GoVirCon 2020

Evolving the Go Memory Manager's RAM and CPU Efficiency

Michael Knyszek

Google, Inc.

mknyszek@google.com

Overview

This is a tutorial.

How does Go manage OS-provided memory?

- What does that mean for you?

Why does it work this way?

- Goals, policy, and design

Overview

This is a fun (?) tutorial.

What went hilariously wrong?

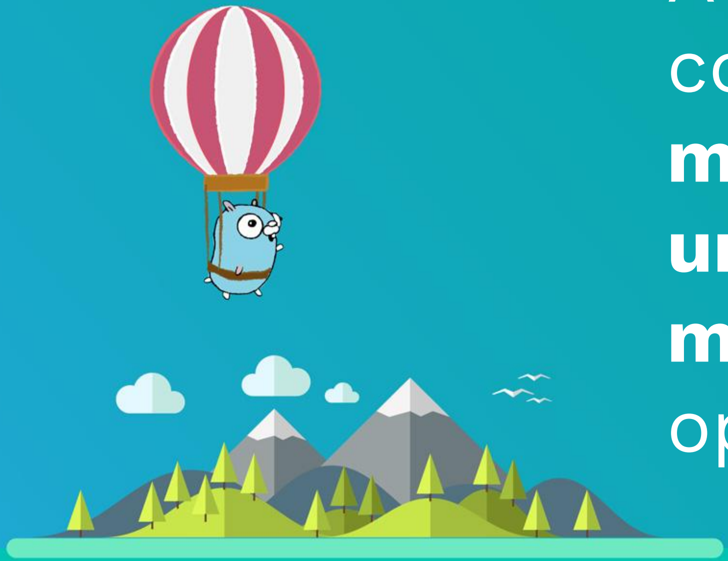
Get to talk about things not even allocator experts talk about very much!

Overview

1. Background (why does this matter?)
2. Interacting with the OS (the building blocks)
3. Managing pages (recycling building blocks)
4. Conclusion

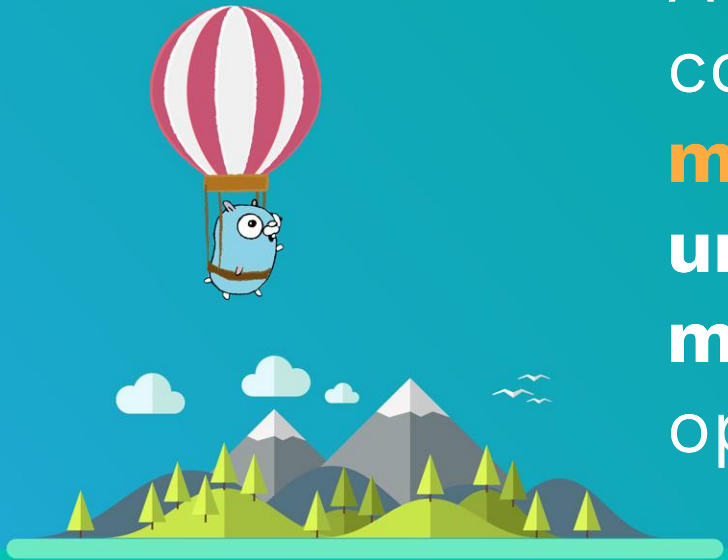
Part 1

Background



“A ***page*** [...] is a **fixed-length** contiguous block of **virtual memory** [...]. It is the **smallest unit of data for memory management** in an [...] operating system.”

-Wikipedia

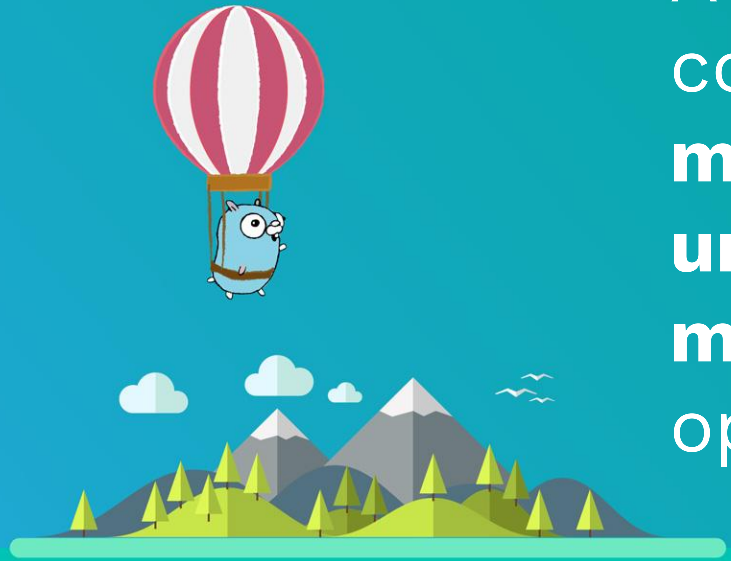


“

A ***page*** [...] is a **fixed-length** contiguous block of **virtual memory** [...]. It is the **smallest unit of data for memory management** in an [...] operating system.

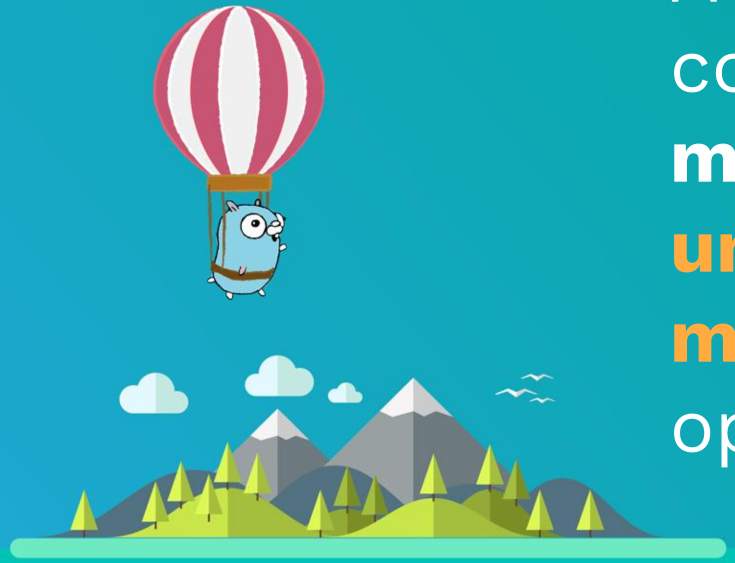
”

-Wikipedia



> A ***page*** […] is a **fixed-length** contiguous block of **virtual memory** […]. It is the **smallest unit of data for memory management** in an […] operating system.

-Wikipedia



“A ***page*** [...] is a **fixed-length** contiguous block of **virtual memory** [...]. It is the **smallest unit of data for memory management** in an [...] operating system.”

-Wikipedia

Dynamic Memory Allocation

We have some data our program wants to keep in memory somewhere.

The Go compiler is unable to determine where to put it.

Let's find a place at runtime!

The motivation.

Dynamic Memory Allocation

The OS gives us ***pages***, so the allocator needs to manage them...

...but the data we want to place is rarely exactly the size of a page.

The conundrum.

Orientation

The Layer Cake

- Go object allocator
- Go page allocator
- Operating system (OS)
- Physical memory (RAM)

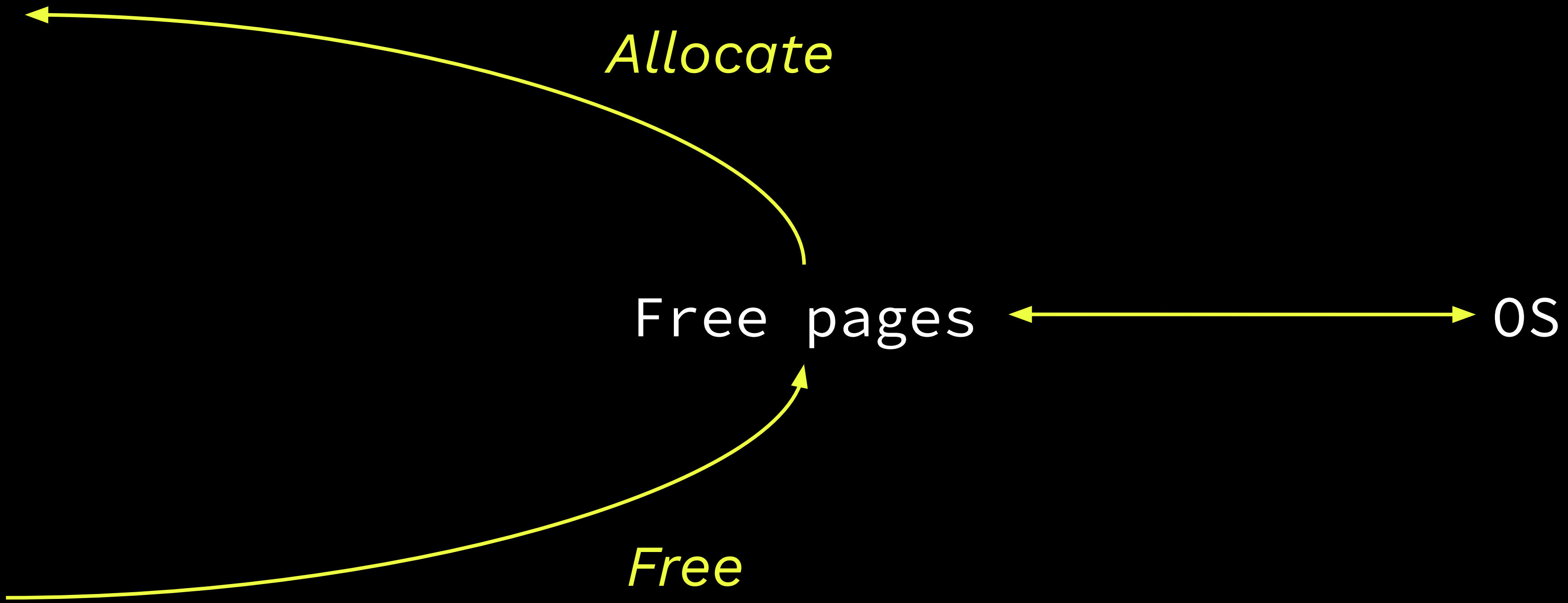
Dynamic Memory Allocation

Take as a given that the allocator needs:

- A "pool" of free pages.
- To allocate contiguous blocks of pages.
- To reuse partially-used pages eagerly.
- To free pages back to the pool eagerly.

The background.

