



An Insight Into Go Garbage Collection

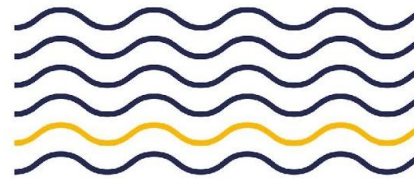
Fabio Falzoi



SAINT PETERSBURG
2019 NOVEMBER 1



Golang
Piter



\$whoami

Senior SwEng @ Develer

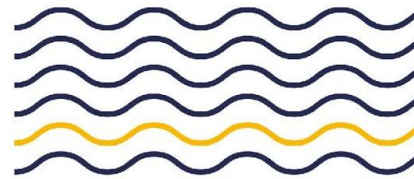
Experience with C, C++, Python and Go

Passionate about low level topics...

... and Garbage Collection!



Golang
Piter



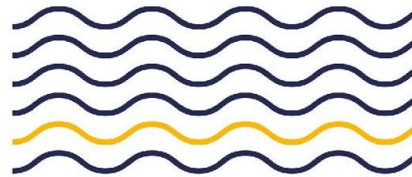
Roadmap

- Memory management in Go
- Go Garbage Collection
- Go GC Performance Impact



Memory Management in Go





Should I stack or should I heap?

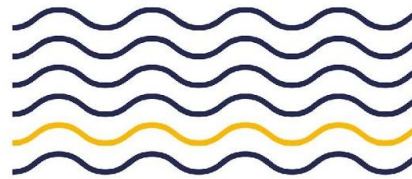
Go compiler uses *escape analysis* to decide where to allocate objects

Go prefers stack allocations, but their **size** and **lifetime** must be known at compile time

Escape Analysis rules are not part of the Go Language Specification. Do not try to guess, ask the compiler instead

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



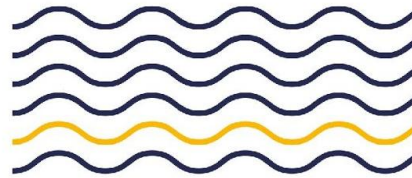


Should I stack or should I heap?

```
type s struct {  
    v int  
}  
  
func newStruct() *s {  
    return &s{10}  
}
```

Will it be allocated on the stack or on the heap?





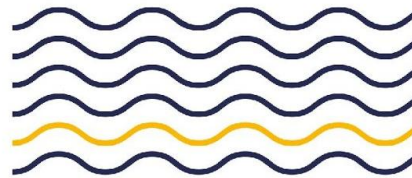
Escape Analysis in action

```
type s struct {  
    v int  
}  
  
func newStruct() *s {  
    return &s{10}  
}  
  
func f() {  
    x := newStruct()  
    _ = x  
}
```

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

```
./main.go:7:6: can inline newStruct as: func() *s { return &s literal }  
./main.go:12:16: inlining call to newStruct func() *s { return &s literal }  
./main.go:8:12: &s literal escapes to heap  
./main.go:12:16: f &s literal does not escape
```





Escape Analysis in action

```
var g *s

type s struct {
    v int
}

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

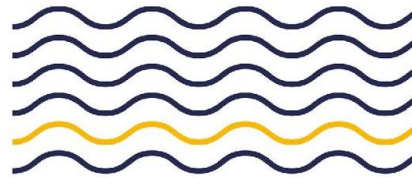
func f() {
    x := newStruct()
    g = x
}
```

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

```
./main.go:9:6: can inline newStruct as: func() *s { return &s literal }
./main.go:14:16: inlining call to newStruct func() *s { return &s literal }
./main.go:10:12: &s literal escapes to heap
./main.go:14:16: &s literal escapes to heap
```




Golang
Piter



Goroutine User Stack

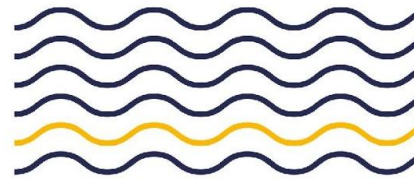
Stack is managed in **frames**: individual memory space for each function call

Creating a new frame and invalidate one is just a matter of bumping up or down the *Stack Pointer* register



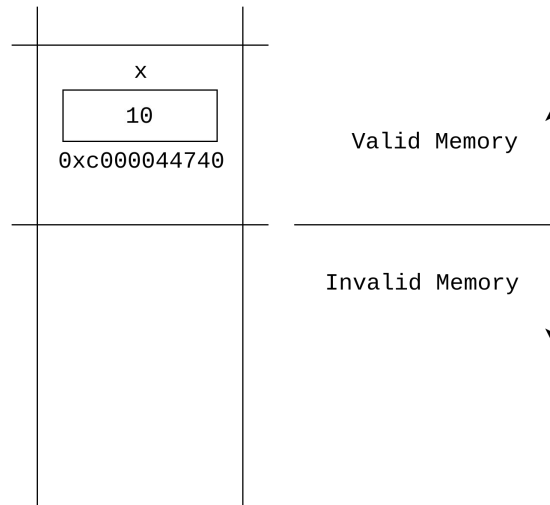


Golang
Piter



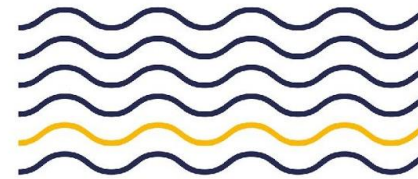
Stack Frames

```
func main() {  
    x := 10  
    f()  
  
    println(&x)  
}  
  
//go:noinline  
func f() {  
    y := 20  
    println(&y)  
}
```



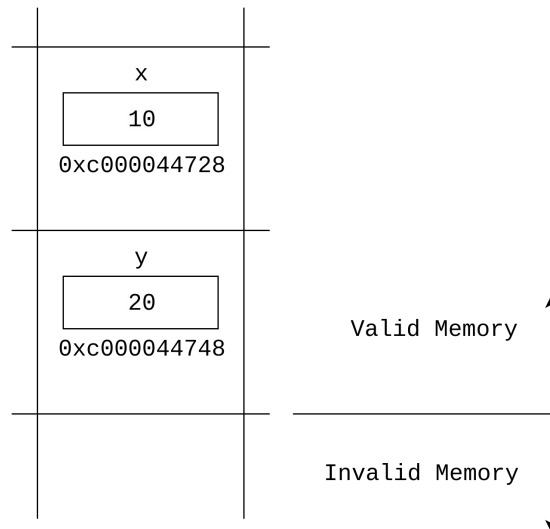


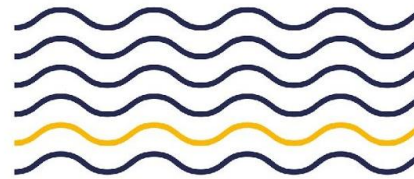
Golang
Piter



Stack Frames

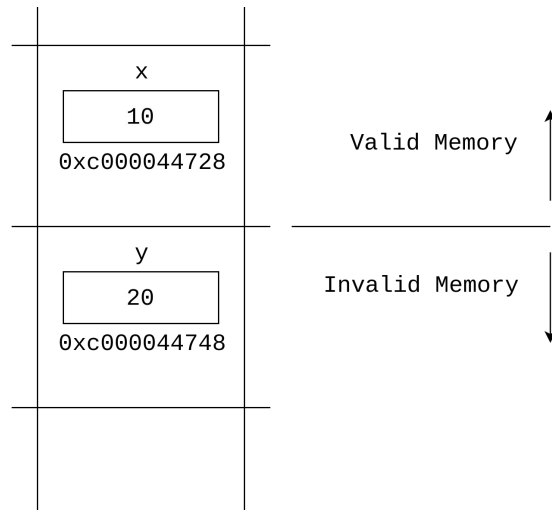
```
func main() {  
    x := 10  
    f()  
  
    println(&x)  
}  
  
//go:noinline  
func f() {  
    y := 20  
    println(&y)  
}
```





Stack Frames

```
func main() {  
    x := 10  
    f()  
  
    println(&x)  
}  
  
//go:noinline  
func f() {  
    y := 20  
    println(&y)  
}
```

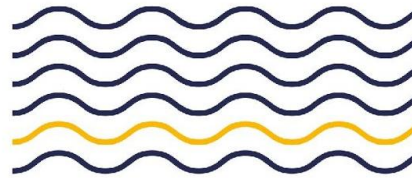


frame for the `f` func is not deleted: the runtime simply updates the *Stack Pointer* register value





