FABIO FALZOI

Senior Software Engineer

An insight into Go Garbage Collection

GoLab

October 21, 2019

develer

$ whoami

Senior SwEng @ Develer

Experience with C, C++, Python and Go

Passionate about low level topics…

… and Garbage Collection!

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”
```

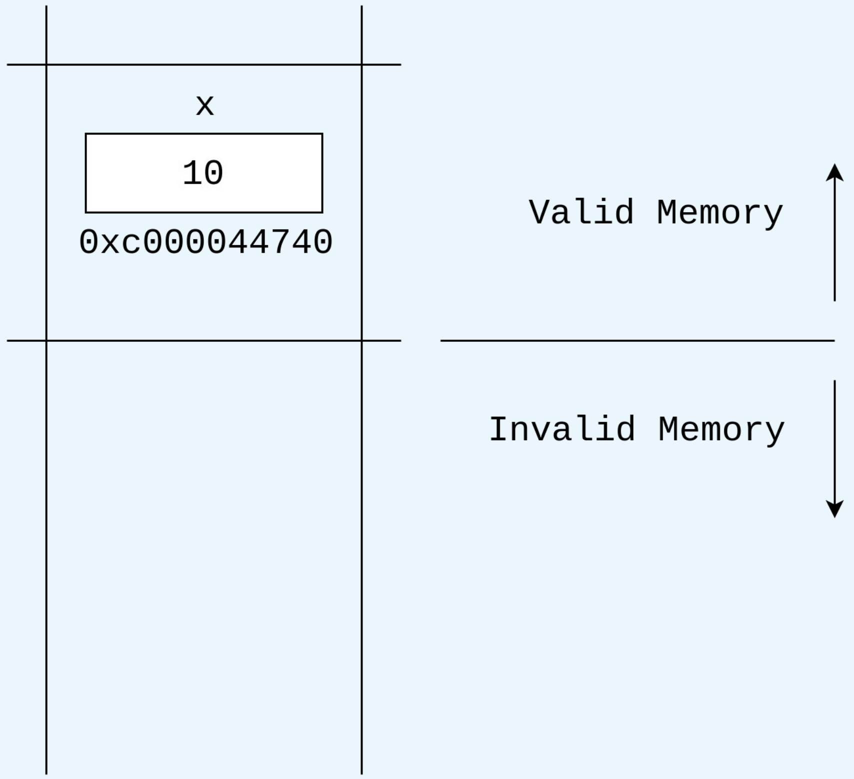
Goroutine User Stack

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

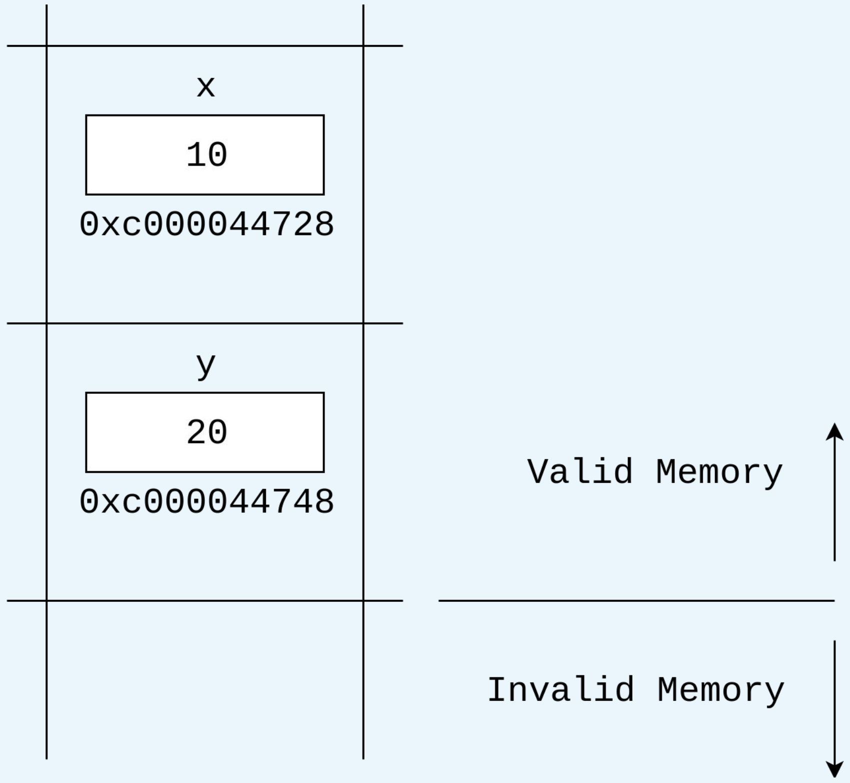
Stack Frames

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

    println(&x)
}

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



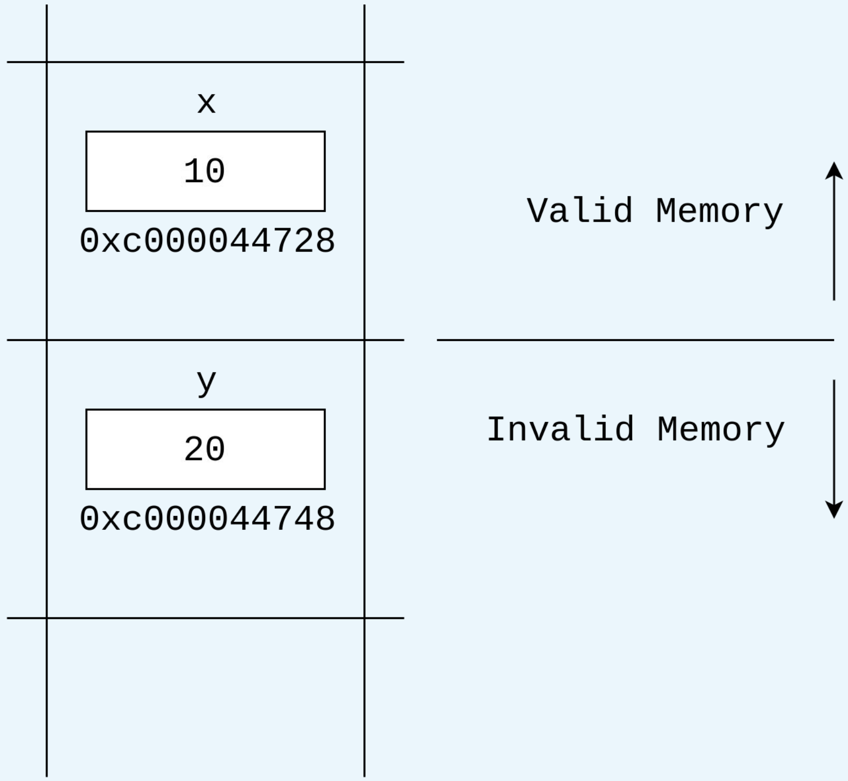
Stack Frames

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

    println(&x)
}

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



Stack Frames

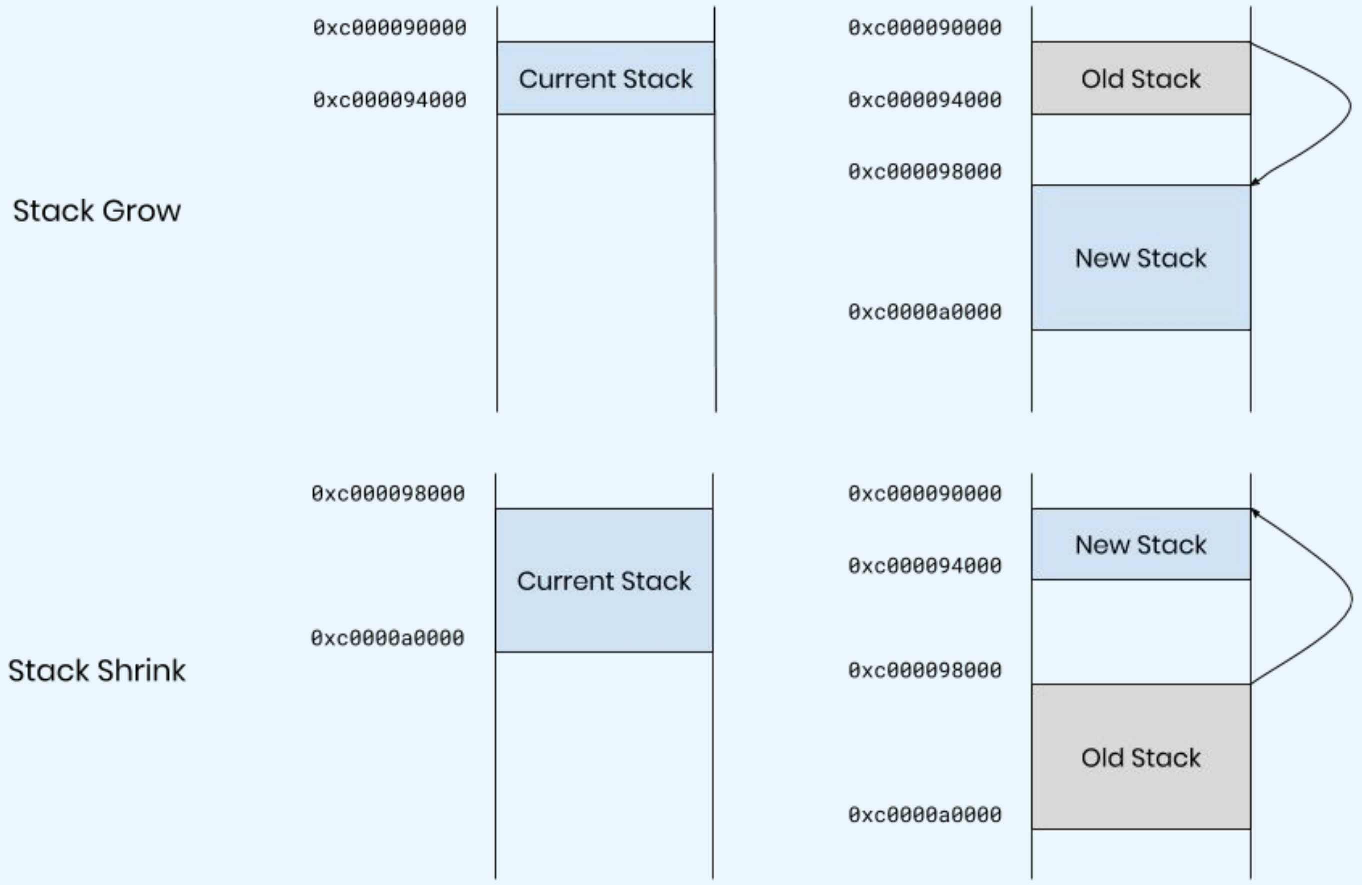
```go
func main() {
    x := 10
    f()