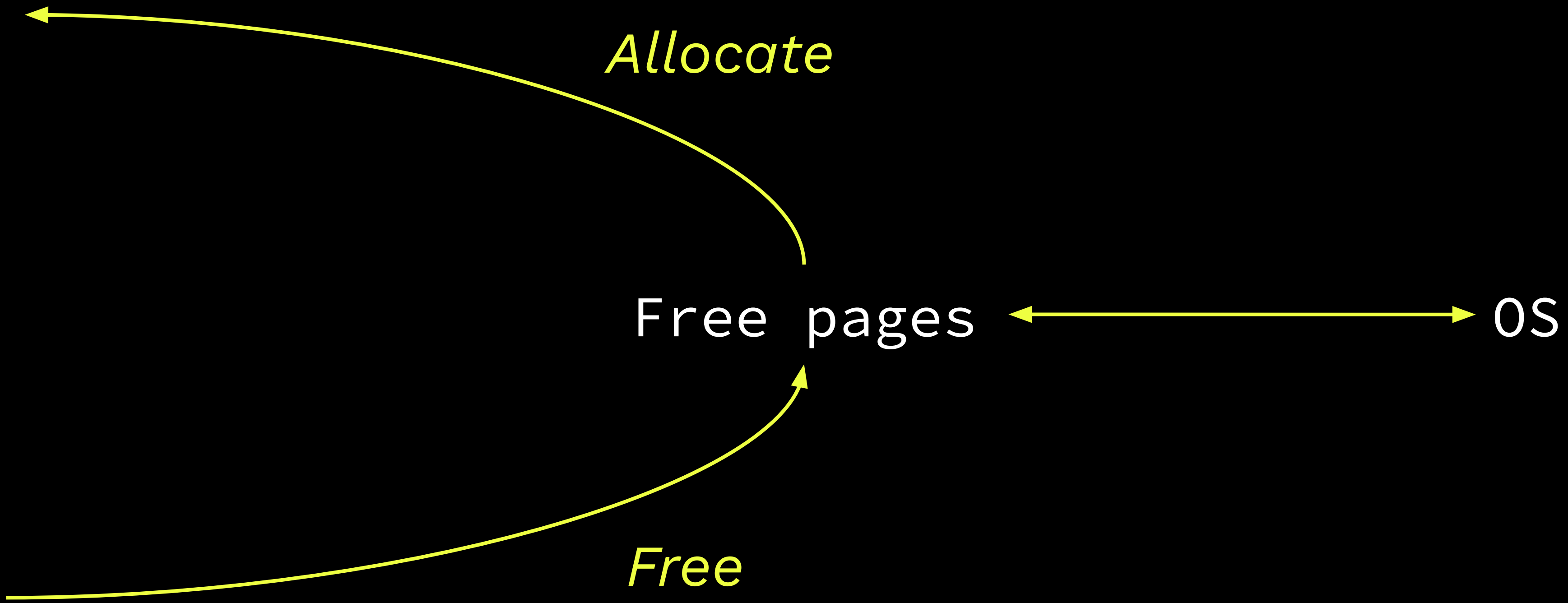
The Heap Memory "Lifecycle"



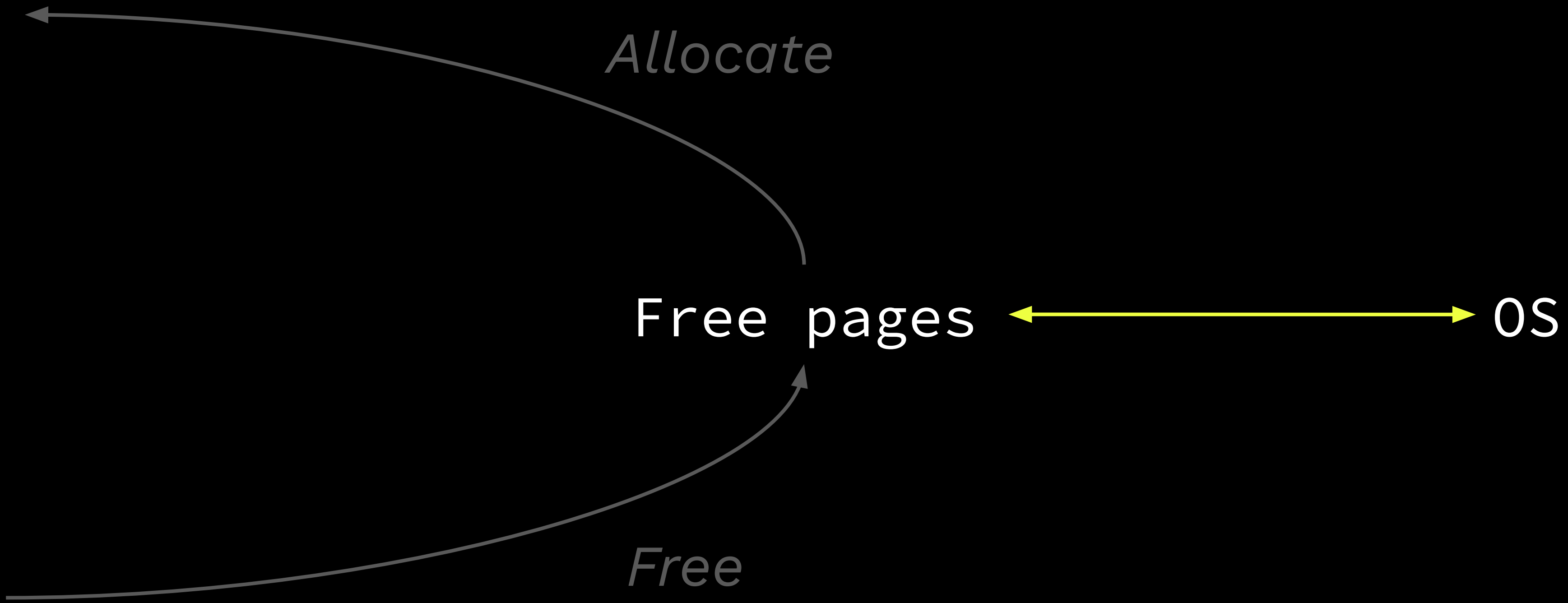
Part 2

Interacting with the OS

The Heap Memory "Lifecycle"



The Heap Memory "Lifecycle"



Obtaining memory from the OS

Request new virtual memory as read-write.

```
// Unix-y systems
mmap(PROT_READ|PROT_WRITE, MAP_ANON|MAP_PRIVATE)

// Windows
VirtualAlloc(MEM_RESERVE|MEM_COMMIT)
```

Backed by physical memory on first use.

How?

Returning memory to the OS

Mapping and unmapping a lot is slow.

In reality, we don't need the *contents* of memory anymore.

```
// Unix-y systems… kind of.
madvise(MADV_DONTNEED)

// Windows
VirtualAlloc(MEM_DECOMMIT)
```

How?

Un-returning memory from the OS

But, those pages are still in our address space! What happens if we use them?

Two schools of thought:

- Unix-y: You touch it you buy it.
- Windows-y: Ask me first.

The results are mostly equivalent.

How?

Talking policy

Obtaining memory.	**When:** we're fresh out. **What:** contiguous.
Returning memory.	???

When and what?

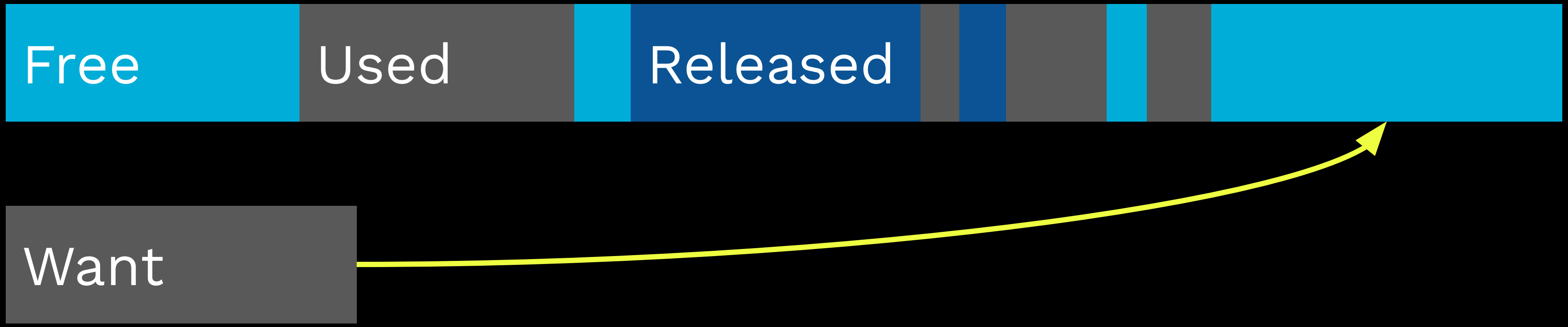
Returning memory to the OS

When:
- Check whole heap every few minutes.

What:
- Pages free and unused for 5 minutes.

Which memory and when? (Go 1.11 and earlier)

Scenario: Large allocation to fragmented heap



We might run out of memory sooner!

Returning memory to the OS

When:
- Check whole heap every few minutes.

What:
- Pages free and unused for 5 minutes.

Which memory and when? (Go 1.11 and earlier)

Returning memory to the OS

When:

- Check whole heap every few minutes.
- When we ask the OS for more memory.

What:

- Pages free and unused for 5 minutes.

Which memory and when? (Go 1.12)

Returning memory to the OS

When:

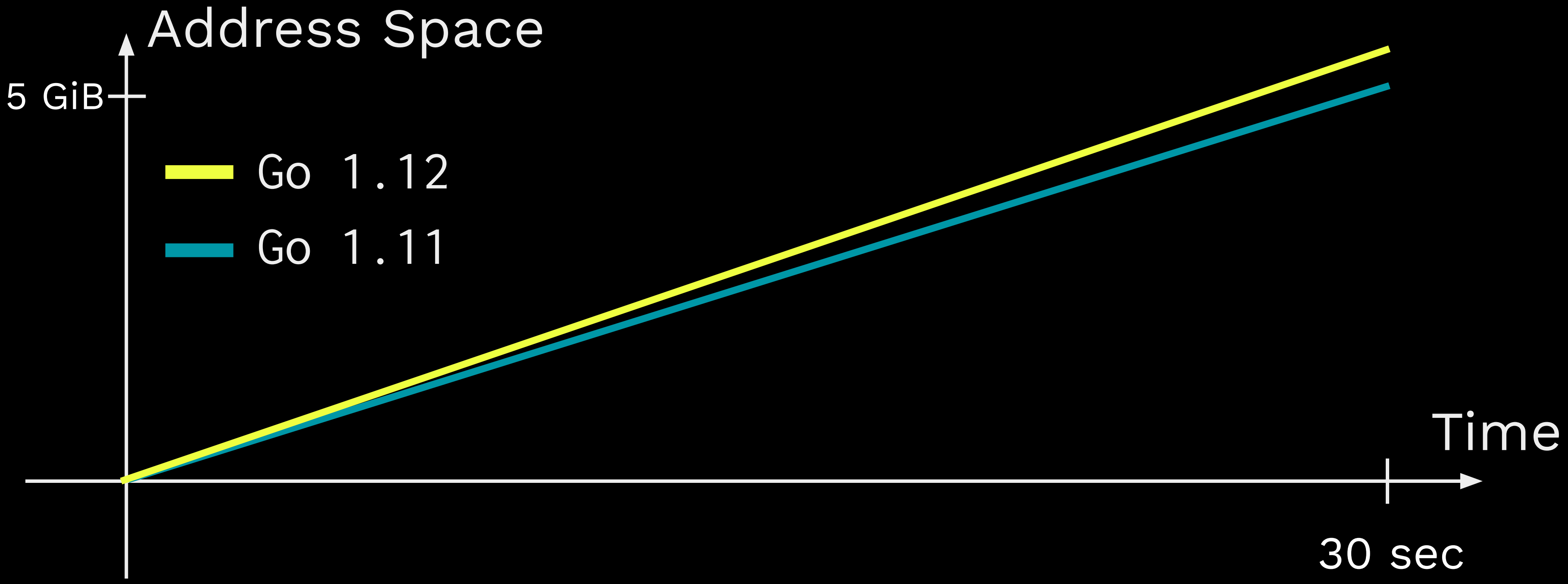
```
- Check whole heap every few minutes.
- When we ask the OS for more memory.
```

What:

```
- Pages free and unused for 5 minutes.
- Largest contiguous block of pages first.
```