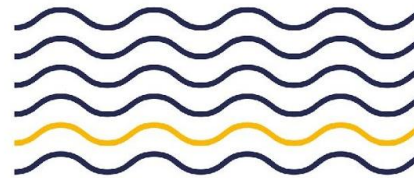
Golang
Piter



Goroutine User Stack

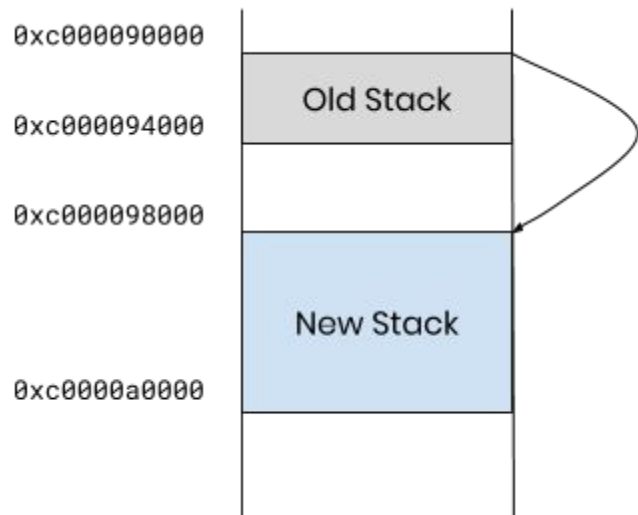
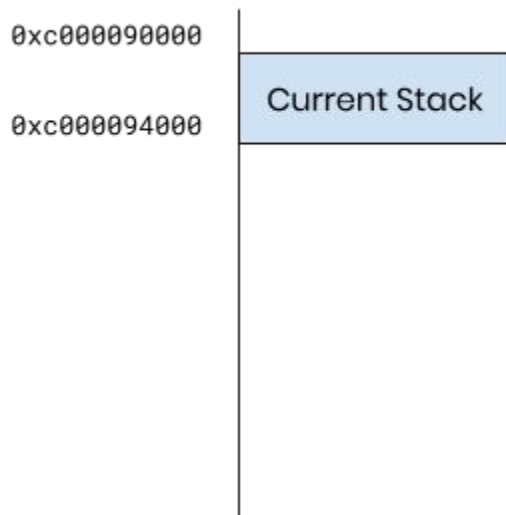
Goroutine user stacks are **dynamically resized** and can grow up to 1GB on amd64





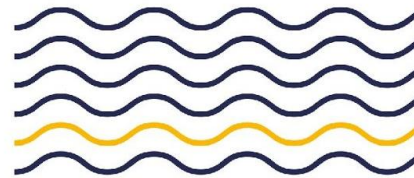
Stack Split

Stack Grow



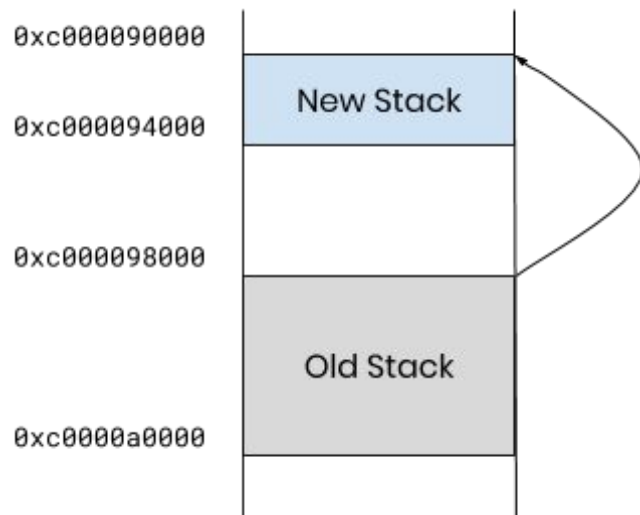
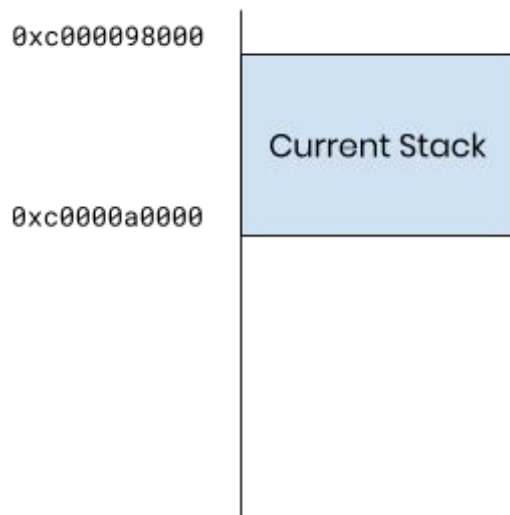


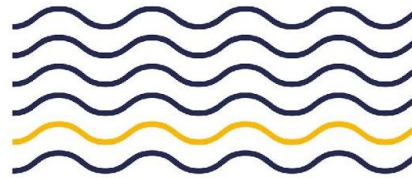
Golang
Piter



Stack Split

Stack Shrink





Stack Allocation Paths

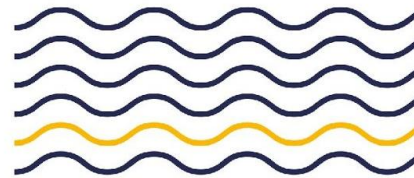
Where does the memory for a growable goroutine stack come from?

Goroutine user stacks are backed by the Go **heap**!

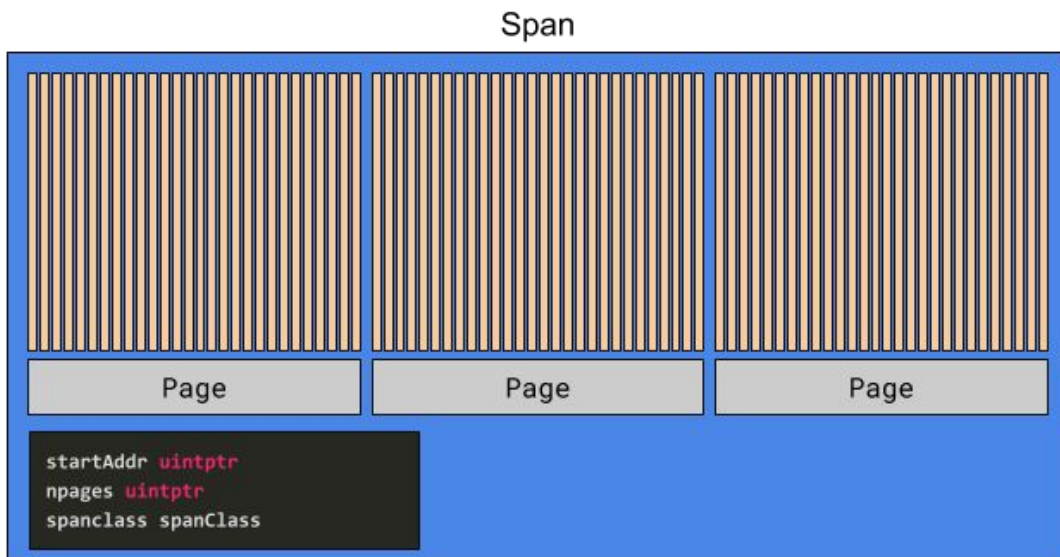
For stack < 32 KB we have a **per M cache**, so to avoid locking (and contention) in the common case

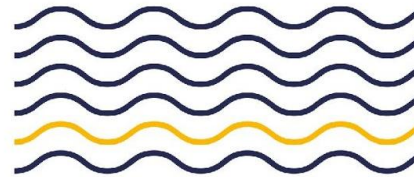
For stack ≥ 32 KB, or when the above cache is empty, we allocate memory from a global pool