    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

Goroutine User Stack Size

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



Stack Allocation Paths

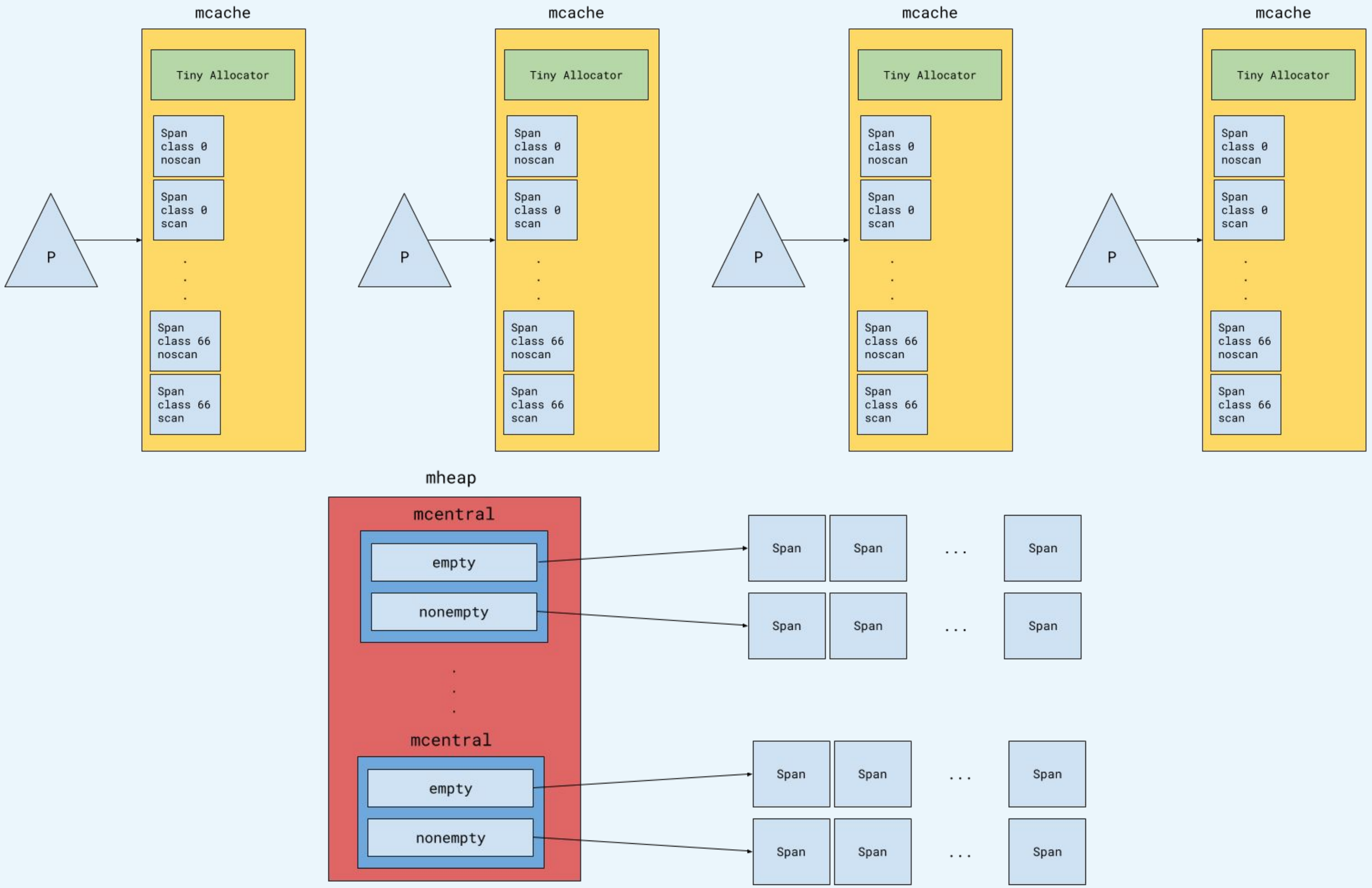
Where does the memory for a growable goroutine stack comes from?

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



Stack vs Heap - which one is cheaper?

For both stack and heap allocations patterns there are:

- caches to avoid locking in the common cases
- fixed-size free lists to reduce fragmentation

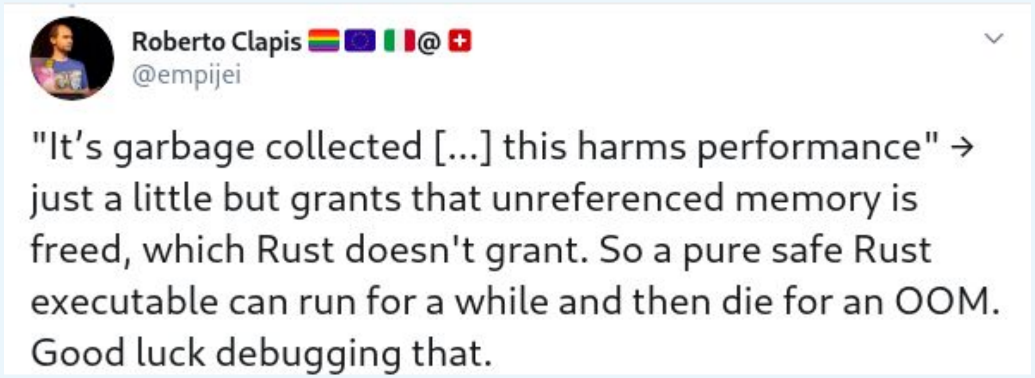
So, why stack allocations should be cheaper?

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

Destroying a stack frame means bumping up the Stack Pointer register
When the size of the live stack is < 25% of the allocated stack, it is shrinked

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

Is Garbage Collection evil?


