Performance and Scalability of Go Applications

24 September 2019

Fabio Falzoi

Before starting

- check your Go environment: compile and run the "Hello World"

```
package main

import "fmt"

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

- download the workshop material

```
git clone --depth 1 git@github.com:Pippolo84/performance-and-scalability-of-go-applications.git
```

- peeking inside the solution folders is not allowed ;-)

Introduction

Big O notation in a nutshell

Adapted from Wikipedia (https://en.wikipedia.org/wiki/Big_O_notation)

```
Let f be a real valued function and g a real valued function. One writes:

f(x) = O(g(x)) for x -> +inf

if for all sufficiently large values of x, the absolute value of f(x) is at most a positive
constant multiple of g(x).

That is, f(x) = O(g(x)) if and only if there exists a positive real number M and
a real number x0 such that

abs(f(x)) <= Mg(x) for all x >= x0
```

Time complexity examples

- *O(1)* constant time (look-up table)
- *O(logn)* logarithmic time (binary search)
- *O(n)* linear time (linear search)
- *O(nlogn)* linearithmic time (quick sort)
- *O(n^2)* quadratic time (bubble sort)

A taste of performance analysis

Try to complete the Isogram exercise!

Follow the instructions inside

```
01-introduction/INSTRUCTIONS.md
```

Happy coding!

Isogram solutions

1) Using a set

```
- iterate over the string
- for each rune: if it is already in the map, return false
- return true
```

2) Using sorting

```
- sort the string
- check each pair of adjacent runes
- if you find a pair of equal runes, return false, otherwise return true
```

3) Using linear searching

```
- iterate over the string
- for each rune, search for it in the remaining part of the string
- if found, return false, otherwise, return true
```

Quick quiz: can you guess the time complexity of each solution?

Isogram solutions time complexities

1) Using a set: *O(n)*
2) Using sorting: *O(nlogn)*
3) Using linear searching: *O(n^2)*

Let's benchmark them!

Inside the solution folder

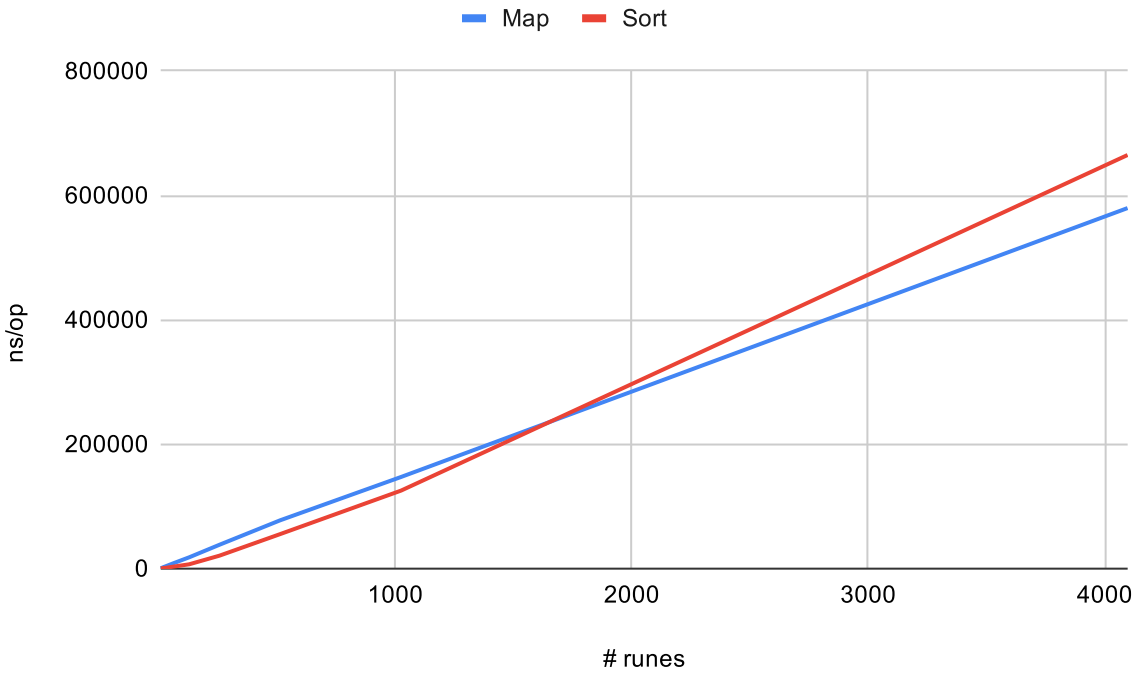
```
$ go test -run=^$ -bench=.
goos: linux
goarch: amd64
pkg: performance-and-scalability-of-go-applications/01-introduction/solution
BenchmarkIsIsogram/01-map-8                200000              8203 ns/op
BenchmarkIsIsogram/02-sorting-8            300000              5534 ns/op
BenchmarkIsIsogram/03-linear-search-8             1000000              1382 ns/op
...
PASS
ok      performance-and-scalability-of-go-applications/01-introduction/solution 6.295s
```

Wait... What!?

Map vs Sort vs Linear Search

For #runes < 256 the linear search outperforms the other implementation

Map vs Sort



For #runes < 1024 the sort implementation is still faster than the map

A closer look to the map data type

There are no generics in Go 1.13

The compiler rewrites lookups, insertion and removal into functions calls to the runtime package

```
v := found[r]
func mapaccess1_*(t *maptype, h *hmap, key uint32) unsafe.Pointer

found[r] = true
func mapassign_*(t *maptype, h *hmap, key uint32) unsafe.Pointer
```

- hmap implements the generic map logic
- maptype is unique to the specific map type declared and contains key and value type descriptors

A closer look to the map data type

```
go tool compile -S isogram.go

0x00df 00223 (isogram.go:18)    MOVL    AX, "".r+40(SP)                 ;; key loaded in AX
0x00e3 00227 (isogram.go:19)    LEAQ    type.map[int32]bool(SB), CX     ;; t *maptype loaded in CX
0x00ea 00234 (isogram.go:19)    MOVQ    CX, (SP)                        ;; t *maptype on the stack
0x00ee 00238 (isogram.go:19)    LEAQ    ""..autotmp_8+112(SP), DX       ;; h *hmap loaded in DX
0x00f3 00243 (isogram.go:19)    MOVQ    DX, 8(SP)                       ;; h *hmap on the stack
0x00f8 00248 (isogram.go:19)    MOVL    AX, 16(SP)                      ;; key on the stack

;; runtime.mapaccess1_fast32(t, h, key)
0x00fc 00252 (isogram.go:19)    CALL    runtime.mapaccess1_fast32(SB)

0x010b 00267 (isogram.go:23)    LEAQ    type.map[int32]bool(SB), AX     ;; t *maptype loaded in CX
0x0112 00274 (isogram.go:23)    MOVQ    AX, (SP)                        ;; t *maptype on the stack
0x0116 00278 (isogram.go:23)    LEAQ    ""..autotmp_8+112(SP), CX       ;; h *hmap loaded in DX
0x011b 00283 (isogram.go:23)    MOVQ    CX, 8(SP)                       ;; h *hmap on the stack
0x0120 00288 (isogram.go:23)    MOVL    "".r+40(SP), DX                 ;; key loaded in DX
0x0124 00292 (isogram.go:23)    MOVL    DX, 16(SP)                      ;; key on the stack

;; runtime.mapassign_fast32(t, h, k)
0x0128 00296 (isogram.go:23)    CALL    runtime.mapassign_fast32(SB)
```

A closer look to sort.Sort

from sort.go:

```
func quickSort(data Interface, a, b, maxDepth int) {
    for b-a > 12 { // Use ShellSort for slices <= 12 elements
        ...
    }
    if b-a > 1 {
        // Do ShellSort pass with gap 6
        // It could be written in this simplified form cause b-a <= 12
        for i := a + 6; i < b; i++ {
            if data.Less(i, i-6) {
                data.Swap(i, i-6)
            }
        }
        insertionSort(data, a, b)
    }
}
```

For small slices, the standard library doesn't even use Quicksort!

A closer look to strings.ContainsRune

from indexbyte_amd64.s:

```
sseloop:
```

```
// Move the next 16-byte chunk of the data into X1.
MOVOU    (DI), X1
// Compare bytes in X0 to X1.
PCMPEQB    X0, X1
// Take the top bit of each byte in X1 and put the result in DX.
PMOVMSKB X1, DX
// Find first set bit, if any.
BSFL    DX, DX
JNZ    ssesuccess
// Advance to next block.
ADDQ    $16, DI
```

Search is done using SIMD instructions

Can we do even better?

Consider the test cases, we can see that:
- all runes are in the ASCII set
- we are only interested in the 26 alphabetical letters

Array as a map

```
func IsIsogram(s string) bool {
    foundRune := [26]bool{} //'a' to 'z'

    for _, r := range s {
        if !unicode.IsLetter(r) {
            continue
        }

        // convert the rune to lowercase to index foundRune
        r = unicode.ToLower(r)
        i := r - 'a'

        if foundRune[i] == true {
            return false
        }
        foundRune[i] = true
    }

