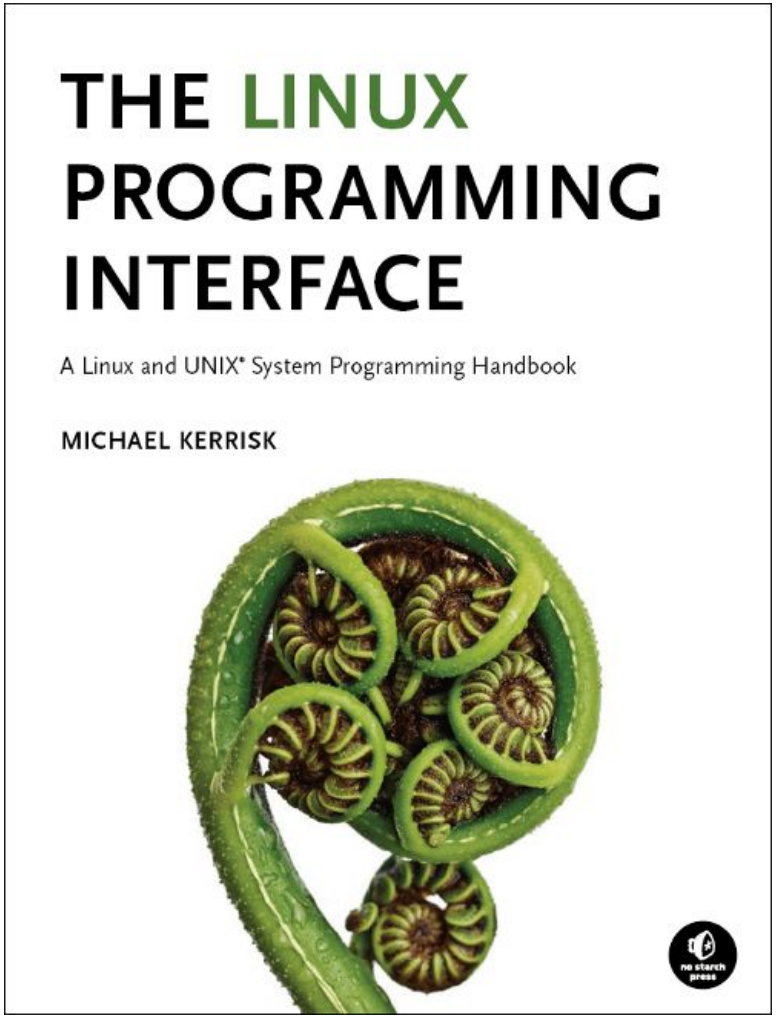
Understanding Go Memory Allocation

André Carvalho
@andresantostc

Developer @ globo.com

tsuru

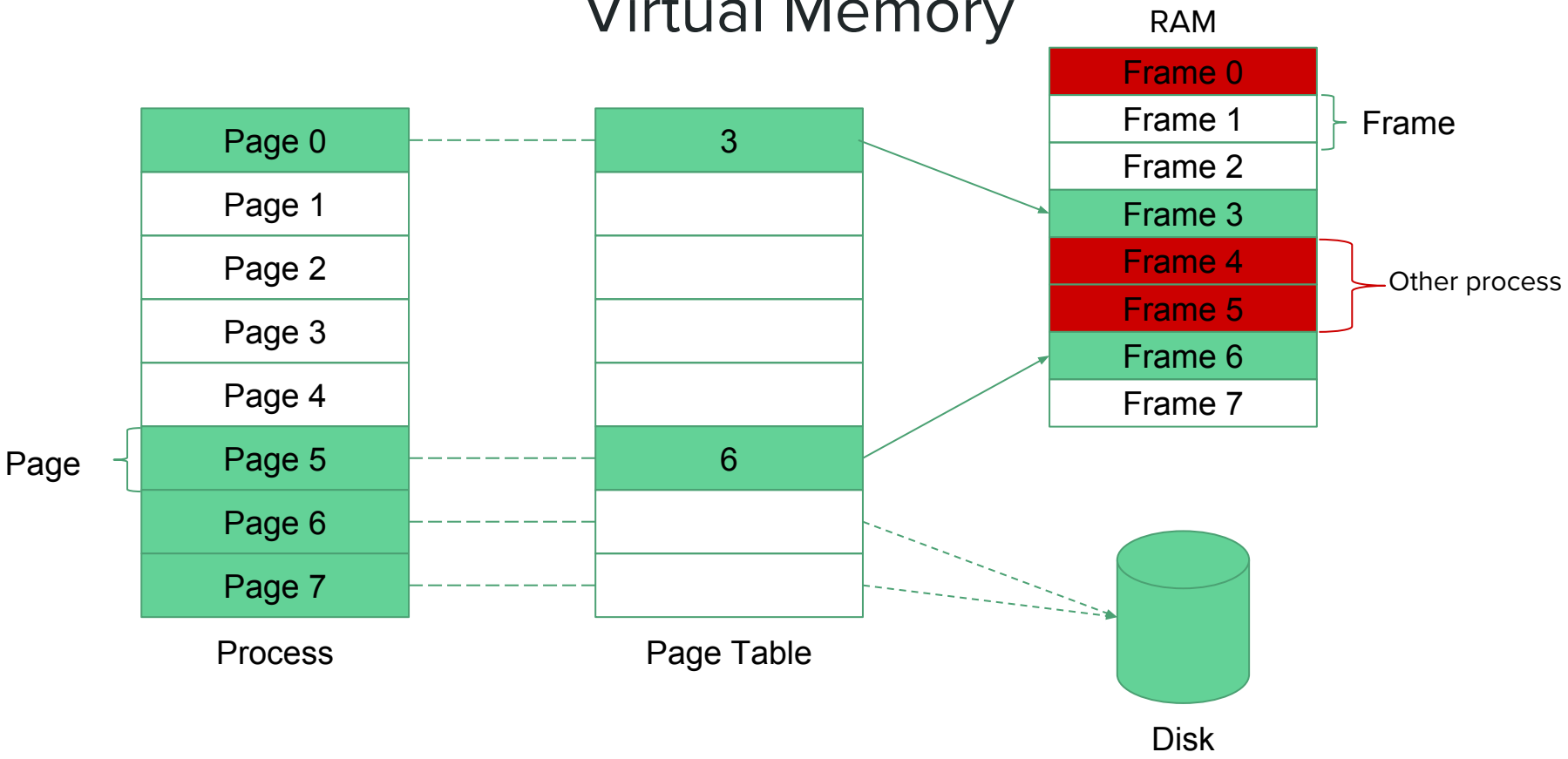
andrestc.com


THE LINUX
PROGRAMMING
INTERFACE
A Linux and UNIX® System Programming Handbook
MICHAEL KERRISK
no starch press

Virtual Memory

- Processes do not read directly from physical memory
  - Security
  - Coordination between multiple processes
- Virtual Memory abstracts that away from the processes
  - Segmentation
  - **Page tables**

Virtual Memory

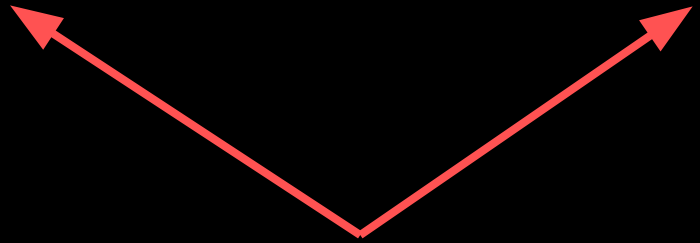


```go
func main() {
    rand.Seed(time.Now().UnixNano())
    i := rand.Intn(100)
    fmt.Printf("%v at %p\n", i, &i)
    for {
    }
}
```

Running two instances at the same time...

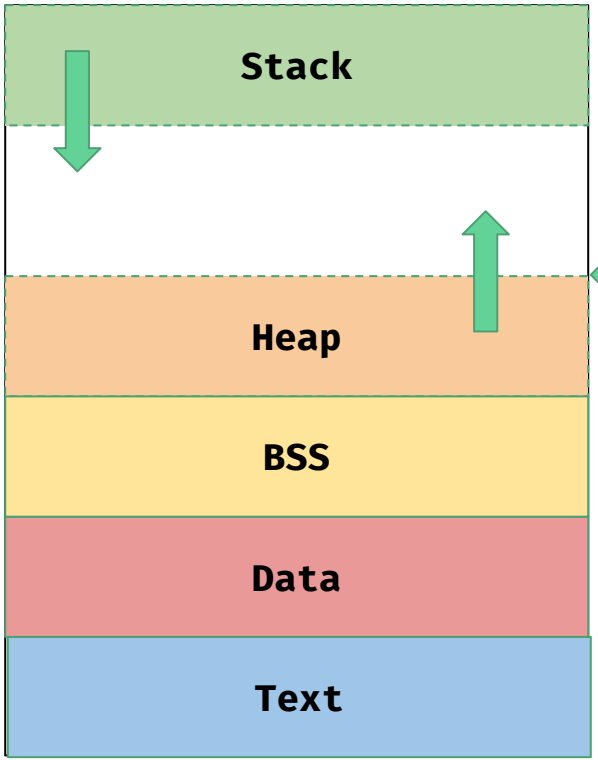
$ ./vmemory
53 at **0xc420016110**

$ ./vmemory
68 at **0xc420016110**



Same virtual address

Process Memory Layout



Function stack frames

Program Break

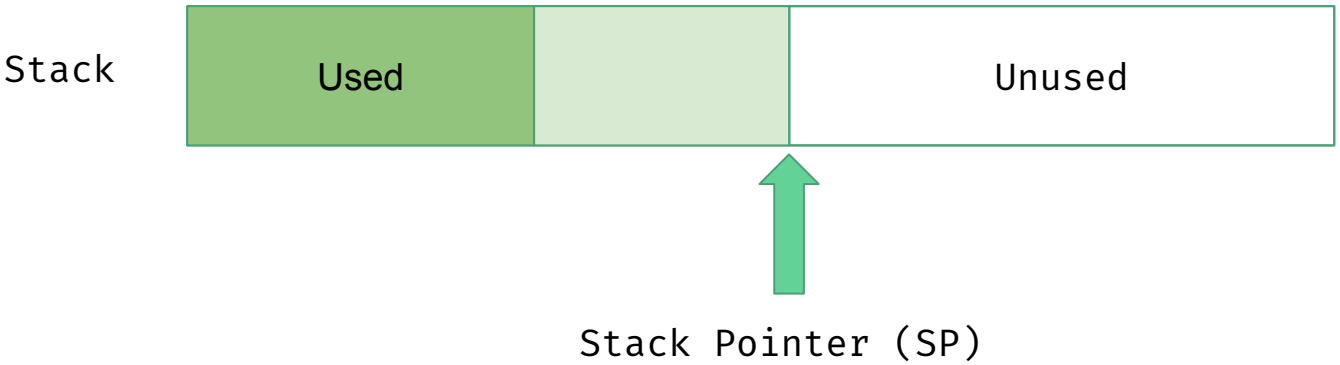
Dynamic allocated variables

Uninitialized static variables

Initialized static variables

Code

Stack Allocation



Allocation

```
SP += size;
return Stack[SP-size];
```

Deallocation

```
SP -= size;
```

Heap Allocation

- For objects with size only known at **runtime**
- C provides **`malloc`** and **`free`**
- C++ provides **`new`** and **`delete`**
- Go uses **escape analysis** and has **garbage collection**