Roberto Clapis 🏳️‍🌈🇪🇺🇮🇹@🇨🇭
@empijei

"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

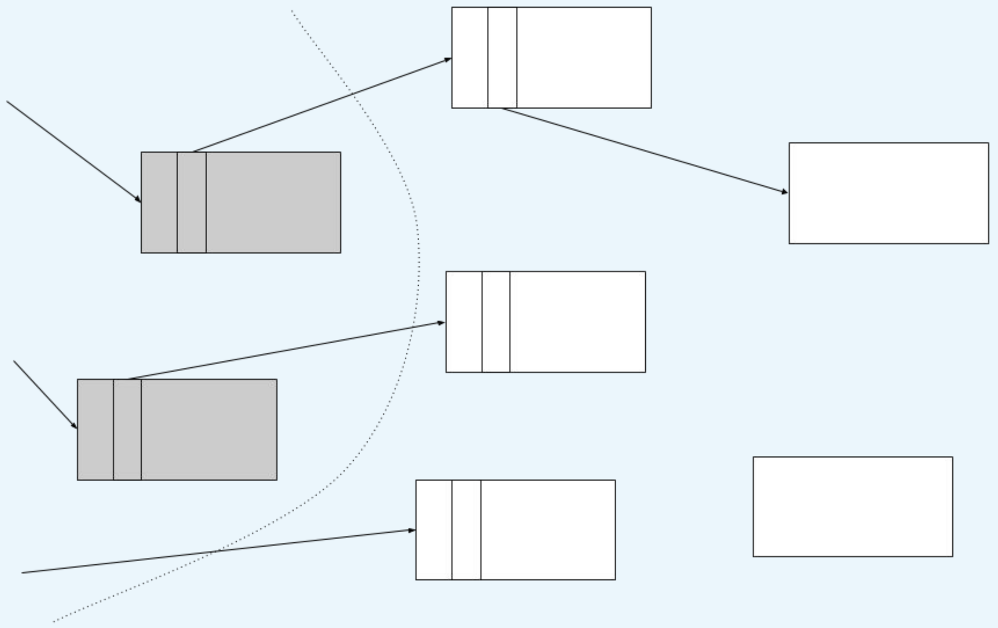
Mark phase & **Sweep** phases

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

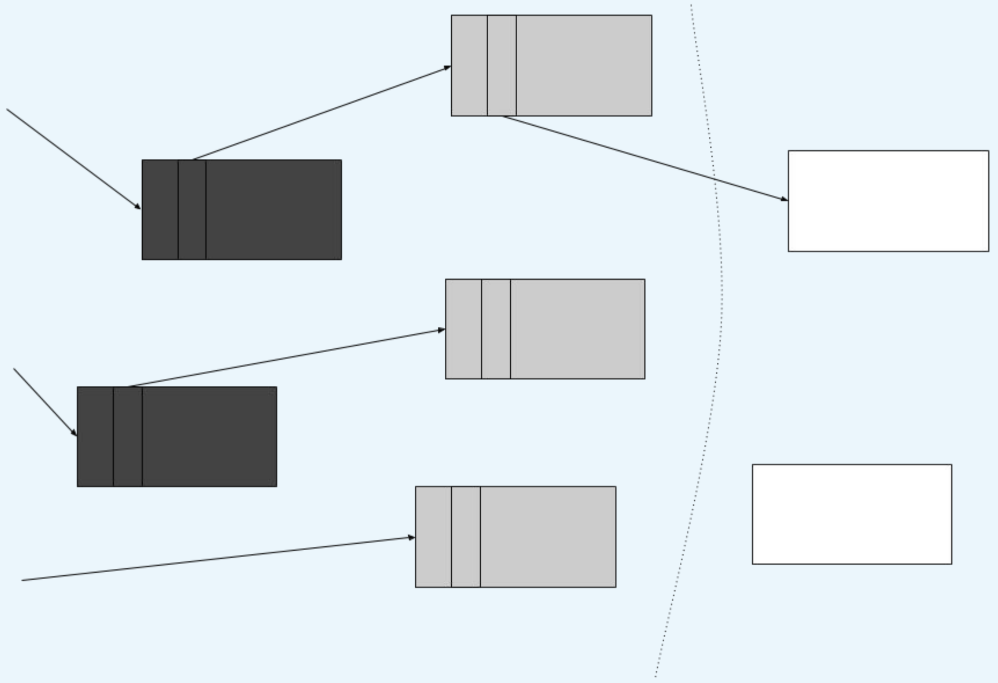
Tricolor Mark & Sweep

- **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



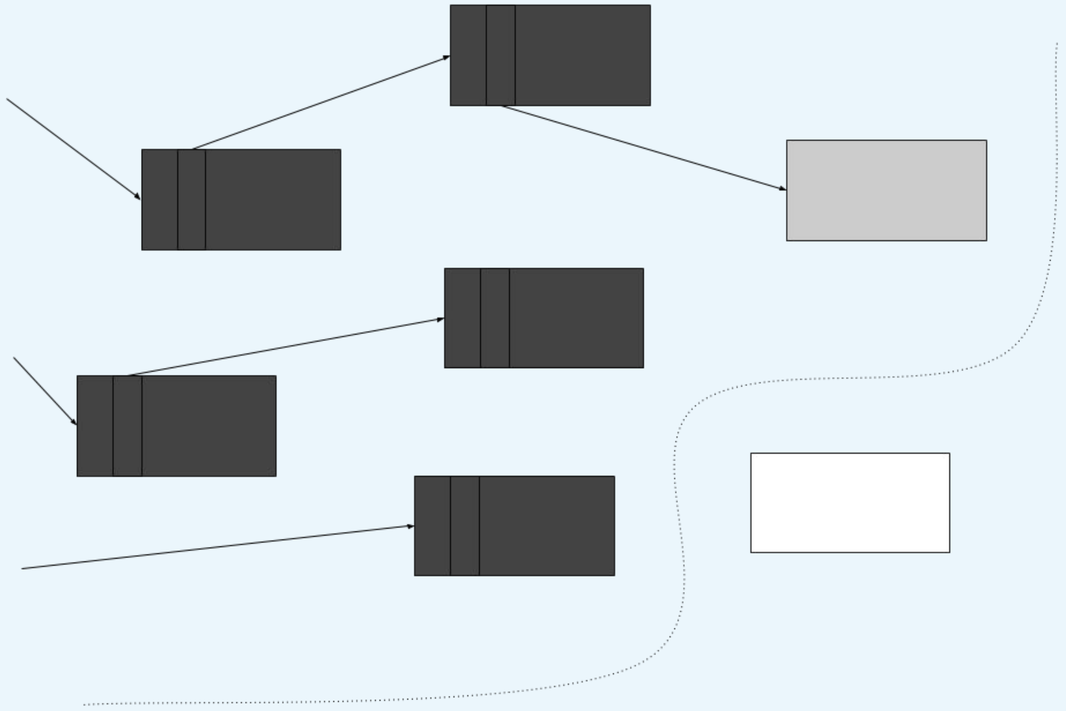
Tricolor Mark & Sweep

- **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



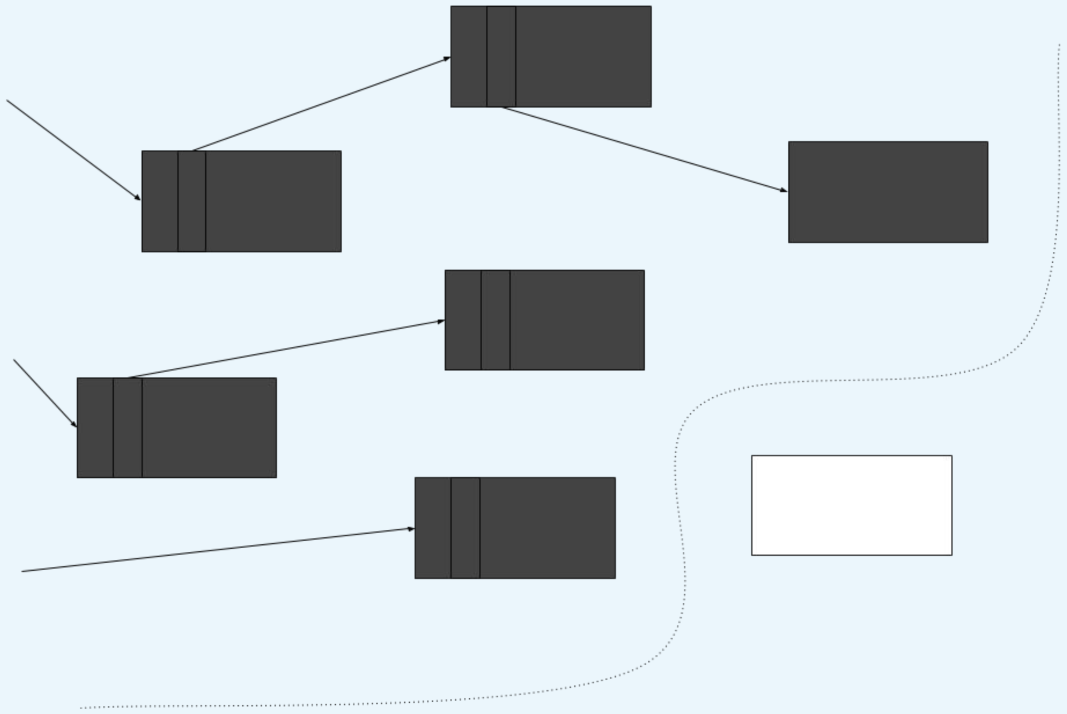
Tricolor Mark & Sweep

- **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



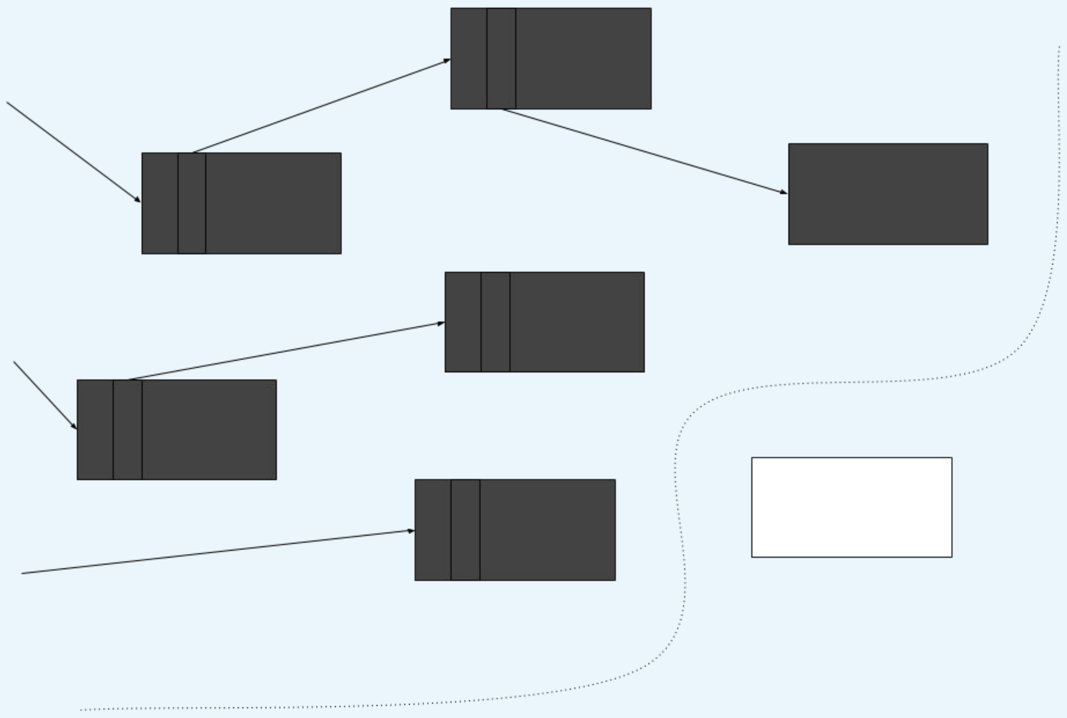
Tricolor Mark & Sweep

- **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

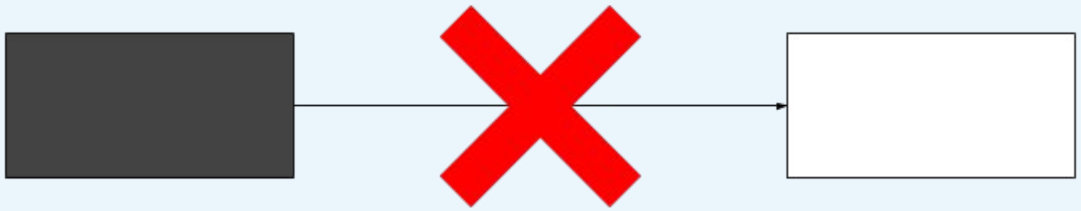