    return true
}
```

Benchmarks

```
BenchmarkIsIsogram/01-map-8                200000              8203 ns/op
BenchmarkIsIsogram/02-sorting-8            300000              5534 ns/op
BenchmarkIsIsogram/03-linear-search-8     1000000              1382 ns/op
BenchmarkIsIsogram/04-bool-array-8        3000000               525 ns/op
```

Pitfalls of the time complexity model

Constant factors are important. What are they determined by?

- memory access pattern
- algorithms implemented in hardware
- characteristics of our input dataset
- ...

Take home message

Time complexity with Big O notation tells us just a part of the story!

If we want to get the most ouf of our applications we need to:

- understand our hardware, the Go runtime internals and their interactions
- learn to profile and benchmark our application with meaningful dataset

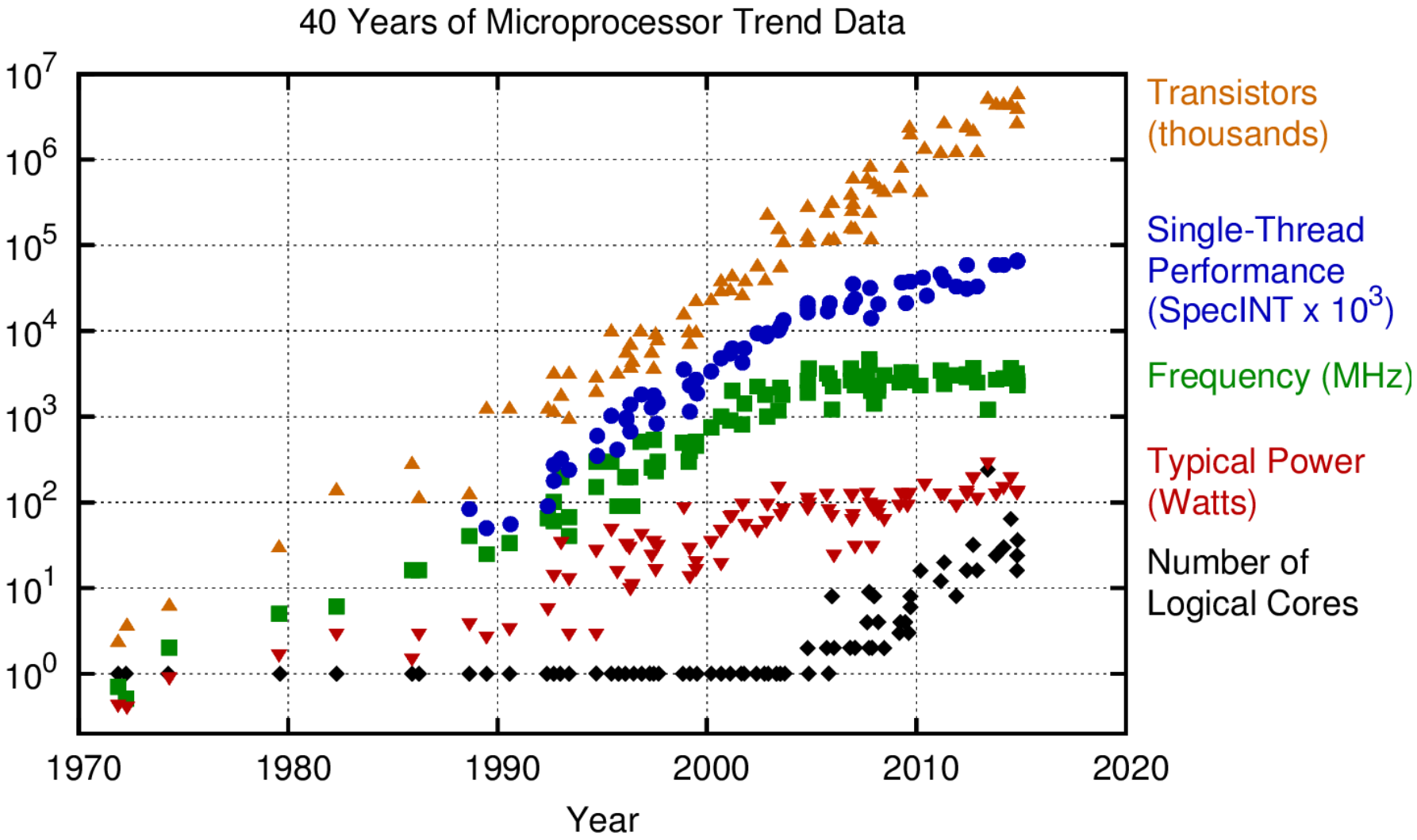
That's why the rest of this workshop is dedicated to that! ;-)

Roadmap

- Modern CPU architecture
- Benchmarking
- Profiling & Tracing
- The Go scheduler
- The Go memory allocator & garbage collector

Modern CPU architecture

Microprocessors evolution



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Single core performance

Single core performance is approaching a limit

CPUs manufacturers need to keep power consumption, and thus heat dissipation, below levels that will damage their CPUs

Number of transistors

Number of transistors on a die continues to increase

But packing more transistors in the same area is getting harder and expensive!

In conclusion, adding cores is not that easy.

Besides, improvements from parallelization are limited by the portion of work that is not parallelizable.

Architectural improvements

Out of order execution to take advantage of *Instruction level parallelism*

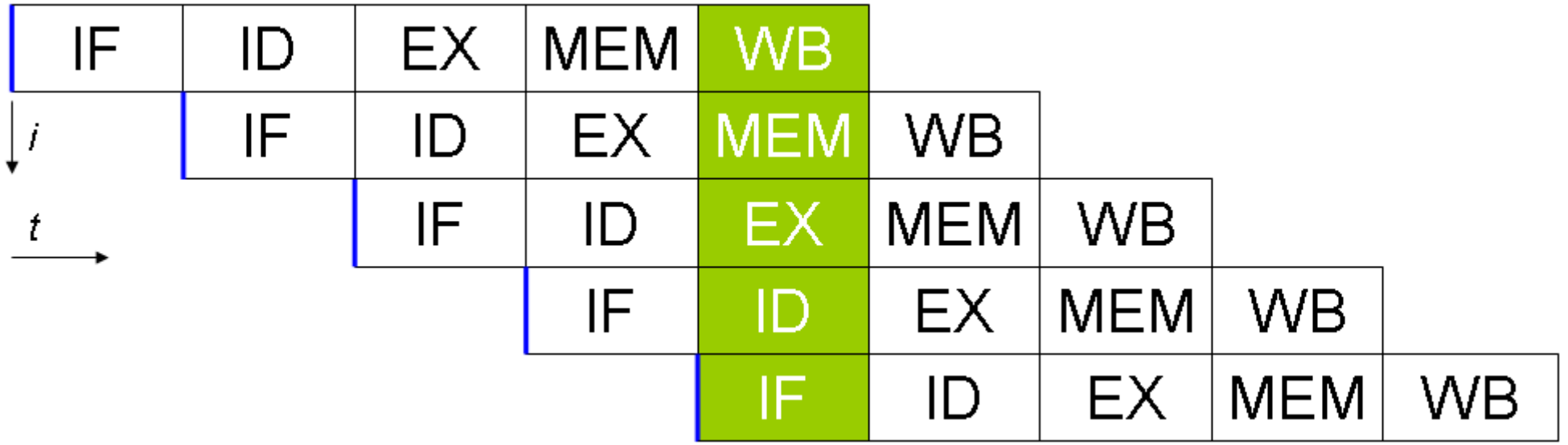
```
1) h = a + b
2) f = c + d
3) g = h * f
```

1) and 2) can be executed in parallel

But connecting each execution unit to all others is costly

Architectural improvements

Speculative execution to guess the outcome of a branch condition and keeps the pipeline full



Performance improvements depend on the rate of branch prediction successes!

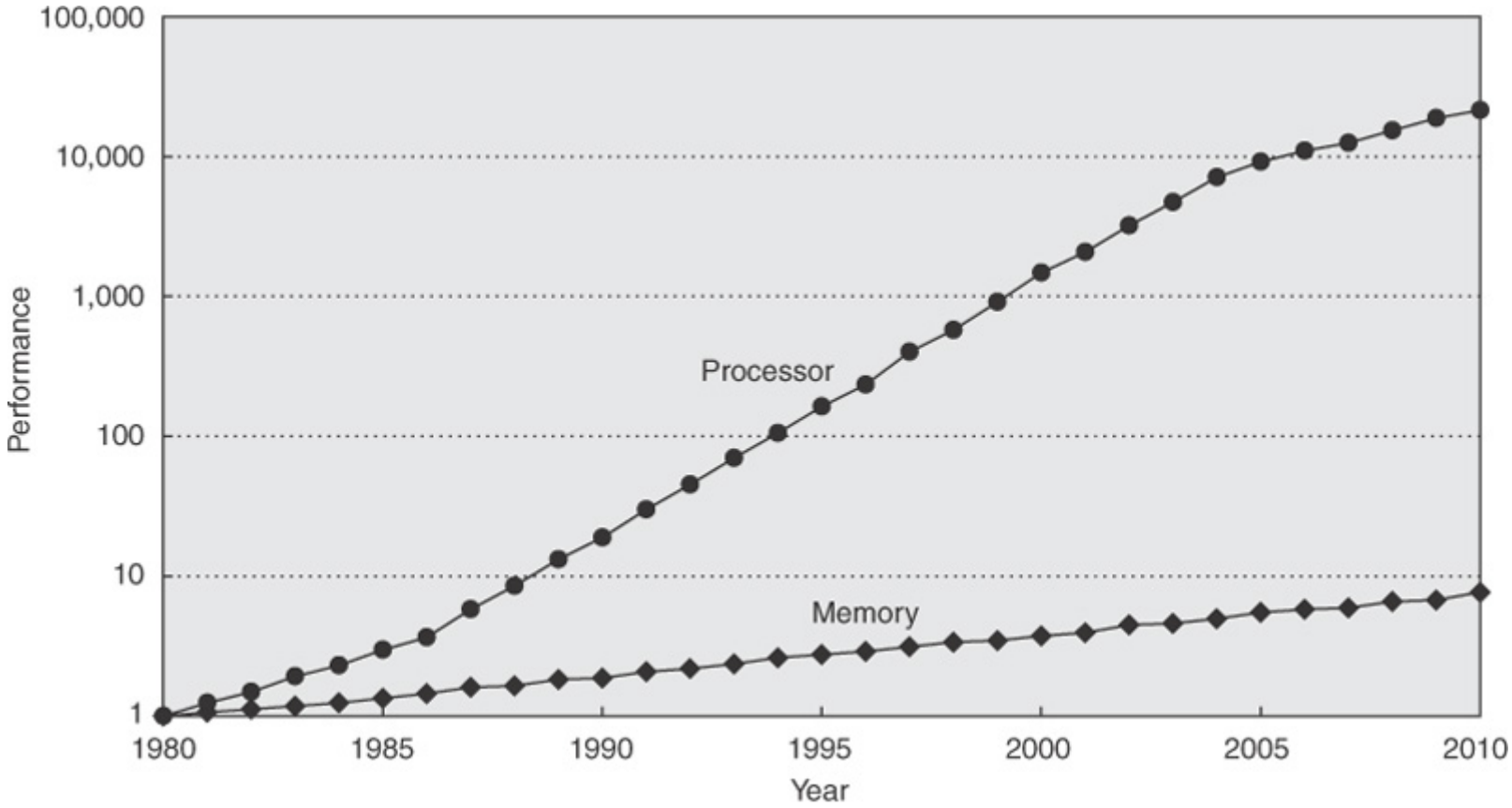
Bulk operations

Modern CPUs are optimized for **bulk** operations

- memory is loaded per multiple of caches lines
- SIMD instructions

We've seen this effects in the introductory exercise!

Memory latency



Memory latency

L1 cache reference ~1ns
Branch mispredict ~3ns
L2 cache reference ~4ns
Main memory reference ~100ns

from Latency Numbers Every Programmer Should Know (https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html)

Caches are critical to reduce memory latencies, unfortunately, their size are limited by their physical size on CPU die and power consumption

Conclusions

future programmers will not longer be able to rely on faster hardware to fix slow programs or slow programming languages

from The free lunch is over (http://www.gotw.ca/publications/concurrency-ddj.htm) - Herb Sutter

Why Go?

- efficient type system: it does not pretend every number is an ideal float
- efficient implementation of structs: in contrast with java objects that store references
- easily scales to multiple cores for highly parallelizable applications

Benchmarking

A stable environment

When benchmarking, try to make your environment as stable as possible:

- avoid shared hardware, virtual machines and shared cloud hosting
- close all resource hungry applications so that CPU utilization is lower than ~5%

Specifically

- do not browse the Web
- do not check emails
- do not use your editor to write stuff

and above all...

A stable environment

- resist the urge to answer on Slack using Giphy

Quick quiz: try to guess the increase in CPU utilization due to this GIF, then measure it with the top utility

Advanced HW tweaking to improve stability

- disable Turbo Boost

```
# Intel
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
# AMD
echo 0 > /sys/devices/system/cpu/cpufreq/boost
```

- Set scaling_governor to "performance"

```
echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
echo performance > /sys/devices/system/cpu/cpu1/cpufreq/scaling_governor
echo performance > /sys/devices/system/cpu/cpu2/cpufreq/scaling_governor
...
```

- Set process priority

```
nice -n 10 ./isogram.bench -test.run=^$ -test.bench=. -test.benchmem
```

More on this here (https://easyperf.net/blog/2019/08/02/Perf-measurement-environment-on-Linux)

Expensive HW tweaking to improve stability

Buy dedicated HW to run your benchmarks

Benchmarking for humans

- create a test executable to do a before/after comparison
- use benchstat (https://godoc.org/golang.org/x/perf/cmd/benchstat)

more on this later!

Go benchmark tooling

Very similar to testing

```
func BenchmarkIsIsogram(b *testing.B) {
    for i := 0; i < b.N; i++ {
        IsIsogram("input test string")
    }
}
```

You can execute them with:

```
go test -bench=.
```

This will first run the tests, and then the benchmarks

Go benchmarking output

Sample output produced

```
goos: linux
goarch: amd64
pkg: path/to/package
BenchmarkName-8   100000   1859 ns/op   57 B/op
PASS
ok   path/to/package    2.042s
```

A line starting with the BenchmarkName followed by the number of cores (GOMAXPROCS) available, then the number of times (100000) the benchmark was run.

The next two numbers indicate:

- the time it took to execute the benchmark, in ns/op (nanoseconds per operation)
- the memory usage, in B/op (bytes of memory allocated per operation)

lower is better.

How benchmarks work

Each benchmark function is repeatedly called with different value for b.N, the number of iterations the benchmark should run for.

b.N starts at 1, if the benchmark function completes in less than 1 second, then b.N is increased and the benchmark function is called again.

b.N increases in a sequence adapted by the benchmark framework, up to 1.000.000

(https://github.com/golang/go/commit/03a79e94ac72f2425a6da0b399dc2f660cf295b6)

```
BenchmarkIsIsogram/01-map-8              200000              8203 ns/op
BenchmarkIsIsogram/02-sorting-8          300000              5534 ns/op
BenchmarkIsIsogram/03-linear-search-8   1000000              1382 ns/op
BenchmarkIsIsogram/04-bool-array-8      3000000               525 ns/op
```

Quick quiz: why each benchmark run for different number of iterations (200000, 300000, 1000000 and 3000000)?

Benchmark option -cpu

Change the value of CPUs used

```
-cpu
```

Example

```
go test -v -run=^$ -bench=. -cpu=1,4,8

goos: linux
goarch: amd64
pkg: performance-and-scalability-of-go-applications/01-introduction/isogram/solution
BenchmarkIsIsogram/01-map                 200000          6886 ns/op
BenchmarkIsIsogram/01-map-4               200000          7246 ns/op
BenchmarkIsIsogram/01-map-8               200000          7951 ns/op
...
PASS
ok      performance-and-scalability-of-go-applications/01-introduction/isogram/solution    19.518s
```

Benchmark option -benchtime

By default, Go will run the benchmark, increasing b.N, until it takes more than 1s to complete.

You can change this behaviour using -benchtime option

Examples

- run the benchmark until it hits the 10s threshold

```
go test -v -run=^$ -bench=. -benchtime=10s
```

- make exactly 100 iterations

```
go test -v -run=^$ -bench=. -benchtime=100x
```

Benchmark option -count

By default, each benchmark is repeated just once.

You can change this behaviour using -count option

Example

- run each benchmark twice

```
go test -v -run=^$ -bench=. -count=2
```

benchstat

benchstat (https://godoc.org/golang.org/x/perf/cmd/benchstat) is a tool to compute statitics about benchmarks

To download it

```
go get -u -v golang.org/x/perf/cmd/benchstat
```

How it works

For each different benchmark listed in an input file, benchstat computes the mean, minimum, and maximum run time, after removing outliers

It greatly helps to understand the statistical significance of your benchmarks

benchstat on a single input file

If invoked on a single input file, benchstat prints the per-benchmark statistics for that file

```
$ go test -v -run=^$ -bench=. -count=10 > isogram.txt
$ benchstat isogram.txt
name                          time/op
IsIsogram/01-map-8            7.60µs ± 3%
IsIsogram/02-sorting-8        5.34µs ± 1%
IsIsogram/03-linear-search-8  1.31µs ± 2%
IsIsogram/04-bool-array-8      529ns ± 0%
```

It computes the mean value and gives the variation across the samples

Lower variation means **better** stability!

benchstat on multiple input files

benchstat is very useful to analyze the impact of your changes

```
$ go test -v -run=^$ -bench=. -count=10 > old.txt

// changes to the code

$ go test -v -run=^$ -bench=. -count=10 > new.txt
$ benchstat old.txt new.txt
name                          old time/op  new time/op  delta
IsIsogram/01-map-8            7.52µs ± 2%  7.40µs ± 5%    ~     (p=0.105 n=10+10)
IsIsogram/02-sorting-8        5.23µs ± 3%  5.13µs ± 3%  -1.95%  (p=0.016 n=10+10)
IsIsogram/03-linear-search-8  1.32µs ± 1%  1.32µs ± 1%    ~     (p=0.669 n=10+10)
IsIsogram/04-bool-array-8      530ns ± 0%   533ns ± 0%  +0.71%  (p=0.000 n=9+10)
```

A ~ means that the difference between the benchmarks is not statistically significant

Benchmarking allocations

Using the -benchmem option, go will report the heap allocations per operation

```
$ go test -v -run=^$ -bench=. -benchmem
goos: linux
goarch: amd64
pkg: performance-and-scalability-of-go-applications/01-introduction/isogram/solution
BenchmarkIsIsogram/01-map-8                200000              7458 ns/op            1273 B/op         11 allocs/
BenchmarkIsIsogram/02-sorting-8            300000              5266 ns/op             928 B/op         25 allocs/
BenchmarkIsIsogram/03-linear-search-8             1000000              1315 ns/op              48 B/op          3
BenchmarkIsIsogram/04-bool-array-8                3000000               532 ns/op               0 B/op          0
PASS
ok      performance-and-scalability-of-go-applications/01-introduction/isogram/solution 6.692s
```

Note how less allocations means better performance... More on this later!

Creating a benchmark "gold standard" file

Instead of saving the textual output of a benchmark run, you can save a binary to reproduce the benchmark

```
$ go test -c -o isogram.bench
```

This way you will be able to benchmark your code (with whatever set of options you like) for later comparison

```
$ ./isogram.bench -test.run=^$ -test.bench=. -test.benchmem
```

Benchmarking pitfalls

Consider the following benchmark

```
func NaturalNumbersSum(n int) int {
    return n * (n + 1) / 2
}

func BenchmarkNaturalNumbersSum(b *testing.B) {
    for i := 0; i < b.N; i++ {
        NaturalNumbersSum(i)
    }
}
```

And its results

```
...
BenchmarkNaturalNumbersSum-8    2000000000               0.30 ns/op
PASS
```

0.30 ns/op is the time of a single clock cycle...

Benchmarking pitfalls

Using

```
go test -v -run=^$ -bench=. -gcflags="-m"
```

We can see that the call to NaturalNumbersSum is inlined by the compiler...