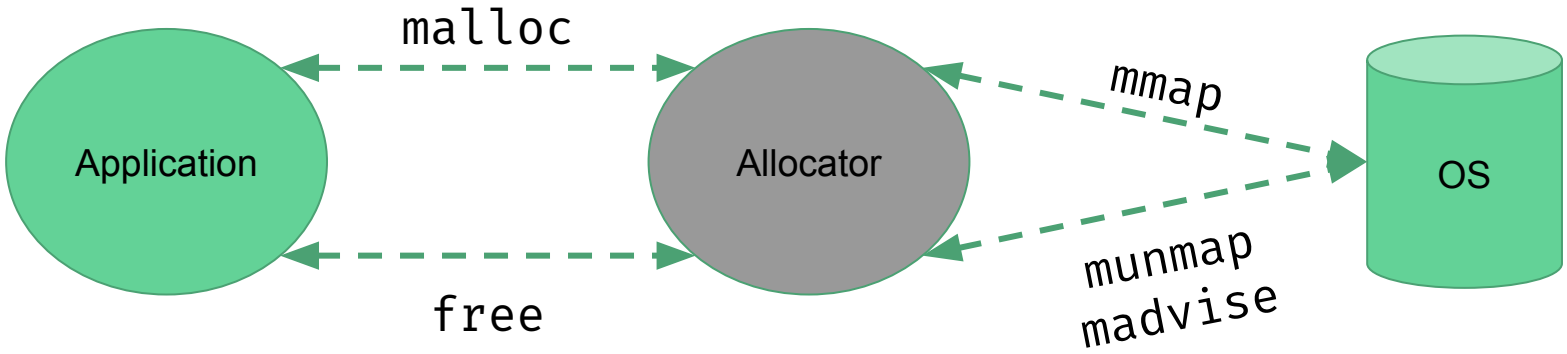
Minimal Allocator

Minimal Allocator

We need to implement two functions

```
void* malloc(size_t size)

void free(void *ptr)
```

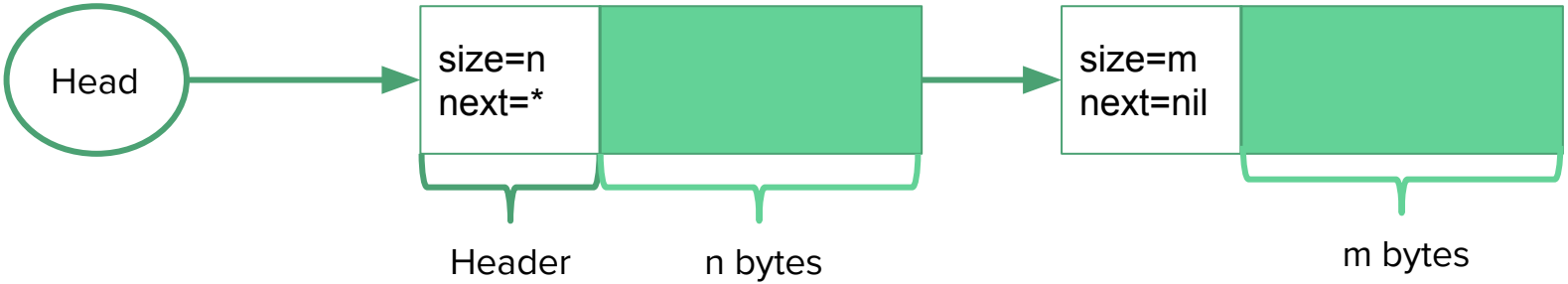
Minimal Allocator



Allocator uses syscalls like mmap/munmap to allocate/deallocate

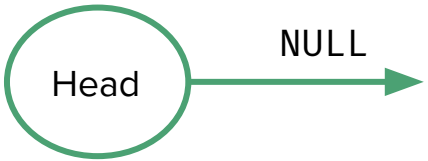
Minimal Allocator

Linked list with free objects



Minimal Allocator - Allocating

```
malloc(10)
```



Minimal Allocator - Allocating

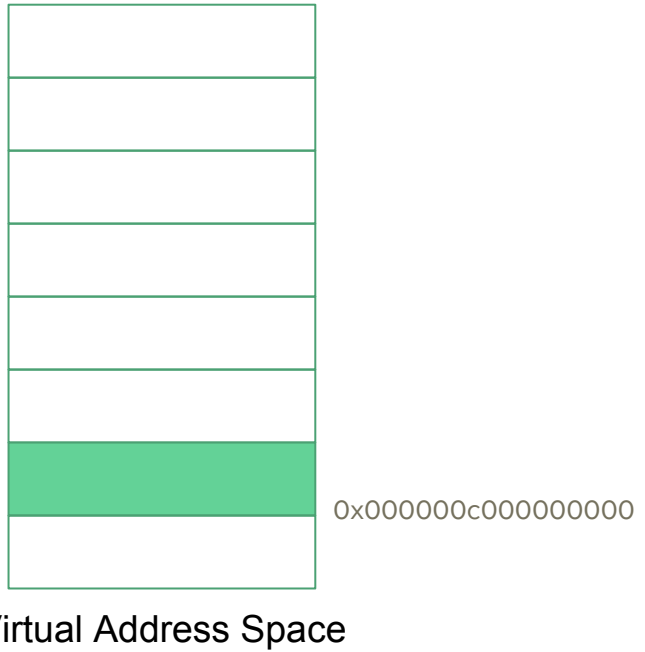
```
mmap(
    0x000000c000000000,
    4096,
    PROT_WRITE | PROT_READ,
    MAP_PRIVATE | MAP_ANONYMOUS,
...)
```