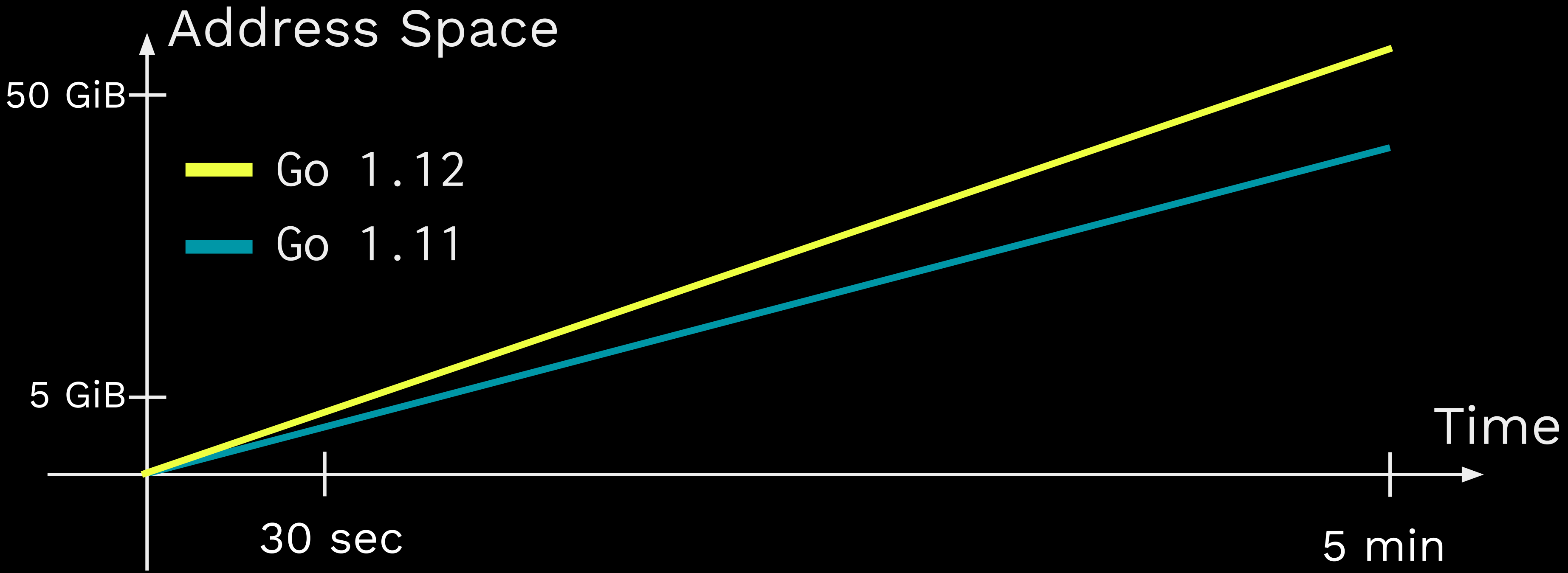
Which memory and when? (Go 1.12)

Validating the implementation



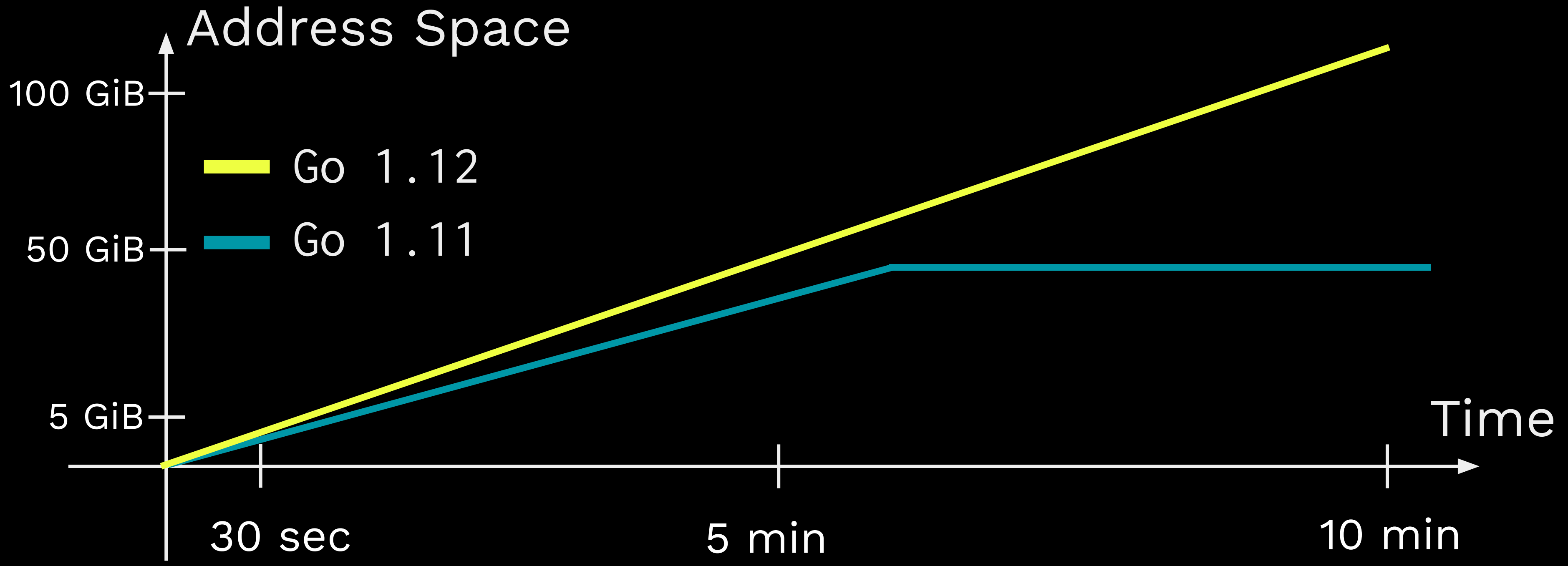
So far so good...

Validating the implementation



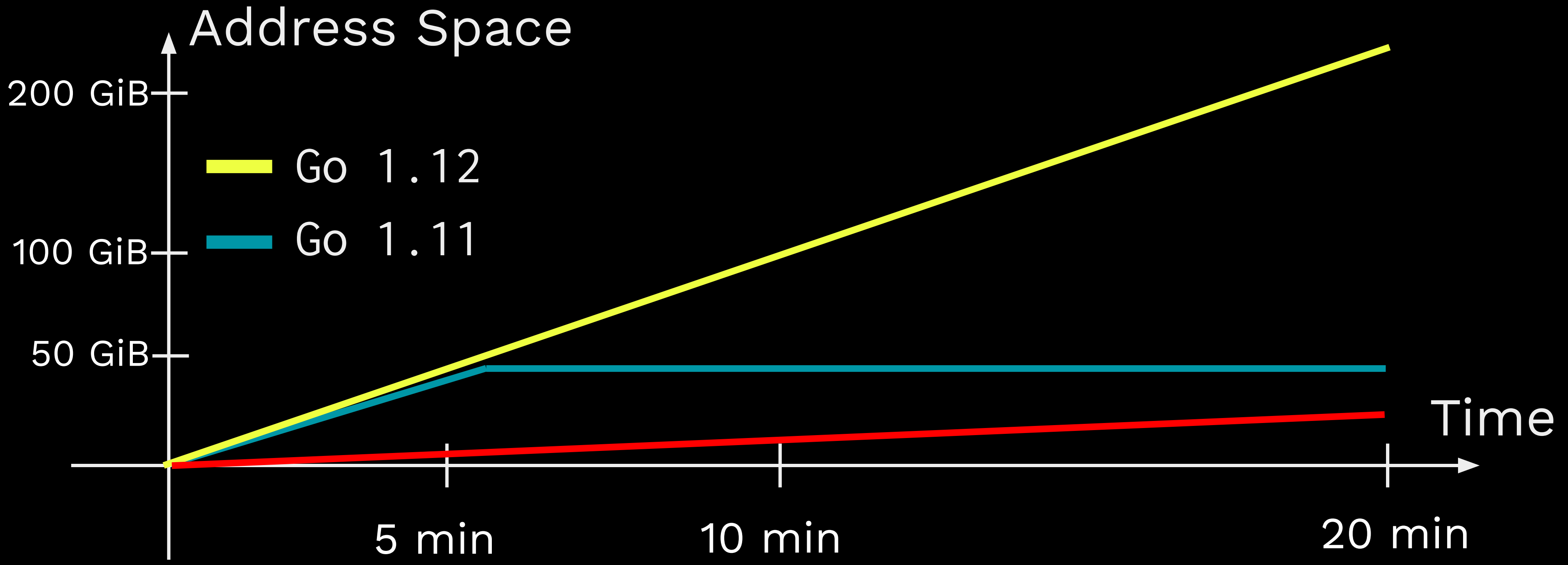
A little fast, but OK as long as it stabilizes...

Validating the implementation



Uhhh...

Validating the implementation



Oh no...

“

The bugs we found along the way.

Free pages are organized in a tree…

…but only if they're above a certain size…

…but I put *everything* in the tree…

…and the `find` operation was *wrong*.

LGTM

```go
func (root *tree) findBestFit(npages uintptr) *node {
	t := root.tree
	for t != nil {
		if t.npages < npages {
			t = t.right
		} else if t.left != nil && t.left.npages >= npages {
			t = t.left
		} else {
			return t
		}
	}
	return nil
}
```

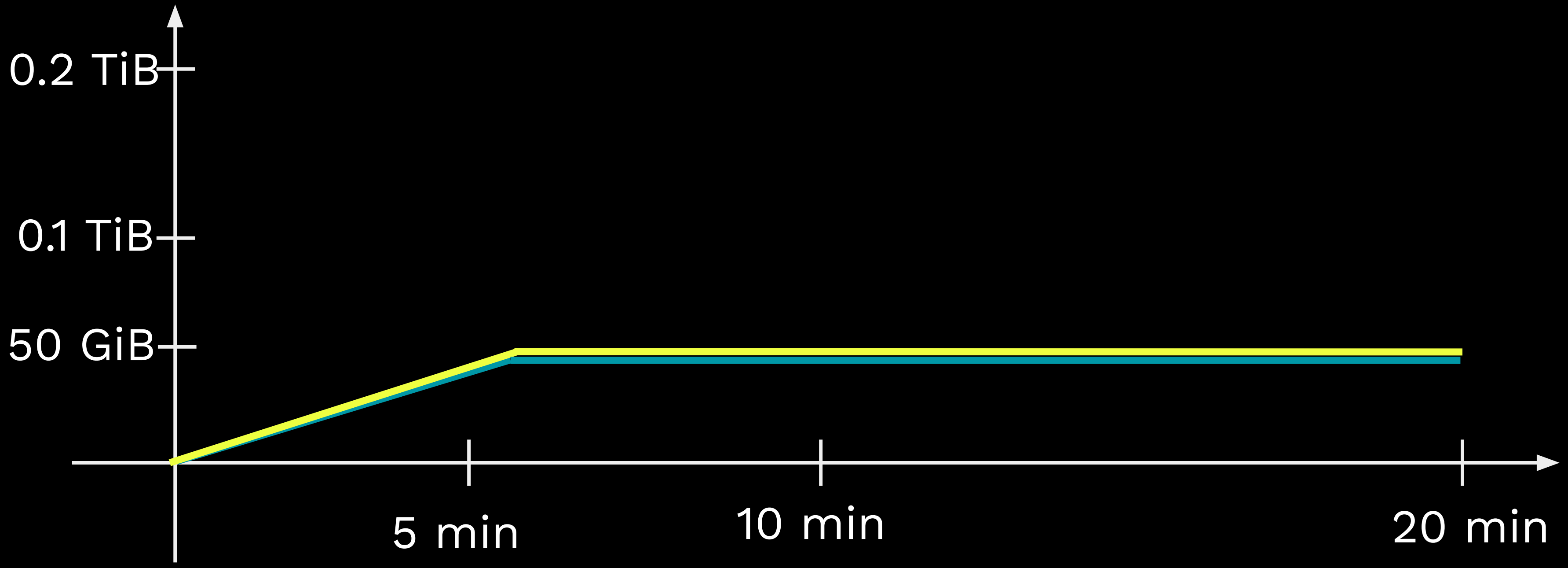
Many eyeballs looked at this code.