Tricolor Mark & Sweep

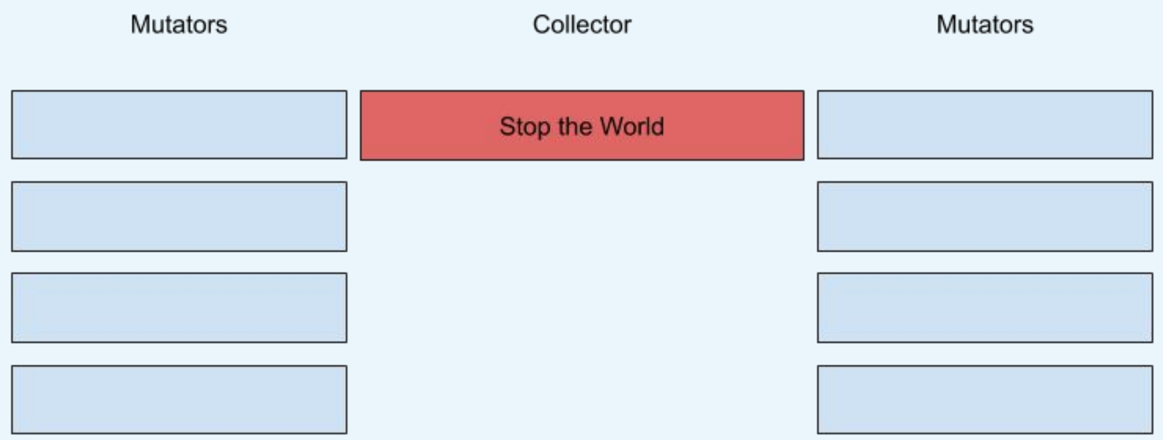
- **White set** objects not marked
- **Grey set** objects marked, but we have not yet scanned all their referents
- **Black set** objects marked along with all their referents



Strong Tricolor Invariant



Tricolor Mark & Sweep Pros & Cons



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?

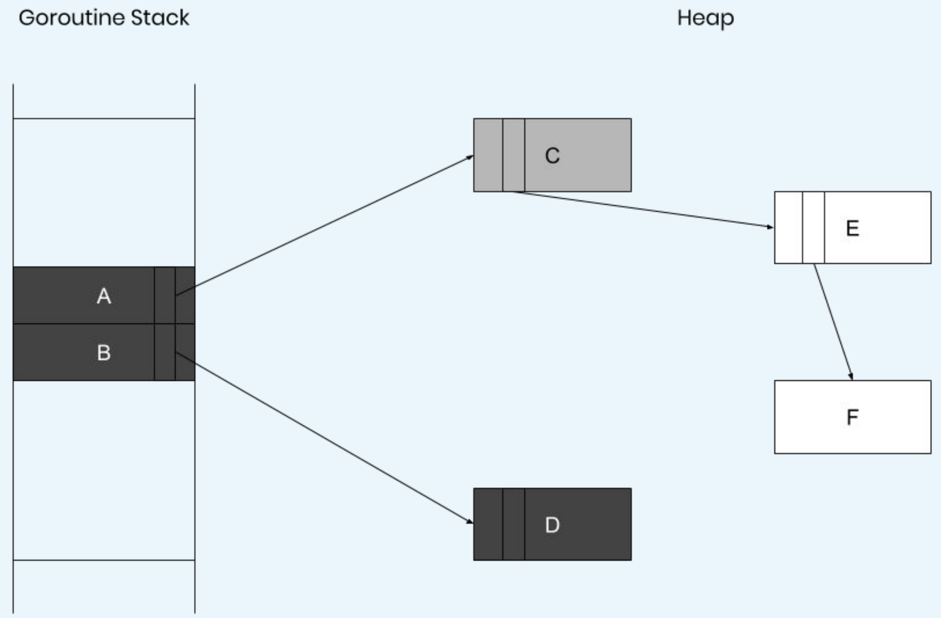
Since Go 1.5, the runtime executes GC **concurrently** to the mutators code, trading throughput for latency

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

What about the **marking phase**?

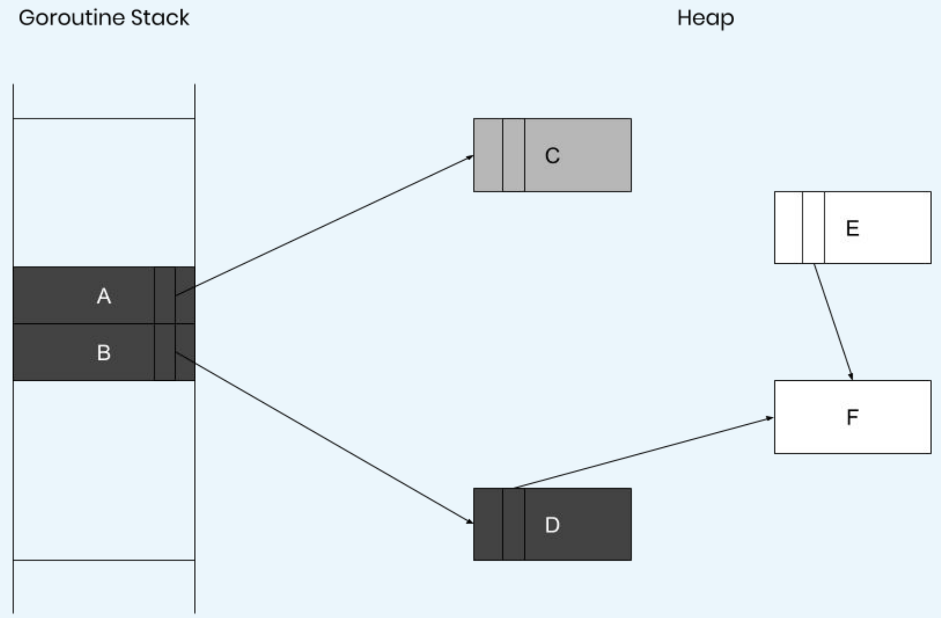
Concurrent Marking

```go
// Mutator code

type obj struct {
    // ...
    p *obj
    // ...
}

D.p = E.p
C.p = nil
```