```
...
./gauss.go:3:6: can inline NaturalNumbersSum
./gauss_test.go:7:20: inlining call to NaturalNumbersSum
...
```

Benchmarking pitfalls

And since the body of the function has no side effects, it is optimized away by the CPU!

```
"".BenchmarkNaturalNumbersSum STEXT nosplit size=23 args=0x8 locals=0x0
    0x0000 00000 (gauss_test.go:5)       TEXT    "".BenchmarkNaturalNumbersSum(SB), NOSPLIT|ABIInternal, $0-8
    ...
    0x0000 00000 (gauss_test.go:6)       MOVQ    "".b+8(SP), AX
    0x0005 00005 (gauss_test.go:6)       XORL    CX, CX
    0x0007 00007 (gauss_test.go:6)       JMP     13
    0x0009 00009 (gauss_test.go:6)       INCQ    CX
    0x000c 00012 (gauss_test.go:7)       XCHGL   AX, AX
    0x000d 00013 (gauss_test.go:6)       CMPQ    264(AX), CX
    0x0014 00020 (gauss_test.go:6)       JGT     9
    ...
    0x0016 00022 (<unknown line number>)    RET
```

Fixing the benchmark

Force the body of the for loop to produce effects visible outside the loop itself

```
var result int

func BenchmarkNaturalNumbersSum(b *testing.B) {
    var r int
    for i := 0; i < b.N; i++ {
        r = NaturalNumbersSum(i)
    }
    result = r
}
```

inspired by issue #14183 (https://github.com/golang/go/issues/14813#issue-140603392)

Fixing the benchmark

```
"".BenchmarkNaturalNumbersSum STEXT nosplit size=54 args=0x8 locals=0x0
        0x0000 00000 (gauss_test.go:7)       TEXT    "".BenchmarkNaturalNumbersSum(SB), NOSPLIT|ABIIn
ternal, $0-8
    ...
        0x0000 00000 (gauss_test.go:10)      MOVQ    "".b+8(SP), AX
        0x0005 00005 (gauss_test.go:10)      XORL    CX, CX
        0x0007 00007 (gauss_test.go:10)      XORL    DX, DX
        0x0009 00009 (gauss_test.go:10)      JMP     37
        0x000b 00011 (gauss_test.go:11)      XCHGL   AX, AX
        0x000c 00012 (gauss.go:4)    LEAQ    1(CX), BX
        0x0010 00016 (gauss.go:4)    IMULQ   BX, CX
        0x0014 00020 (gauss.go:4)    MOVQ    CX, SI
        0x0017 00023 (gauss.go:4)    SHRQ    $63, CX
        0x001b 00027 (gauss.go:4)    LEAQ    (CX)(SI*1), DX
        0x001f 00031 (gauss.go:4)    SARQ    $1, DX
        0x0022 00034 (gauss_test.go:10)      MOVQ    BX, CX
        0x0025 00037 (gauss_test.go:10)      CMPQ    264(AX), CX
        0x002c 00044 (gauss_test.go:10)      JGT     11
    ...
        0x002e 00046 (gauss_test.go:14)      MOVQ    DX, "".result(SB)
        0x0035 00053 (gauss_test.go:15)      RET
```

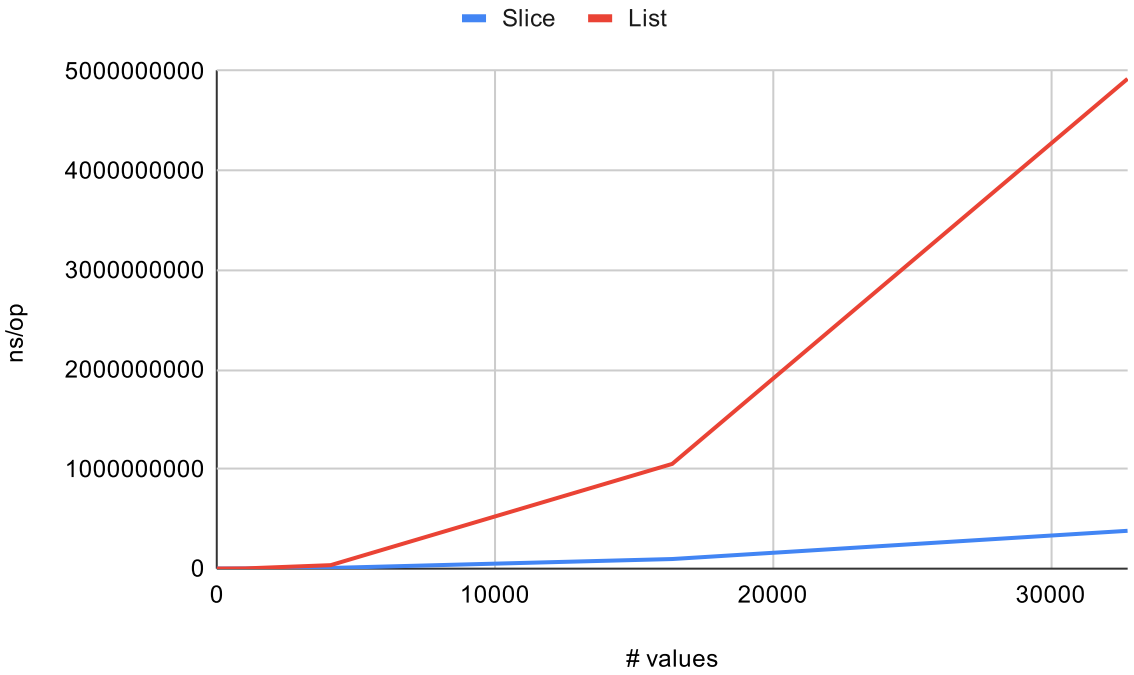
Insert and Remove exercise

Follow the instructions inside

```
03-benchmarking/INSTRUCTIONS.md
```

Happy coding!

Insert and Remove exercise solution



Predictability of memory access makes slice the winner!

Insert and Remove exercise solution

Useful things:

- benchstat
- -count
- -benchtime

to check benchmark stability and get more meaningful results for longer execution test cases

Take home message(s)

- don't store data unnecessarily (more on that later)
- keep data compact (again... more on that later)
- access memory in a predictable manner
- don't make statements about performance without measurements

"I emphasize the importance of cache effects. In my experience, all but true experts tend to forget those when algorithms are discussed."

Bjarne Stroustrup - from Are lists evil? (https://isocpp.org/blog/2014/06/stroustrup-lists)

Profiling & Tracing

Benchmarking vs Profiling

Benchmarking allows us to inspect the performance of a specific piece of code.

Profiling gives us a way to understand where our application is spending the most of the time and using the majority of resources.

What is a profile?

A Profile is a collection of stack traces showing the call sequences that led to instances of a particular event, such as allocation.

Types os profiles

- CPU
- Heap
- Block
- Mutex

In this workshop we'll focus on CPU profiles and Heap profiles

How the CPU profiler works

An handler to the `SIGPROF` signal is registered and a timer is started. The timer will count only when the program is actively consuming CPU cycles.
Every 10ms, the signal `SIGPROF` is delivered, and the handler unwinds the stack, registering the stack trace.

How the Heap profiler works

These profiles don't require the delivery of any signals, the runtime itself will collect samples while allocating memory on the heap.
Not all allocations are sampled, if you like to see every one of them, you should set

```
runtime.MemProfileRate
```

to 1 as early as possibile in your program.

Profiling benchmarks

You can generate profiles directly when launching benchmarks

```
go test -run=^$ -bench=. -cpuprofile cpu.prof
go test -run=^$ -bench=. -memprofile mem.prof
```

Profiling web servers

For web servers, you can use the `net/http/pprof` package to expose an endpoint where you can collect various profiles type

1) Import `pprof` package with a blank import

```
import _ net/http/pprof
```

2) If your application is not already running an HTTP server, then start one

```
http.ListenAndServe(":8080", nil)
```

You can find an example in

```
04-profiling/profilingexamples/server
```

Profiling whole programs

To profile a whole program you can use the `runtime/pprof` package.

Just add this lines at the start of your `main` function

```
cpuProfile := flag.String("cpuprofile", "", "write cpu profile to file")
flag.Parse()

if *cpuProfile != "" {
    f, err := os.Create(*cpuProfile)
    if err != nil {
        log.Fatal(err)
    }
    pprof.StartCPUProfile(f)
    defer pprof.StopCPUProfile()
}
```

Then launch your program with the flag

```
-cpuprofile cpu.prof
```

To save a CPU profile inside `cpu.prof` file

pprof live demo

To analyse the profile you generated, you can use the Go `pprof` tool, with a textual interface

```
go tool pprof <profile>
```

or a web based one

```
go tool pprof -http=:9090 <profile>
```

Let's explore both of them with a live demo!

Tracing

Excerpt from the Go documentation (https://golang.org/pkg/runtime/trace/)

"The execution trace captures a wide range of execution events related to goroutines, syscalls, GC and heap size, processor start/stop, etc.
A precise nanosecond-precision timestamp and a stack trace is captured for most events."

While profiling helps us understand **what** happened, tracing is very useful to understand **when** things happened

Tracing benchmarks

You can generate execution traces directly when launching benchmarks

```
go test -run=^$ -bench=. -trace trace.out
```

Tracing web servers

1) Import `pprof` package with a blank import

```
import _ net/http/pprof
```

2) If your application is not already running an HTTP server, then start one

```
http.ListenAndServe(":8080", nil)
```

3) Collect the trace

```
go tool trace http://localhost:8080/debug/trace?seconds=10 > trace.out
```

Tracing whole programs

To trace a whole program you can use the `runtime/trace` package.

Just add this lines at the start of your `main` function

```
traceFile := flag.String("trace", "", "trace file name")
flag.Parse()

if *traceFile != "" {
    f, err := os.Create(*traceFile)
    // check and handle error
    if err := trace.Start(f); err != nil {
        log.Fatalf("could not start trace: %v", err)
    }
    defer trace.Stop()
}
```

Then launch your program with the flag

```
-trace trace.out
```

To save an execution trace inside `trace.out` file

Tracing cheatsheet

To analyze a trace

```
go tool trace <trace>
```

- use ? to show Tracing Help
- navigate using `W`, `S`, `A` and `D` keys
- hold `Shift` to drag a box with your mouse pointer to ease selection of tiny events

WARNING: It can be viewed **only** using the Chrome web browser... may our RAM forgive us!

Let's explore it with a live demo!

Run Length Encoding exercise

Follow the instructions inside

```
04-profiling/INSTRUCTIONS.md
```

Happy coding!

Run Length Encoding exercise solution

The program opens an input file, read all its content one byte at a time to encode or decode it, then write the results in an output file.

Run Length Encoding exercise solution

As a reference, let's compare runlengthencoding with the gzip/gunzip utilities.

```
$ time gzip divina-commedia.txt
real    0m0,056s
user    0m0,052s
sys     0m0,004s

$ time gunzip divina-commedia.txt.gz
real    0m0,043s
user    0m0,006s
sys     0m0,006s

$ time ./runlengthencoding e divina-commedia.txt
real    0m1,617s
user    0m0,568s
sys     0m1,065s

$ time ./runlengthencoding d encoded.rle
real    0m2,075s
user    0m0,845s
sys     0m1,249s
```

Lot of room for improvement!

Run Length Encoding exercise solution

Let's inspect this code adding a `-cpuprofile` option to store the profile in a file

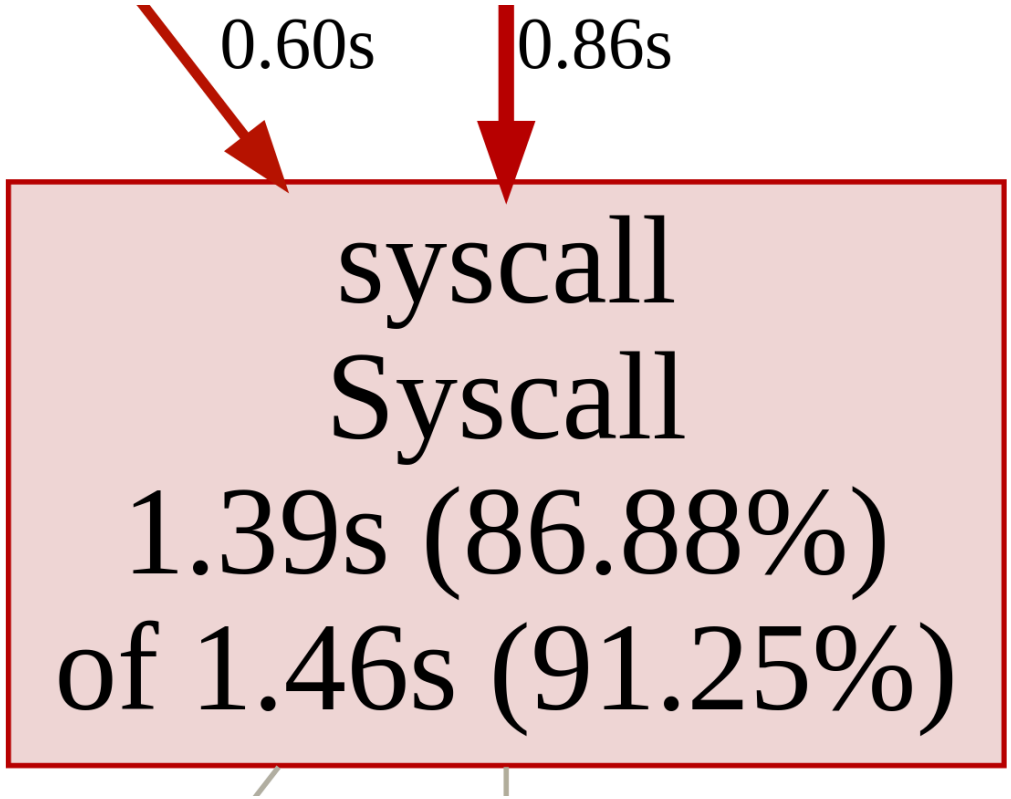
```
./runlengthencoding -cpuprofile cpu.prof e divina-commedia.txt
```

Now, we can inspect this profile with:

```
go tool pprof -http=:8080 cpu.prof
```

What can you see from this profile?

Run Length Encoding exercise solution



Run Length Encoding exercise solution

We are spending the majority of the time doing syscalls.

A read syscall for very few bytes in memory can cost us from ~50 to ~100ns (http://arkanis.de/weblog/2017-01-05-measurements-of-system-call-performance-and-overhead) (and Meltdown/Spectre mitigation patches are making it worse (https://www.zdnet.com/article/linux-meltdown-patch-up-to-800-percent-cpu-overhead-netflix-tests-show/))

We are using unbufferead read and write, 1 byte at a time.

For the encoding, the total number of syscalls equals the number of encoded bytes.

Run Length Encoding exercise solution

To solve the problem, we can simply buffer our read operations using bufio.NewReader

(https://golang.org/pkg/bufio/#NewReader)

```
$ time ./solution e divina-commedia.txt
real    0m0,020s
user    0m0,017s
sys     0m0,004s

$ time ./solution d encoded.rle
real    0m0,024s
user    0m0,020s
sys     0m0,005s
```

Here, you should note how the sys time has dropped down!

Parallel quicksort exercise solution

Live demo

The Go scheduler

Requirements & constraints

- lightweight (up to 1 million goroutines per process)
- parallel and scalable
- simple (minimal API with just a single keyword)

```
go func() {
    // ...
}()
```

- efficient handling of IO calls and syscalls
- one goroutine per connection model, without callbacks

Why a new scheduler?

- OS thread context switch latency: from ~1500 to ~2000ns
- Goroutines context switch latency: from ~150 to ~200ns

Tested on my laptop, using Linux 4.19 and Go 1.13

More info about the benchmarks here (https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads)

Considering a medium frequency of 3 GHz CPU, and an *Istructions Per Cycle* of ~4, we can execute ~12 instructions per ns

- OS thread context switch cost ranges from ~18000 to ~24000 instructions
- Goroutines context switch cost ranges from ~1800 to ~2400 instructions

A ten times improvement!

M:P:N scheduler

Based on a M:N threading models, where we **multiplex** N goroutines onto M OS threads

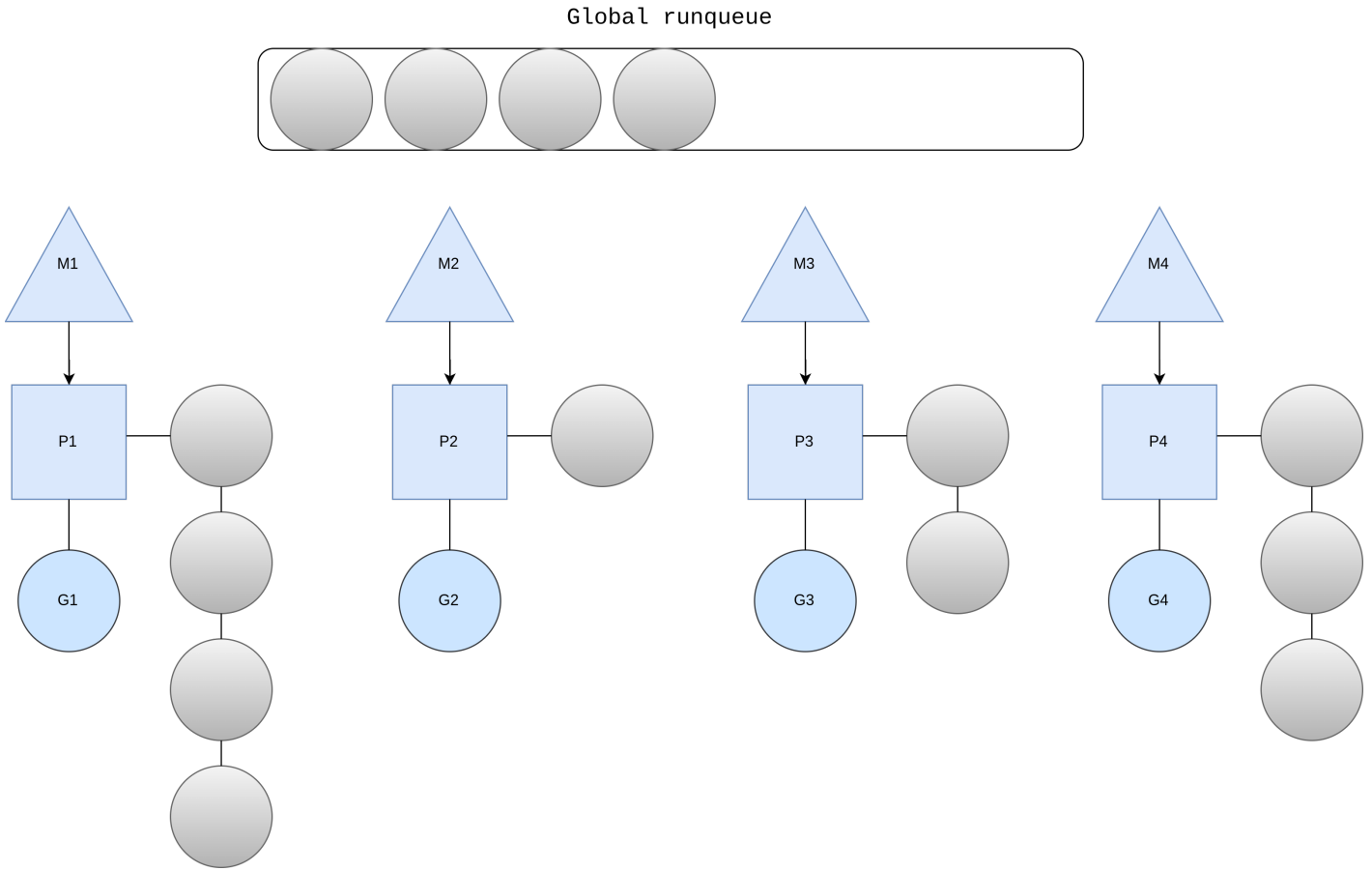
- **M** stands for *machine* and represents the OS thread
- **P** stands for *processor* and is an execution context for Go code
- **G** stands for *goroutine*

P holds the list of runnable goroutines and other Go execution contexts caches

G holds the Instruction Pointer and the goroutine stack

To execute a G, a M needs to hold a P

Scheduler steady state



At startup, the Go runtime creates **GOMAXPROCS** Ps, where **GOMAXPROCS** is set to the number of available cores

Cooperative scheduling

Each G is assigned a time slice of 10ms, when it is exhausted, the G is put in the global runqueue to ensure **fairness**

Though the use of a time slice, the preemption checks are inserted by the compiler at:
- function calls (..but not every ones!)
- memory allocations
- communication primitives (creating goroutines, acquiring mutexes, sending data to a channel, etc.)

Even if the Go scheduler behaves as **preemptive**, it is **cooperative**

Cooperative scheduling gotcha

Quick quiz: can you guess the output of this?

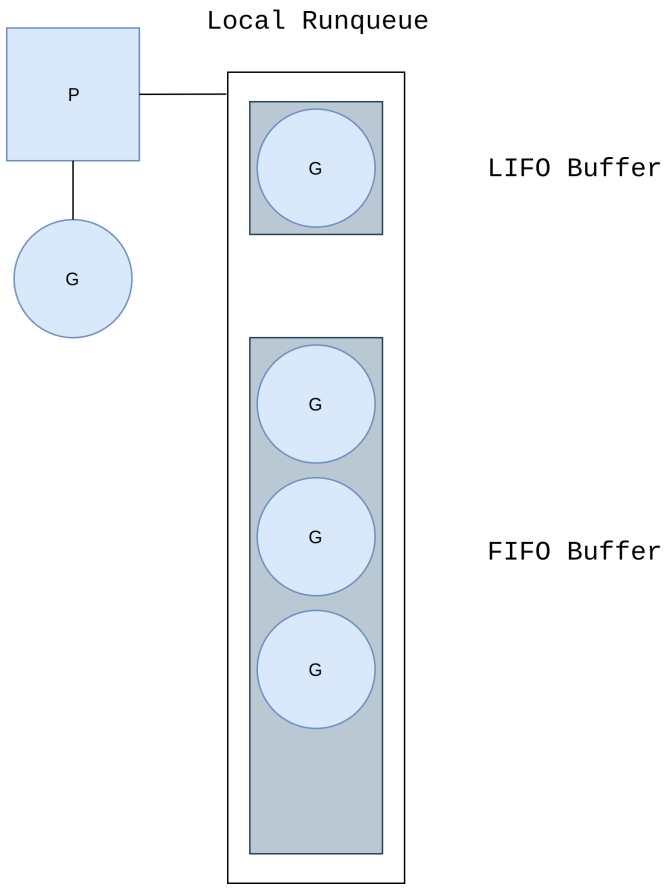
```
package main

import "fmt"
import "time"
import "runtime"

func main() {
    var x int

    for i := 0; i < runtime.GOMAXPROCS(0); i++ {
        go func() {
            for { x++ }
        }()
    }

    time.Sleep(time.Second)
    fmt.Println(x)
}
```

Runqueues architecture

- one global runqueue
- one local runqueue for each P

Quick quiz: why distributed runqueues instead of a single global runqueue?

Architecture of a local runqueue



Quick quiz: Why using that additional LIFO slot?

Network I/O

For network I/O, Go uses the asynchronous interface of the underlying OS, like **epoll** on Linux.

Whenever you open or accept a connection in Go, the file descriptor that backs it is set to **non-blocking** mode.

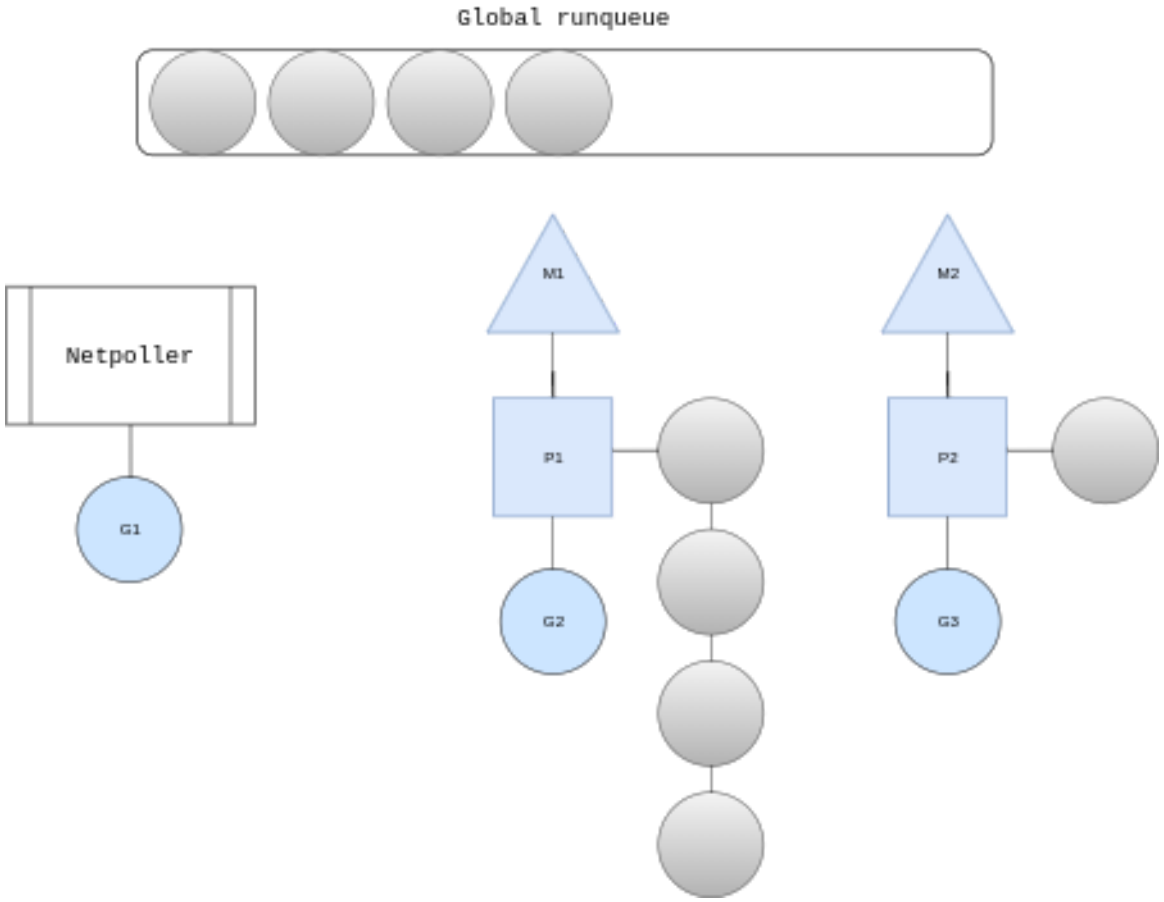
When a G tries to read or write to a connection, if the file descriptor isn't ready, G is scheduled out of the M and enqueued in the **netpoller** queue.

The netpoller is periodically cheched: when a file descriptor is ready, the netpoller returns the G blocked on that file descriptor to be scheduled.

Take home message

If you think at the above scenario from the point of view of the OS, you'll realize that Go is turning an **I/O bound** load in a **CPU bound** one

The netpoller



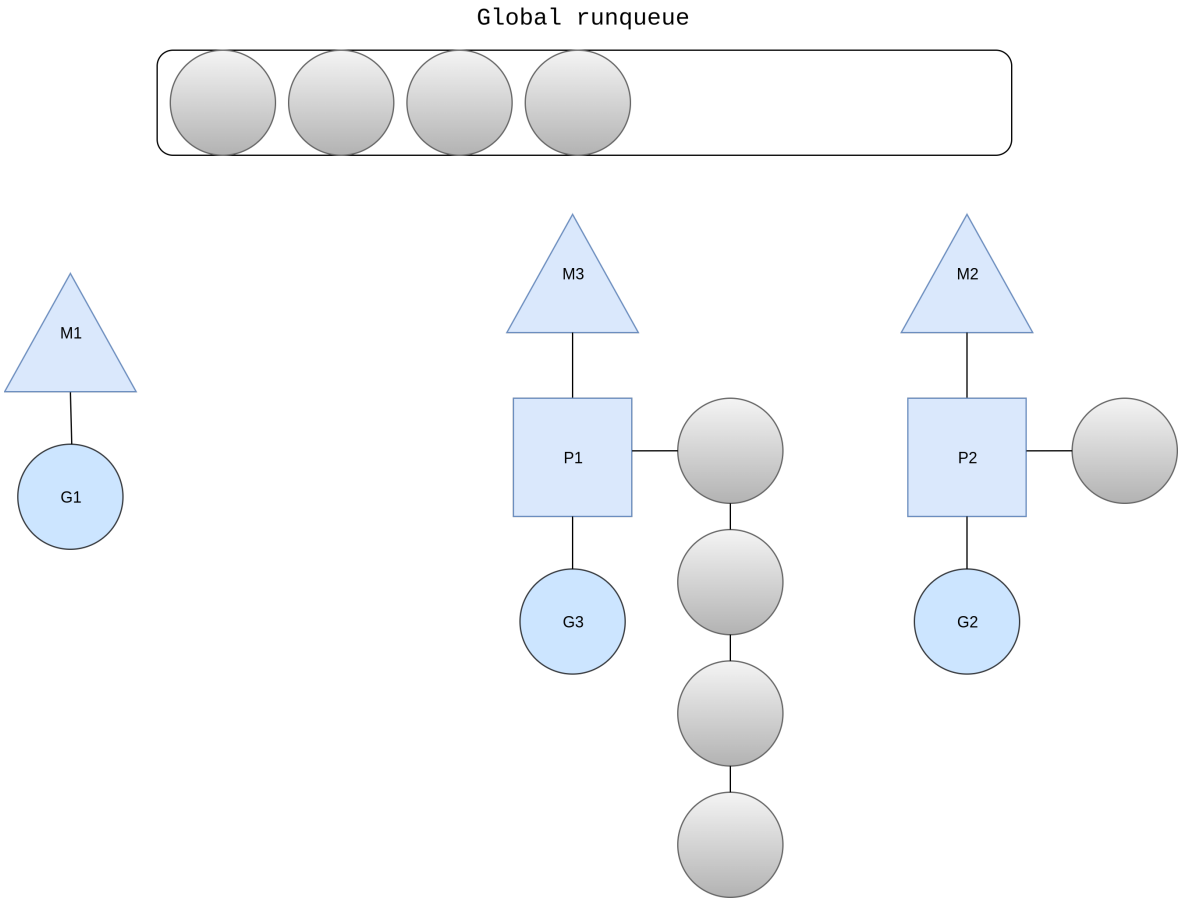
Syscalls

Suppose that G1 wants to call into a blocking syscall. How the runtime will manage this?

- Releases the current P2 from M2 (syscall isn't Go code, it doesn't need a P to execute!)
- Spawns a new M3, and hand off P3
- blocks M3 on the syscall
- using M3, P3 can continue scheduling the Gs in the local runqueue
- When the OS is done with the syscall, M2 is put to sleep in a thread pool
- G2 is put on the global runqueue

This is why, even if **GOMAXPROCS** is equal to 1, Go programs will run multiple threads

Syscalls



Scheduling round decision