Start Address

Size

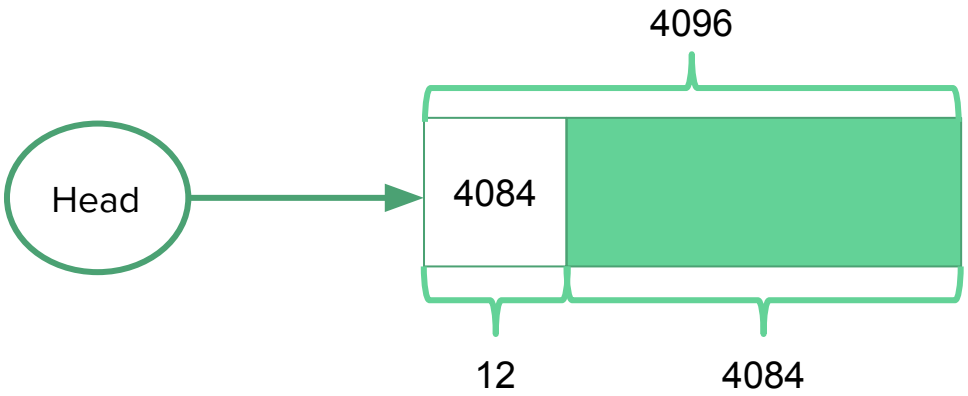
Permission

Flags



Minimal Allocator - Allocating

`malloc(10)`



Minimal Allocator - Allocating

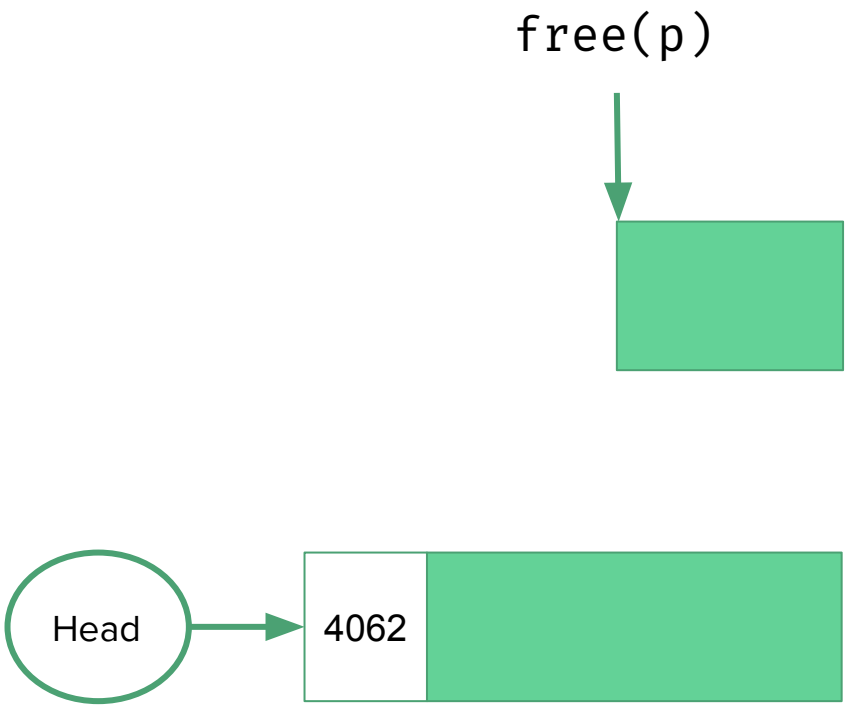
`malloc(10)`

Minimal Allocator - Allocating

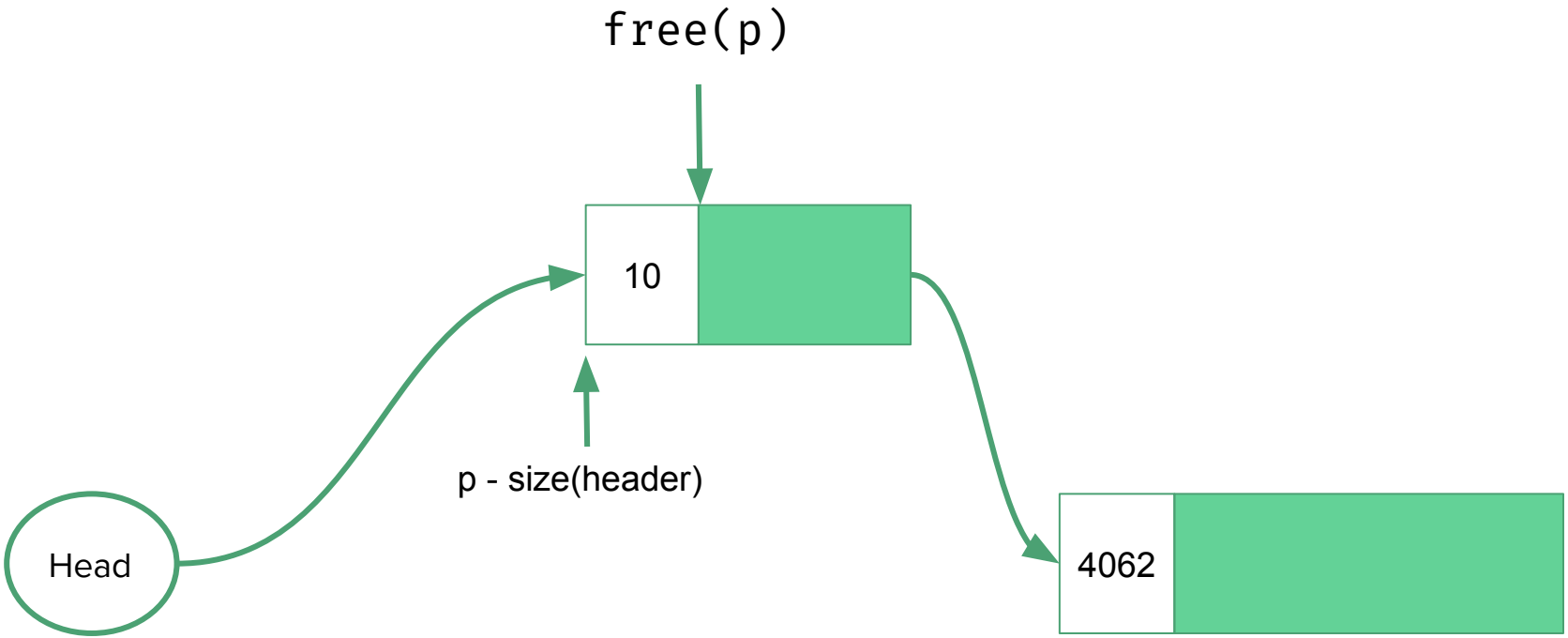
```
malloc(10)
```

Allocator returns p, which points right after the header

Minimal Allocator - Deallocating



Minimal Allocator - Deallocating



Minimal Allocator

- Can be implemented in a few hundred LOCs
- Issues
  - Fragmentation
  - Corruption
  - Releasing memory back to OS
    - When?
    - How? `munmap`, `madvise`...
  - Multi-thread
  - ...

Go Runtime Allocator

- TCMalloc
- Invoking the Allocator
- Go’s Allocator

Thread-Caching Malloc (TCMalloc)

- Originally implemented for the C language by Google
- Served as basis for Go’s runtime allocator
- Reduces lock contention for multithreaded programs

TCMalloc

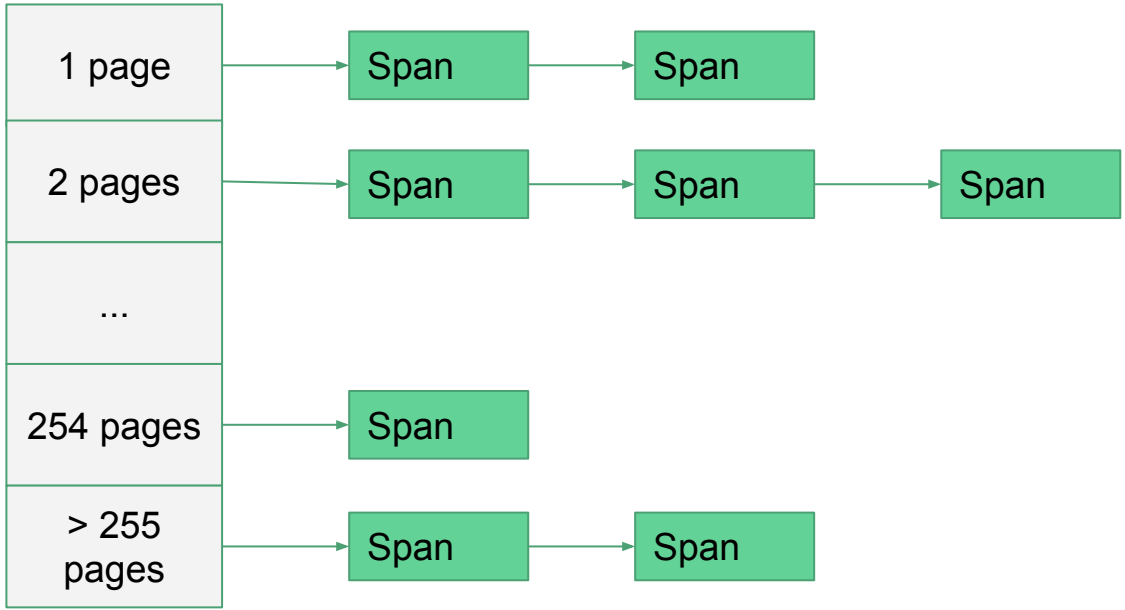
- Each **thread** has a local **cache**
- Two types of allocations
  - **Small** allocations (<= 32 kB)
  - **Large** allocations
- Manages memory in units called **Spans**
  - Runs of **contiguous** memory pages
  - Metadata is **kept separated from the allocation arena**

TCMalloc - Large Allocations

- Served by the central heap
- Requested size is rounded up to number of pages (4kB)

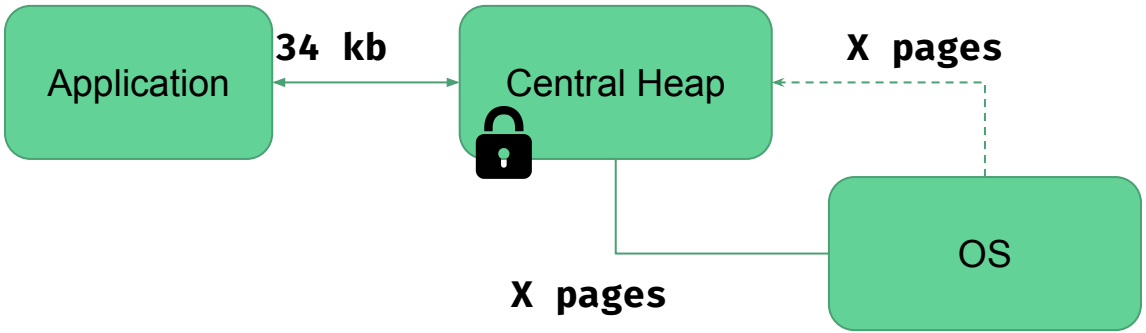
```
malloc(34 kB) ⇒ malloc(36 kB) ⇒ 9 pages

malloc(33 kB) ⇒ malloc(36 kB) ⇒ 9 pages
```

TCMalloc - Large Allocations



Central Heap

TCMalloc - Large Allocations

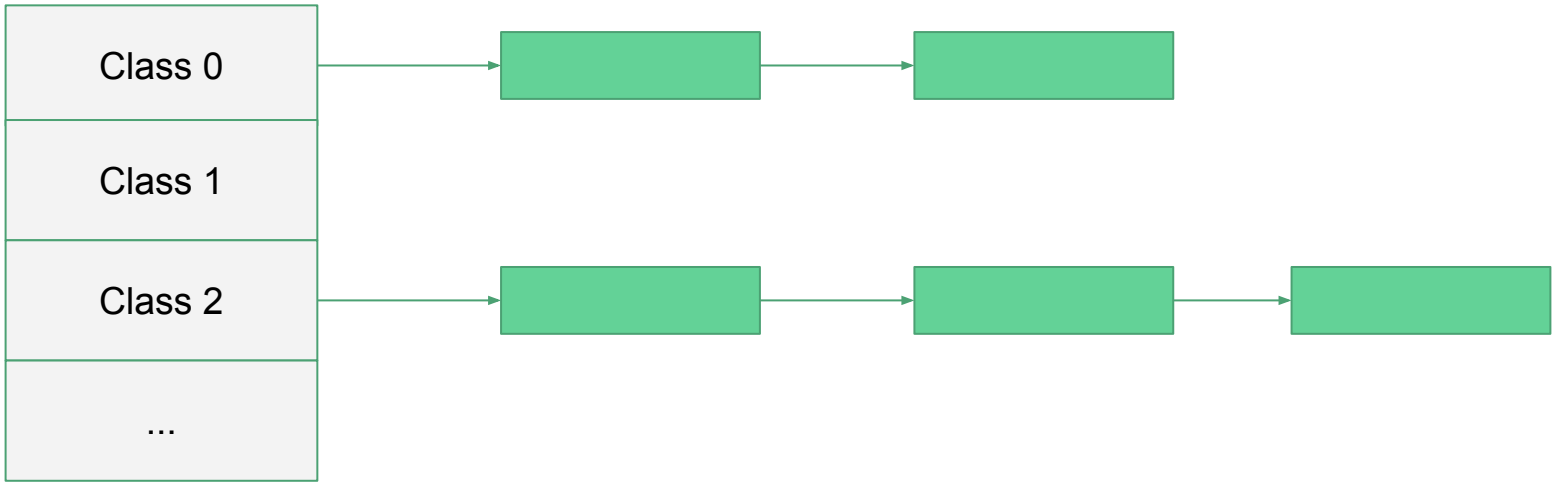


TCMalloc - Small Allocations

- Served by the local thread cache
- Requested size is rounded up to one of the size classes