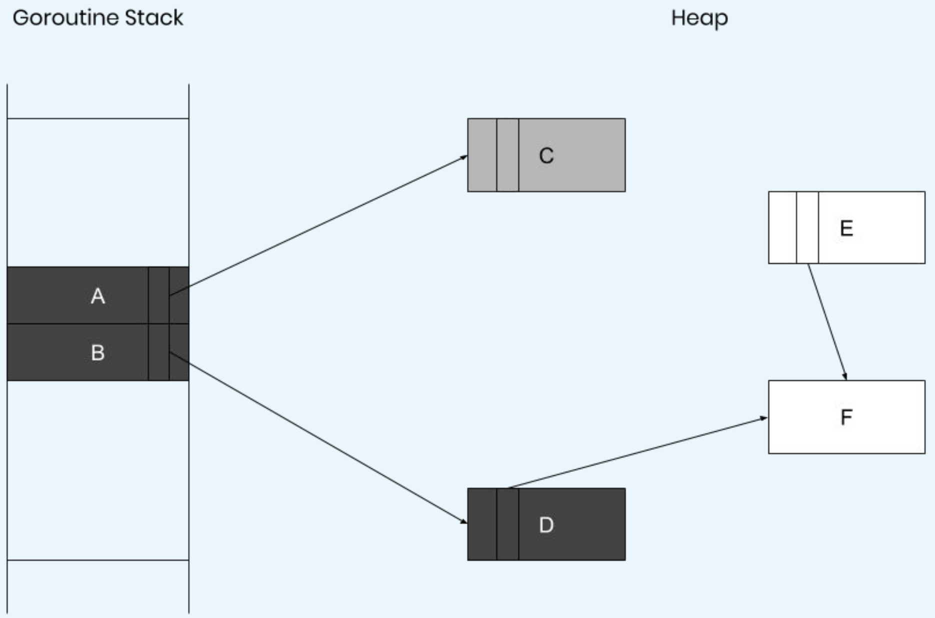
Concurrent Marking

```
// Mutator code

type obj struct {
    // ...
    p *obj
    // ...
}

D.p = E.p
C.p = nil
```



Concurrent Marking

```
// Mutator code

type obj struct {
    // ...
    p *obj
    // ...
}

D.p = E.p
C.p = nil
```



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

GC Barriers

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

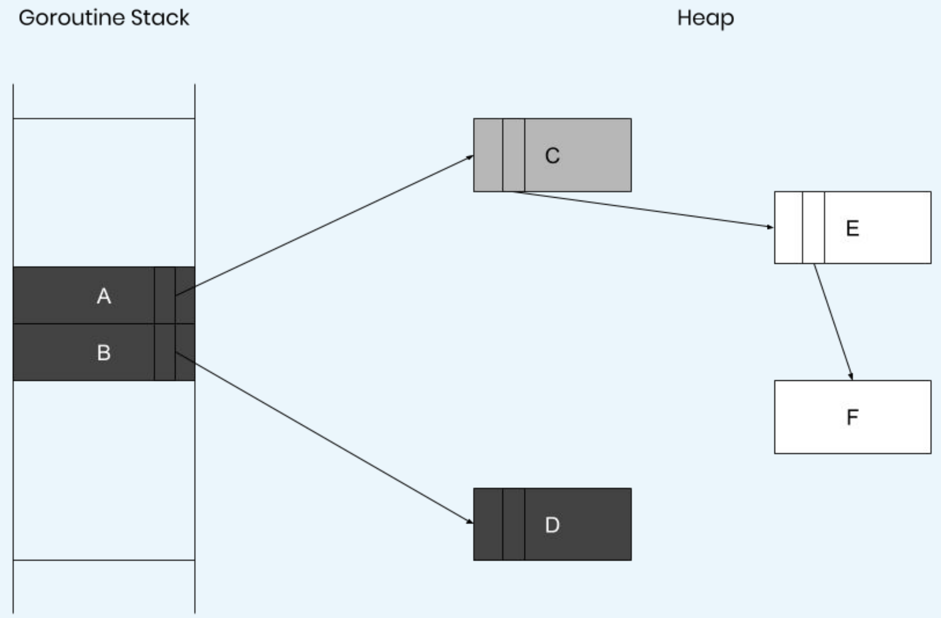
```
*slot = ptr
```

$\Rightarrow$

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

Dijkstra Write Barrier

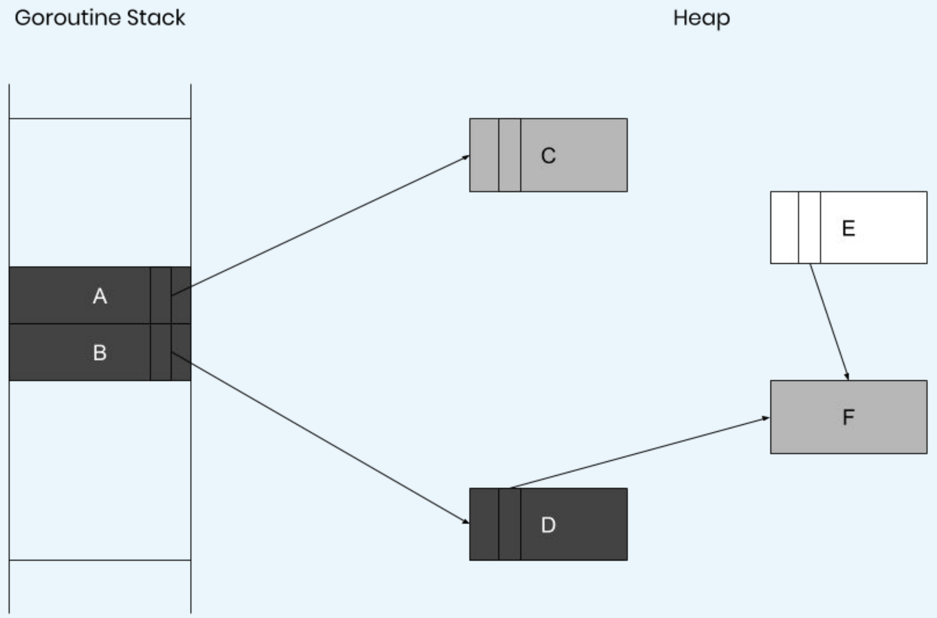
```
// Mutator code
D.p = E.p
C.p = nil

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



develeR

Dijkstra Write Barrier

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

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



Dijkstra Write Barrier

Pros

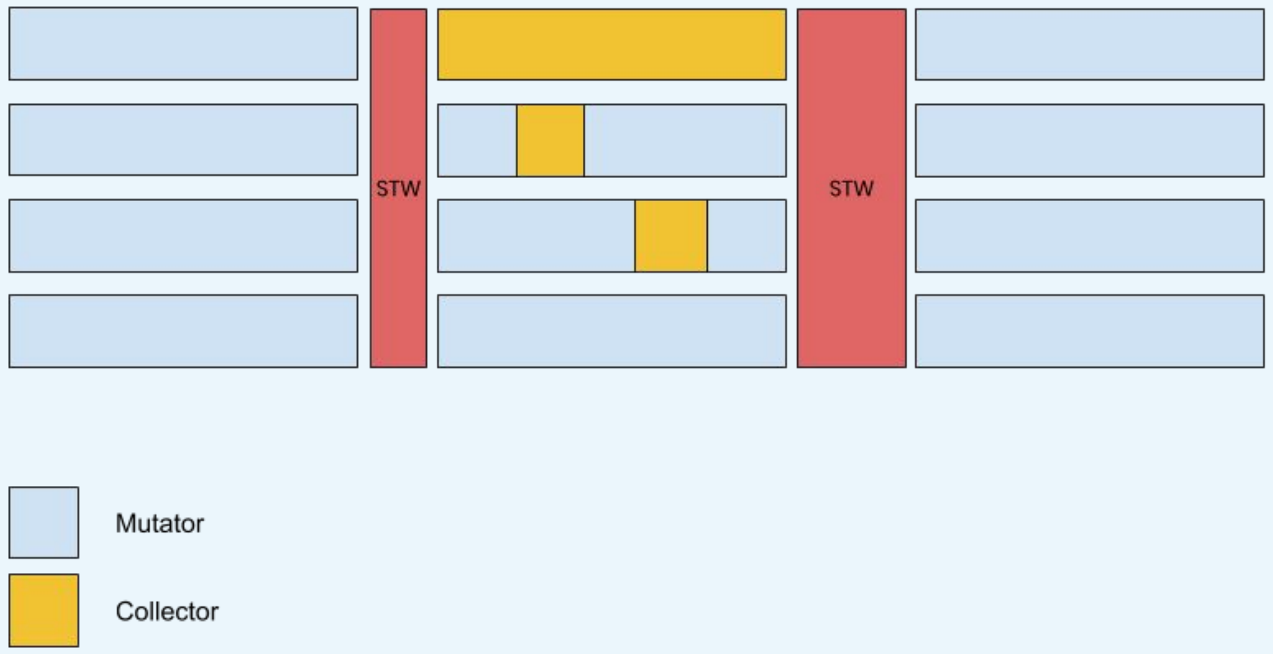
- ensures the **strong tricolor invariant**
- ensures **forward progress**

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**!

develeler

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

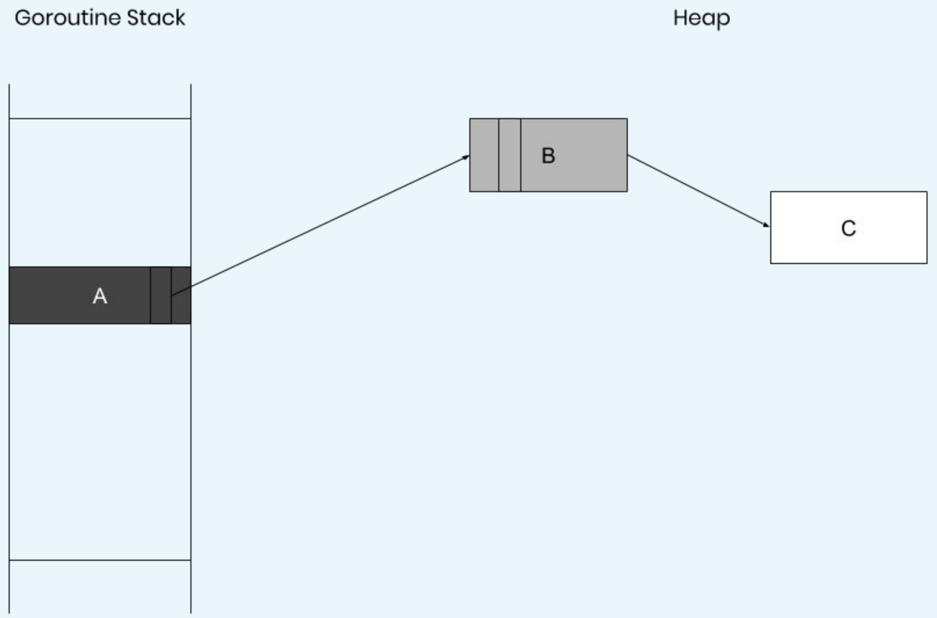
Yuasa - style barrier requires STW stack scanning at the **begin** of marking

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
}
```