LGTM

```
func (root *tree) findBestFit(npages uintptr) *node {
	var best *tree
	t := root.tree
	for t != nil {
		if t.npages >= npages {
			best = t
			t = t.left
		} else {
			t = t.right
		}
	}
	return best
}
```

Now fixed.

Validating the implementation



Oh no...

Returning memory to the OS

When:

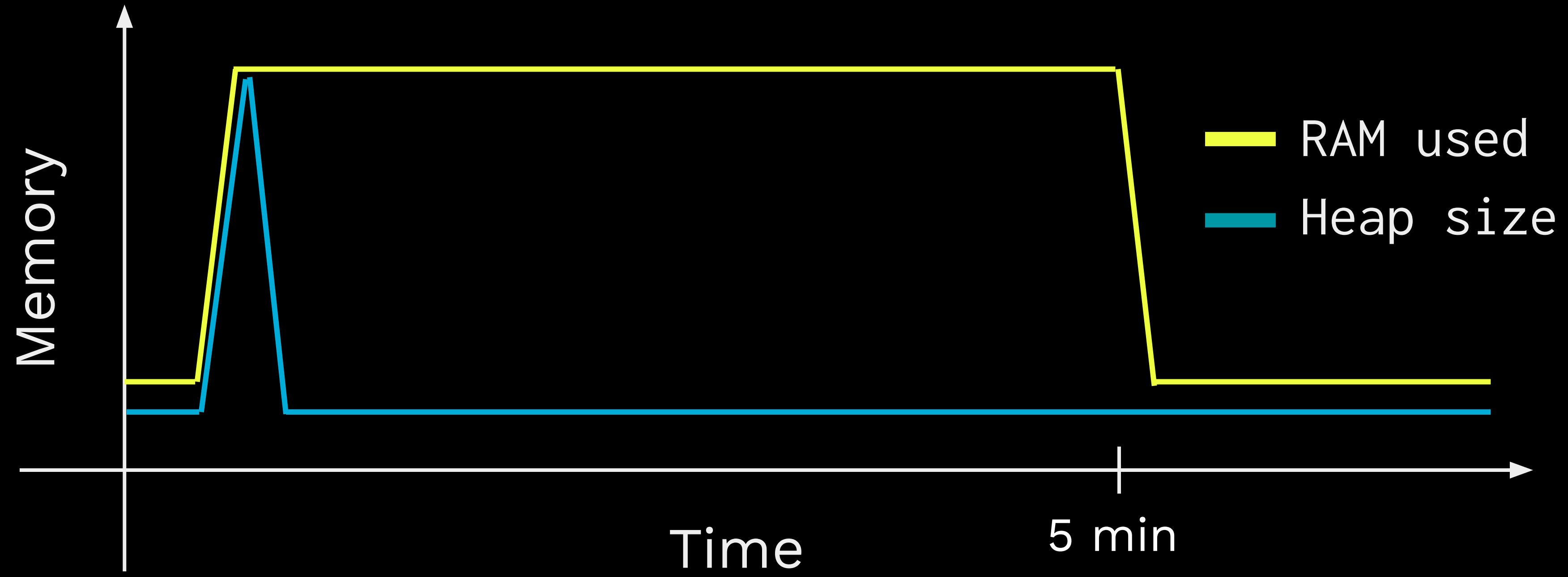
- Check whole heap every few minutes.
- When we ask the OS for more memory.

What:

- Pages free and unused for 5 minutes.
- Largest contiguous block of pages first.

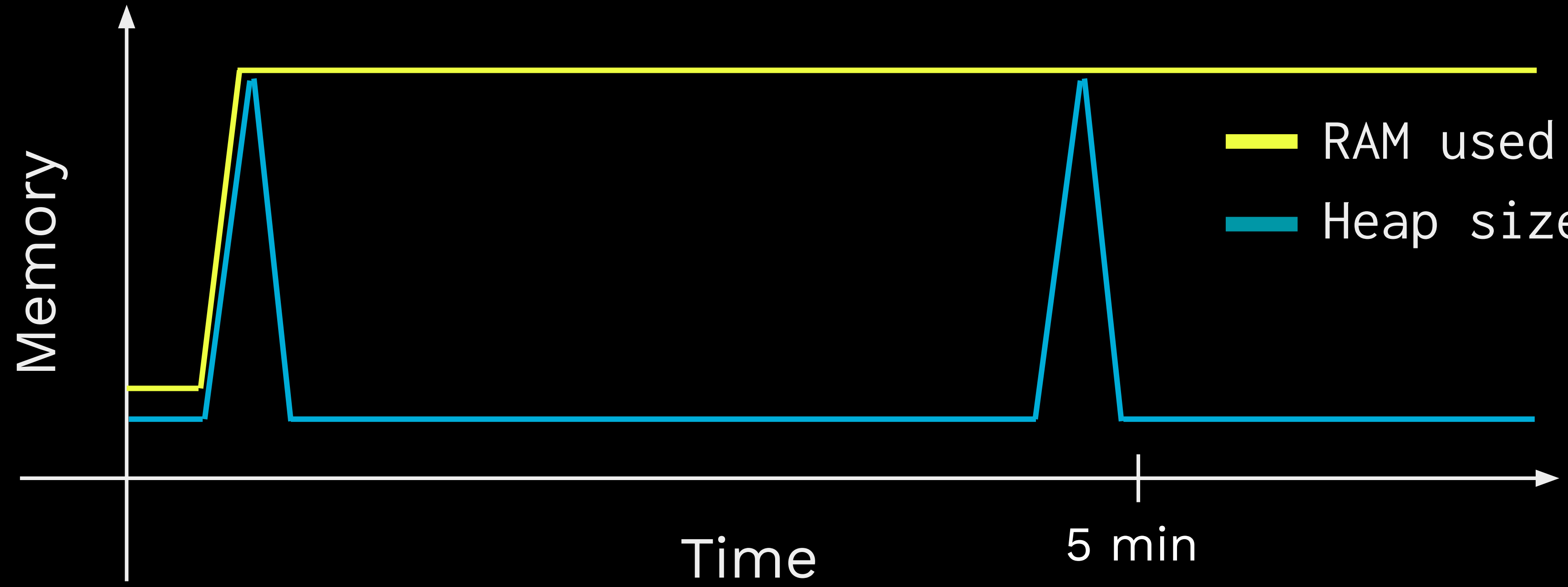
Which memory and when? (Go 1.12)

Scenario: Heap Spike



Very delayed effect...

Scenario: Heap Spikes



Memory might never get returned!

Returning memory to the OS

When:

- Check whole heap every few minutes.
- When we ask the OS for more memory.

What:

- Pages free and unused for 5 minutes.
- Largest contiguous block of pages first.

Which memory and when? (Go 1.12)

Returning memory to the OS

When:

~~- Check whole heap every few minutes.~~

- When we ask the OS for more memory.

What:

- Pages free and unused for 5 minutes.
- Largest contiguous block of pages first.

Which memory and when? (Go 1.12)

Returning memory to the OS

When:

- Gradually, at a fixed rate.
- When we ask the OS for more memory.

What:

- Pages free and unused for 5 minutes.
- Largest contiguous block of pages first.

Which memory and when? (Go 1.12)

Returning memory to the OS

When:

- Gradually, at a fixed rate.
- When we ask the OS for more memory.

What:

- ~~Pages free and unused for 5 minutes.~~
- ~~Largest contiguous block of pages first.~~

Which memory and when? (Go 1.13 and later)

Picking a policy

Allocation Policy	Return Policy
Best-fit	Largest first ?
Address-ordered First-fit	Highest address first

The right fit.

Returning memory to the OS

When:

- Gradually, at a fixed rate.
- When we ask the OS for more memory.

What:

- ~~Pages free and unused for 5 minutes.~~
- ~~Largest contiguous block of pages first.~~

Which memory and when? (Go 1.12)

Returning memory to the OS

When:

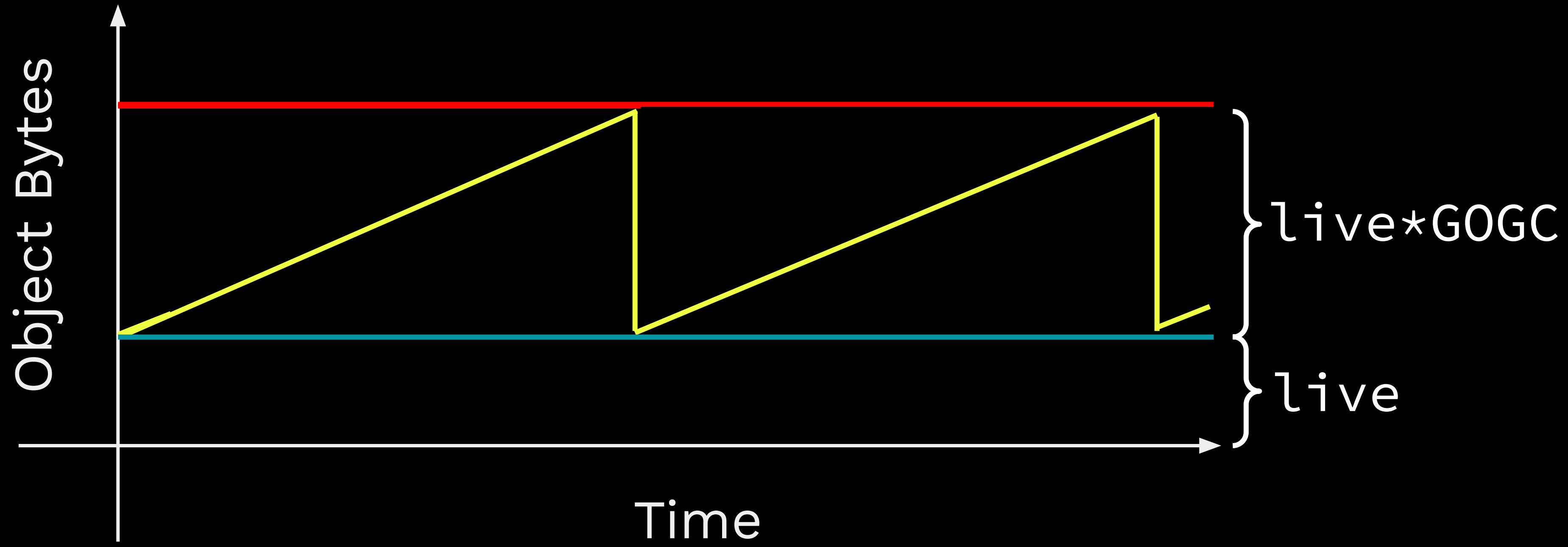
- Gradually, at a fixed rate.
- When we ask the OS for more memory.

What:

~~- Pages free and unused for 5 minutes.~~
- Highest address first.

Which memory and when? (Go 1.13 and later)

GOGC



GOGC controls Go GC.

Returning memory to the OS

When:

- Gradually, at a fixed rate.
- When we ask the OS for more memory.

What:

- ~~Pages free and unused for 5 minutes.~~
- Highest address first.