```
runtime.schedule() {
    // only 1/61 of the time, check the global runnable queue for a G.

    // if not found, check the local queue

    // if not found call findrunnable
}

runtime.findrunnable() {
    // if not found, check the global runqueue

    // check the network poller

    // Steal work from other P's.
}
```

Since Go uses distributed runqueues, it needs a mechanism to balance load between Ps **work stealing**

When a P with a non empty runqueue is found, the current P steals half of it

Work stealing and M:P:N threading model

Quick quiz: why should we separate P and M entities?

Scheduling exercises

Follow the instructions inside

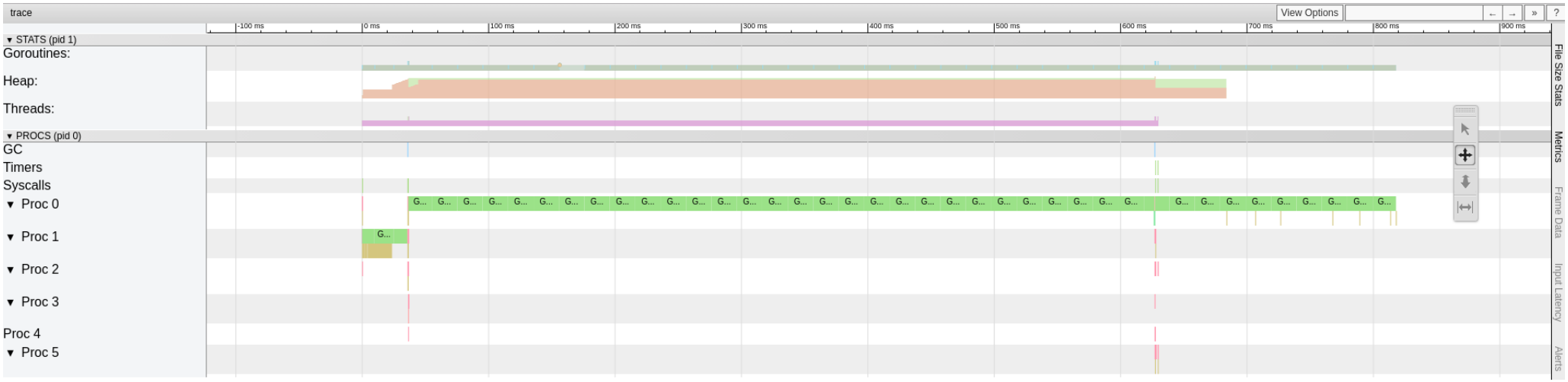
```
05-scheduler/INSTRUCTIONS.md
```

Happy coding!

Gaussian Blur exercise solution

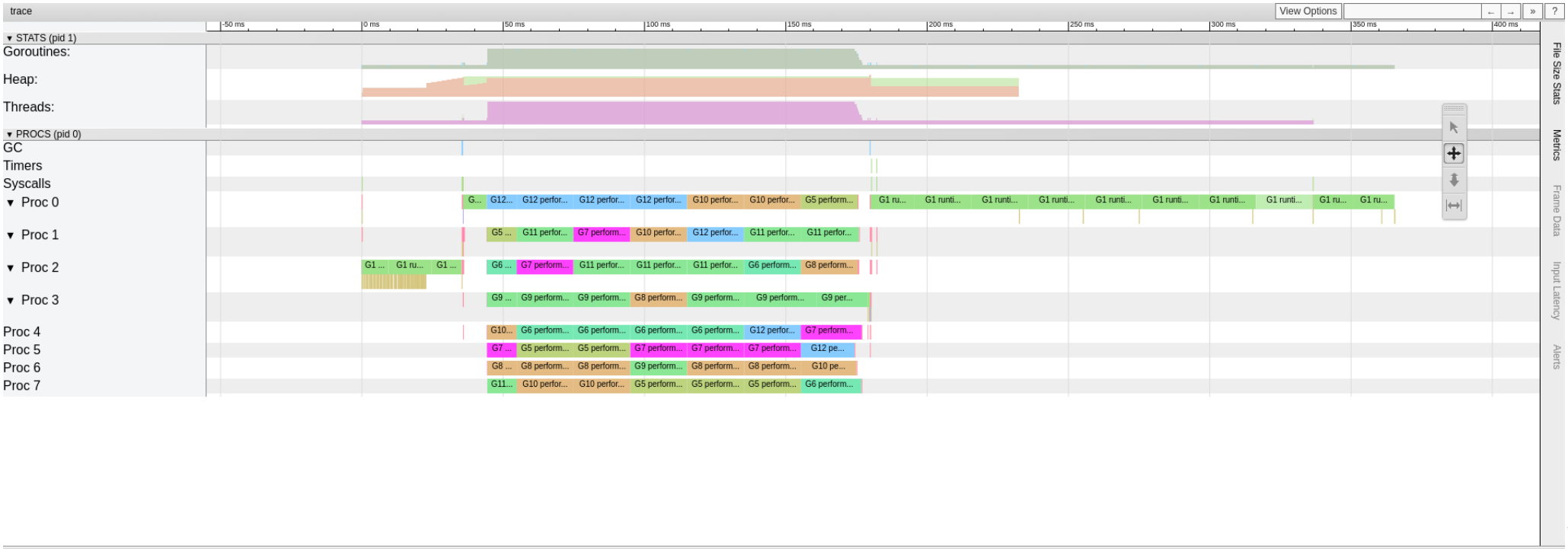
This is a **CPU bound** load

Using the go trace tool, we can see that with a single goroutine we get low CPU utilization



Gaussian Blur exercise solution

Let's write a concurrent version of `Convolve` and set the number of Goroutines equal to **GOMAXPROCS**



Gaussian Blur exercise solution

Furtherly increasing the goroutines (i.e.: one per row) does not affect performance too much

Instead, we can see a worsening if Gs are too much (i.e.: one per pixel), due to goroutine management overhead

Note that there is still a substantial amount of the trace (writing the transformed image) where we are using just one G and one M

Even if we increase the HW cores, thus increasing the parallelism level of the convolution part, there will be a point in which we won't gain anything. This is due to the sequential part dominating the execution time.

This effect is described by the Amdahl law (https://en.wikipedia.org/wiki/Amdahl%27s_law)

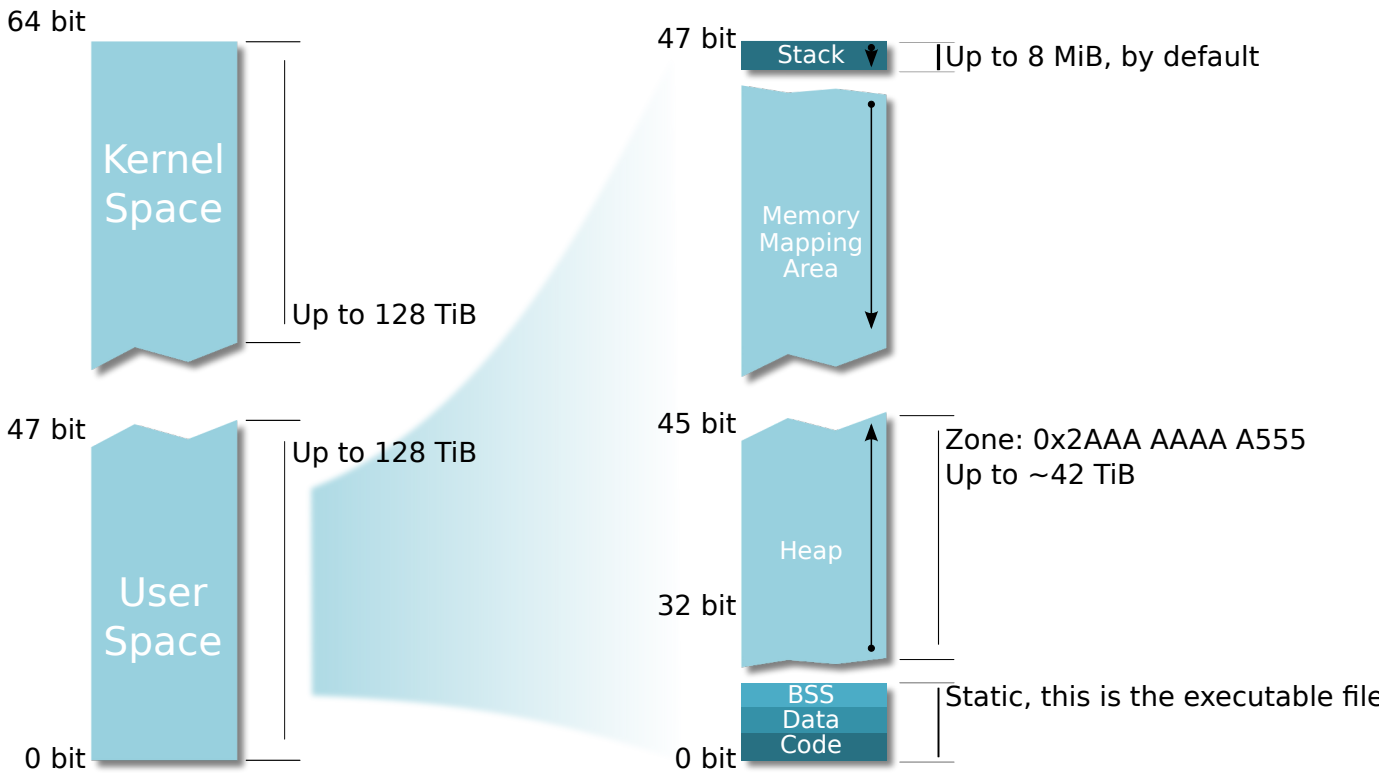
WordLineCount exercise solution

What about an I/O bound load? We get a great performance improvement using concurrency, but increasing the parallelism level does not change anything

```
$ time GOMAXPROCS=1 ./wordlinecount -word This
real    0m10,551s
user    0m0,278s
sys     0m0,067s

$ time GOMAXPROCS=8 ./wordlinecount -word This
real    0m10,002s
user    0m0,280s
sys     0m0,091s

$ time GOMAXPROCS=1 ./solution -word This -goroutines 8
real    0m5,977s
user    0m0,220s
sys     0m0,091s