Go Hybrid Write Barrier - Deletion

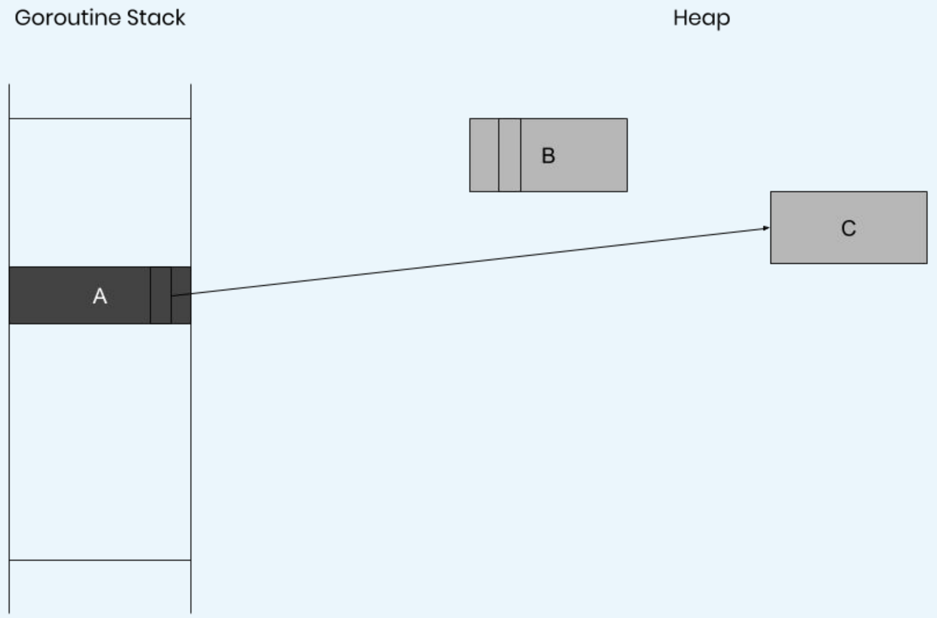
```
// Mutator code
A.p = B.p
B.p = nil

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



Hiding an object by moving the sole pointer to it from the heap to a black object in the stack

Go Hybrid Write Barrier - Deletion

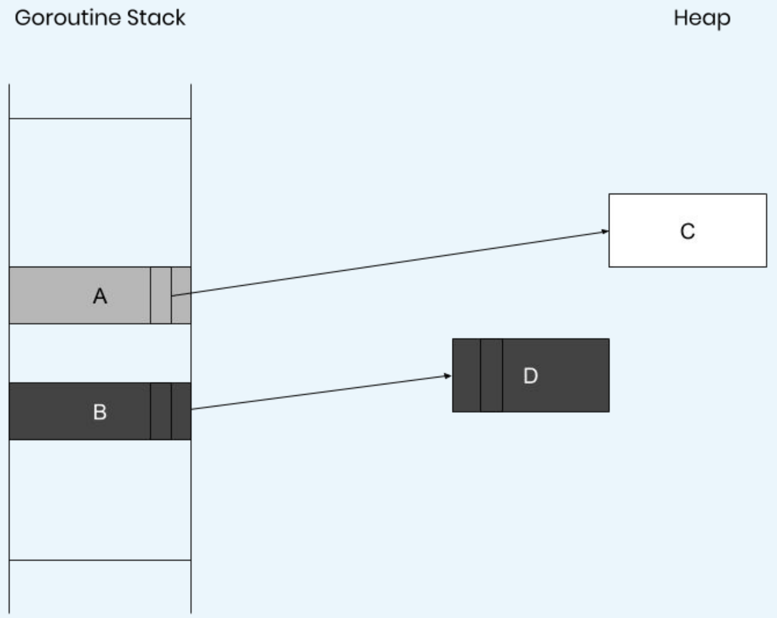
```
// Mutator code
A.p = B.p
B.p = nil

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



Hiding an object by moving the sole pointer to it from the heap to a black object in the stack

Go Hybrid Write Barrier - Insertion

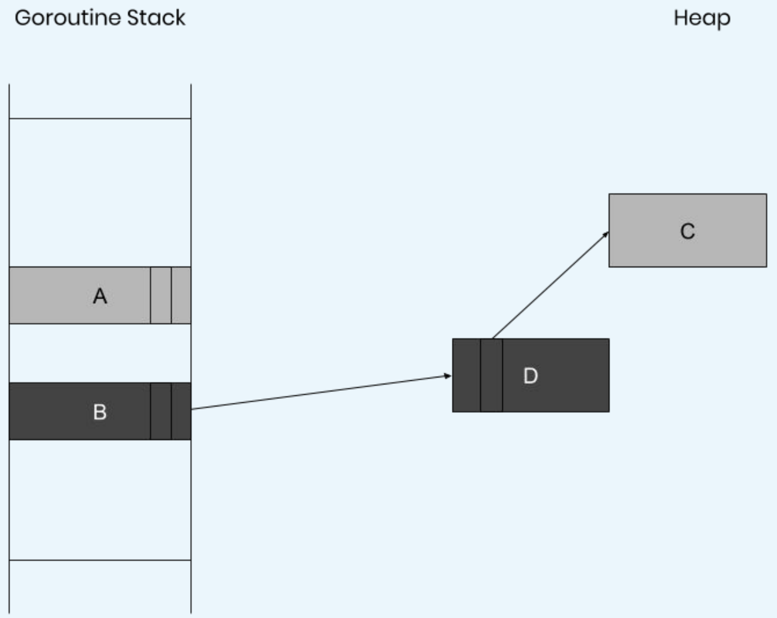
```
// Mutator code
D.p = A.p
A.p = nil

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



Hiding an object by moving the sole pointer to it from its stack to a black object in the heap

Go Hybrid Write Barrier - Insertion

```
// Mutator code
D.p = A.p
A.p = nil

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



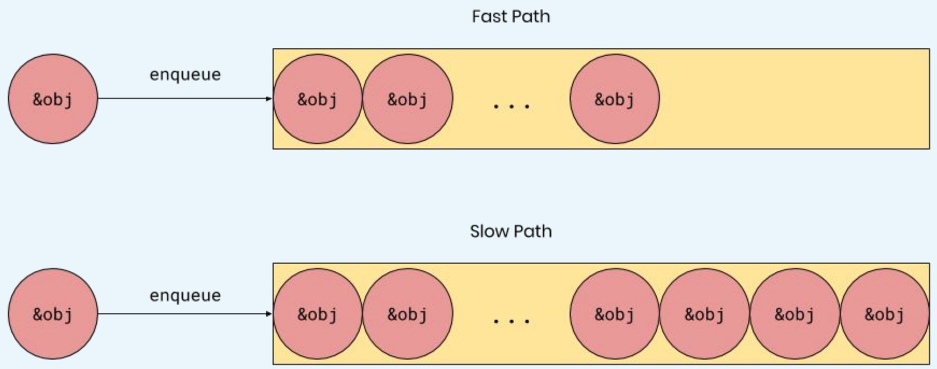
Hiding an object by moving the sole pointer to it from its stack to a black object in the heap

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**

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



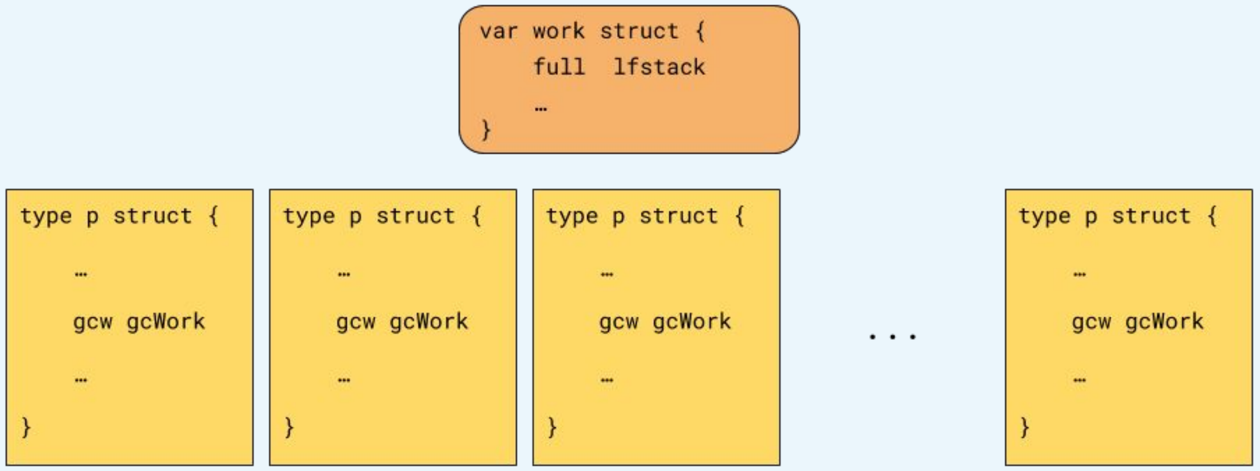
When it is full, the hybrid write barrier jumps to the **slow path**, where it flushes its buffer and greys all the pointers as usual!