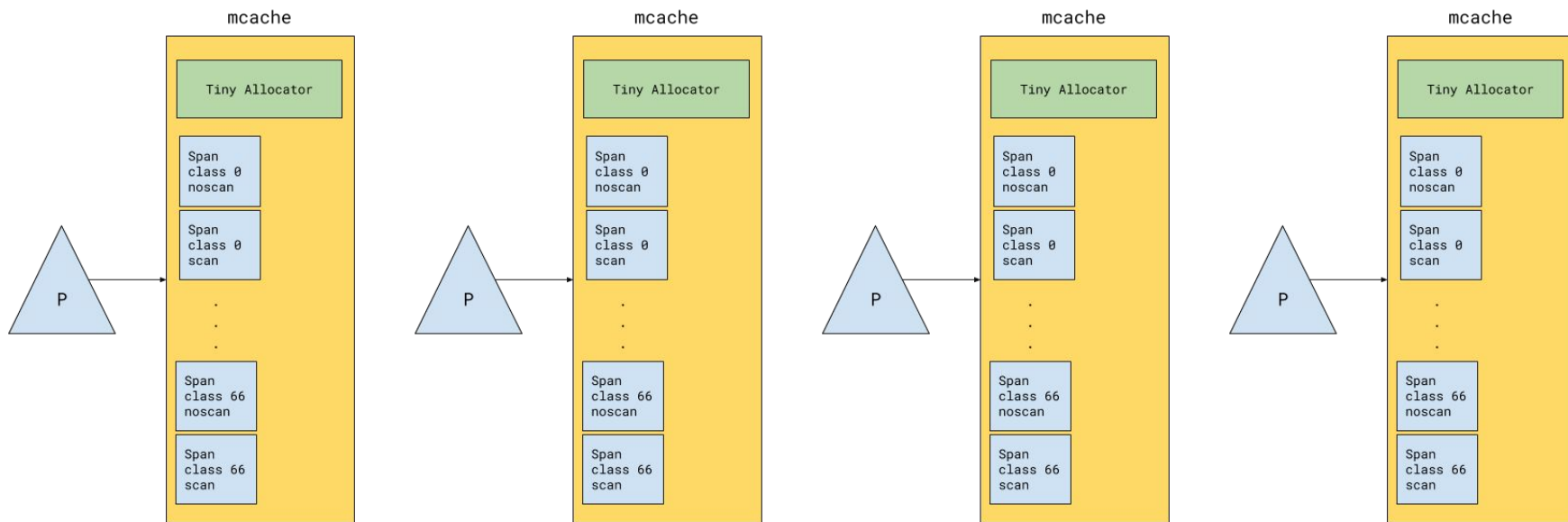


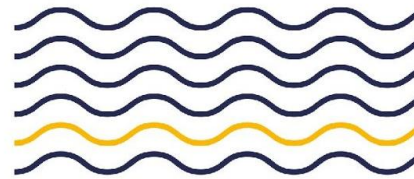
Heap Allocations



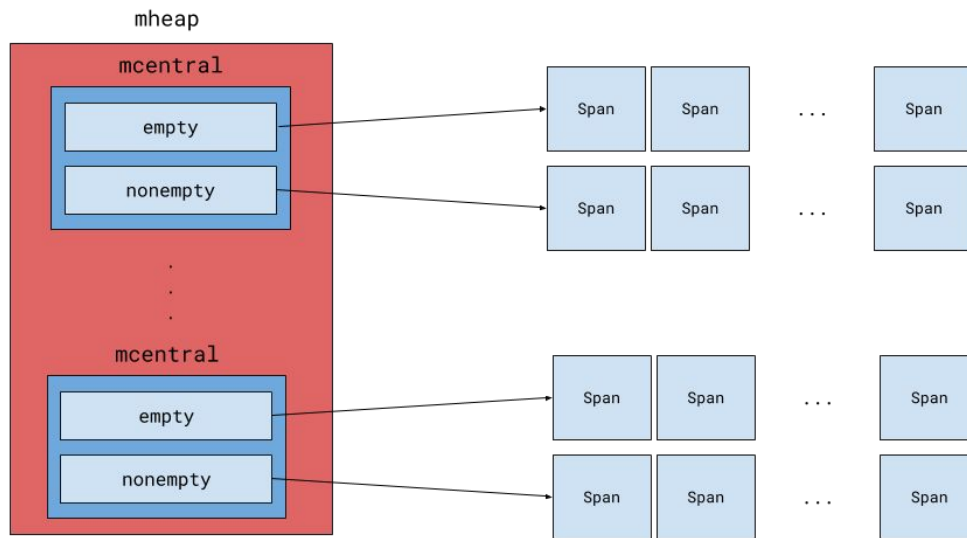


Heap Allocations





Heap Allocations





Stack vs Heap – which one is cheaper?

Stack and Heap allocation patterns are similar

So, why stack allocations should be cheaper?

The main difference does not lie in the allocations, but in the **deallocations**

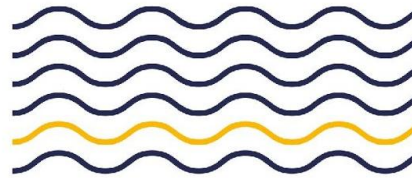
Destroying a stack frame means bumping up the Stack Pointer register

Instead, heap allocated objects are reclaimed through **Garbage Collection!**





Golang
Piter



Is Garbage Collection evil?



Roberto Clapis 🇧🇪 🇪🇺 🇮🇹 @
@empijej

"It's garbage collected [...] this harms performance" → just a little but grants that unreferenced memory is freed, which Rust doesn't grant. So a pure safe Rust executable can run for a while and then die for an OOM. Good luck debugging that.

Memory leaks are **hard** to debug

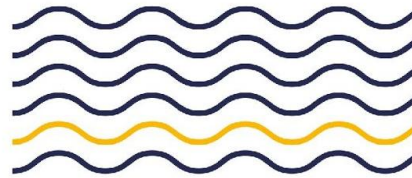
Go Garbage Collection is optimized for very low latency





Go Garbage Collection





Mark & Sweep

Mark phase & **Sweep** phases

Mark phase Start from roots (global variables and goroutine stacks) and mark each reachable object as alive

Sweep phase Check each allocated object, freeing it if it is not marked

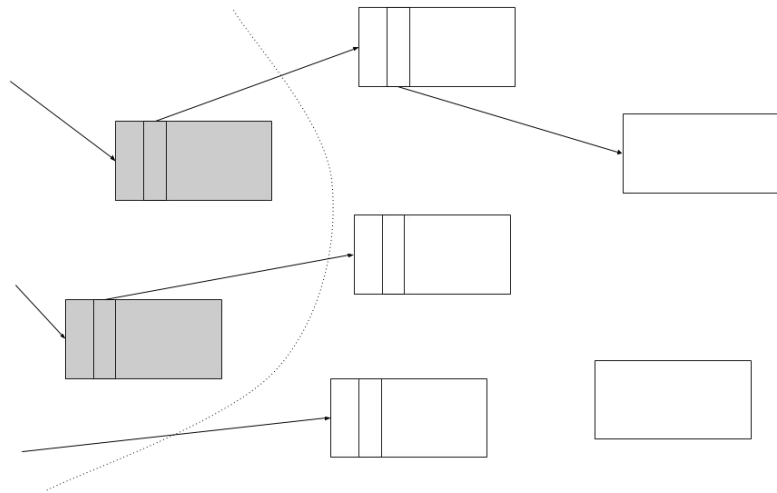




Golang
Piter

Tricolor Mark & Sweep

- **White set** objects not marked
- **Grey set** objects marked, but we have not yet scanned all their referents
- **Black set** objects marked along with all their referents

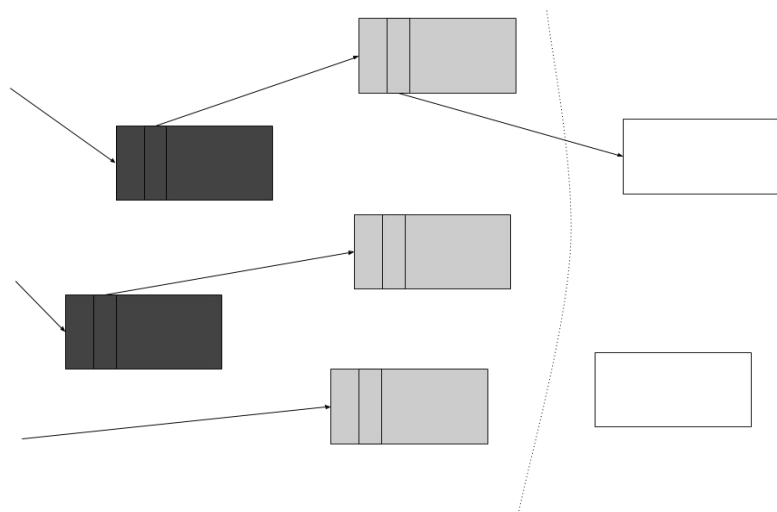




Golang
Piter

Tricolor Mark & Sweep

- **White set** objects not marked
- **Grey set** objects marked, but we have not yet scanned all their referents
- **Black set** objects marked along with all their referents

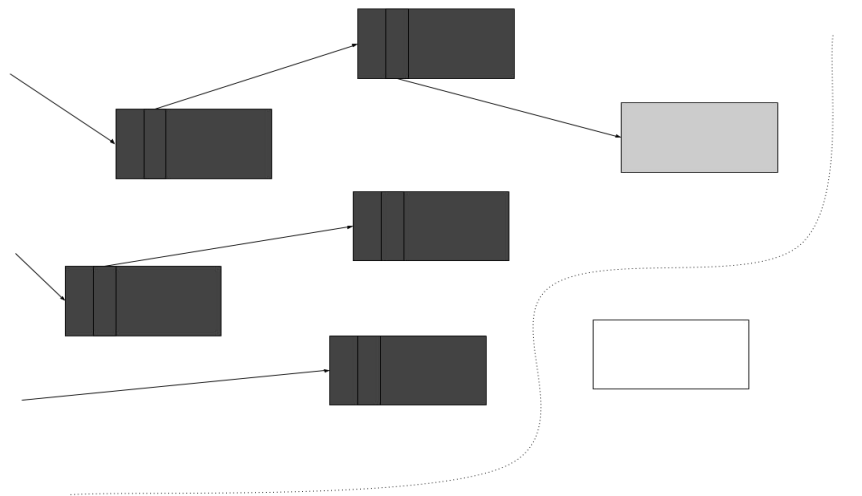




Golang
Piter

Tricolor Mark & Sweep

- **White set** objects not marked
- **Grey set** objects marked, but we have not yet scanned all their referents
- **Black set** objects marked along with all their referents