$ time GOMAXPROCS=8 ./solution -word This -goroutines 8
real    0m6,254s
user    0m0,291s
sys     0m0,085s
```

The Go memory allocator & garbage collector

Process virtual memory layout



So Kernel + User Spaces add for 256 TiB which is a tiny part of the 16 777 216 TiB addressable over 64 bit!

Stack frame

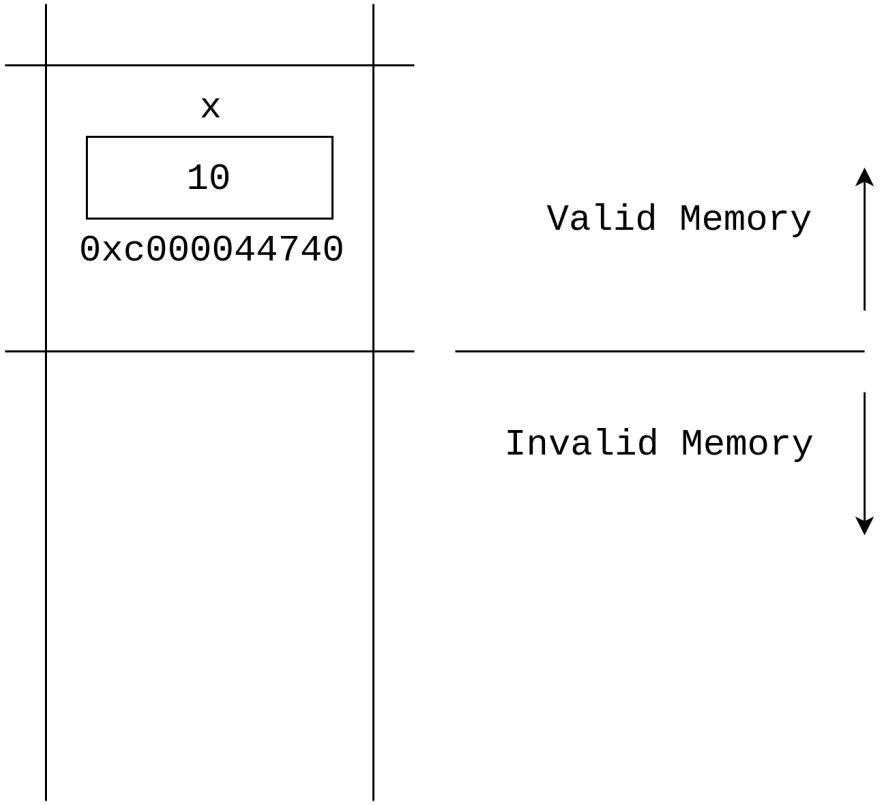
Stack is managed in **frames**: individual memory space for each function call

Reserving a new frame and invalidate one can be done just bumping up or down the *Stack Pointer* register

```
func main() {
    x := 10
    f()

    println(&x)
}

//go:noinline
func f() {
    y := 20
    println(&y)
}
```

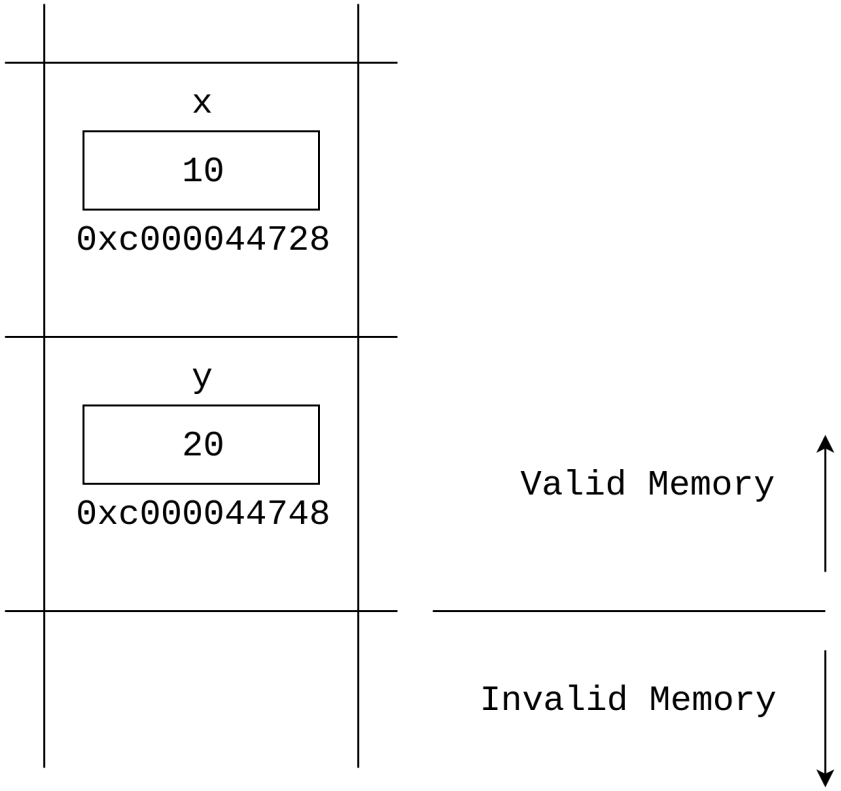
Stack frame

frame for the `main` function



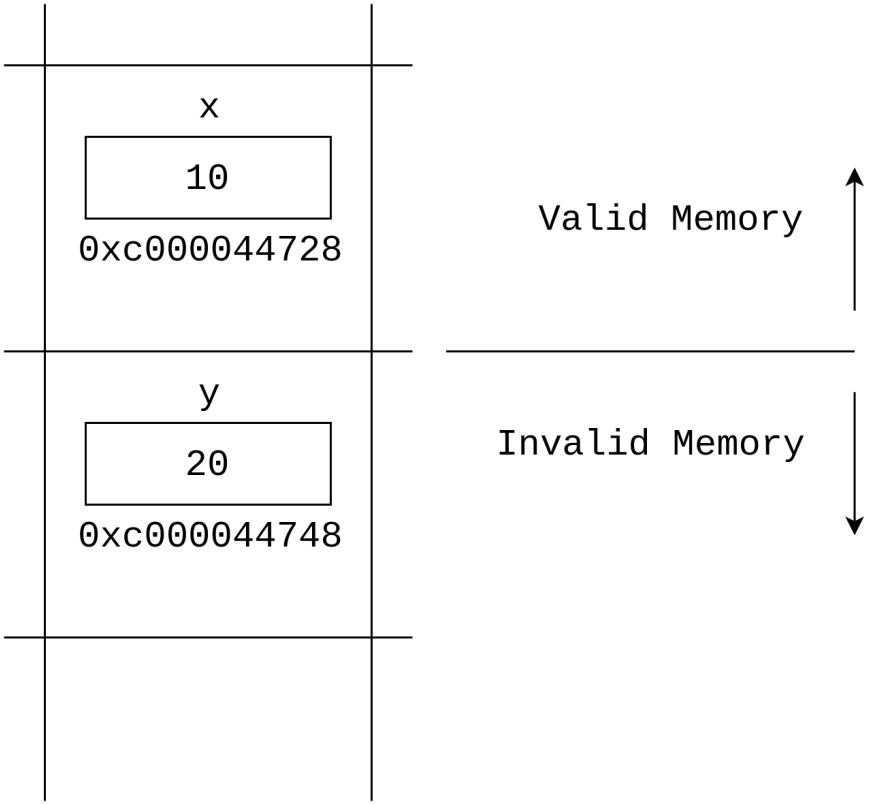
Stack frame

frame for the f function



Stack frame

after returning from the f function



f frame is not deleted, the runtime simply update the *Stack Pointer* register value

Goroutine stack size

```
import "fmt"

type S struct {
        a, b int
}

// String implements the fmt.Stringer interface
func (s *S) String() string {
        return fmt.Sprintf("%s", s) // Sprintf will call s.String()
}

func main() {
        s := &S{a: 1, b: 2}
        fmt.Println(s)
}
```

from Why is a Goroutine's stack infinite ? (https://dave.cheney.net/2013/06/02/why-is-a-goroutines-stack-infinite)

Goroutine stack size

```
$ go run stack-size.go
runtime: goroutine stack exceeds 1000000000-byte limit
fatal error: stack overflow
```

from src/runtime/proc.go

```
// Max stack size is 1 GB on 64-bit, 250 MB on 32-bit.
if sys.PtrSize == 8 {
    maxstacksize = 1000000000
} else {
    maxstacksize = 250000000
}
```

But

```
$ ulimit -s
8192
```

Quick quiz: how is it possible to have a 1 GB stack size if thread stack size is limited to 8 MB?

System stack vs User stack

Excerpt from src/runtime/HACKING.md

```
Every non-dead G has a _user_stack_ associated with it, which is what user Go code executes on.
User stacks start small (2K) and grow or shrink dynamically.

Every M has a _system_stack_ associated with it and, on Unix platforms, a signal stack.
System and signal stacks cannot grow, but are large enough to execute runtime and cgo code
(8K in a pure Go binary).
```

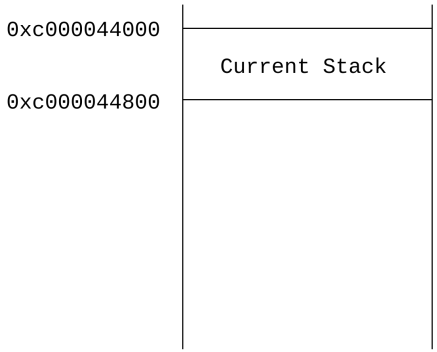
Goroutines *user stacks* are allocated on the **heap** and **dynamically resized**!

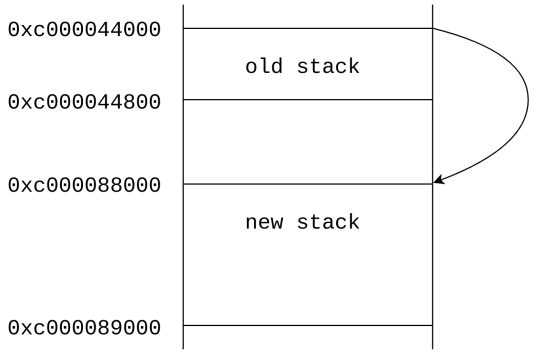
Quick quiz: can you guess why goroutines cannot share stack allocated values just passing pointers?

Go copying stacks - grow

Set `stackdebug` runtime variable to 1 and execute again the code above

```
// copy stack, size is increased from 2048 to 4096 bytes
copystack gp=0xc000000180 [0xc000044000 0xc000044350 0xc000044800] ->
    [0xc000088000 0xc000088b50 0xc000089000]/4096
stack grow done
...
// copy stack, size is increased from 4096 to 8192 bytes
copystack gp=0xc000000180 [0xc000088000 0xc0000882f8 0xc000089000] ->
    [0xc000078000 0xc0000792f8 0xc00007a000]/8192
stack grow done
```





Go copying stacks - grow

Go compiler will insert stack size check at functions prologue, if needed

```
;; stack split prologue
0x0000 00000 MOVQ    (TLS), CX
0x0009 00009 CMPQ    SP, 16(CX)
0x000d 00013 JLS    142

;; function body

;; stack split epilogue
0x008e 00142 NOP
0x008e 00142 PCDATA    $1, $-1
0x008e 00142 PCDATA    $0, $-1
0x008e 00142 CALL    runtime.morestack_noctxt(SB)
```

Go compiler is smart enough to identify functions that don't need a growing stack, marking them as `NOSPLIT`

```
TEXT    "".f(SB), NOSPLIT|ABIInternal, $0-0
```

Go copying stacks - shrink

The stack is shrinked, if needed, by the garbage collector

```
//go:noinline
func f() {
    var x [1000]int
    println(&x)
}

func main() {
    f()
    println("before GC")
    runtime.GC()
}
```

will output

```
copystack ... -> [0xc00008c000 0xc00008ff38 0xc000090000]/16384
stack grow done
...
before GC
...
shrinking stack 16384->8192
copystack ... -> [0xc000054000 0xc000055eb8 0xc000056000]/8192
```

Heap allocations

Consider the following snippet

```
type s struct {
    v int
}

func newStruct() *s {
    return &s{10}
}
```

Quick quiz: where that `s` object will be allocated? On the stack or on the heap?

Heap allocations

The Go Programming Language Specification (https://golang.org/ref/spec) does not define where items will be allocated

However, certain requirements will exclude some choices: stack allocation requires that

- the lifetime of a variable can be determined at compile time
- the memory footprint of a variable can be determined at compile time

To decide where variables should be allocated, the Go compiler uses a technique called **escape analysis**

We can inspect the compiler decisions using