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



```
var work struct {
    full  lfstack
    …
}
```

```
type p struct {
    …
    gcw gcWork
    …
}
```

```
type p struct {
    …
    gcw gcWork
    …
}
```

```
type p struct {
    …
    gcw gcWork
    …
}
```

```
type p struct {
    …
    gcw gcWork
    …
}
```

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 allocations 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?

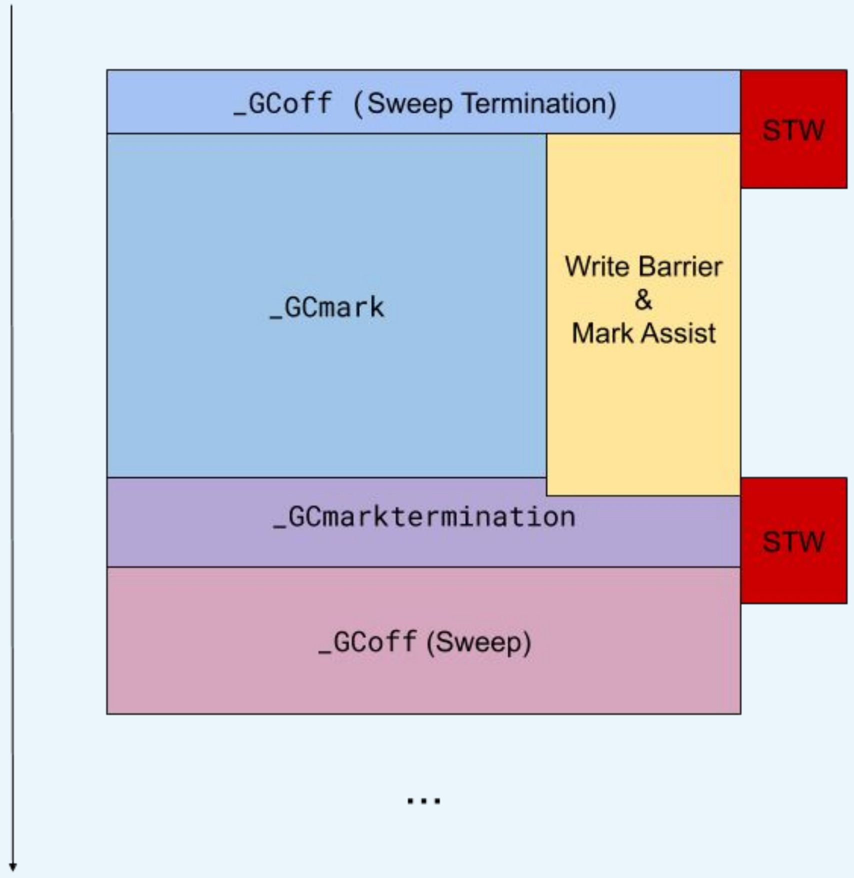
Mark Termination Algorithm rewritten in Go 1.12

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?

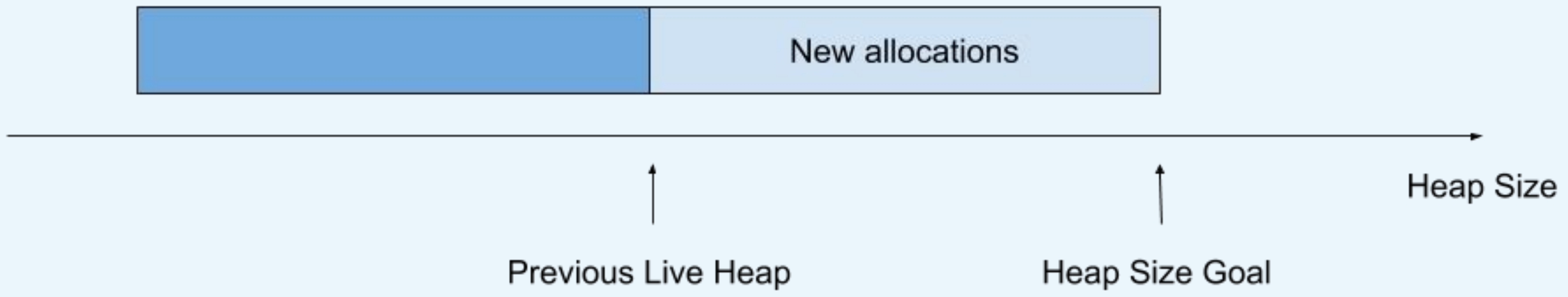
With **GOGC** environment variable the user sets a **heap goal**

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

GOGC default value is 100

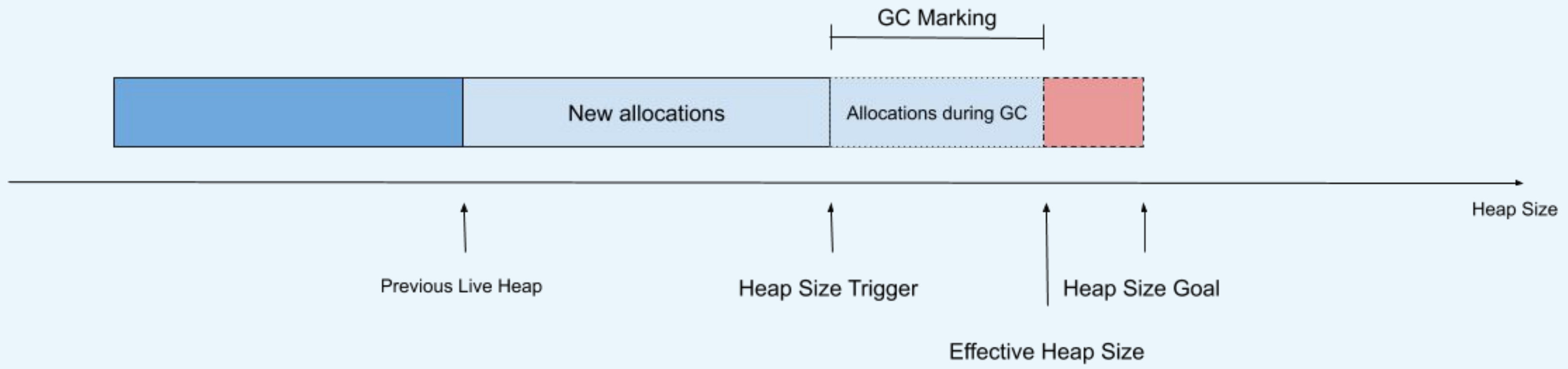
STW Mark and Sweep Trigger

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

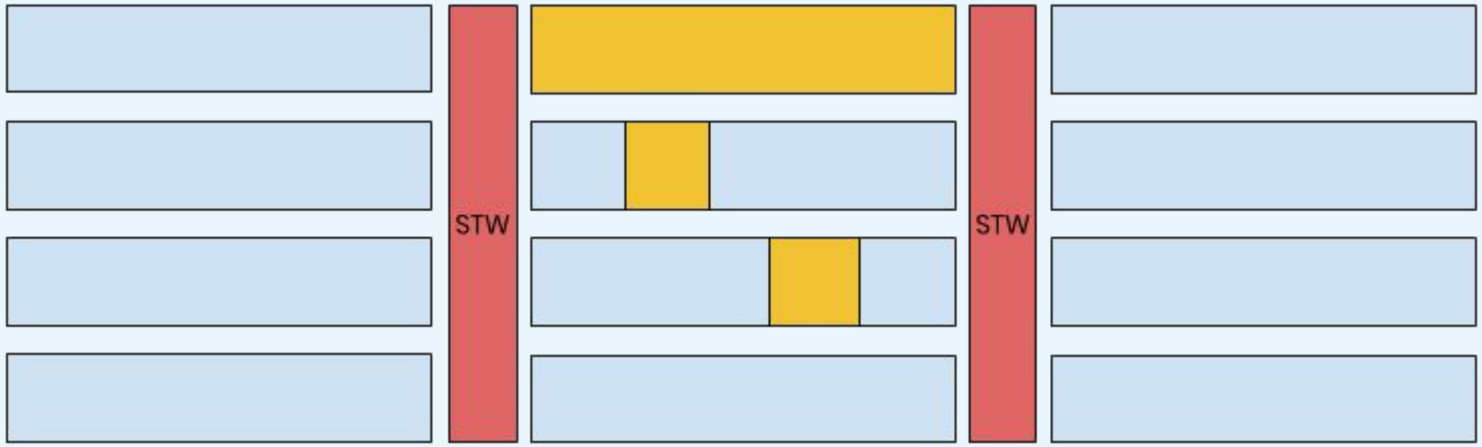
Sweeping rate is based on a budget system just like the proportional marking