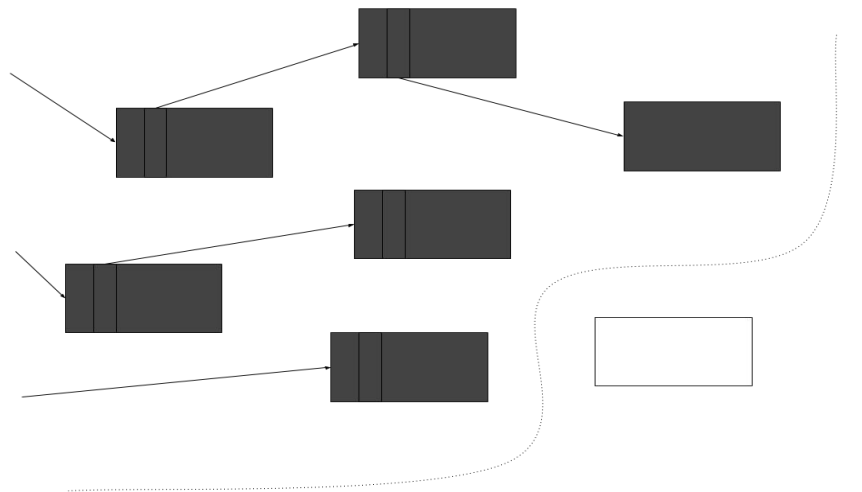




Golang
Piter

Tricolor Mark & Sweep

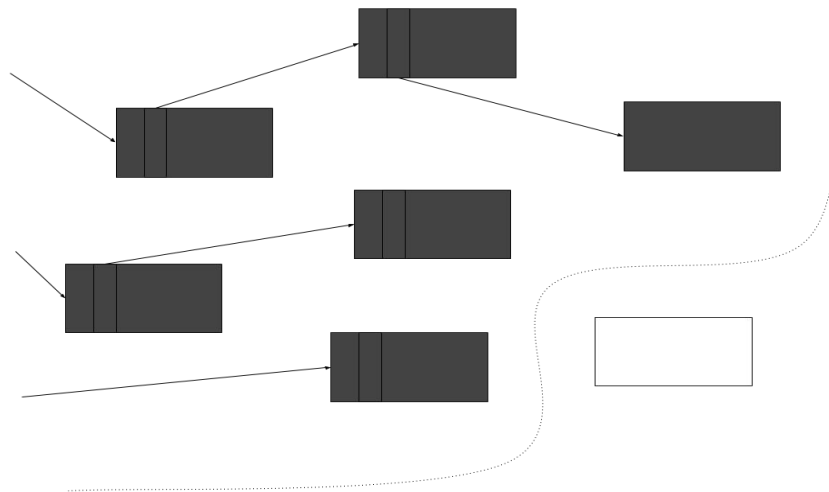
- **White set** objects not marked
- **Grey set** objects marked, but we have not yet scanned all their referents
- **Black set** objects marked along with all their referents





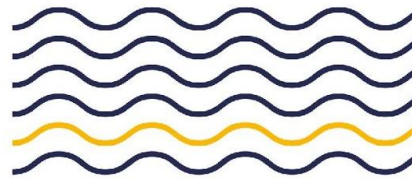
Tricolor Mark & Sweep

- **White set** objects not marked
- **Grey set** objects marked, but we have not yet scanned all their referents
- **Black set** objects marked along with all their referents



Strong Tricolor Invariant





Tricolor Mark & Sweep



Go 1 used a STW Mark & Sweep Garbage Collector

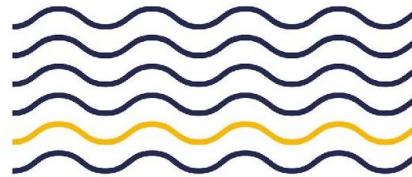
Pros

- easy to implement
- easy to control the heap growth

Cons

- external fragmentation
- **STW latency** proportional to the heap size





Concurrent Tricolor Mark & Sweep

How can we reduce the latencies of Garbage Collection?

Since Go 1.5, the runtime executes GC **concurrently** to the mutators code, trading throughput for latency

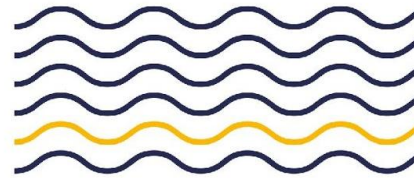
Since garbage is not reachable by user code, the sweep phase can be done concurrently

What about the **marking phase**?





Golang
Piter



Concurrent Marking

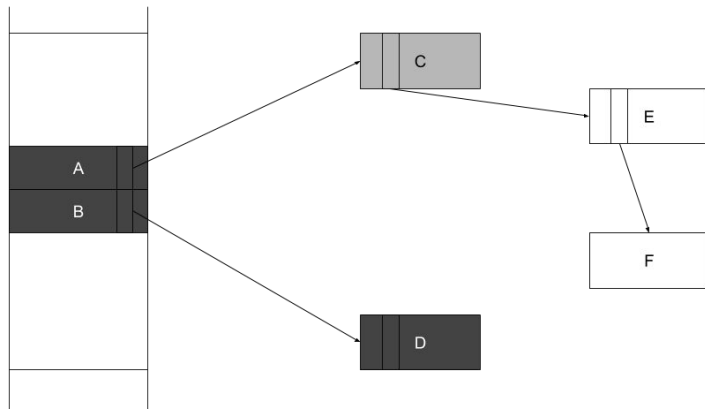
```
// Mutator code
```

```
type Obj struct {  
    // ...  
    next *Obj  
    // ...  
}
```

```
D.next = E.next  
C.next = nil
```

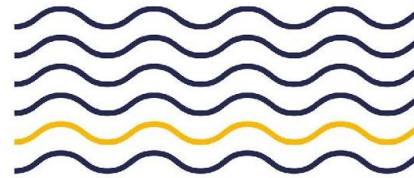
Goroutine Stack

Heap





Golang
Piter



Concurrent Marking

```
// Mutator code
```

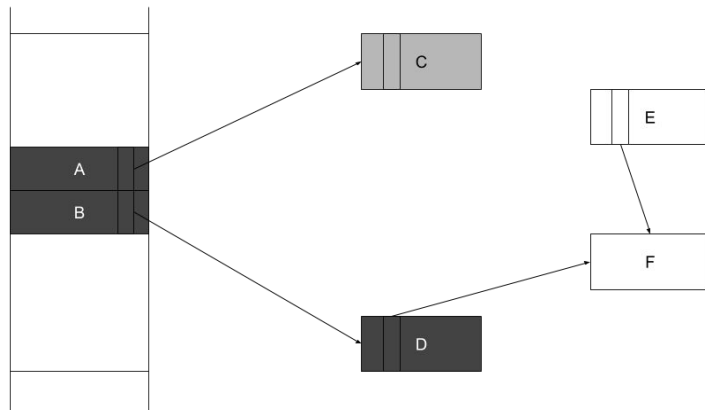
```
type Obj struct {  
    // ...  
    next *Obj  
    // ...  
}
```

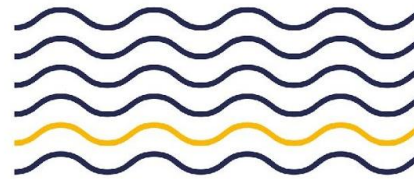
```
D.next = E.next
```

```
C.next = nil
```

Goroutine Stack

Heap





Concurrent Marking

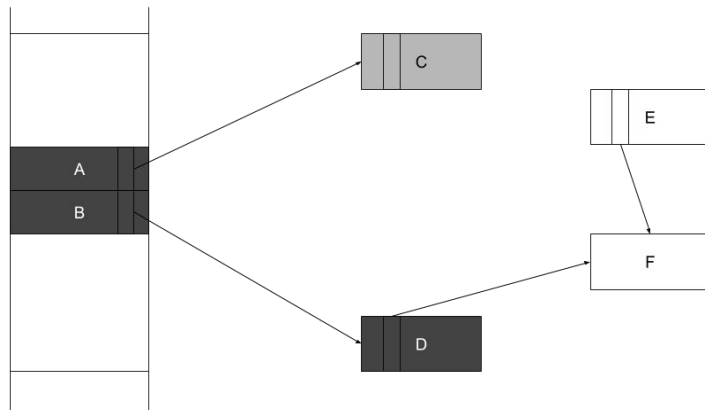
```
// Mutator code
```

```
type Obj struct {  
    // ...  
    next *Obj  
    // ...  
}
```

```
D.next = E.next  
C.next = nil
```

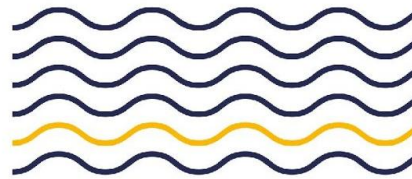
Goroutine Stack

Heap



The tricolor invariant **does not** hold true anymore!