```
go build -gcflags="all=-m"
```

More on this in a moment!

Allocations and performance

From the kernel point of view, a Go program allocates variables using
- the stack (that is, the **system stack**) of each threads the program starts
- the heap, managed with a custom allocator

But, from the perspective of a Go programmer, our goroutines allocate variables
- on the **user stack**, backed by heap allocations in the runtime
- on the heap

Quick quiz: why are stack allocations considered to be cheaper than heap allocations?

Go memory allocations

Go custom allocator guarantees to

1) efficiently satisfy allocations of a given size, avoiding fragmentation

Go allocator uses **spans** of like-sized objects, defining ~70 different class sizes

2) avoid locking in the common case

Go allocator uses a hierarchies of caches, where the first layer is a per P **mcache**

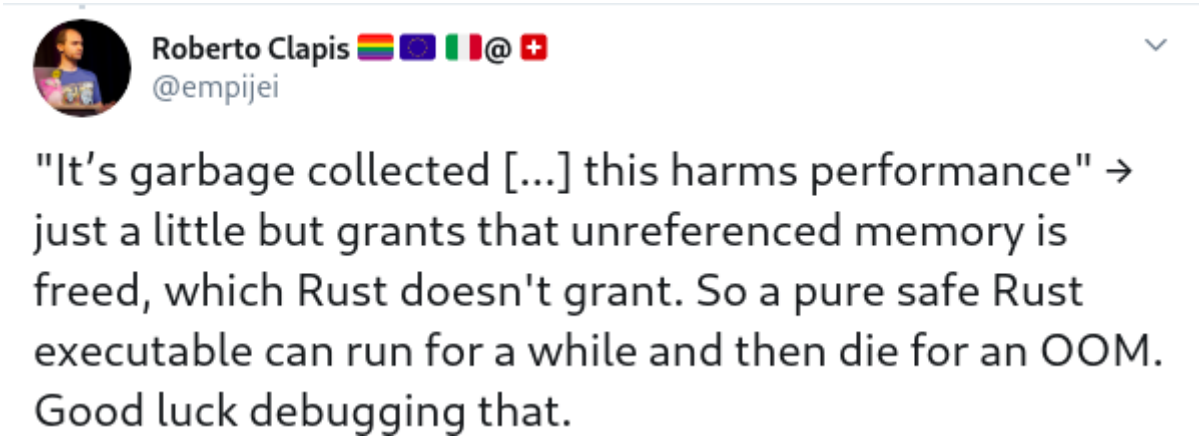
Quick quiz: why each P should get a memory cache?

Go memory reclaiming

- **Stack**: to invalidate a memory frame, we just need to change the value of the *Stack Pointer* register. If needed, the garbage collector kicks in to shrink the stack, but that is a very fast operation
- **Heap**: to reclaim unreachable memory, we need a round of garbage collection

Performance vs safety tradeoff

From a discussion about Go vs Rust



Remember that Go is heavily used to build high-performance web services, so OOMs are to be particularly feared.

Garbage collection in Go

Concurrent Tri-color Mark & Sweep collector

The concurrent implementation guarantees lower latencies at the cost of a reduced throughput, due to an increased total amount of GC work

Tri-color mark & sweep algorithm

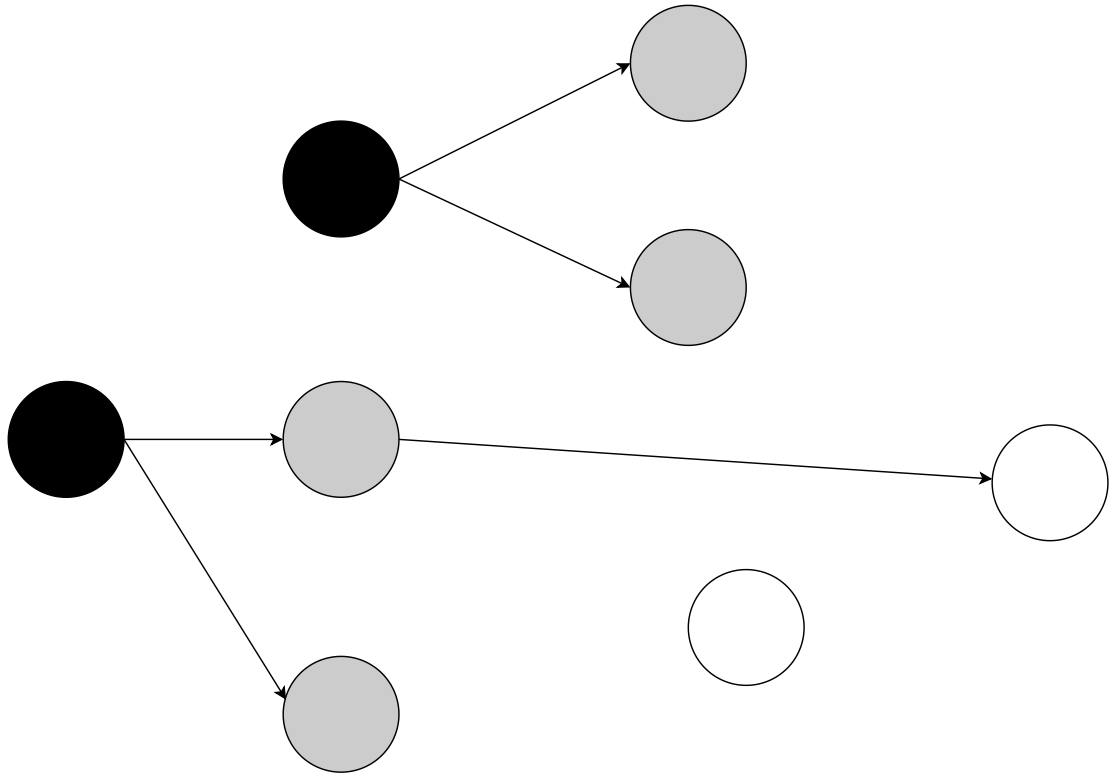
We define three types of object sets

- **white set**: objects that are candidate to be collected
- **black set**: objects that are reachable and that do not have references to objects in the white set
- **grey set**: objects that are reachable, but still not entirely scanned for references to white objects

We start from the root references like top-level variables and stack allocated references

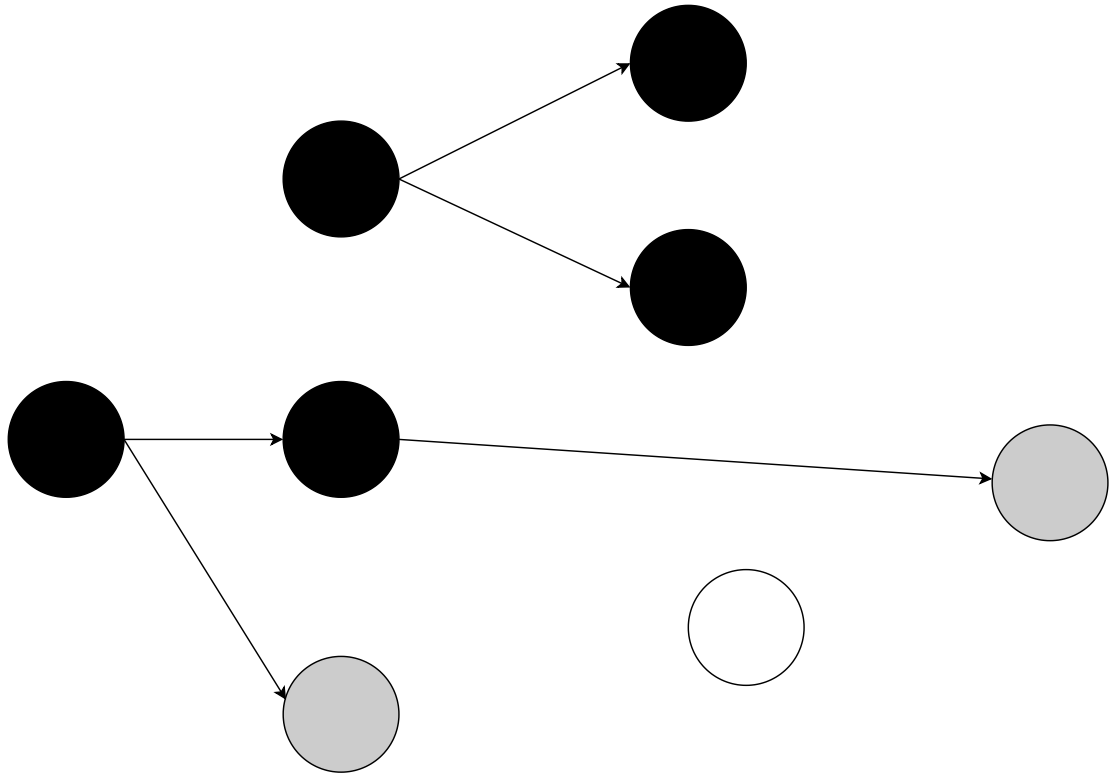
Tri-color mark & sweep algorithm

During the tri-color marking phase, objects:
- start as **white** at the beginning
- become **grey** when they are found to be alive
- finally become **black** when all their references have been traversed



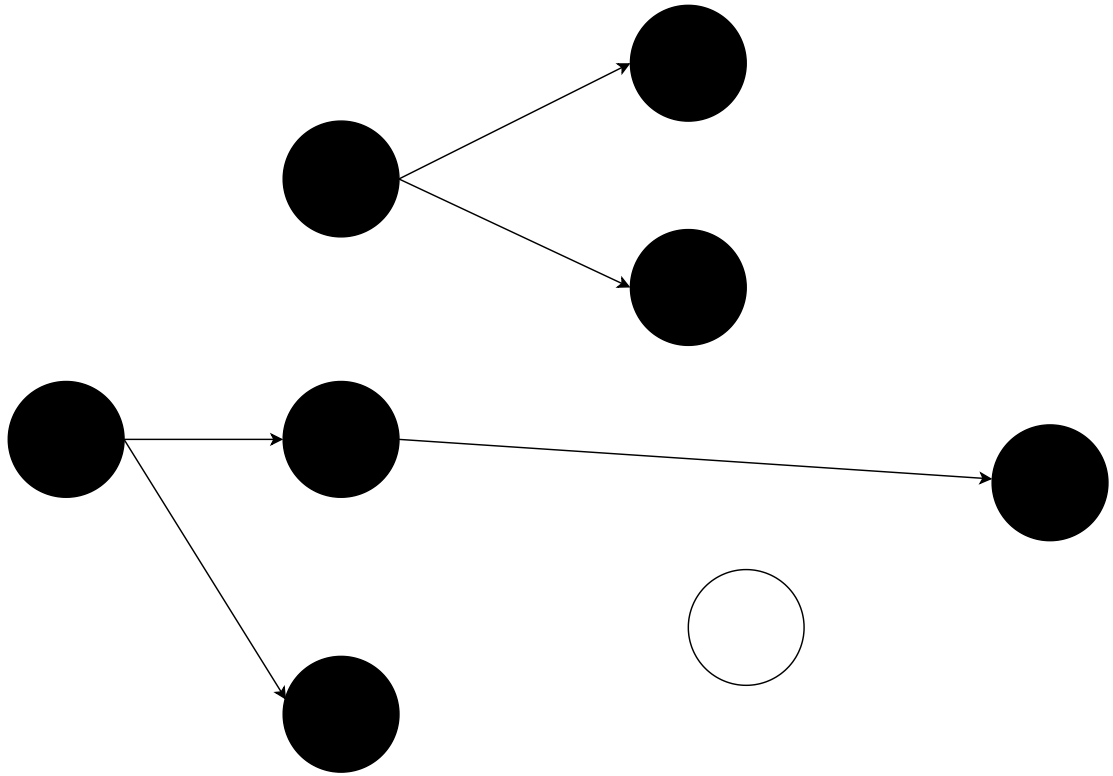
Tri-color mark & sweep algorithm

During the tri-color marking phase, objects:
- start as **white** at the beginning
- become **grey** when they are found to be alive
- finally become **black** when all their references have been traversed



Tri-color mark & sweep algorithm

During the tri-color marking phase, objects:
- start as **white** at the beginning
- become **grey** when they are found to be alive
- finally become **black** when all their references have been traversed



Concurrent marking

The described algorithm must hold the **tri-color invariant** to work properly

at any moment, no black objects should reference white objects

But the marking is happening concurrently to our program goroutines.

Concurrent marking

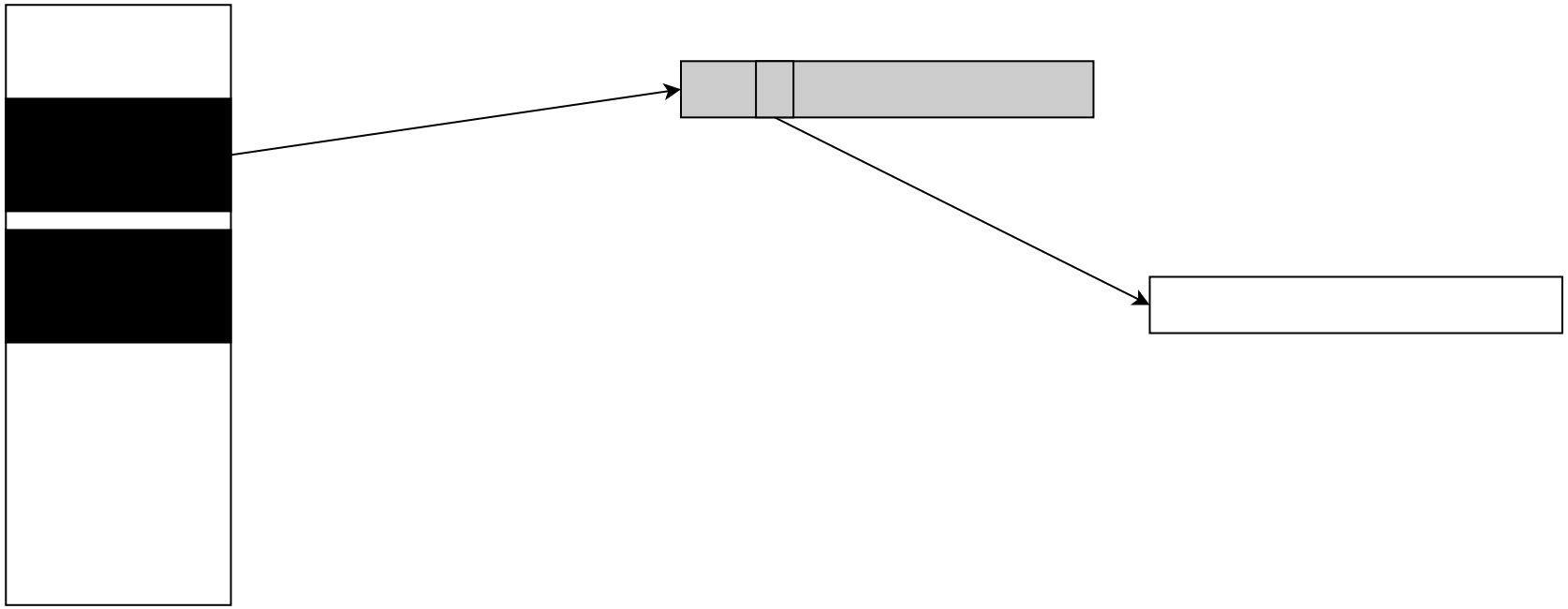
Consider the following scenario