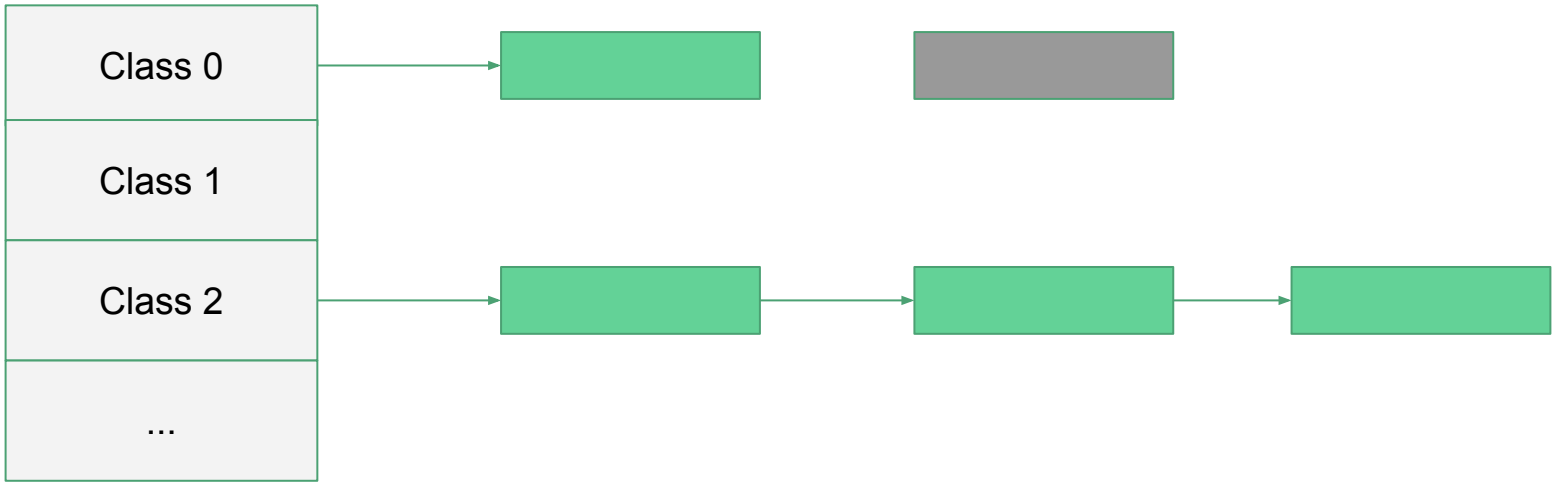
```
malloc(4 bytes)  ⇒ malloc(8 bytes)
malloc(6 bytes) ⇒ malloc(8 bytes)
```