Which memory and when? (Go 1.13 and later)

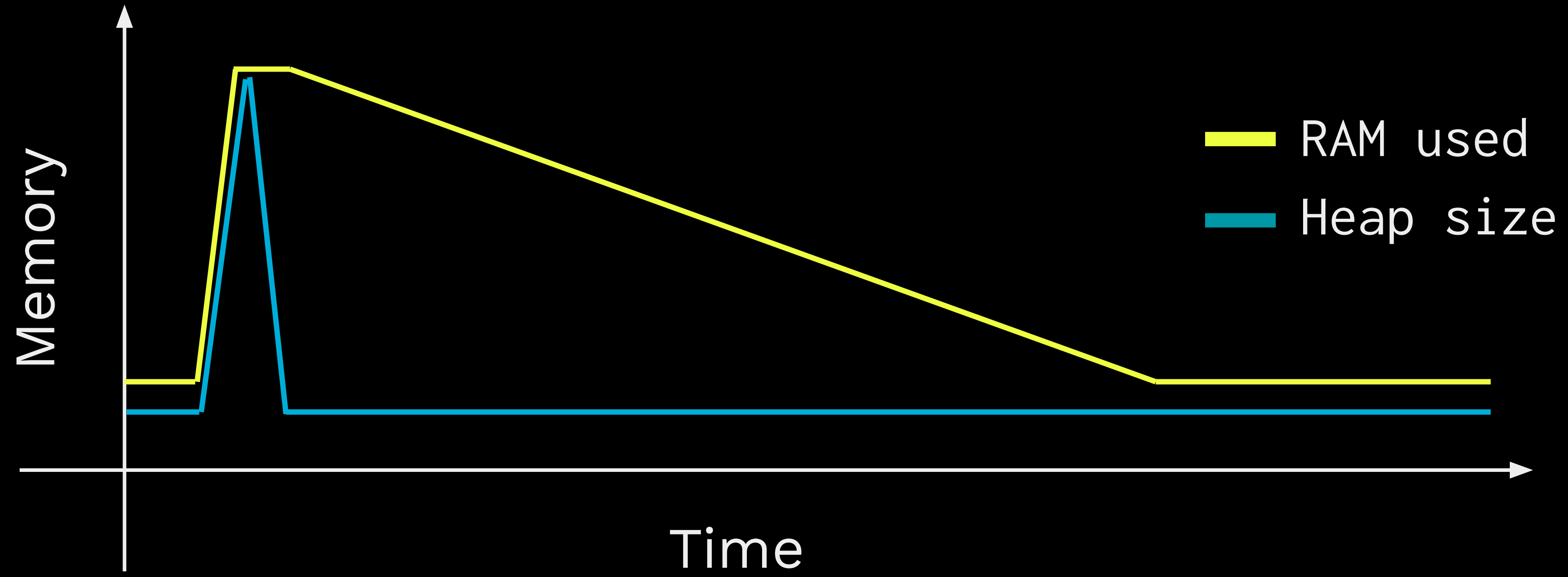
Returning memory to the OS

When:
- Gradually, at a fixed rate.
- When we ask the OS for more memory.

What:
- 10% more than the heap will grow to.
- Highest address first.

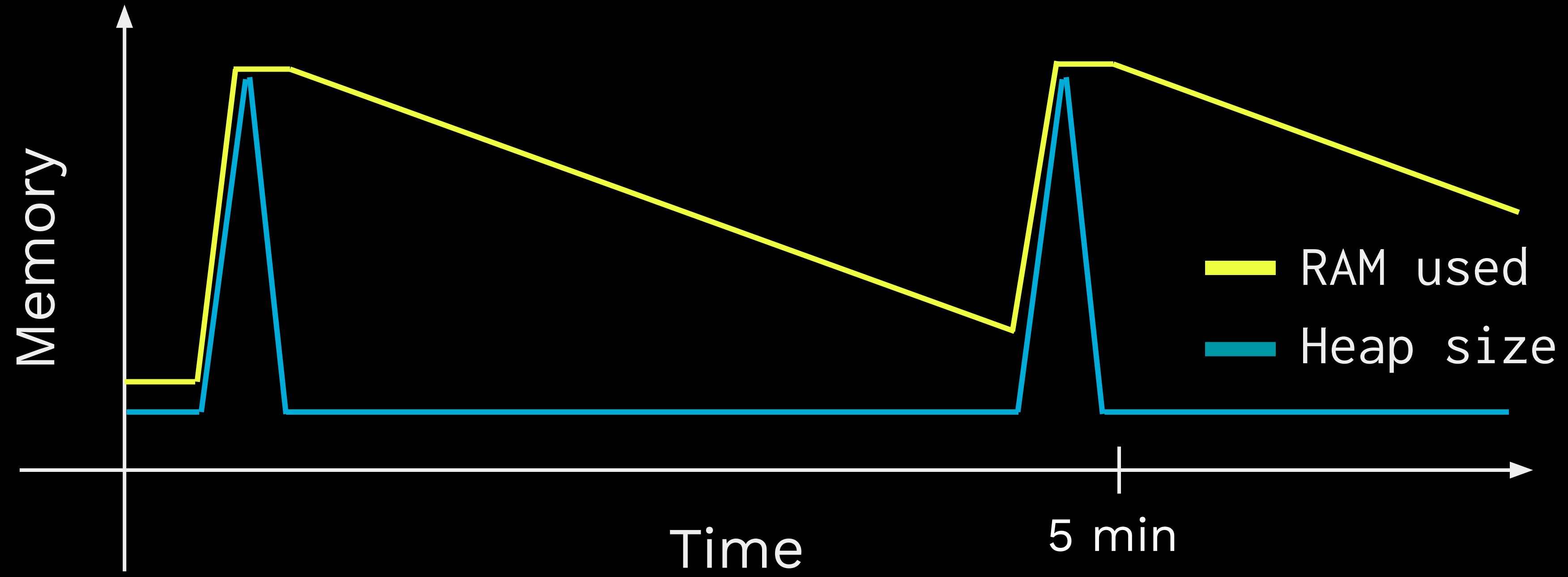
Which memory and when? (Go 1.13 and later)

Scenario: Heap Spike



Memory is gradually returned to the OS.

Scenario: Heap Spikes



Memory is gradually returned to the OS.

"

The bugs we made along the way.

3x increase in k8s API tail latency...

...this happened *twice*...

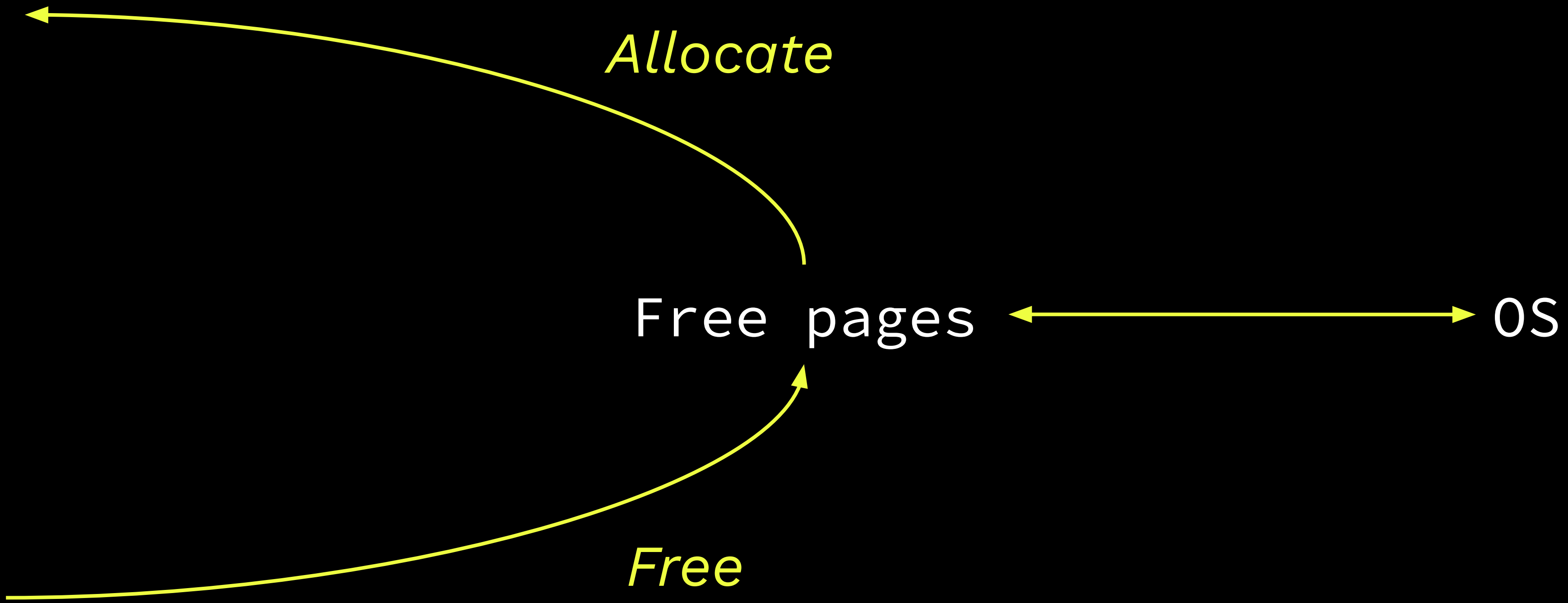
...and then also to gRPC.

Turns out there's a bottleneck!

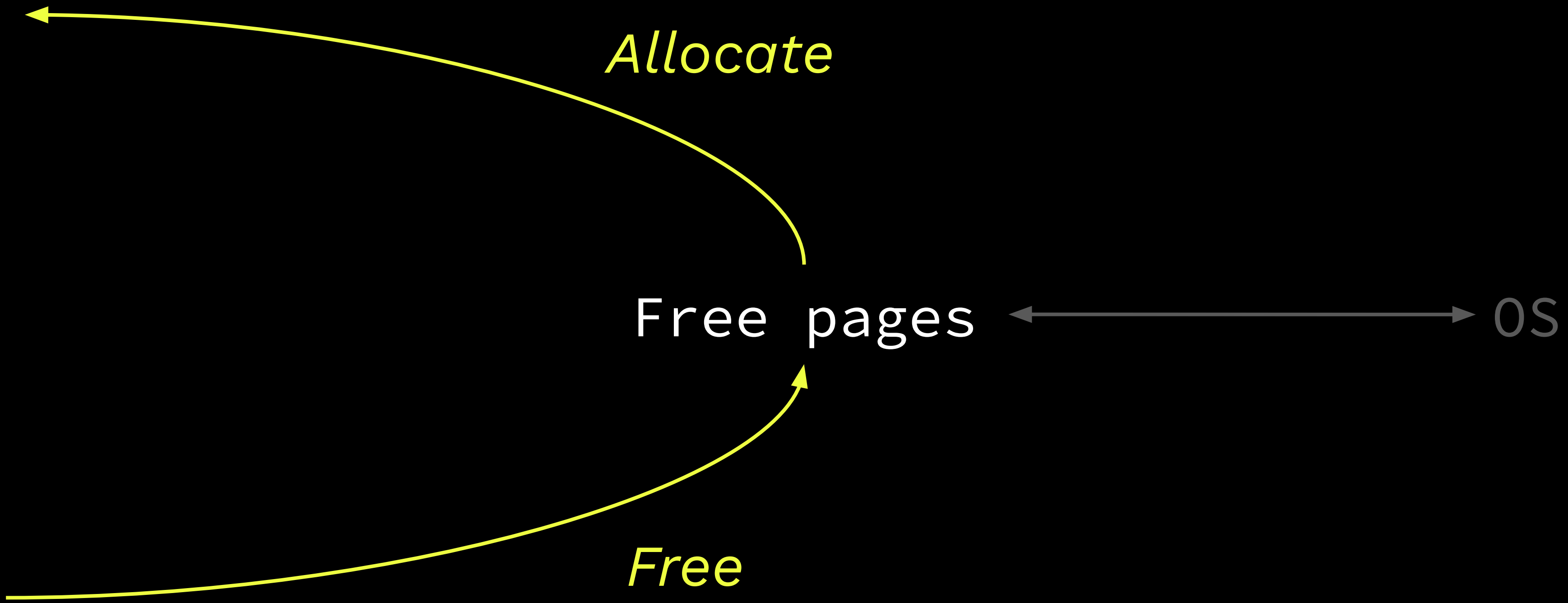
Part 3

Managing pages

The Heap Memory "Lifecycle"



The Heap Memory "Lifecycle"



“

What went wrong?