Is Garbage Collection evil?

How can we preserve GC **correctness** while doing it concurrently?

We need a way for the **mutator** to *inform* the **collector** that it is changing the heap memory graph

Instead of normal pointer operations, the compiler can emit **write** or **read** barriers

```
*slot = ptr
```

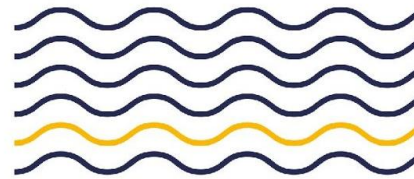


```
func barrier(slot, ptr) {  
    // ...  
}
```





Golang
Piter



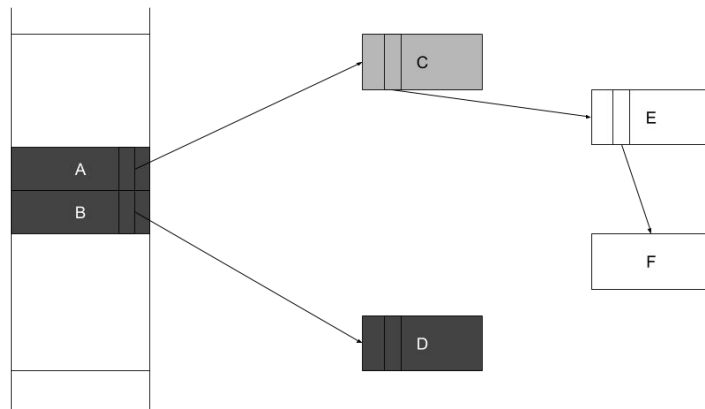
Dijkstra Write Barrier

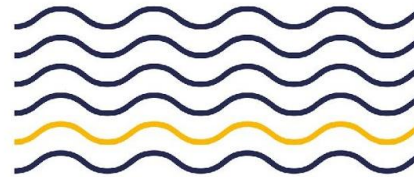
```
// Mutator code
D.next = E.next
C.next = nil

// Write Barrier
func writePointer(slot, ptr) {
    shade(ptr)
    *slot = ptr
}
```

Goroutine Stack

Heap





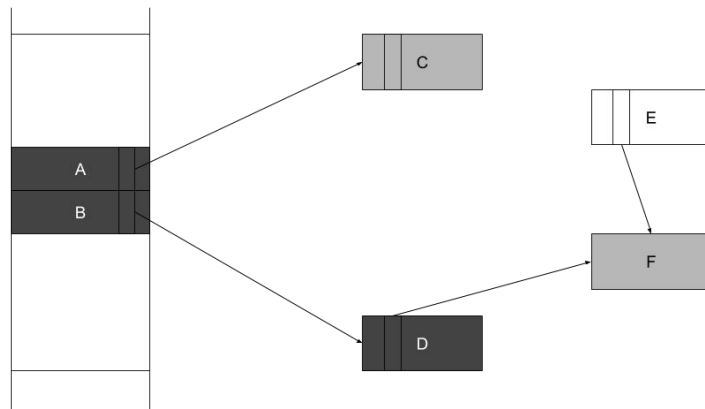
Dijkstra Write Barrier

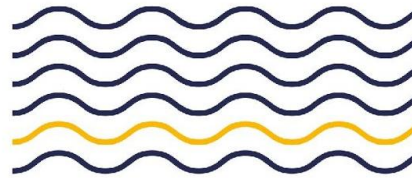
```
// Mutator code
D.next = E.next
C.next = nil

// Write Barrier
func writePointer(slot, ptr) {
    shade(ptr)
    *slot = ptr
}
```

Goroutine Stack

Heap





Dijkstra Write Barrier

Pros

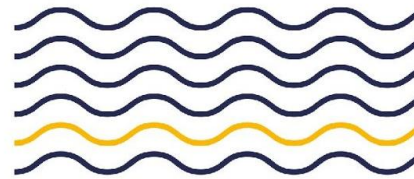
- ensures the **strong tricolor invariant**
- ensures **forward progress**

Cons

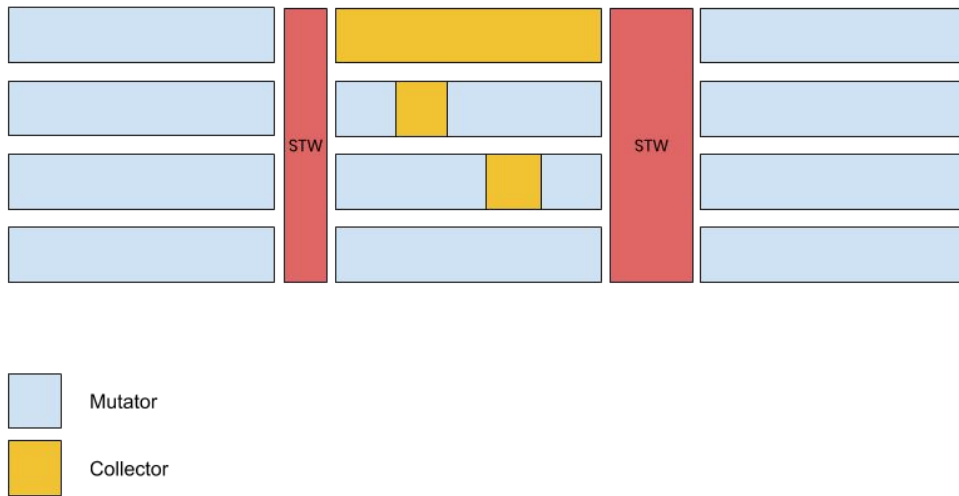
- ***permagrey*** stacks

Permagrey stacks forces us to rescan all the goroutine stacks that have been modified during the marking phase!



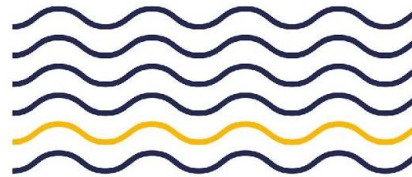


Go 1.7 Concurrent GC



Stack rescanning happens with the world stopped at the end of marking: it is a source of potentially **unbounded latency**!





Go Hybrid Write Barrier

Go 1.8 introduced a **Hybrid Write Barrier**: a combination of Dijkstra - style and Yuasa - style write barriers

Dijkstra - style barrier requires STW stack rescanning at the **end** of marking

Yuasa - style barrier requires STW stack scanning at the **begin** of marking

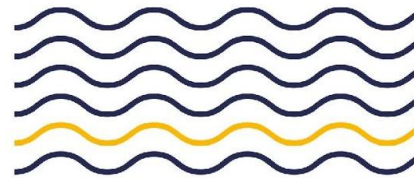
The Hybrid Write Barrier allows **concurrent stack scanning** without rescan!

```
func writePointer(slot, ptr) {  
    shade(*slot)  
    if current_stack_is_grey {  
        shade(ptr)  
    }  
    *slot = ptr  
}
```





Golang
Piter



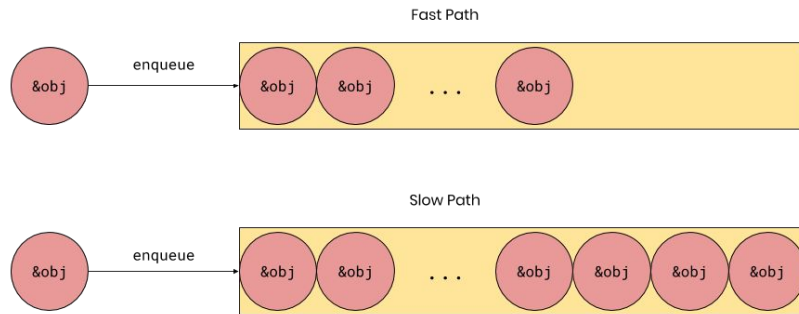
Buffered Write Barrier

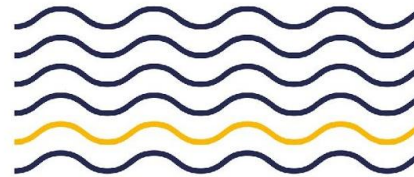
In Go 1.10 the implementation of the Hybrid Write Barrier has been optimized implementing a Buffered Write Barrier

Instead of immediately shading the pointers, these are saved inside a **per P buffer**

