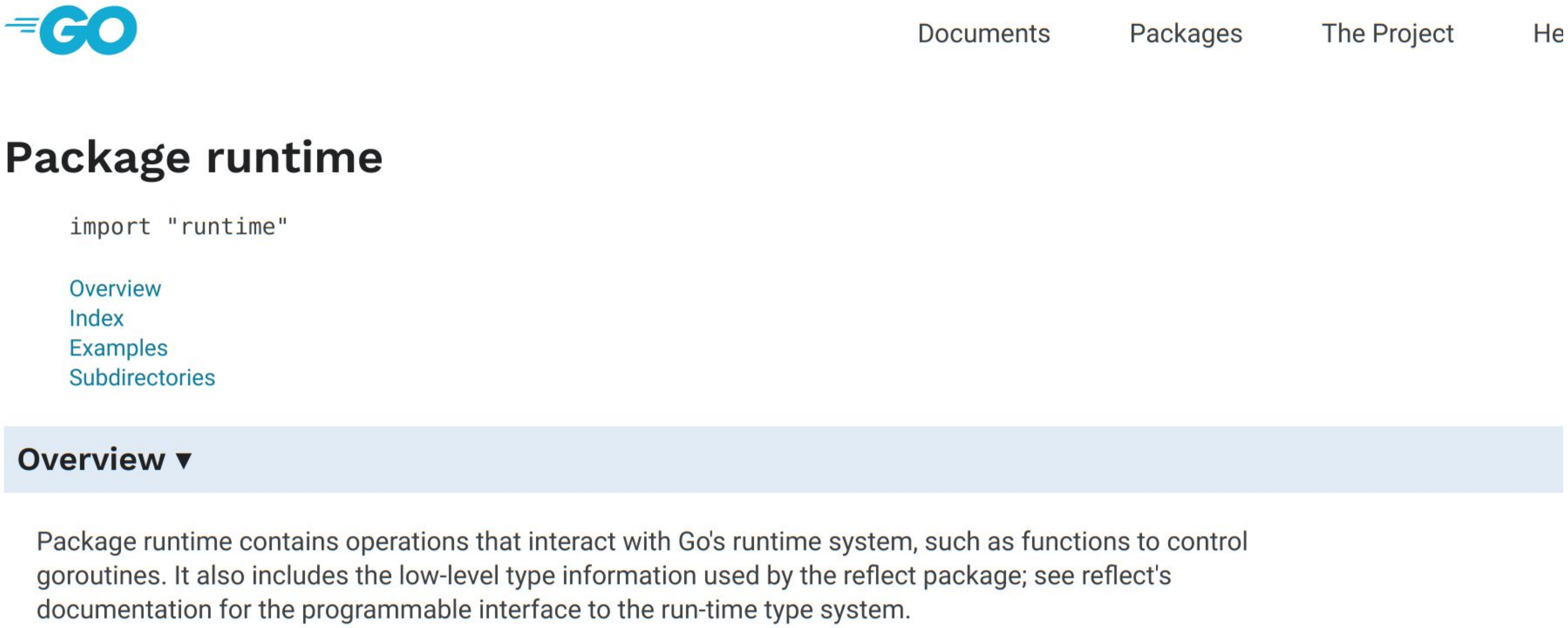
12 July 2018

This is the transcript from the keynote I gave at the International Symposium on Memory Management (ISMM) on June 18, 2018. For the past 25 years ISMM has been the premier venue for publishing memory management and garbage collection papers and it was an honor to have been invited to give the keynote.