TCMalloc - Small Allocations



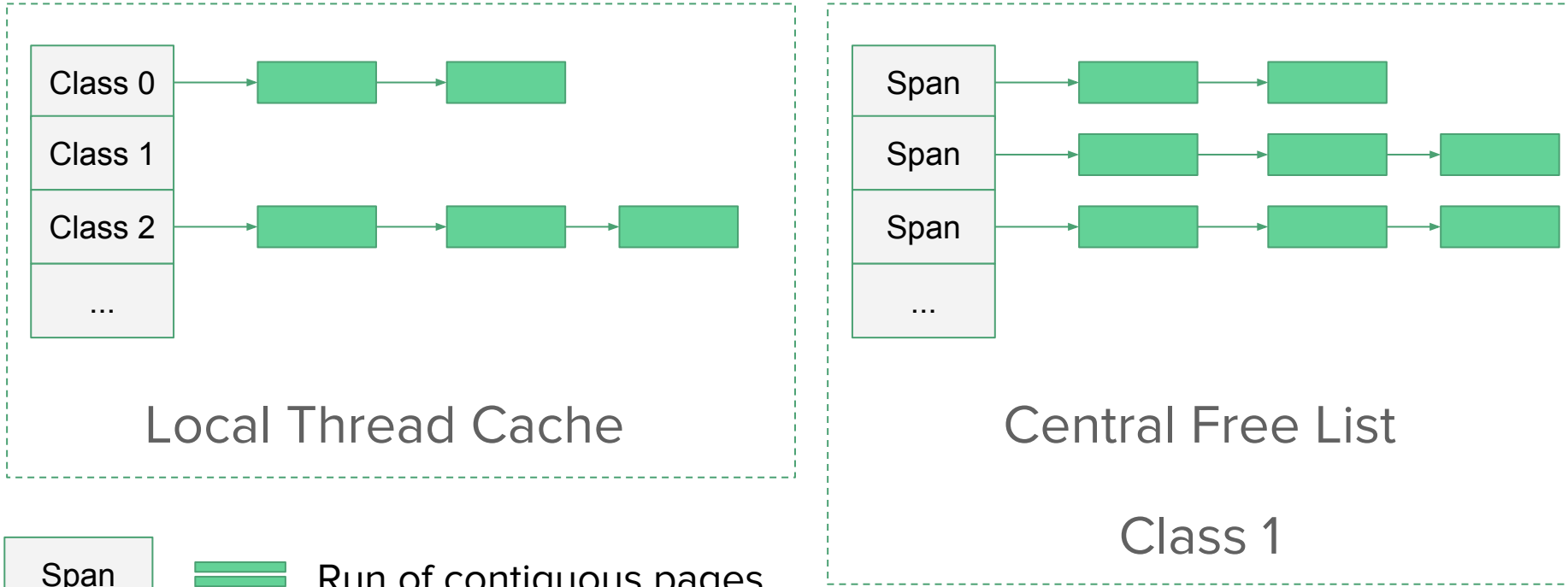
Local Thread Cache

TCMalloc - Small Allocations

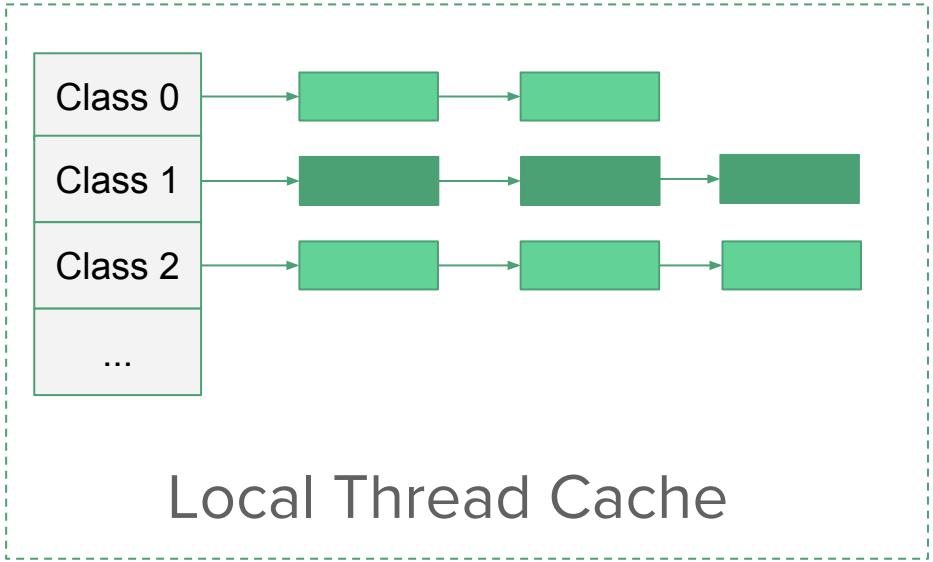


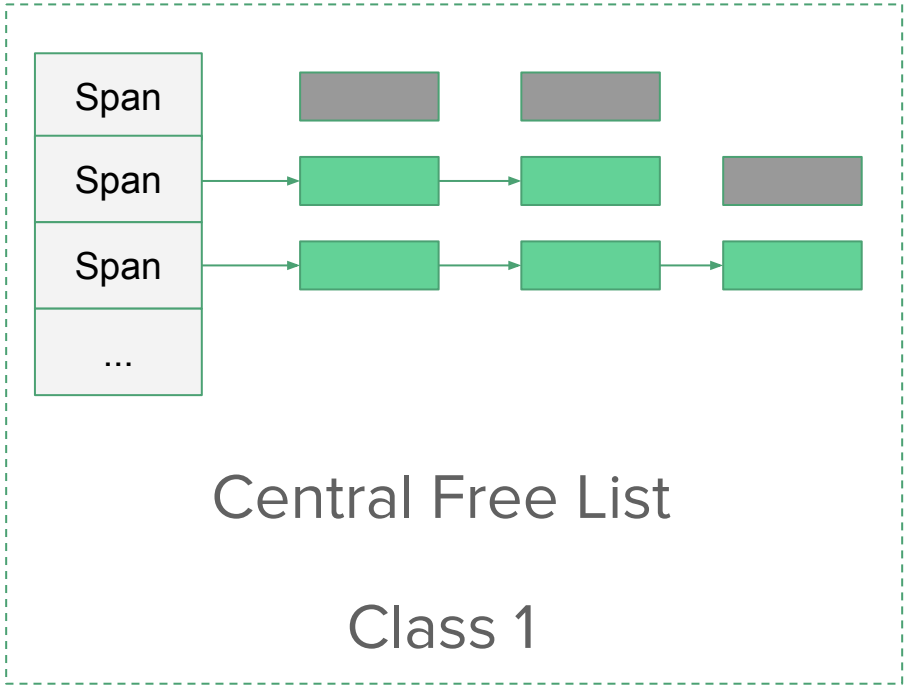
Local Thread Cache

TCMalloc - Small Allocations



TCMalloc - Small Allocations

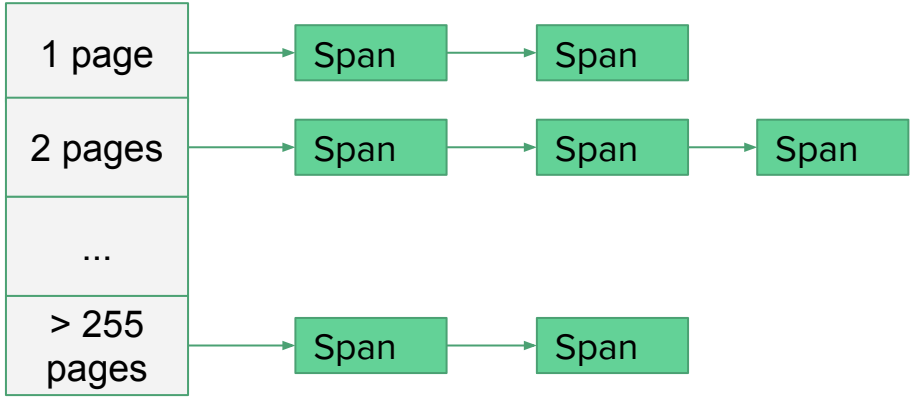




TCMalloc - Small Allocations

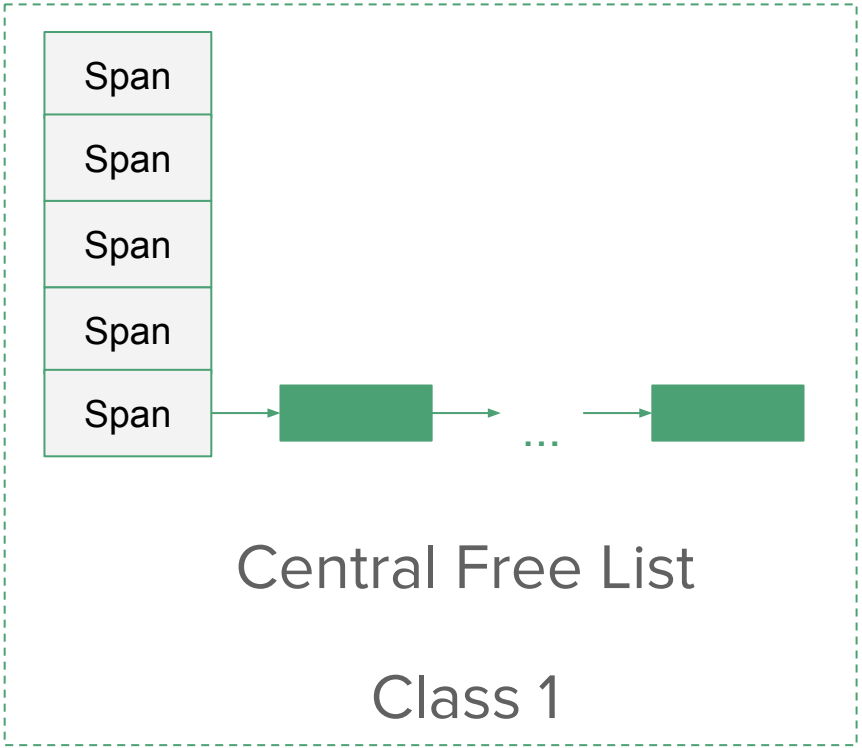
Central Free List

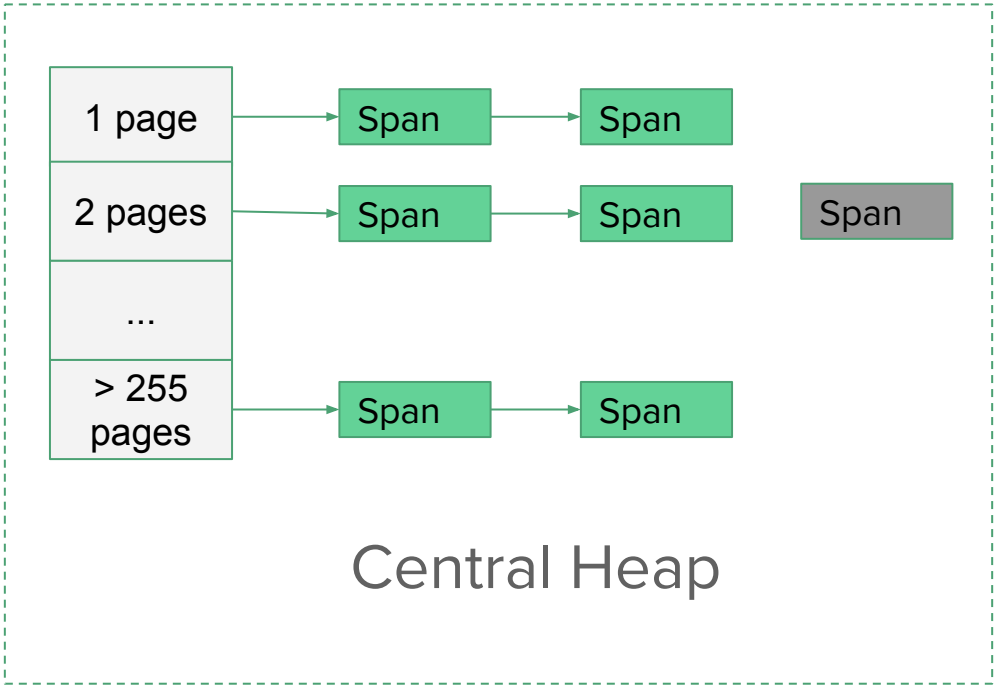
Class 1



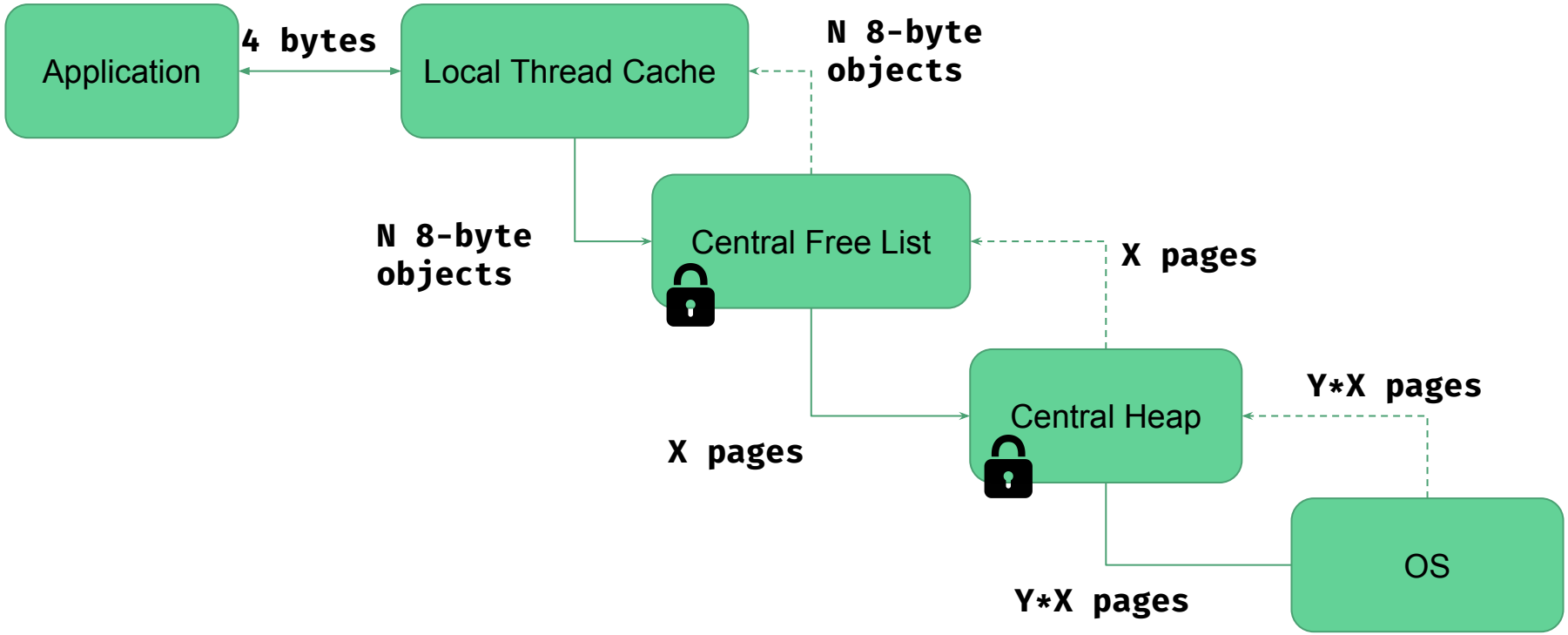
Central Heap

TCMalloc - Small Allocations

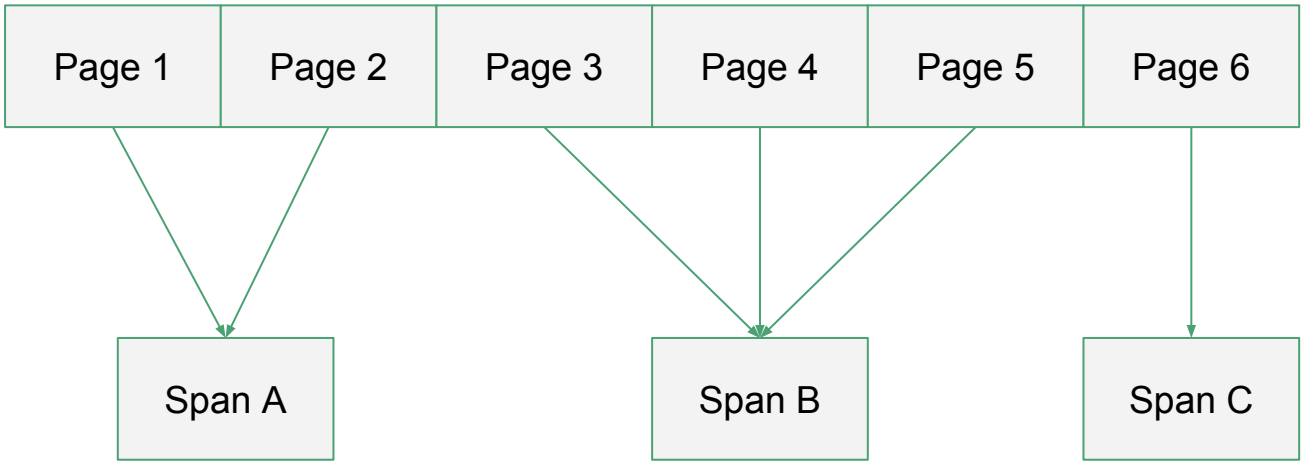




TCMalloc - Small Allocations

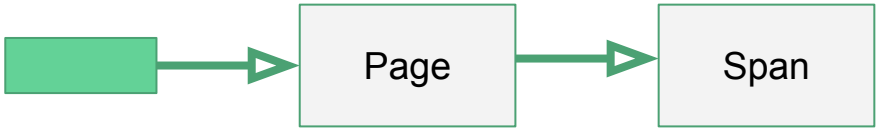


TCMalloc - Deallocation

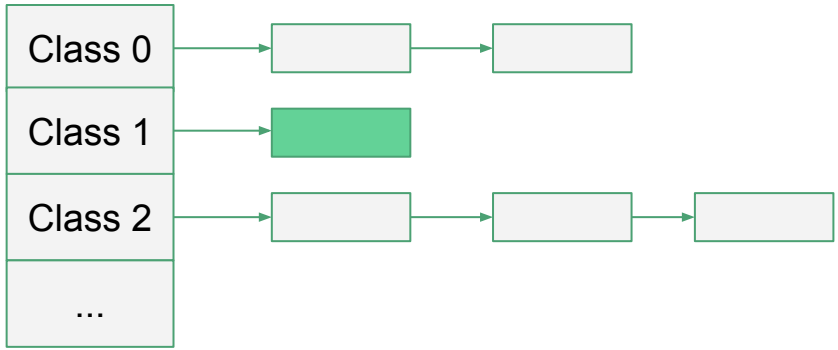


TCMalloc - Deallocation

TCMalloc - Deallocation



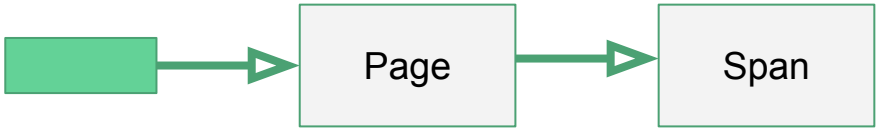
Small object



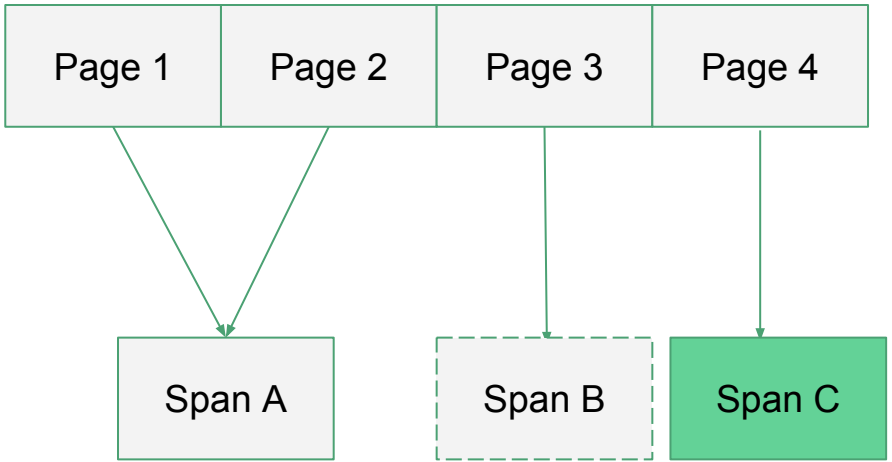
Local Thread Cache

TCMalloc - Deallocation

free()