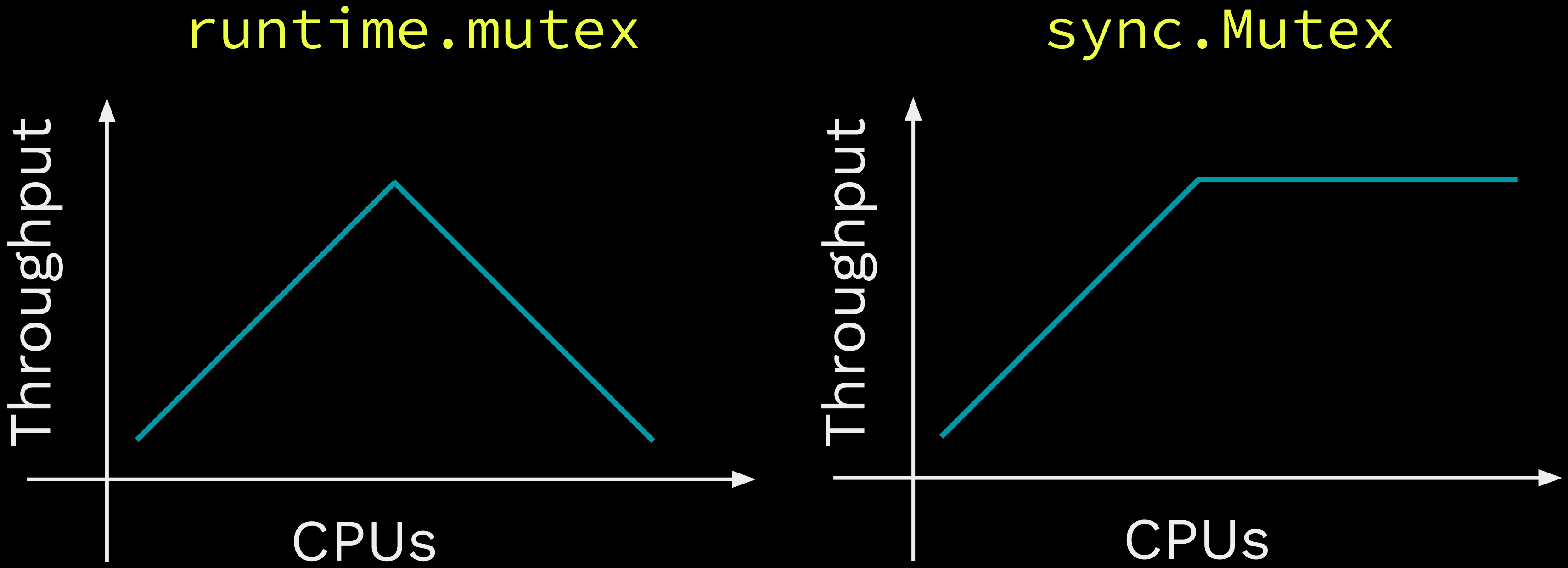
Remember the tree?

...it's protected by a lock...

...and that lock was collapsing...

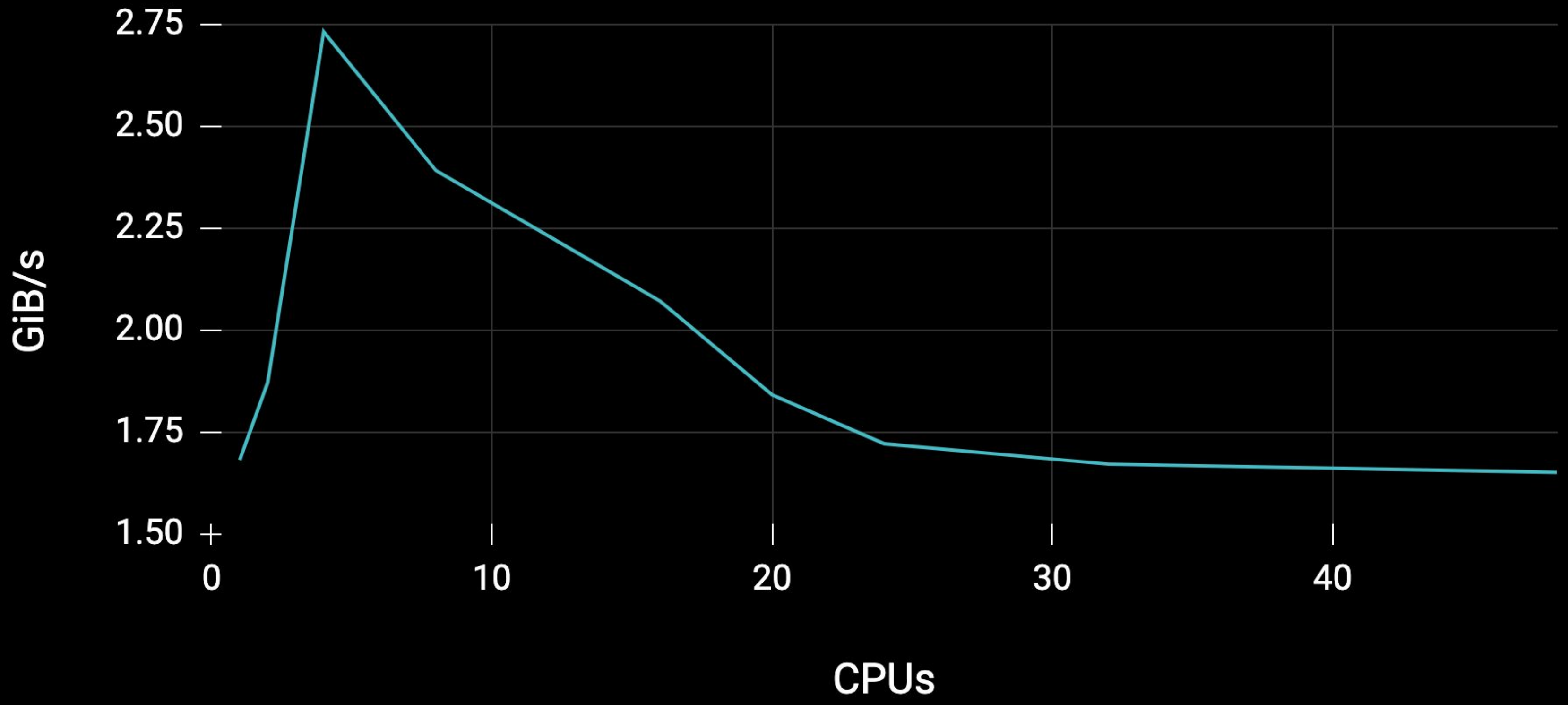
...since **before Go 1.11**.

Lock collapse



More parallelism means worse throughput and latency.

1 KiB Allocation Throughput (Go 1.13)



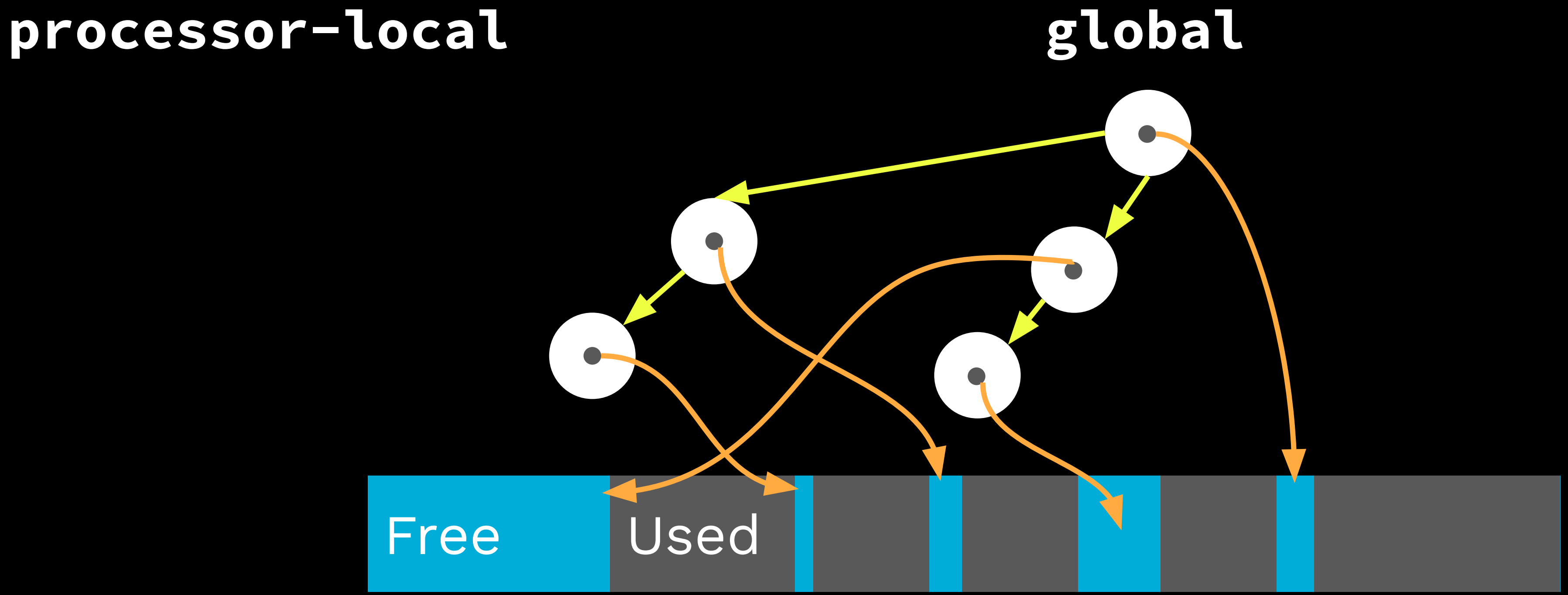
Locking less by caching.

Observation: most allocations are small.

Idea: processor-local page cache.