When it is full, the hybrid write barrier jumps to the **slow path**, where it flushes its buffer and greys all the pointers as usual!

```
type wbBuf struct {  
    next uintptr  
    end  uintptr  
    buf [wbBufEntryPointers * wbBufEntries]uintptr  
    // ...  
}
```



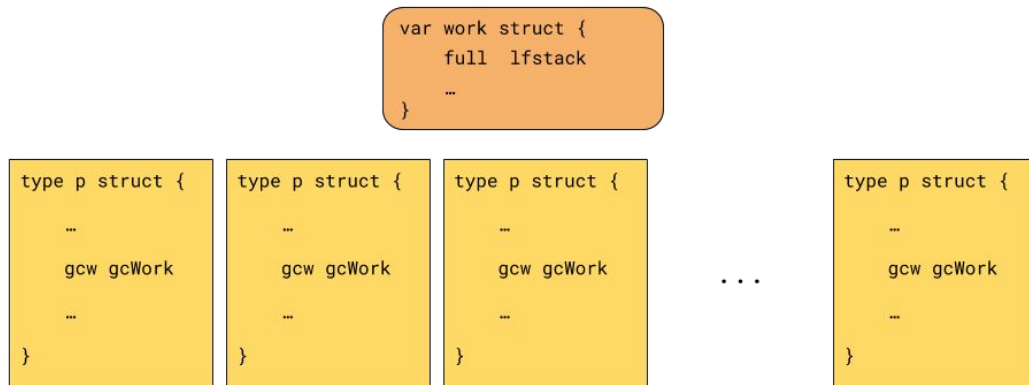


Marking – Grey and Black Objects

Go dedicates **25% of GOMAXPROCS CPUs** to background marking

To reduce contention Go uses a **distributed work pool** to hold objects to scan

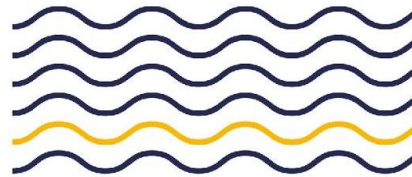
- a global GC work queue
- per P local GC work queues



A **grey** object is one that is marked and on a work queue

A **black** object is one that is marked and not on a work queue





Heavy Allocating Goroutines


What happen if a goroutine allocates too **heavily**?

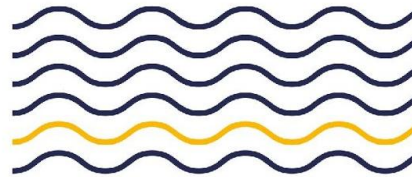
To avoid outrunning the heap size goal, the GC enable **Mark Assist**

```
func gcStart(trigger gcTrigger) {  
    // ...  
    atomic.Store(&gcBlackenEnabled, 1)  
    // ...  
}
```

Mark Assist works as a budget system where each allocation is charged based on the size. What happen when a goroutine exhausts its budget?

First, it tries to steal allocation credits from the background marking goroutines. If there isn't enough, the goroutine is **forced to assist** in marking, slowing down its the allocation rate





How the Mark Phase ends?

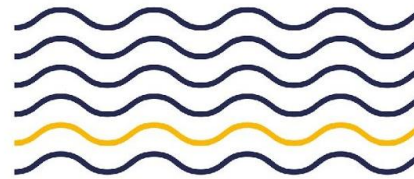
Mark Termination Algorithm rewritten in Go 1.12

Since Go uses a distributed work queue, a distributed **mark completion** algorithm is needed

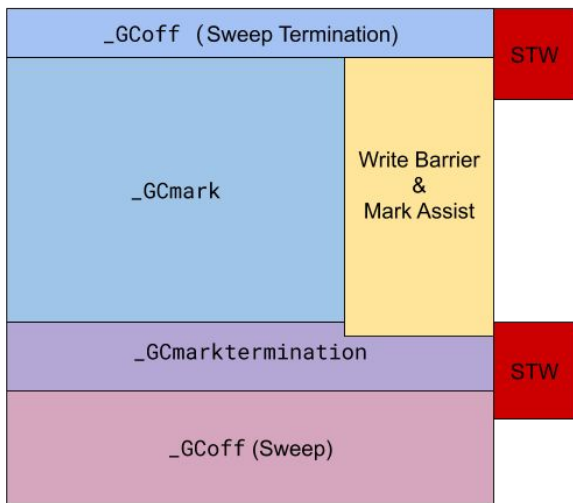
When a P reaches a background mark completion point

- 1) Acquire work.markDoneSema semaphore to make sure no other Ps is running the algorithm
- 2) Check if there is global work to do, if so, abort the algorithm
- 3) On each P
 - a) Flush local write barrier buffer
 - b) Flush local GC work queue
- 4) Check gcMarkDoneFlushed flag to see if at least one P has flushed some work. If so, abort the algorithm, otherwise enter Mark Termination Phase

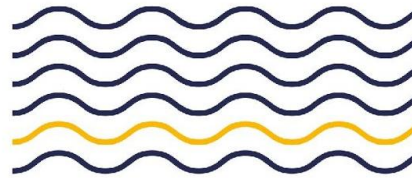




Go Garbage Collector Phases



STW pauses are used to enable/disable the Write Barrier and are not proportional to the heap size anymore!



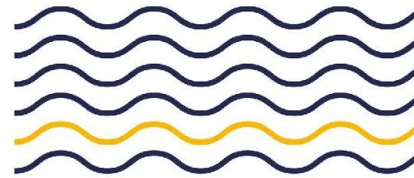
When a collection cycle should start?

With **GOGC** environment variable the user sets a **heap goal**

$$HeapGoal = HeapLive \cdot (1 + \frac{GOGC}{100})$$

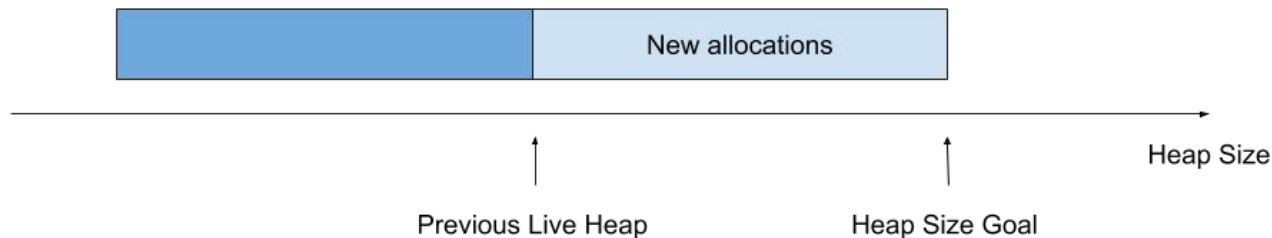
GOGC default value is 100

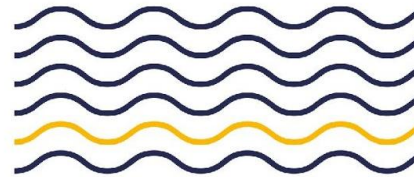




STW Mark and Sweep Trigger

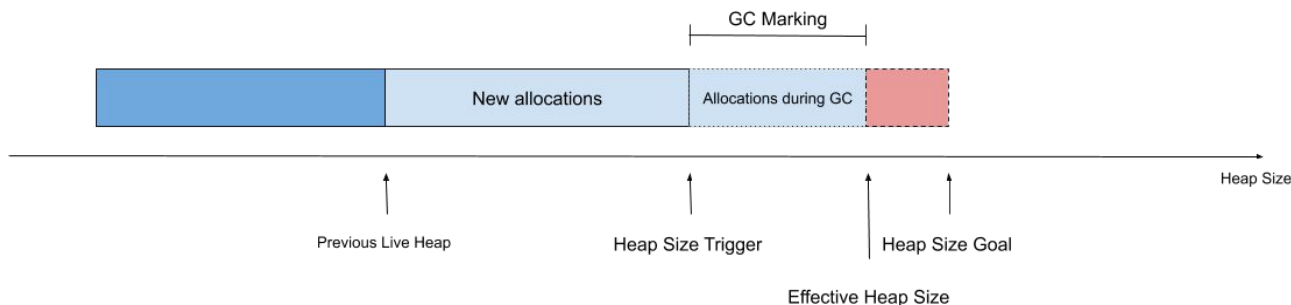
When the Heap Size is equal to the Heap Goal, we stop the world and run a collection!