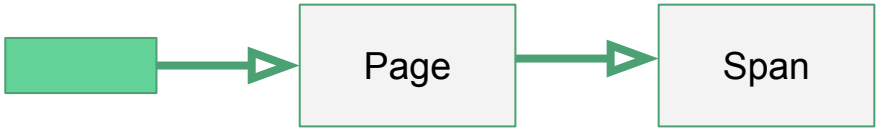


Large object

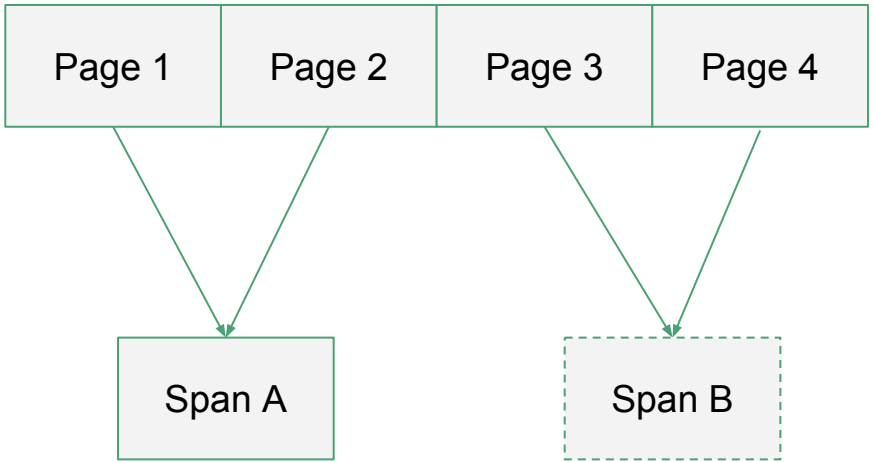


TCMalloc - Deallocation

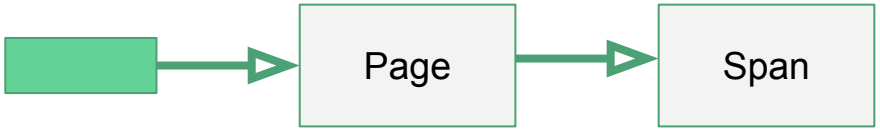
free()



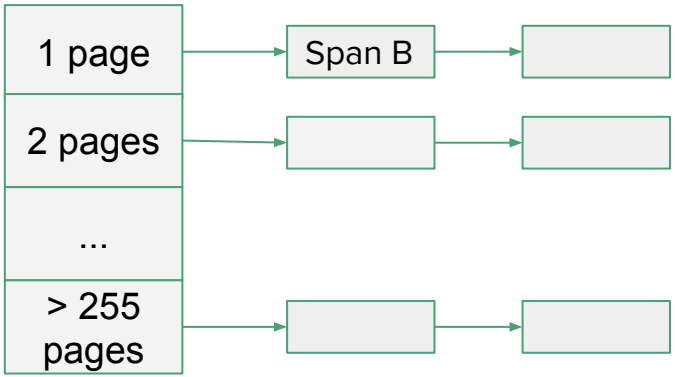
Large object



TCMalloc - Deallocation

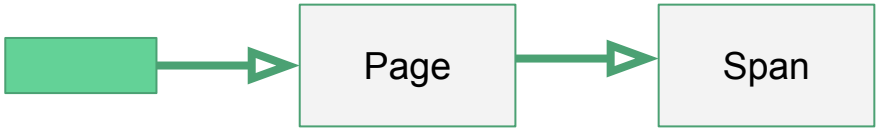


Large object

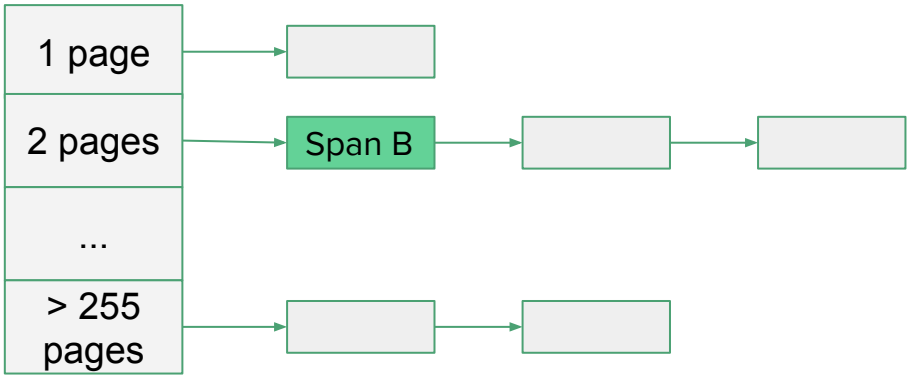


Central Heap

TCMalloc - Deallocation



Large object



Central Heap

Go Runtime Allocator

- TCMalloc
- Invoking the Allocator
- Go’s Allocator

```go
package main

func main() {
    f()
}

//go:noinline
func f() *int {
    i := 10
    return &i
}
```

```go
package main

func main() {
    f()
}

//go:noinline
func f() *int {
    i := 10
    return &i
}
```

```
$ go build -gcflags "-m -m" main.go

# command-line-arguments
./main.go:8:6: cannot inline f: marked
go:noinline
./main.go:3:6: cannot inline main: non-leaf
function
./main.go:10:9: &i escapes to heap
./main.go:10:9:    from ~r0 (return) at
./main.go:10:2
./main.go:9:2: moved to heap: i
```

```
$ go tool compile -S main.go
...
0x001d 00029 (main.go:9)      LEAQ  type.int(SB), AX
0x0024 00036 (main.go:9)      MOVQ  AX, (SP)
0x0028 00040 (main.go:9)      PCDATA      $0, $0
0x0028 00040 (main.go:9)      CALL  runtime.newobject(SB)
...
```

```
$ go tool compile -S main.go
...
0x001d 00029 (main.go:9)      LEAQ  type.int(SB), AX
0x0024 00036 (main.go:9)      MOVQ  AX, (SP)
0x0028 00040 (main.go:9)      PCDATA      $0, $0
0x0028 00040 (main.go:9)      CALL  runtime.newobject(SB)
...
```

```
func newobject(typ *_type) unsafe.Pointer {
      return mallocgc(typ.size, typ, true)
}
```

Go Runtime Allocator

- TCMalloc
- Invoking the Allocator
- Go’s Allocator

Go’s Allocator

- Based of TCMalloc
- Garbage Collector
  - Tightly coupled with the allocator
  - Makes hard (impossible?) to replace with other implementations
- Three types of allocations
  - Tiny Allocations (size < 16 bytes, no pointers)
  - Small Allocations (size <= 32 kbytes)
  - Large Allocations

Go’s Allocator - Sweeping

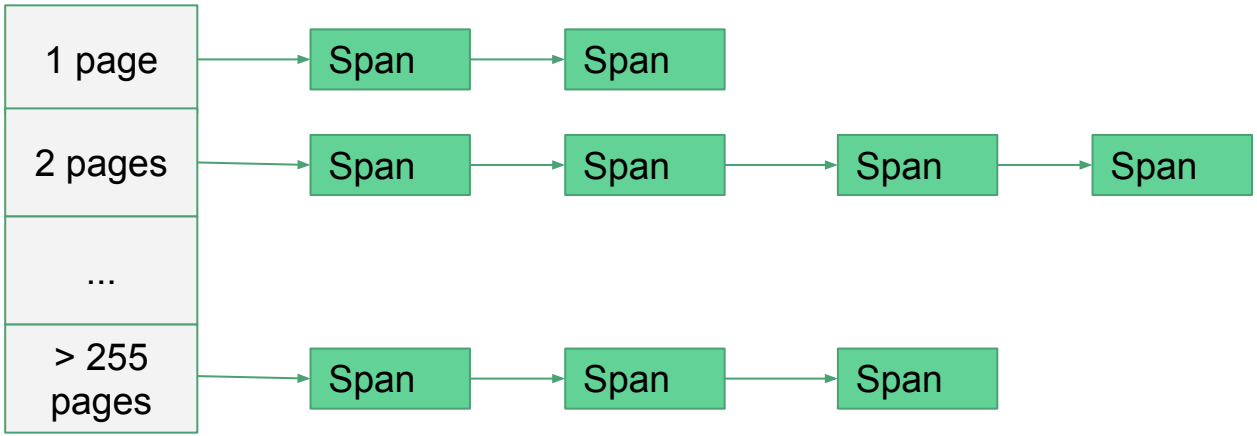
Garbage Collector ⇒ Concurrent mark and sweep

1. Scan all objects
2. **Mark** objects that are live
3. **Sweep** objects that are not live
    a. In background
    b. In response to allocations