```
type S struct {
    p *int
}

func f(s *S) *int {
    r := s.p
    s.p = nil
    return r
}
```

from Eben Freeman - Allocator Wrestling (https://www.youtube.com/watch?v=M0HER1G5BRw)

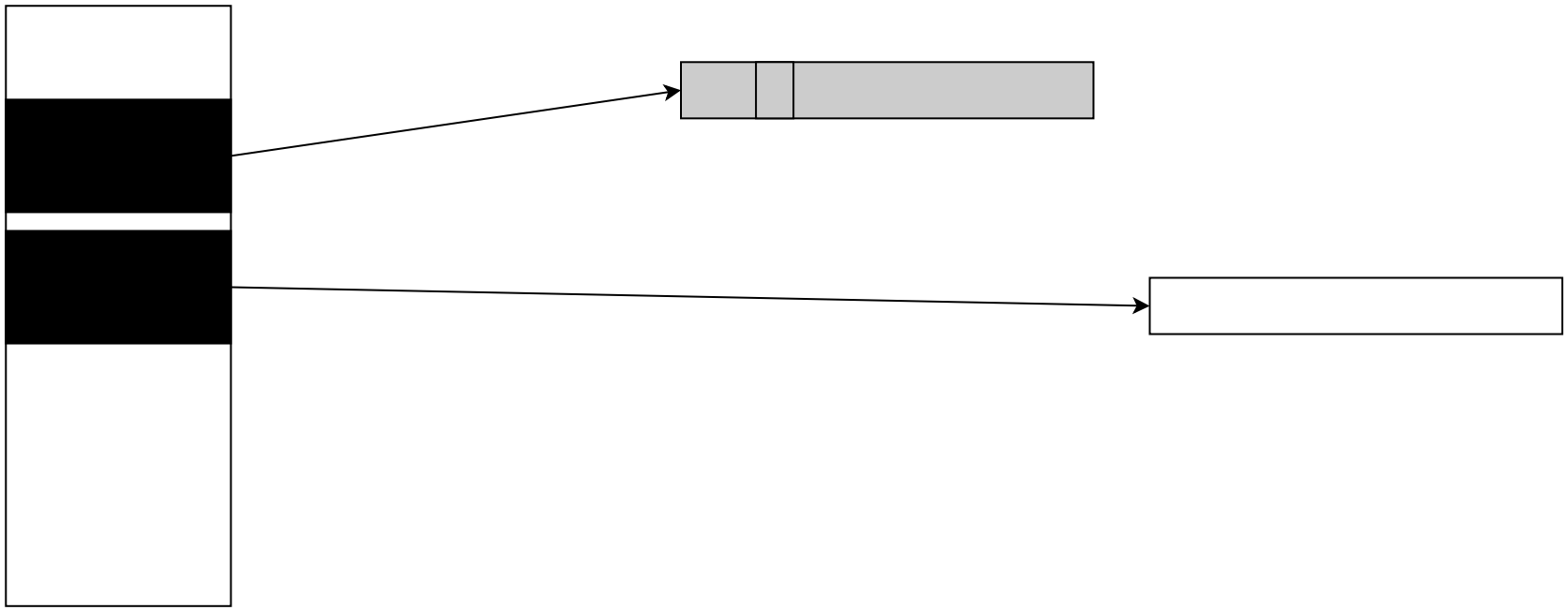
Concurrent marking

A concurrent mark phase is running, this is the current state



Concurrent marking

We execute the f function and



The tri-color invariant is violated!

Write barrier

During the mark phase, the runtime enables the **write barrier** for each pointer write

To enable and disable the write barrier, the Go runtime needs to **stop the world**, in other words, to block all goroutines.

STW phases typical latencies

Mark Setup: 10 to 30 microseconds on average
Mark Termination: 60 to 90 microseconds on average

STW phases and tight loops

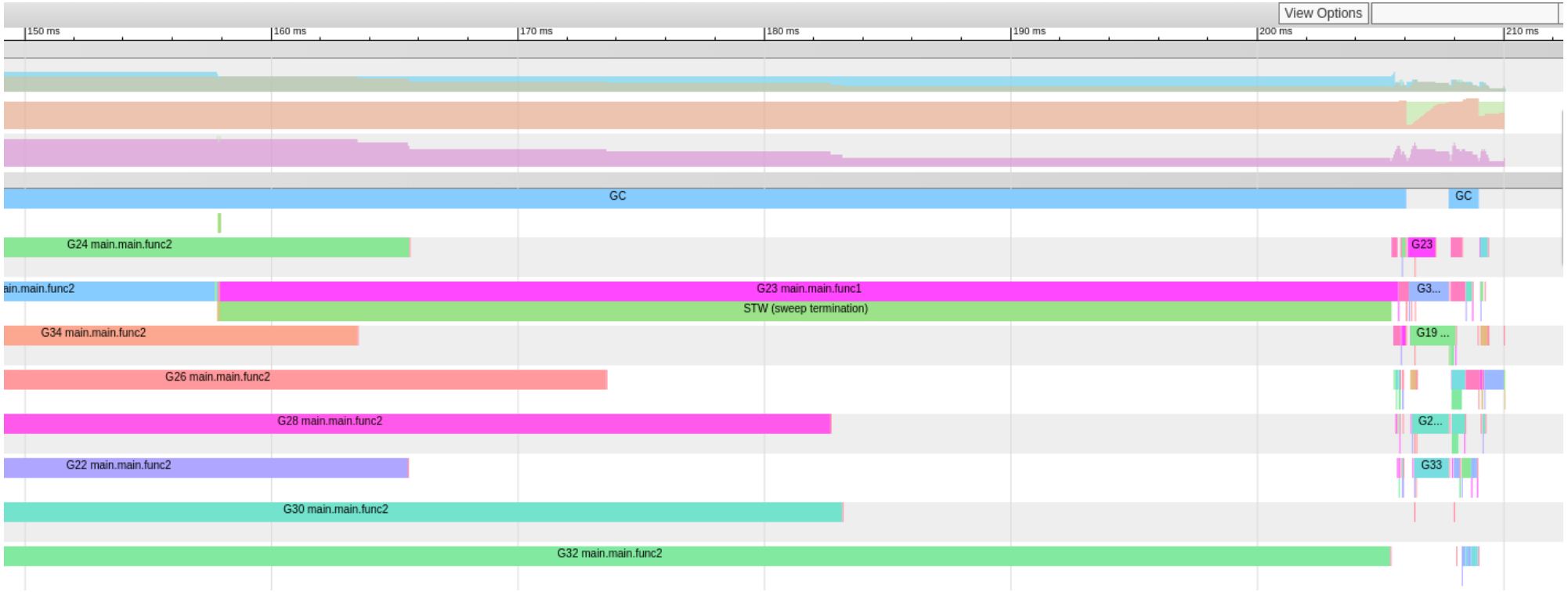
Take a look at the example in `06-memory-and-gc/gctightloopexample/main.go`

There are some goroutines that allocate heavily and trigger the GC

There are other goroutines crunching numbers in a tight loop

Quick quiz: can you guess what is going to happen?

STW phases and tight loops



The duration of the STW sweep termination phase is **huge**: > 100ms

The GC is waiting for the goroutine 32 (the bottom line marked as G32) to finish its work and so give a chance to the runtime to complete the sweep phase

Pacing algorithm

The main task of the GC is giving the impression of an infinite heap, while avoiding OOM errors

To do this it is fundamental to start the gc cycles at the right time

When does a garbage collection cycle start in Go?

At the end of every GC, the runtime configures the target heap size as a function of the current heap live size

The target value depends on the **GOGC** environment variable through the following formula

goal = reachable * (1+ GOGC/100)

By default, **GOGC** is set to 100, so for a live heap of 256 MB, the target for the next collection would be set to

256 MB * (1 + 100/100) = 512 MB

Pacing algorithm

After defining the target size, the runtime starts the next collection so that when the mark phase ends, the size of the heap in use is equal to the goal set

We can trace the garbage collection cycles of an application with

```
GODEBUG=gctrace=1 ./main

gc 406 @12.463s 2%: 0.062+1.4+0.018 ms clock, 0.50+1.1/2.6/0+0.14 ms cpu, 8->9->4 MB, 10 MB goal, 8 P

gc 407 @12.466s 2%: 0.018+1.8+0.011 ms clock, 0.14+1.1/3.1/0+0.093 ms cpu, 7->8->4 MB, 9 MB goal, 8 P

gc 408 @12.469s 2%: 0.075+4.0+0.008 ms clock, 0.60+2.4/2.8/0+0.071 ms cpu, 7->9->5 MB, 8 MB goal, 8 P
```

The format is described here (https://golang.org/pkg/runtime/#hdr-Environment_Variables) under "gctrace"

Pacing algorithm

Every line shows info about a GC cycle. Consider the last one, the following values

```
7->9->5 MB, 8 MB goal
```

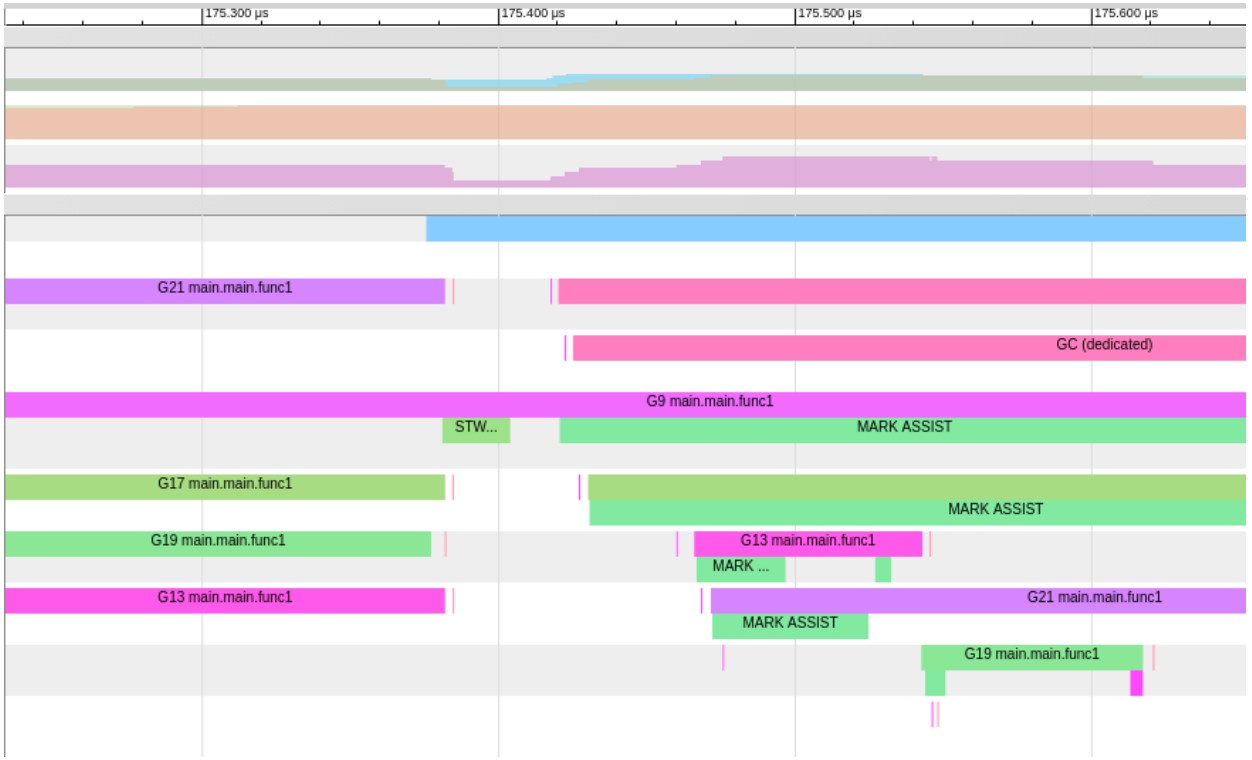
means:

- 7 MB heap memory in use before the mark phase started
- 9 MB heap memory in use after the mark phase finished
- 5 MB heap memory marked as live after the mark phase
- 8 MB next collection goal

Mark assist

What happen if the application goroutines allocate heavily during the concurrent mark phase? They can, potentially, outrun the GC, leading to OOM!

To slow down the allocation rate, the GC recruits the allocating goroutines to assist with the marking



Garbage collection performance impact

To summarize, the garbage collection inflicts latencies to our running application through these means

- 25% of the CPU capacity dedicated to GC during mark phase
- STW pauses before and after the mark phase
- write barrier enabled during mark phase
- mark assist for goroutines allocating heavily during mark phase

Sweep phases are negligible

Performance guidelines

You can tune the garbage collector modifying **GOGC** environment variable

If you increase it, you will slow down the garbage collector pace

Quick quiz: can you guess why this is not the best thing to do in most cases?

Performance guidelines

The work of the GC is proportional to the scannable heap

1) Allocate less memory
avoid memory allocations in hot path or use struct packing (http://golang-sizeof.tips/)

2) Prefer stack allocations
use escape analysis to understand where and why an allocation escapes to the heap

3) Avoid as much as possible the use of pointers
non pointer types are not scanned by the GC

4) Consider reusing memory
`Sync.Pool` has been improved in Go 1.13

A primer on escape analysis

We've seen how to inspect the compiler decisions

```
go build -gcflags="-m"
```

If you want a more verbose output

```
go build -gcflags="-m -m"
go build -gcflags="-m -m -m"
```

and so on

Escape analysis examples

Have a look at

```
escapeanalysisexample/main.go
```

Take home message

Do **not** try to guess compiler decisions. Use escape analysis and profiling to understand allocations patterns.

General guidelines

These patterns usually lead to variables escaping on the heap

- Sending pointers or values containing pointers to channels
- Using slices with size unknown at compile time
- Calling methods on an interface type

Memory and GC exercises

Follow the instructions inside

```
06-memory-and-gc/INSTRUCTIONS.md
```

Happy coding!

Word Frequency exercise solution

to found the problem, first look at a CPU profile

1) start the server
2) start profiling

```
go tool pprof -http=:9090 http://localhost:8080/debug/pprof/profile
```

3) load the server

```
hey -m GET -c 10 -n 1000 "http://localhost:8080/search?user=1"
```

N.B.: **in this order**, since the cpu profiler samples data every 10ms

You'll see **a huge amount** of time spent in WordsFreq, in particular in `strings.Fields`!

Word Frequency exercise solution

Then look at the heap profile

```
go tool pprof -http=:9090 http://localhost:8080/debug/pprof/heap
```

remember to select `alloc_space` samples type!

You'll see **a huge number** of allocations in `strings.Fields`!

Word Frequency exercise solution

We can look at the trace too!

1) add this code to the start of service.Search function

```
//start tracing program events to trace.out file
f, err := os.Create("trace.out")
if err != nil {
    panic(err)
}
defer f.Close()
if err := trace.Start(f); err != nil {
    panic(err)
}
defer trace.Stop()
```

Even tracing just a single request, we can see that:
- the heap is growing significantly
- there are goroutines in *MARK ASSIST*

Word Frequency exercise solution

2) inspect the work of the GC

```
GODEBUG=gctrace=1 ./wordcounter
```

to confirm what we've seen with the traces

Word Frequency exercise solution

To reduce allocations and GC pressure, a possible solution is to avoid `strings.Fields`

Just loop over the text to search for separators (only ' ' and '\n'), extract the words and account them in `wfreq`

To prove the solution, repeat the steps above. Specifically, check how many requests/sec we are now able to fulfill with hey

Strings allocations exercise solution

Have a look at the proposed solution: there are various way to fulfill the task

To compare them

```
go test -v . -bench=. -benchmem -count=5 > bench.txt
benchstat bench.txt
```

Preallocated slice is the winner: less allocations -> better performance!

Bonus material

Go 1.13

sync.Pool (https://github.com/golang/go/issues/22950)

mid-stack inliner (https://blog.filippo.io/efficient-go-apis-with-the-inliner/)

Optimization workflow

Benchmark. Analyze. Improve. Verify. Iterate. (https://github.com/dgryski/go-perfbook/blob/master/performance.md#concrete-optimization-tips)

Thank you

Fabio Falzoi
Senior Software Engineer @ Develer (mailto:Senior%20Software%20Engineer%20@%20Develer)
fabio.falzoi84@gmail.com (mailto:fabio.falzoi84@gmail.com)
https://github.com/Pippolo84 (https://github.com/Pippolo84)
@Pippolo84 (http://twitter.com/Pippolo84)