Concurrent Mark and Sweep Trigger

Since we are marking concurrently, the **Heap Trigger** must be set **before** the Heap Goal

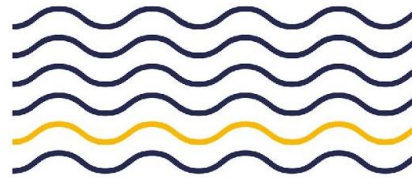


The **GC Pacer** algorithm decides the trigger trying to

- minimize distance between **Heap Size Goal** and **Effective Heap Size**
- minimize distance between **CPU Utilization Goal** and **Effective CPU Utilization**

The GC pacer **estimates** the marking work based on the last GC marking cycle





Sweep Phase

Each mspan holds two metadata fields

- allocBits pointer to a bitmap of allocated objects in span
- gcmarkBits pointer to a bitmap of marked objects in span

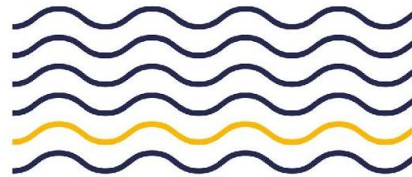
Sweep a span simply means assigning gcmarkBits to allocbits and allocate a zeroed gcmarkBits ready for the next marking phase

```
s.allocBits = s.gcmarkBits  
s.gcmarkBits = newMarkBits(s.nelems)
```

Sweep a span is very **fast** but...

... since sweeping modifies the span metadata it **must be completed** before the next marking phase!





Proportional Sweeping

To avoid delays in the enabling of mutator assists, Go uses

- lazy sweeping while allocating
- background concurrent sweeping

Sweeping rate is based on a budget system just like the proportional marking

The sweeping rate is decided by the GC Pacer, taking into account

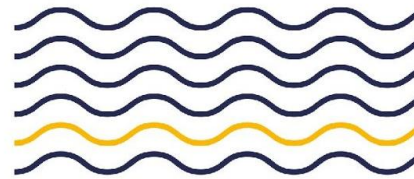
- number of sweepable pages
- distance between heap live at the end of the last marking and the heap trigger



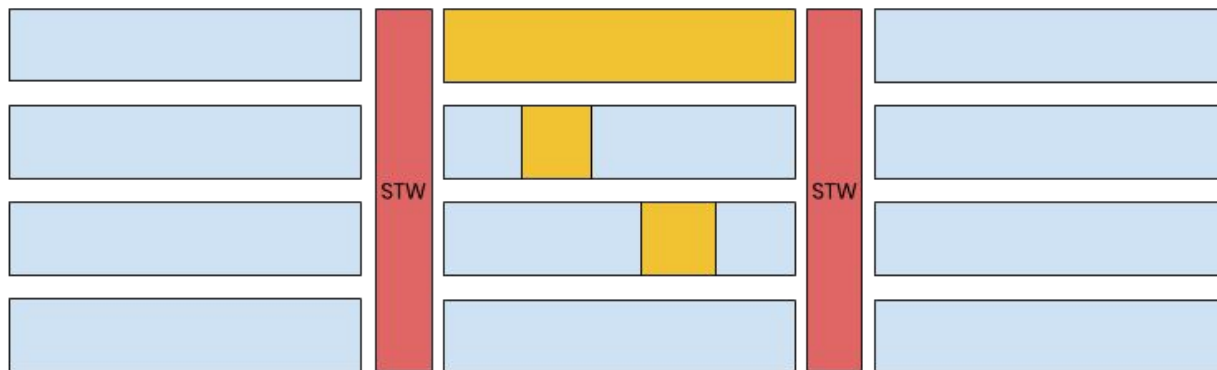


Go GC Performance Impact





GC Impact Summary



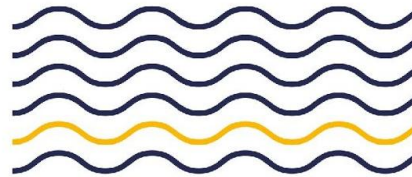
- STW pauses at the beginning and at the end of each cycle
- 25% CPUs dedicated to Background Marking
- Mark Assist
- Write Barrier on during each cycle
- Background and lazy sweeping





Golang
Piter

Go GC SLOs



Go GC Service Level Objectives for 2018 from Rick Hudson's ISMM
Keynote “**Getting to Go**”

```
$ ./garbage
pkg: golang.org/x/benchmarks
goos: linux
goarch: amd64
```

```
BenchmarkGarbage/benchmem-MB=64-8      5000      ... 75430 STW-ns/GC ...
```

2018

25% of the CPU *during* GC cycle

Heap 2X live heap or max heap

Two <500 μ s STW pauses per GC

Goroutines allocation \propto GC assists

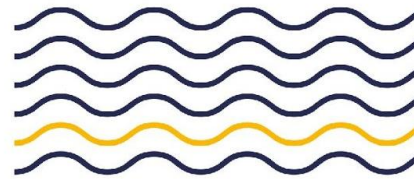
Minimal GC assists in steady state

Typical STW pauses ~ **tenths of microseconds**



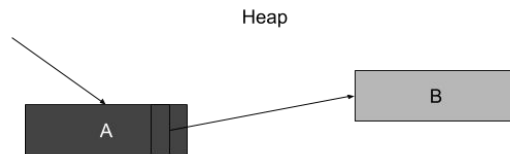


Golang
Piter



Throughput and Floating garbage

The collector marks A and B objects as shown



The mutator deletes the pointer to B

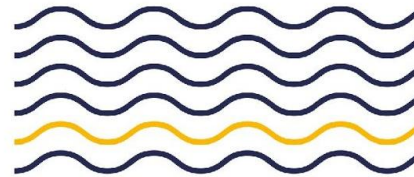


At the end of marking, B is black



The GC retains objects that are reachable **at some point** during marking, even if **they are not** at the end of the cycle, due to the mutator executing concurrently

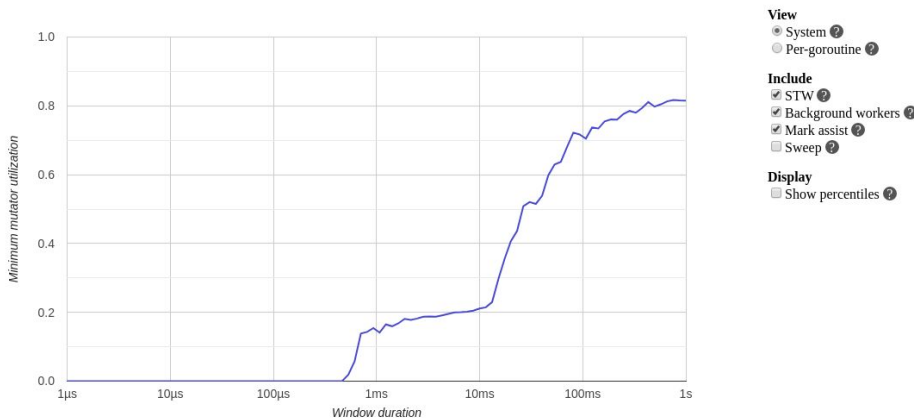




Am I experiencing GC pressure?

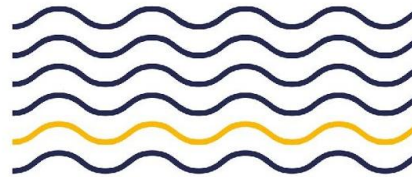
Minimum Mutator Utilization curve

- x axis \Rightarrow time
- y axis \Rightarrow fraction of CPU time spent in the mutator (CPU utilisation)
- y-intercept \Rightarrow mutators' overall share of processor time
- x-intercept \Rightarrow maximum pause time





Golang
Piter



Am I experiencing GC pressure?

GC Trace

```
$ GODEBUG=gctrace=1 ./garbage
```

```
gc 1 @0.006s 0%: 0.015+0.21+0.020 ms clock, 0.12+0/0.14/0.27+0.16 ms cpu, 0->0->0 MB, 4 MB goal, 8 P (forced)
gc 2 @0.007s 1%: 0.012+0.16+0.015 ms clock, 0.097+0/0.17/0.25+0.12 ms cpu, 0->0->0 MB, 4 MB goal, 8 P (forced)
gc 3 @0.028s 1%: 0.014+2.0+0.023 ms clock, 0.11+0.087/2.6/5.5+0.18 ms cpu, 4->4->2 MB, 5 MB goal, 8 P
...
```

Format described [here](#)





Golang
Piter

In a nutshell

Less allocations on the heap



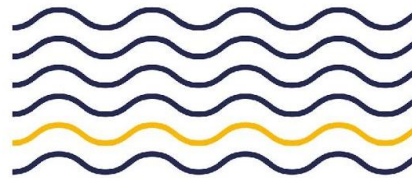
Less marking work

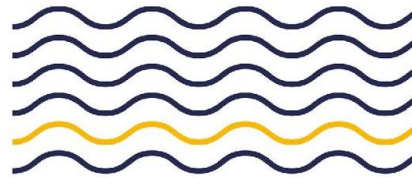


Shorter GC cycle



Write Barrier on for less time, less assist and less floating garbage





Value vs Pointers

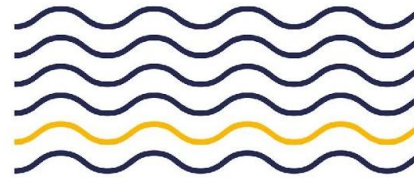
Scanning time is roughly linear in the number of pointers scanned