Go’s Allocator - Large Allocations

mheap

Busy Spans

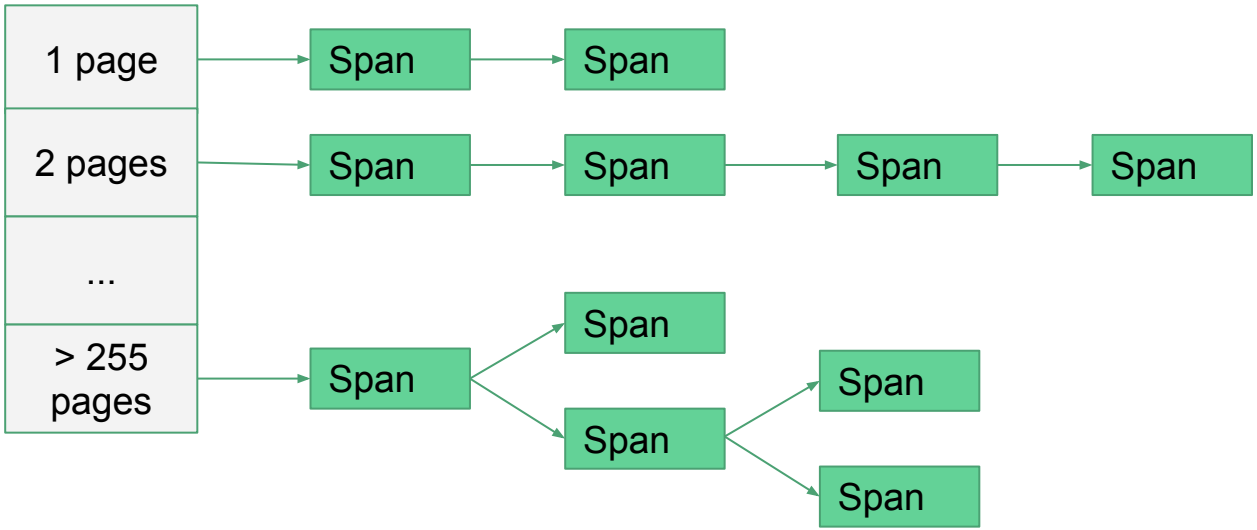


Before allocating, mheap **sweeps** the requested number of pages

Go’s Allocator - Large Allocations

mheap

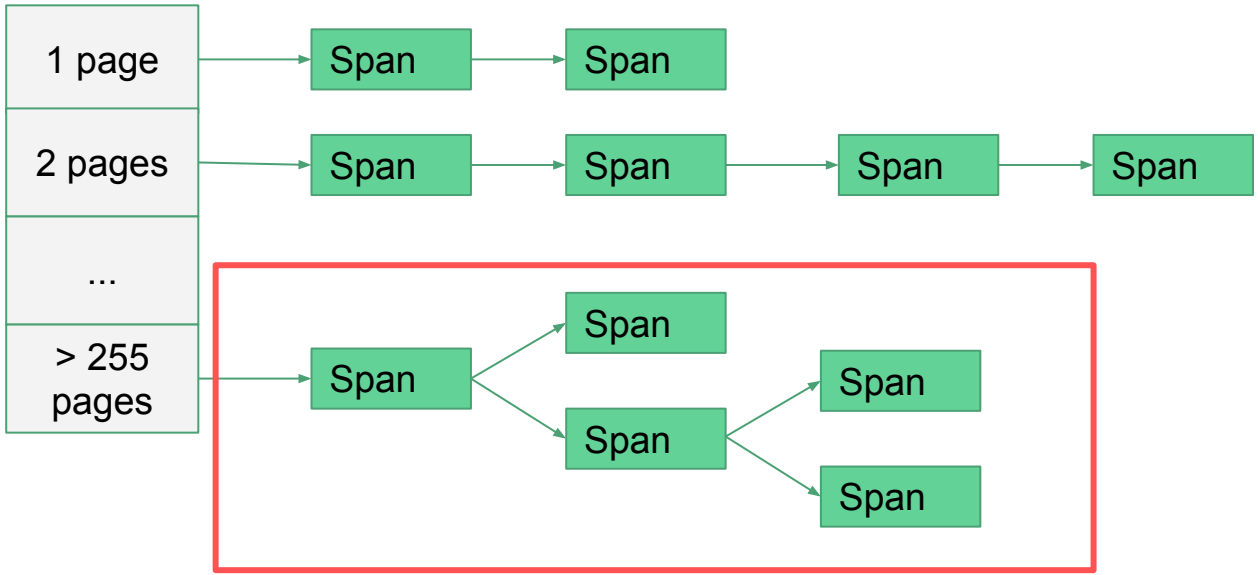
Free Spans



Go’s Allocator - Large Allocations

mheap

Free Spans



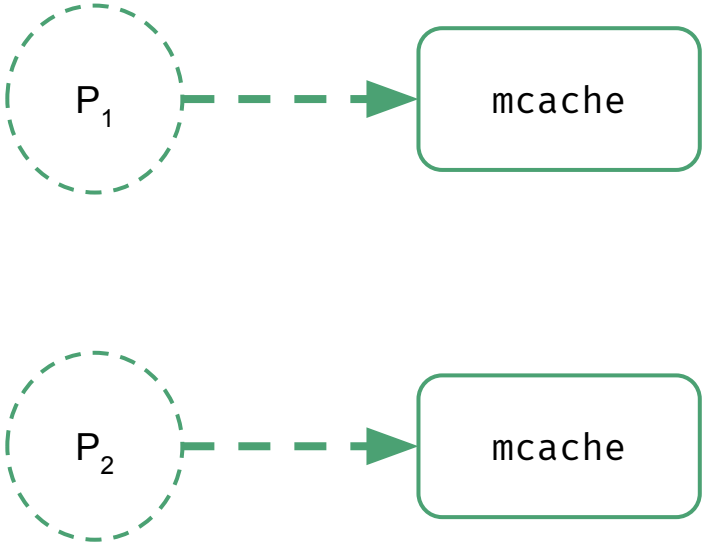
`mtreap` ⇒ randomized binary tree

Go’s Allocator - Large Allocations

After allocating, depending on the total amount of live memory...

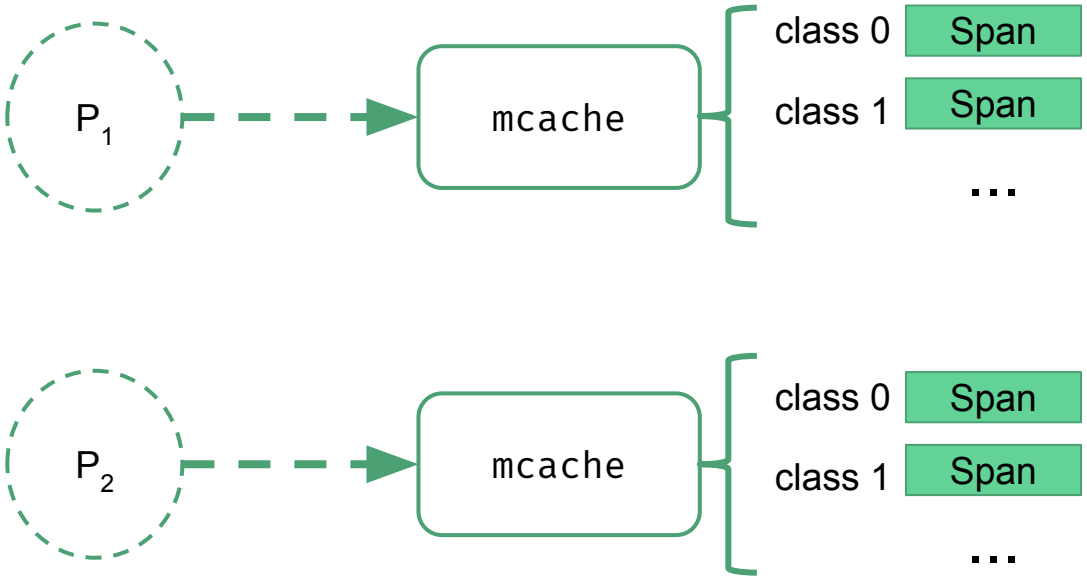
The goroutine may perform additional work for the GC!

Go’s Allocator - Small Allocations



Each logical processor (P) has a local cache (`mcache`)

Go’s Allocator - Small Allocations

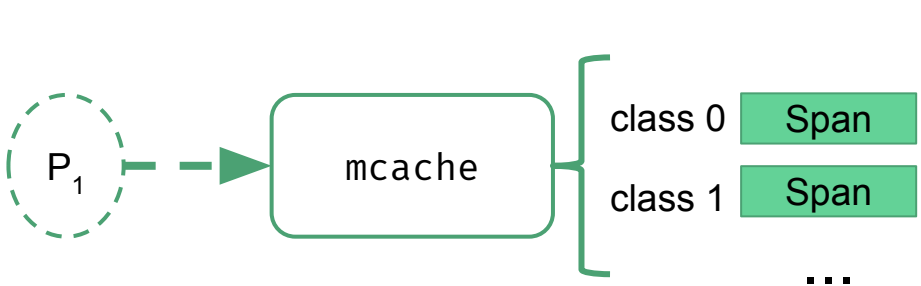


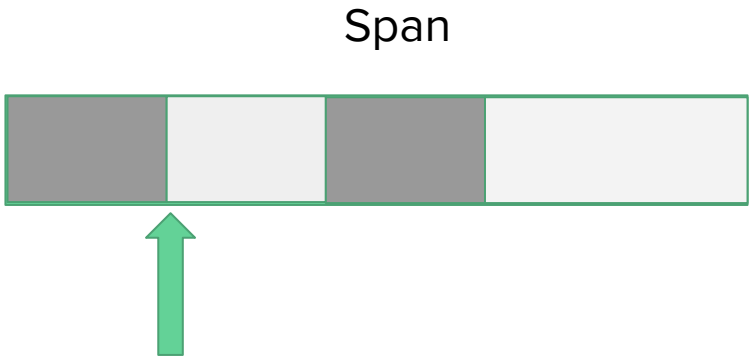
Each `mcache` maintains a span for each **size class**

Go’s Allocator - Small Allocations

class	bytes/obj	bytes/span	objects
1	8	8192	1024
2	16	8192	512
3	32	8192	256
4	64	8192	170
...			
65	28672	57344	2
66	32768	32768	1