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

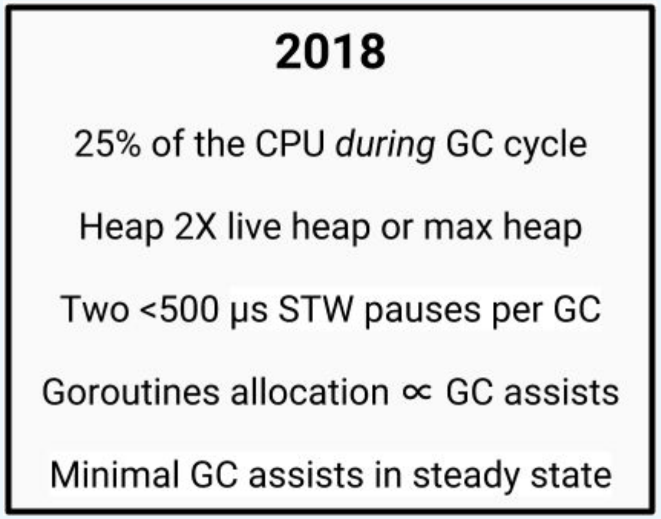


- STW pauses at the beginning and at the end of each cycle
- 25% CPUs dedicated to Background Marking
- Mark Assist
- Write Barrier on during each cycle
- Background and lazy sweeping

develer

Go GC SLOs

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



2018

25% of the CPU *during* GC cycle

Heap 2X live heap or max heap

Two <500 µs STW pauses per GC

Goroutines allocation ∝ GC assists

Minimal GC assists in steady state

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

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

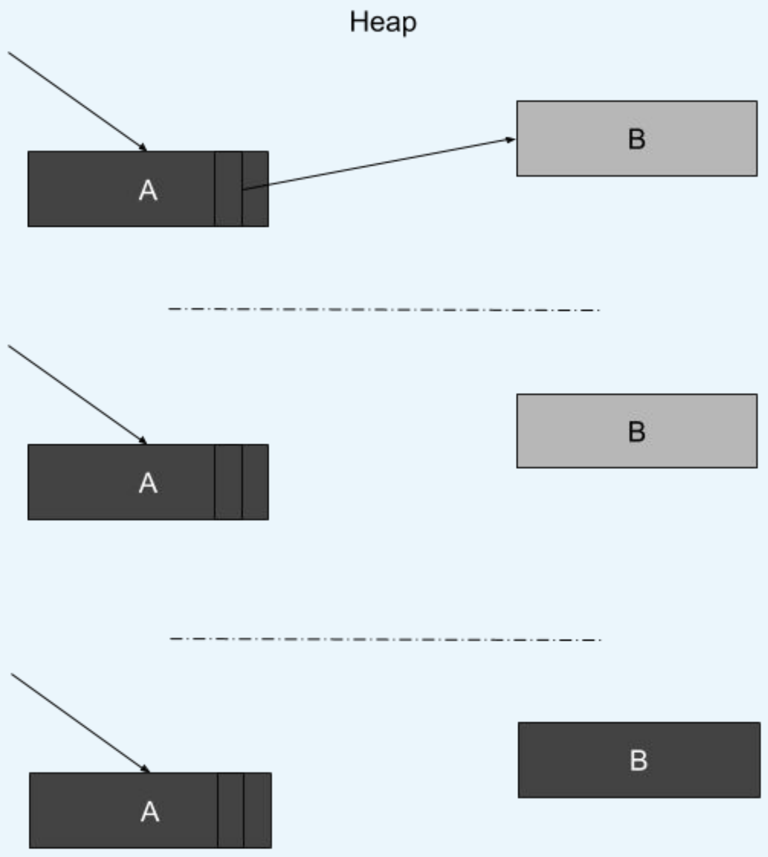
Typical STW pauses ~ **tenths of microseconds**

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



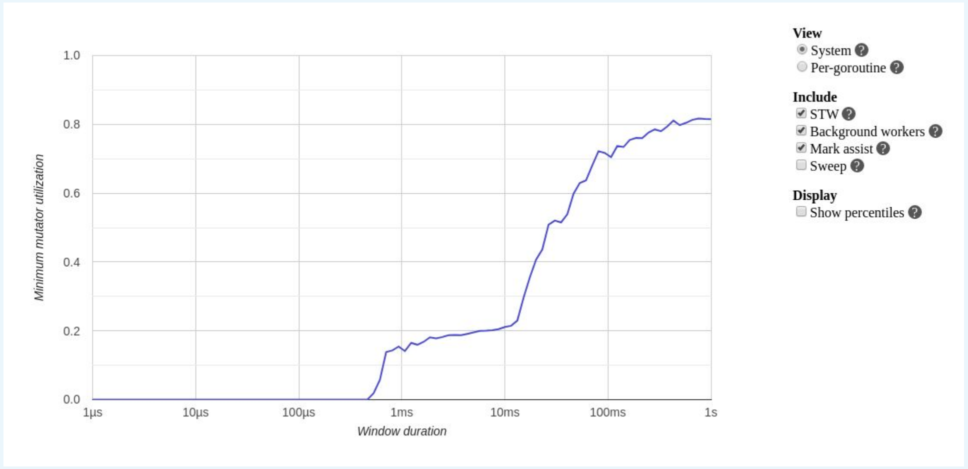
The GC retains objects that are reachable **at some point** during marking, even if **they are not** at the end of the cycle, due to the mutator executing concurrently

Tools of the trade - Am I experiencing GC pressure?

Minimum Mutator Utilization curve

The x axis represents time
The y axis the fraction of CPU time spent in the mutator (CPU utilisation)

y-intercept is the mutators' overall share of processor time
x-intercept is the maximum pause time



View
● System
○ Per-goroutine

Include
✓ STW
✓ Background workers
✓ Mark assist
☐ Sweep

Display
☐ Show percentiles

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

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?

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

Reuse Memory

```go
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**

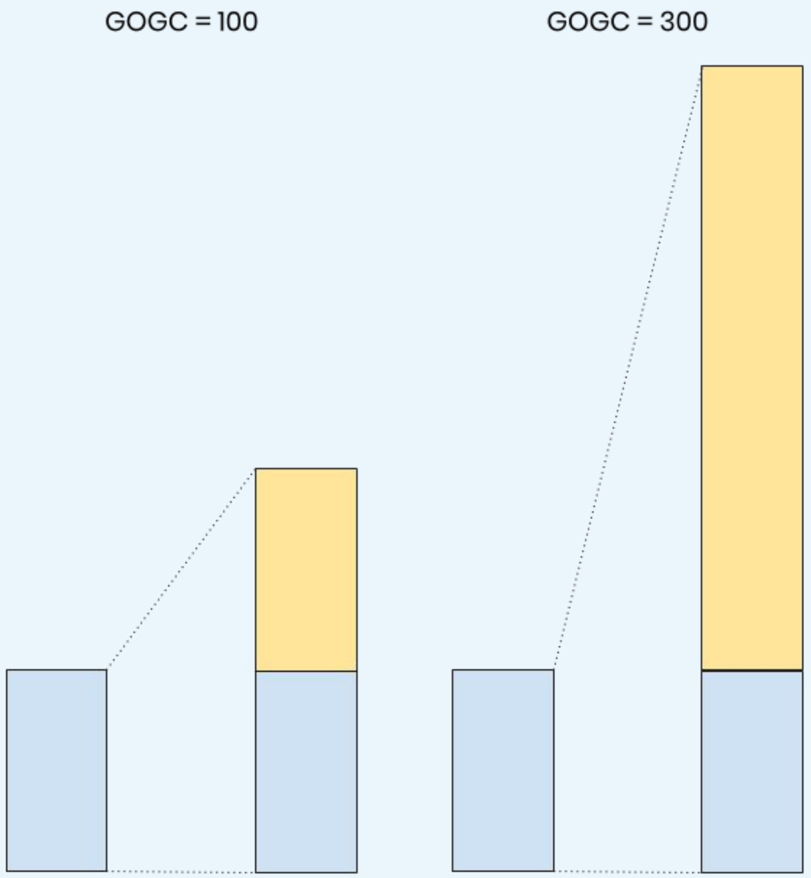
Garbage Collector Tuning

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

GOGC knob: trading memory for CPU utilization

You can change it at runtime too:
`runtime/debug.SetGCPercent`

The heap growth becomes **harder** to control



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!

Questions?

CONTACTS

Fabio Falzoi

fabio@develer.com

github.com/Pippolo84

@Pippolo84

develer

www.develer.com