- Use escape analysis to ask the compiler where it is allocating and why
- Prefer copying values instead of passing a pointer
- Consider refactoring to avoid pointers in your types

Example: did you know about the pointer in the Time type?

```
type Time struct {  
    wall uint64  
    ext  int64  
    loc  *Location  
}
```





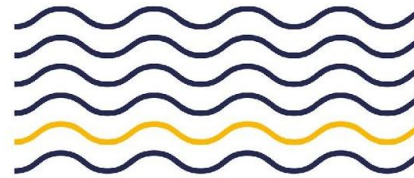
Struct Layout

Consider **struct packing**: check out [Golang Sizeof Tips](#) for a visual explanation

```
// Type size: 80 bytes
type S struct {
    a string
    b bool
    c string
    d *string
    e byte
    f []byte
}
```

Fields	Alignment
a string	
b bool	
padding	
c string	
d pointer	
e byte	
padding	
f slice	





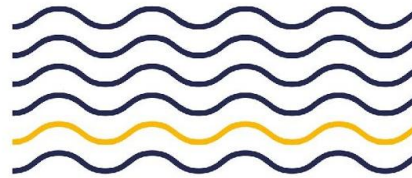
Struct Layout

Consider **struct packing**: check out [Golang Sizeof Tips](#) for a visual explanation

```
// Type size: 72 bytes
type S struct {
    e byte
    b bool
    a string
    c string
    d *string
    f []byte
}
```

Fields	Alignment
e byte	
b bool	
padding	
a string	
c string	
d pointer	
f slice	





Reuse Memory

```
var pool = sync.Pool{
    New: func() interface{} {
        return make([]byte, 1024)
    },
}

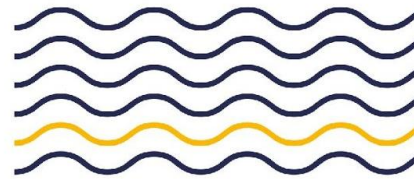
func f() {
    buf := pool.Get().([]byte) // reuses from pool or calls New
    // do work
    pool.Put(buf) // returns it to the pool
}
```

sync.Pool has been updated in Go 1.13 introducing a **victim cache**





Golang
Piter



Garbage Collector Tuning

$$HeapGoal = HeapLive \cdot (1 + \frac{GOGC}{100})$$

GOGC = 100

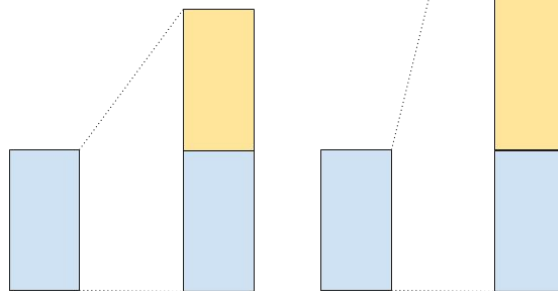
GOGC = 300

GOGC knob: trading memory for CPU utilization

You can change it at runtime too:

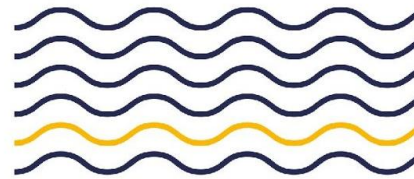
`runtime/debug.SetGCPercent`

The heap growth becomes **harder** to control



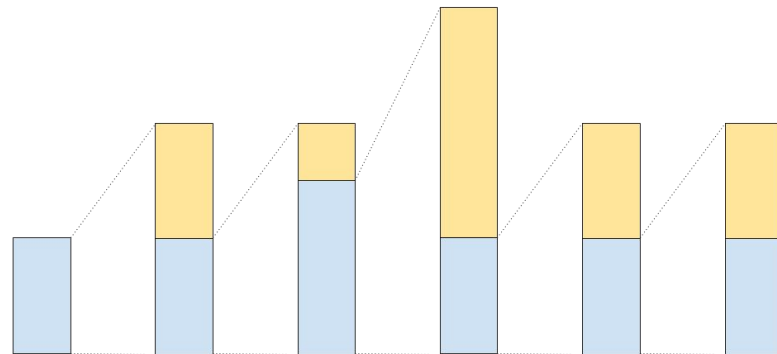
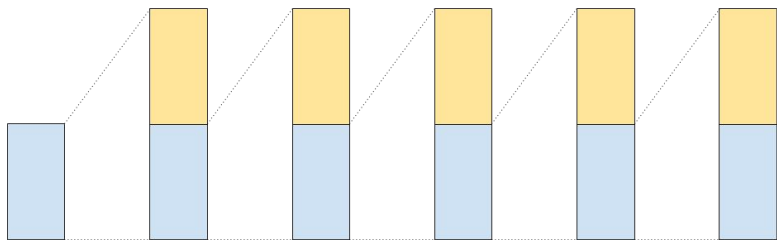


Golang
Piter



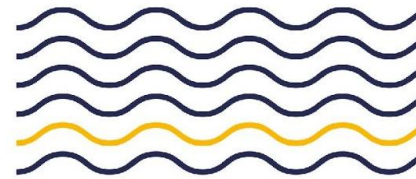
Increasing GOGC

GOGC = 100



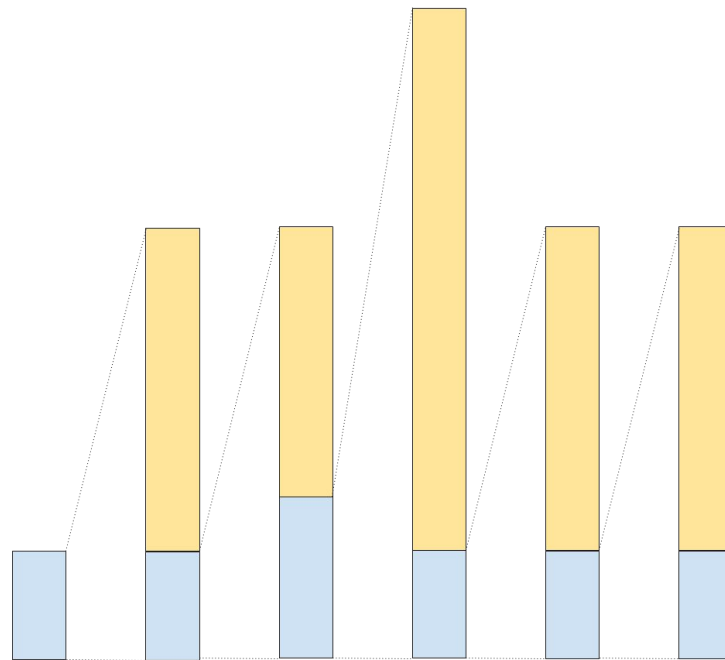
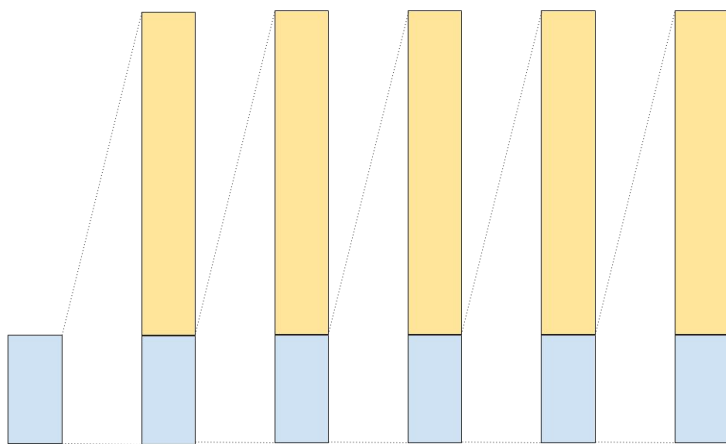


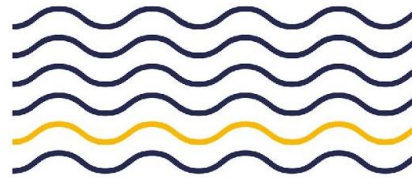
Golang
Piter



Increasing GOGC

GOGC = 300





A Glimpse of the (possible) Future

```
runtime.SetMaxHeap
```

Targeting the heap size instead of the heap growing ratio is handy If your application:

- have a small live heap
- a very high allocation rate
- enough free memory to use

Currently, the GC has no knowledge of the total available heap memory, but it may know it with the proposed API.

See issue [#16843](#) for more details!



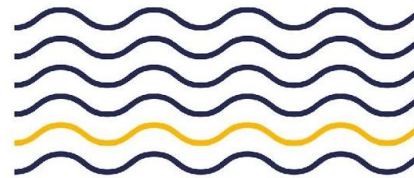


Thank You!





Golang
Piter



Contacts

fabio.falzoi84@gmail.com

github.com/Pippolo84

@Pippolo84