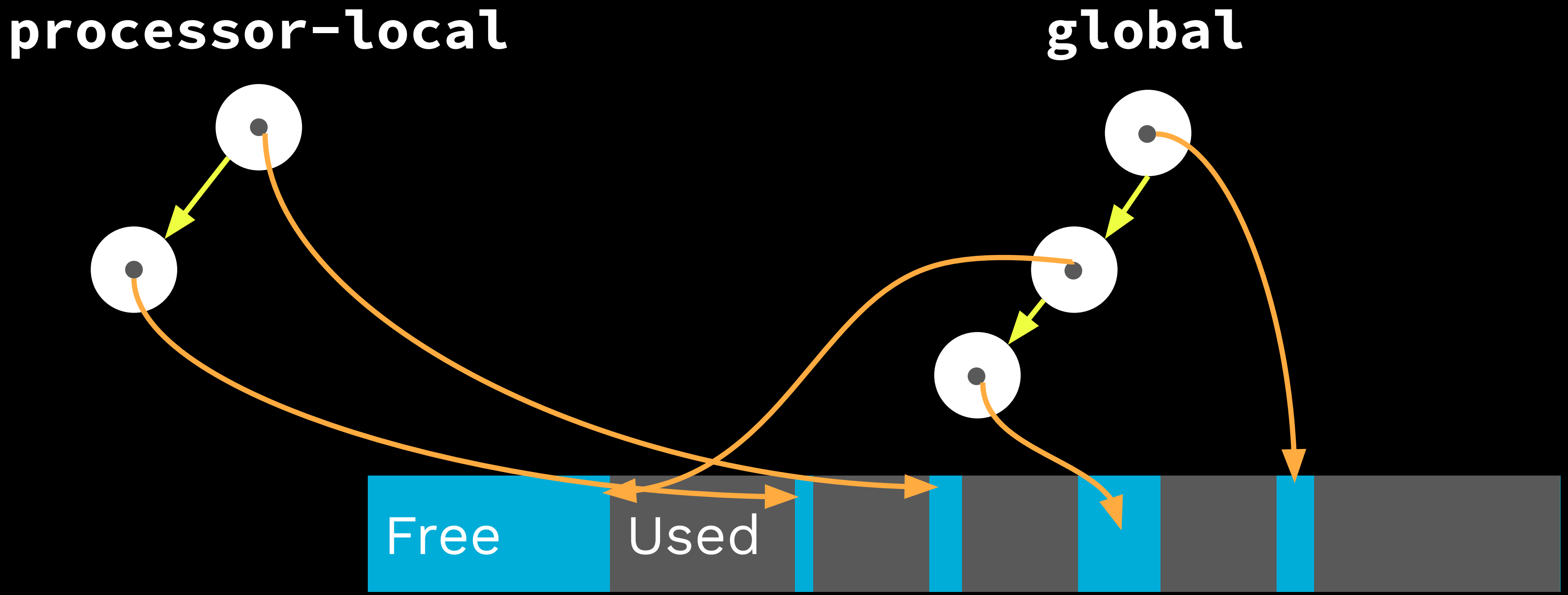
Grab the lock once and cache >1 page.

Trying with a tree



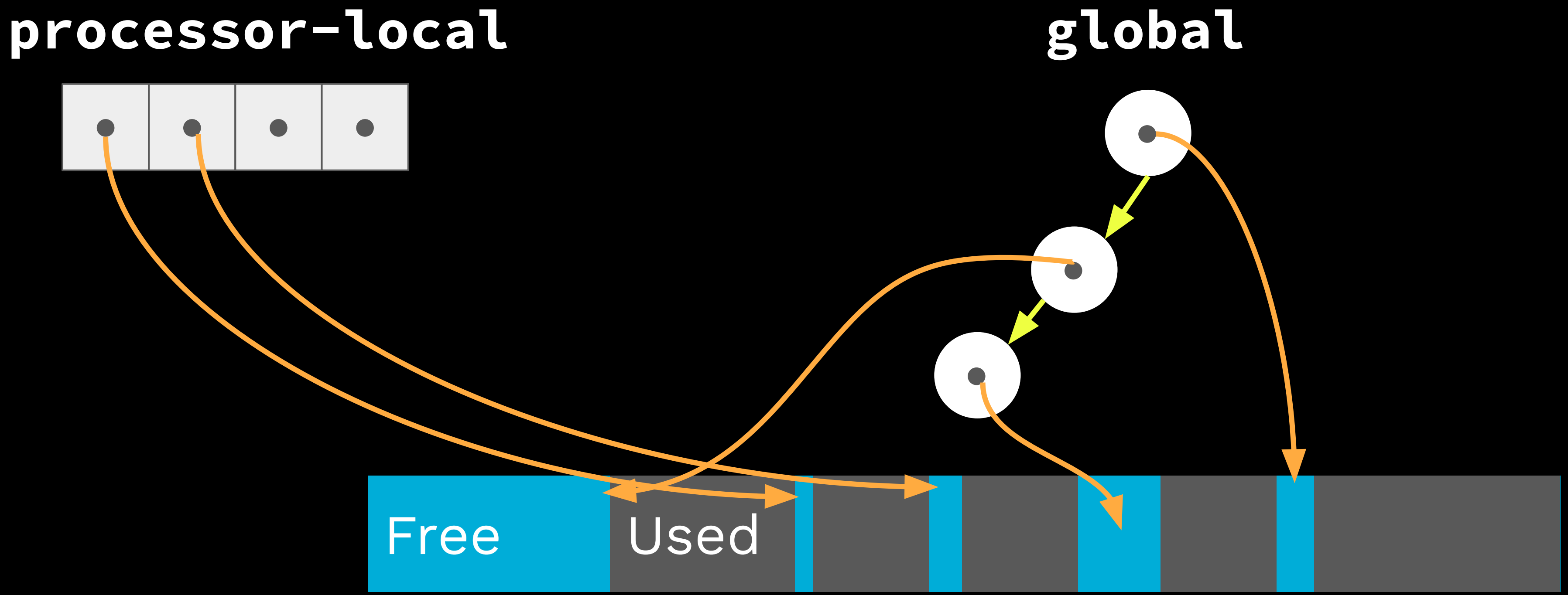
It works, but...

Trying with a tree



It works, but...

Trying with a tree



It works, but...

“

Thinking bigger.

Observations: CPUs don't like trees...

...but CPUs *do* like bitmaps.

Idea: use a bitmap.

Bitmaps!

Operations:

0 = free
1 = in-use

0000 0000 0000 0000

The main idea.

Bitmaps!

0 = free
1 = in-use

```
0000 0000 0000 0000
```

Operations:
- Alloc 1 page

The main idea.

Bitmaps!

0 = free
1 = in-use

```
1000 0000 0000 0000
```

Operations:

- Alloc 1 page

The main idea.

Bitmaps!

0 = free
1 = in-use

```
1000 0000 0000 0000
```

Operations:

- Alloc 1 page
- Alloc 10 pages

The main idea.

Bitmaps!

```
0 = free
1 = in-use
```

```
1111 1111 1110 0000
```

```
Operations:
 -  Alloc 1 page
 -  Alloc 10 pages
```

The main idea.

Bitmaps!

0 = free
1 = in-use

1111 1111 1110 0000

Operations:

- Alloc 1 page
- Alloc 10 pages
- Free 1 page

The main idea.

Bitmaps!

0 = free
1 = in-use

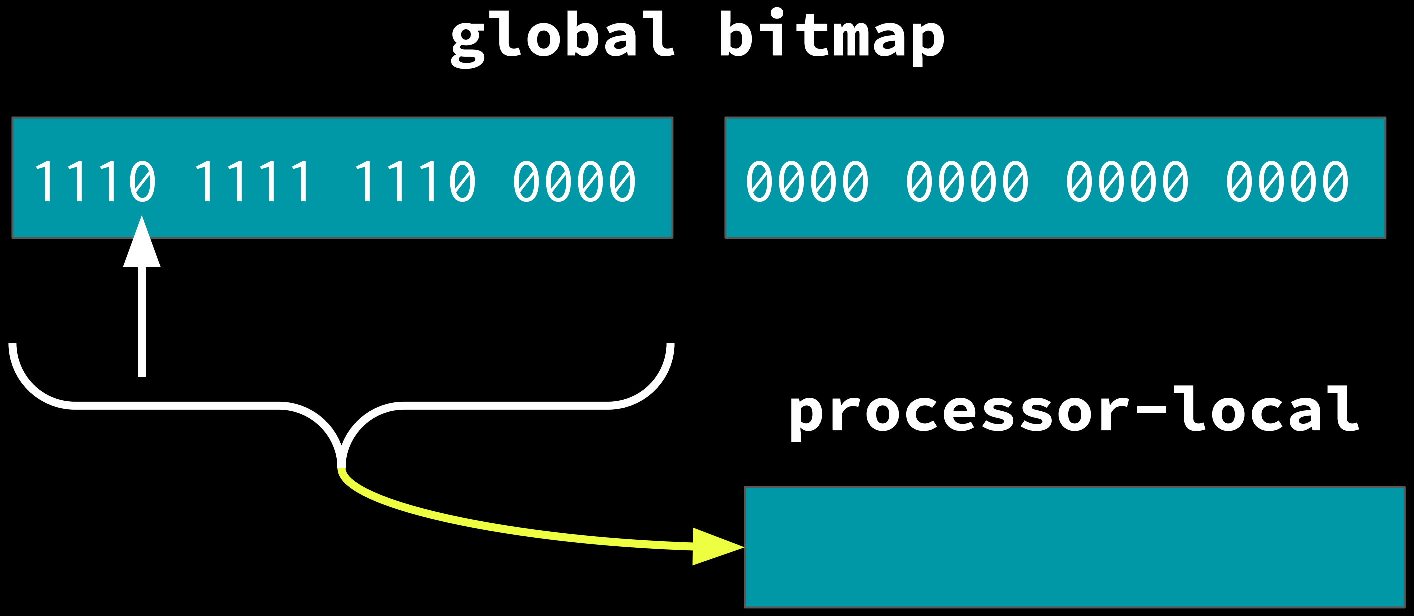
1110 1111 1110 0000

Operations:

- Alloc 1 page
- Alloc 10 pages
- Free 1 page

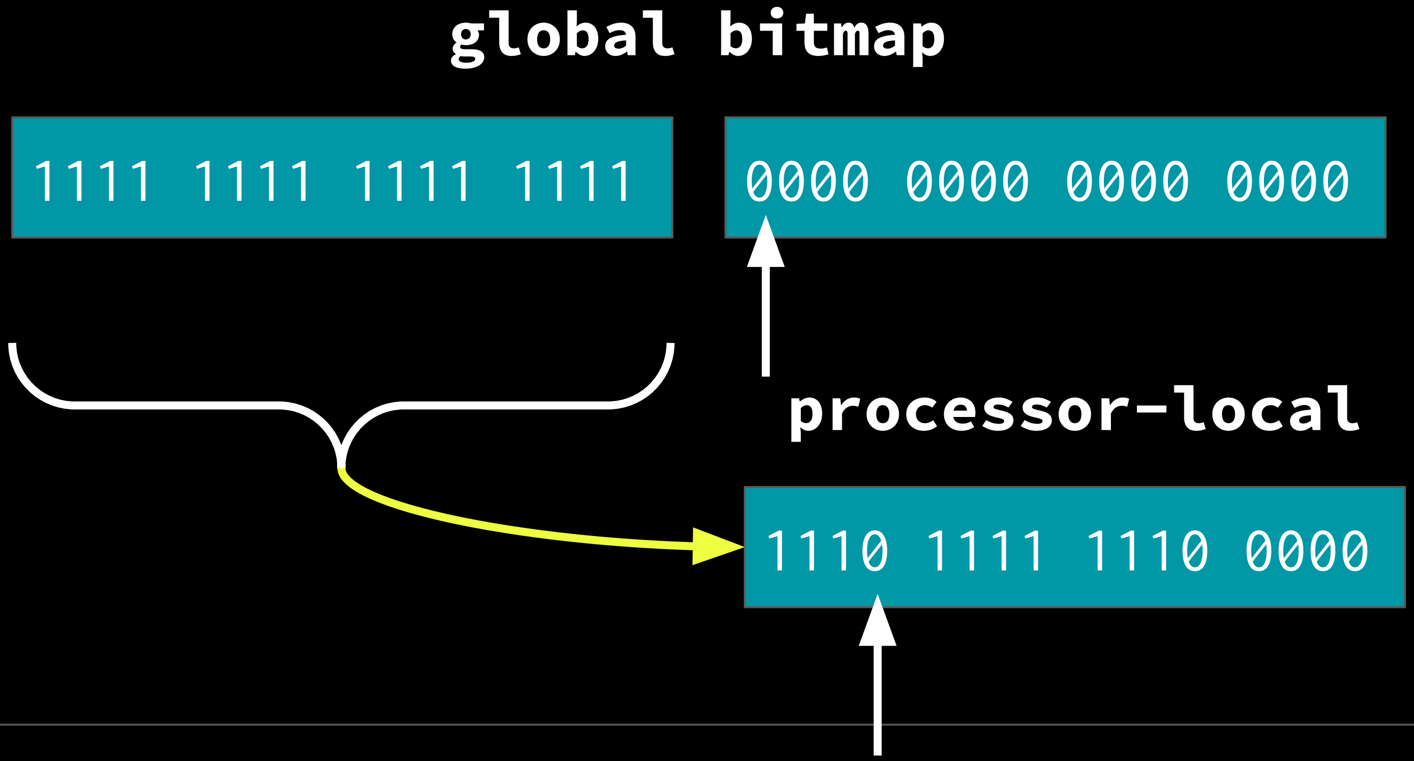
The main idea.