`https://blog.golang.org/ismmkeynote`

Interacting with runtime

Interacting with runtime



Package runtime

```
import "runtime"
```

Overview
Index
Examples
Subdirectories

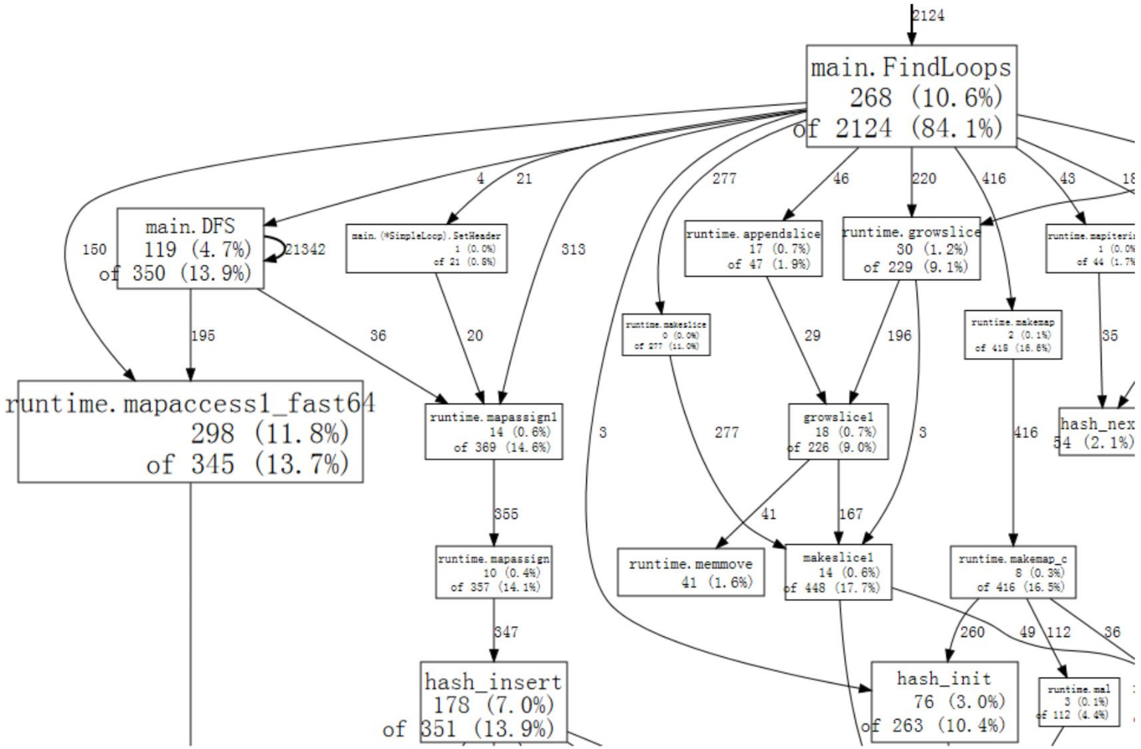
Overview ▾

Package runtime contains operations that interact with Go's runtime system, such as functions to control goroutines. It also includes the low-level type information used by the reflect package; see reflect's documentation for the programmable interface to the run-time type system.

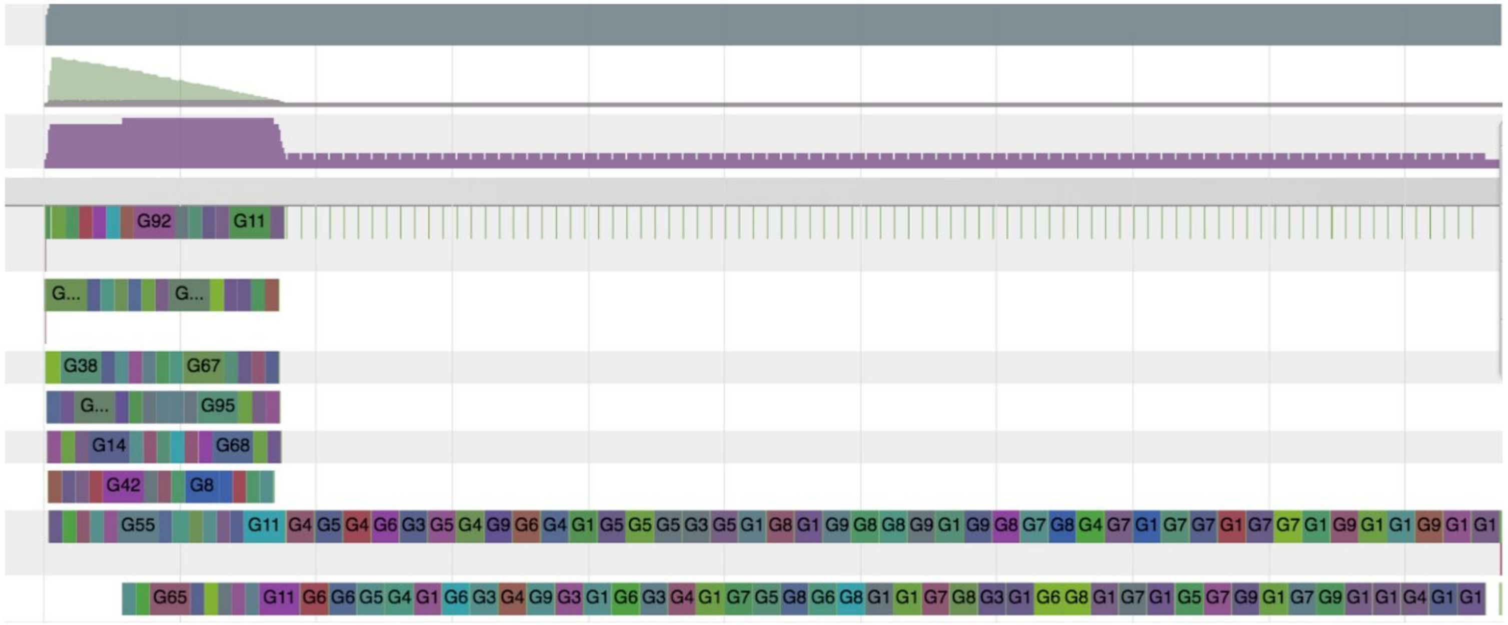
Monitoring

```
"runtime/pprof"

"runtime/trace"
```