Caching pages



The main idea.

Caching pages



The main idea.

Thinking even bigger.

Bitmap scales poorly for large allocations.

Can we do better?

Summarizing the bitmap

start:0 max:5 end:3

start:0 max:2 end:1

start:4 max:4 end:3

1110 1111 1110 0110

0000 1111 1111 1000

Summarily...

Summarizing the bitmap

start:0 max:5 end:3

start:0 max:2 1 | start:4 max end:3

110 11 1110 0110 | 0000 1111 1111 1000

OBJECTION!

Summarily...

“

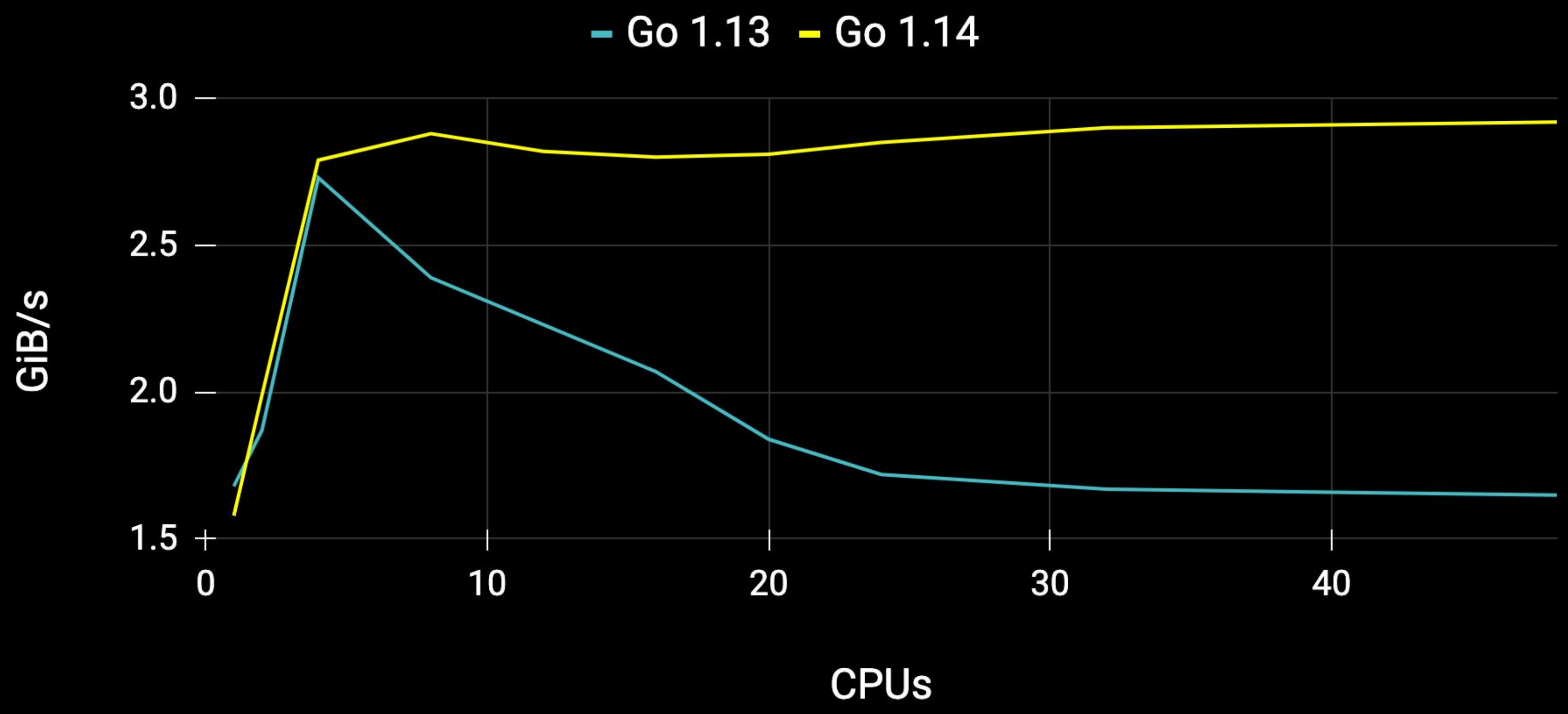
Wow! So fast!

Real-world server benchmarks:

+30% throughput

-20% 99th %ile latency

1 KiB Allocation Throughput



Too fast...?

Faster allocation ⇒ more GC pressure

⇒ ... ⇒ ... ⇒ ... ⇒ ... ⇒ ... ⇒ ... ⇒ ... ⇒ ...

⇒ More memory used (**unbounded**)

Fix: Limit the GC's feedback loop.

Part 4

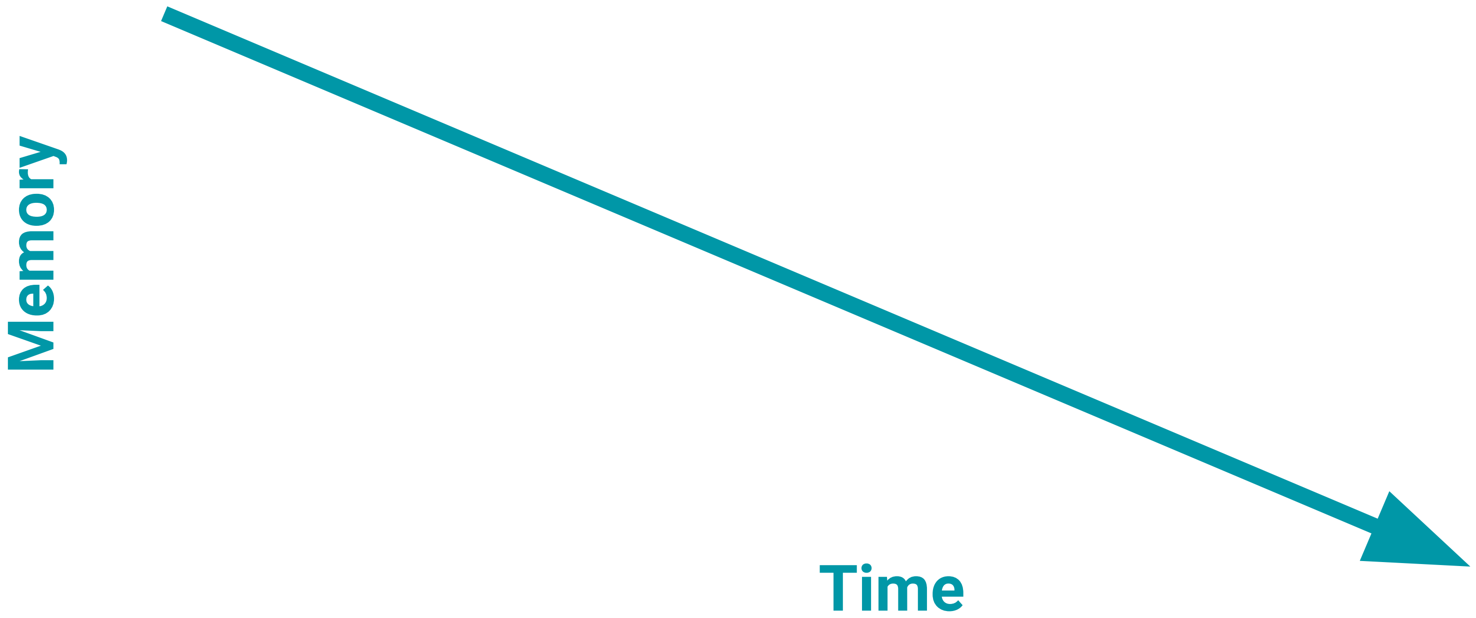
Conclusion

Takeaways

Memory management is like an onion.

Takeaways

As of Go 1.13:



“

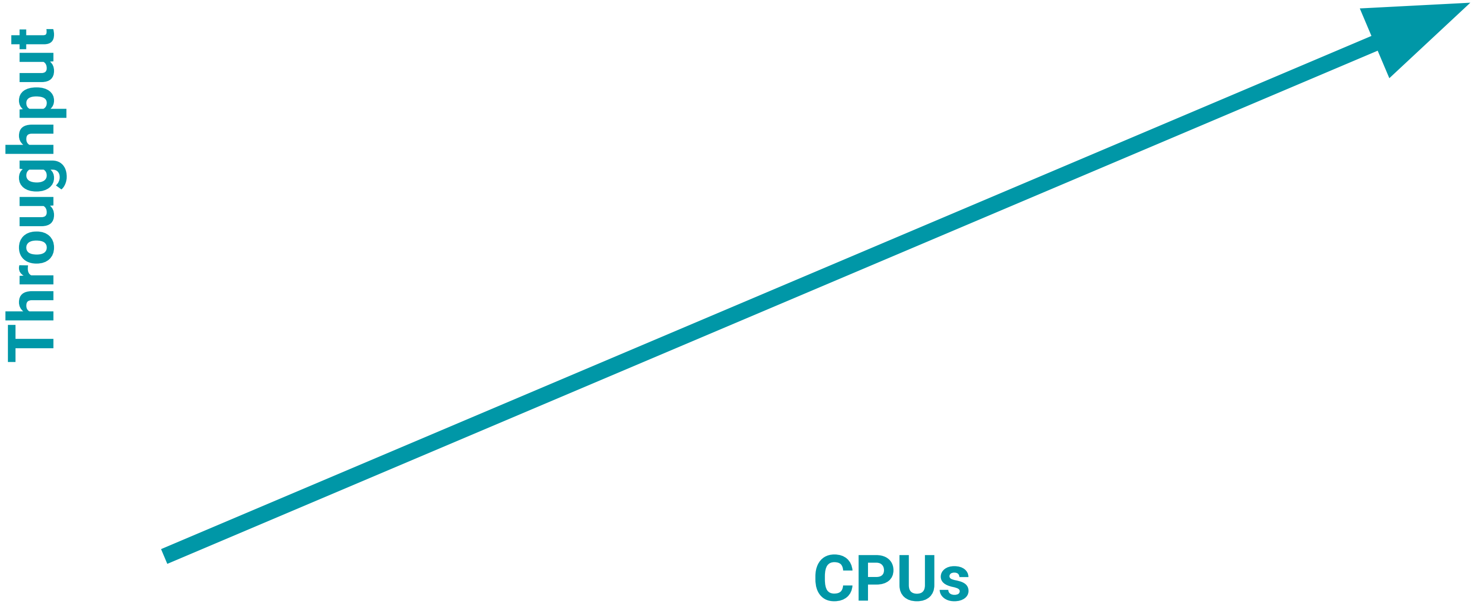
Summary

The runtime **gradually** returns the **highest-addressed** free memory to the OS, leaving **headroom** for the application.

Less memory on average, more robust.

Takeaways

As of Go 1.14:



Summary

The runtime manages free pages in a **summarized bitmap**, allowing it to easily **cache** pages for **lockless** access.

Scales better to many cores.

Takeaways

More people should talk about this stuff!

No two allocators return memory the same way.

Most allocators manage pages the same way.

That's the end.

Thank you!