Monitoring - pprof


2124
main.FindLoops 268 (10.6%) of 2124 (84.1%)
main.DFS 119 (4.7%) of 350 (13.9%)
21342
main.(*SimpleLoop).SetHeader 1 (0.0%) of 21 (0.8%)
runtime.appendslice 17 (0.7%) of 47 (1.9%)
runtime.growslice 30 (1.2%) of 229 (9.1%)
runtime.mapiteri 1 (0.0% of 44 (1.7%
runtime.makeslice 0 (0.0%) of 277 (11.0%)
runtime.makemap 2 (0.1%) of 418 (16.6%)
runtime.mapaccess1_fast64 298 (11.8%) of 345 (13.7%)
runtime.mapassign1 14 (0.6%) of 369 (14.6%)
growslice1 18 (0.7%) of 226 (9.0%)
hash_nex 54 (2.1%)
runtime.mapassign 10 (0.4%) of 357 (14.1%)
runtime.memmove 41 (1.6%)
makeslice1 14 (0.6%) of 448 (17.7%)
runtime.makemap_c 8 (0.3%) of 416 (16.5%)
hash_insert 178 (7.0%) of 351 (13.9%)
hash_init 76 (3.0%) of 263 (10.4%)
runtime.mal 3 (0.1%) of 112 (4.4%)
150
4
21
313
277
46
220
416
43
18
195
36
20
29
196
35
3
277
3
416
355
41
167
347
260
49
112
36

Monitoring - trace


G92 G11
G... G...
G38 G67
G... G95
G14 G68
G42 G8
G55 G11 G4 G5 G4 G6 G3 G5 G4 G9 G6 G4 G1 G5 G5 G5 G3 G5 G1 G8 G1 G9 G8 G8 G9 G1 G9 G8 G7 G8 G4 G7 G1 G7 G7 G1 G7 G7 G1 G9 G1 G1 G9 G1 G1
G65 G11 G6 G6 G5 G4 G1 G6 G3 G4 G9 G3 G1 G6 G3 G4 G1 G7 G5 G8 G6 G8 G1 G1 G7 G8 G3 G1 G6 G8 G1 G7 G1 G5 G7 G9 G1 G7 G9 G1 G1 G4 G1 G1

Go Scheduler

`runtime.Gosched()`

`runtime.LockOSThread()`

`runtime.UnlockOSThread()`

`GOMAXPROCS`

Garbage Collector

```
runtime/debug/SetGCPercent()

runtime.FreeOSMemory()

runtime.KeepAlive()

runtime.SetFinalizer()
```

Statistics

`runtime.ReadMemStats`

`runtime/debug.ReadGCStats`

`runtime/debug.Stack`

`runtime/debug.WriteHeapDump`

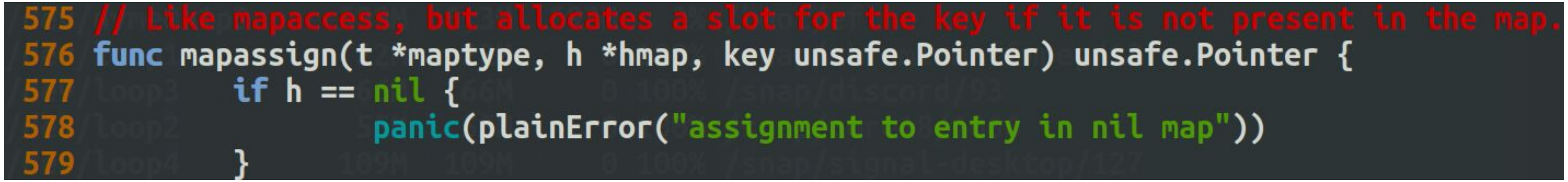
`runtime.NumGoroutine`

`runtime.*`

`GODEBUG`

Some Runtime Panics

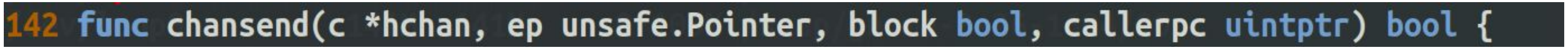
assign key to nil map



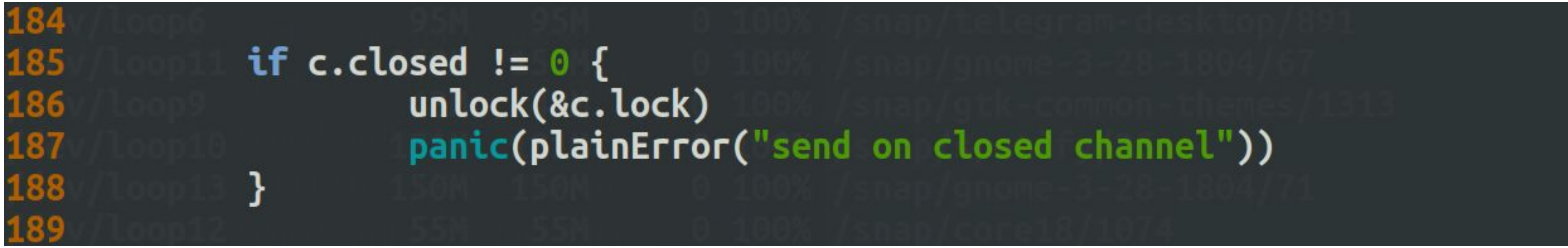
```go
// Like mapaccess, but allocates a slot for the key if it is not present in the map.
func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
	if h == nil {
		panic(plainError("assignment to entry in nil map"))
	}
```

$GOROOT/src/runtime/map.go (1.12.5)

send to closed channel



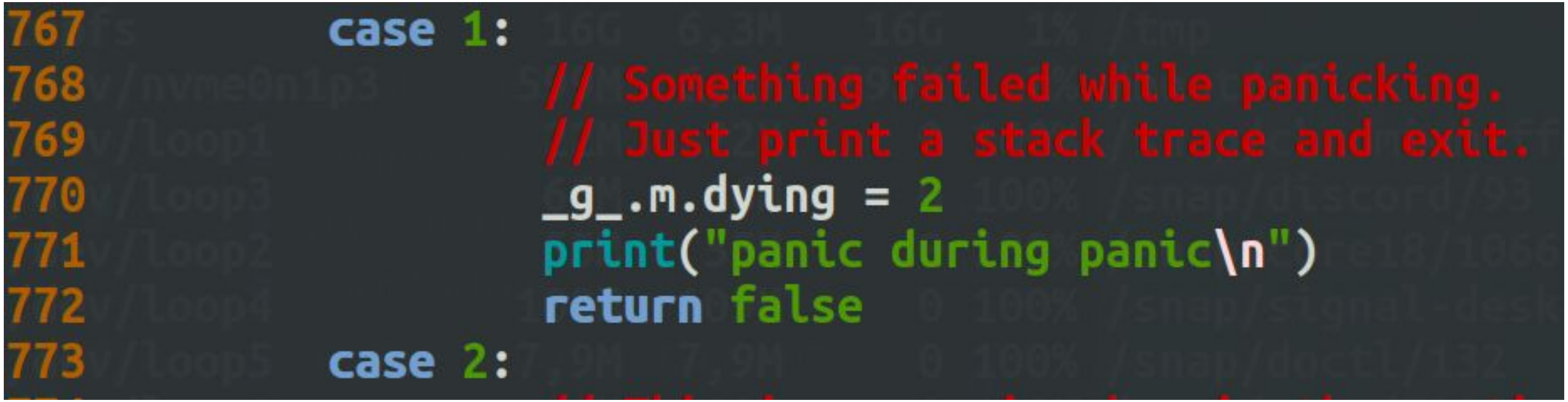
```go
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
```



```go

        if c.closed != 0 {
                unlock(&c.lock)
                panic(plainError("send on closed channel"))
        }

```

$GOROOT/src/runtime/chan.go (1.12.5)

panic during panic?



```go
        case 1:
                // Something failed while panicking.
                // Just print a stack trace and exit.
                _g_.m.dying = 2
                print("panic during panic\n")
                return false
        case 2:
```

$GOROOT/src/runtime/panic.go (1.12.5)

Do Go programmers should know runtime?

Q&A

Thank You!