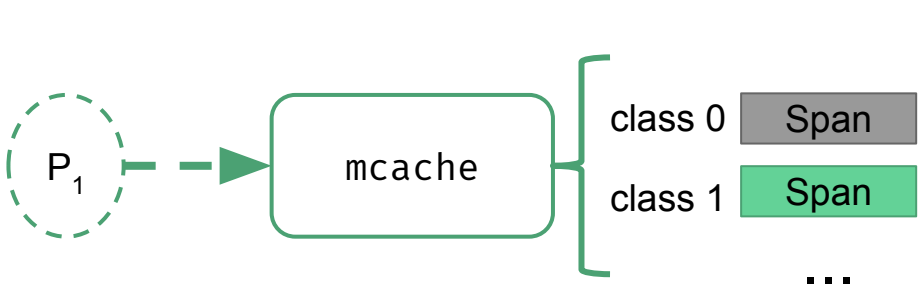
Go’s Allocator - Small Allocations

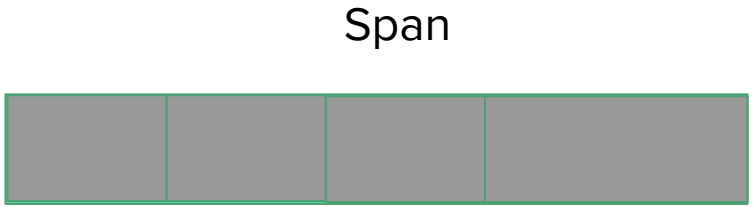




`mcache` returns the address for a free object on the span

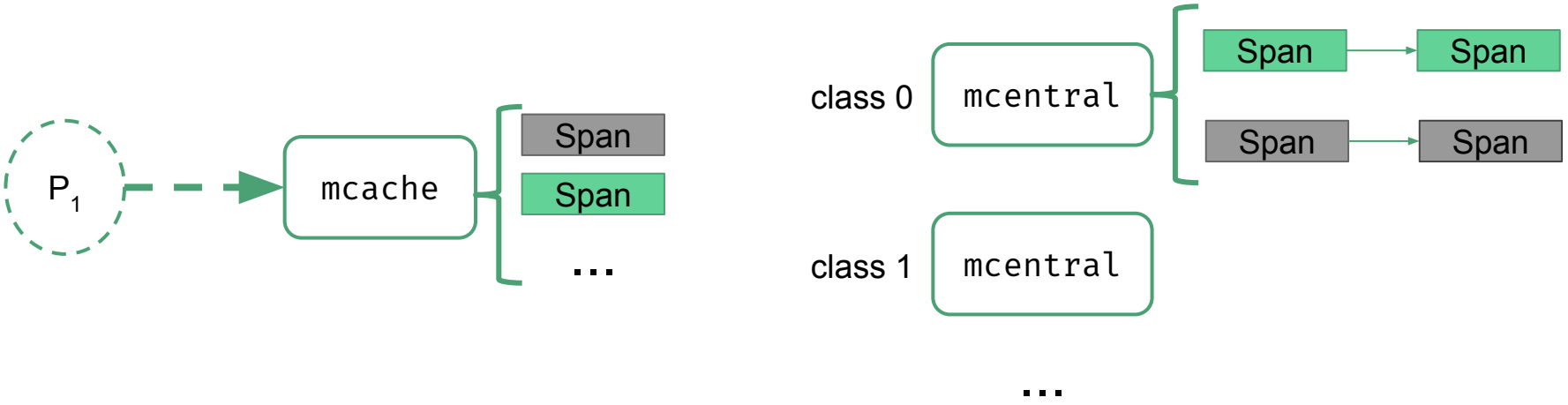
Go’s Allocator - Small Allocations





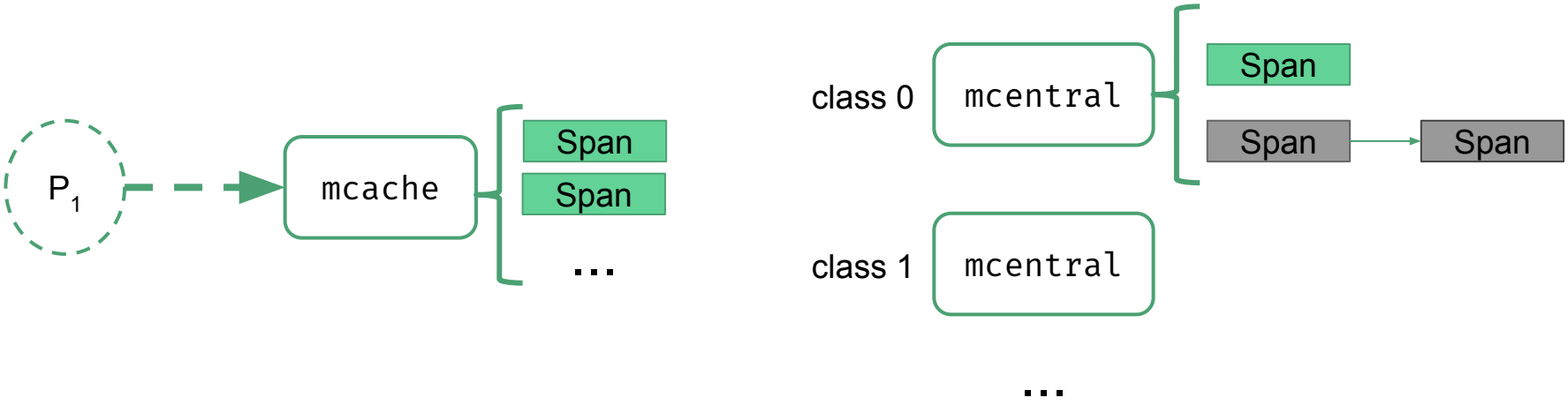
`mcache` request a new span from `mcentral` for this **size class**

Go’s Allocator - Small Allocations



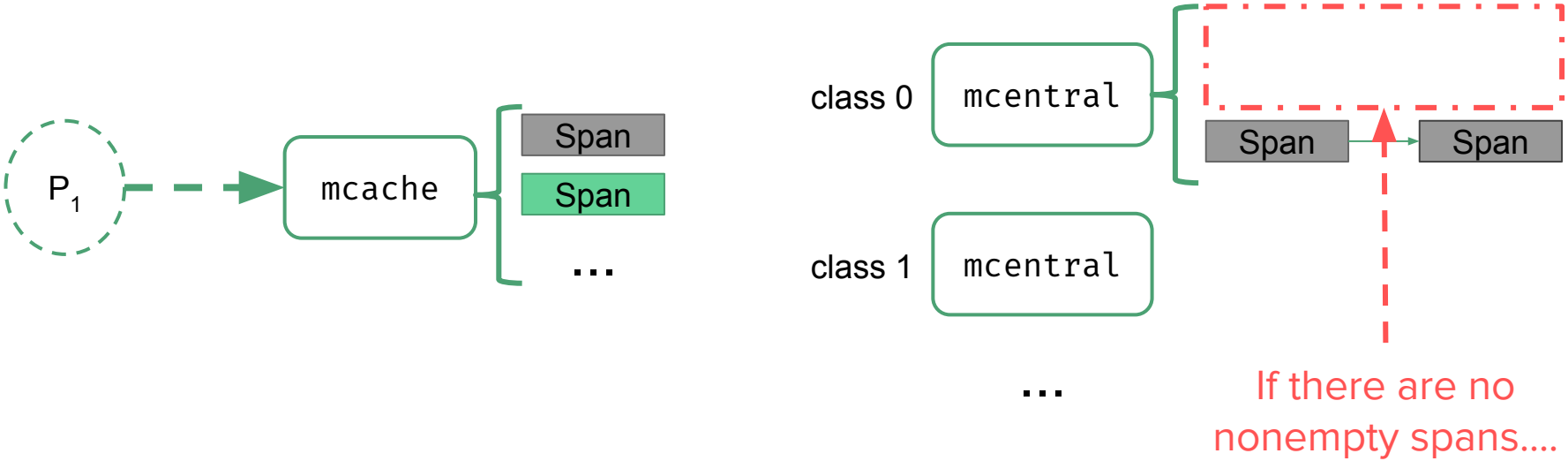
Each `mcentral` has two linked lists, empty and nonempty spans

Go’s Allocator - Small Allocations



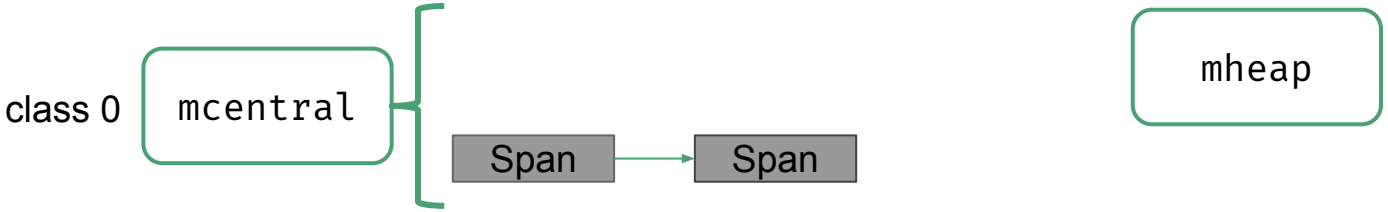
Span with free objects will be given to the `mcache`

Go’s Allocator - Small Allocations



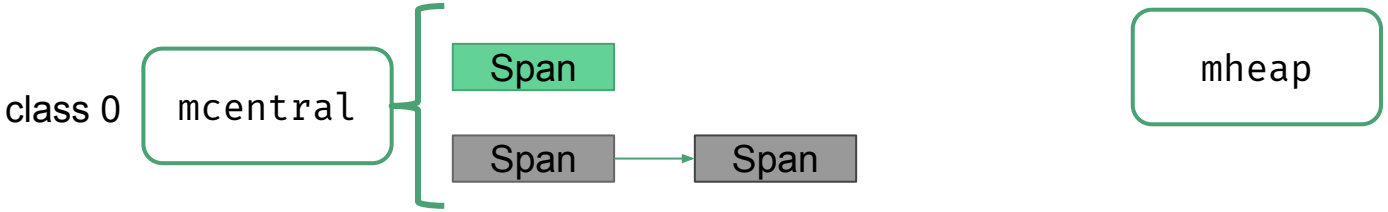
mcentral will try to sweep existing spans

Go’s Allocator - Small Allocations



As a last resort, `mcentral` will ask for a new span from `mheap`

Go’s Allocator - Small Allocations



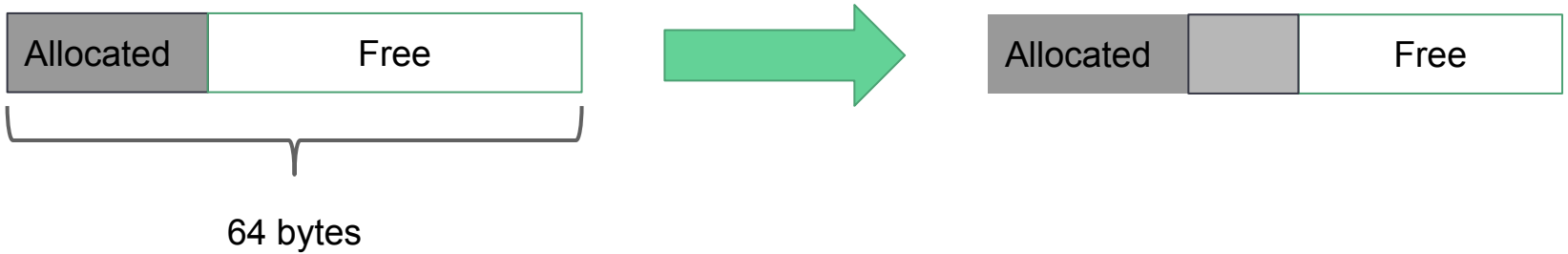
`mcentral` will give this span to `mcache`

Go’s Allocator - Tiny Allocations

Allocations for objects with **no** pointers and size < **16 bytes**

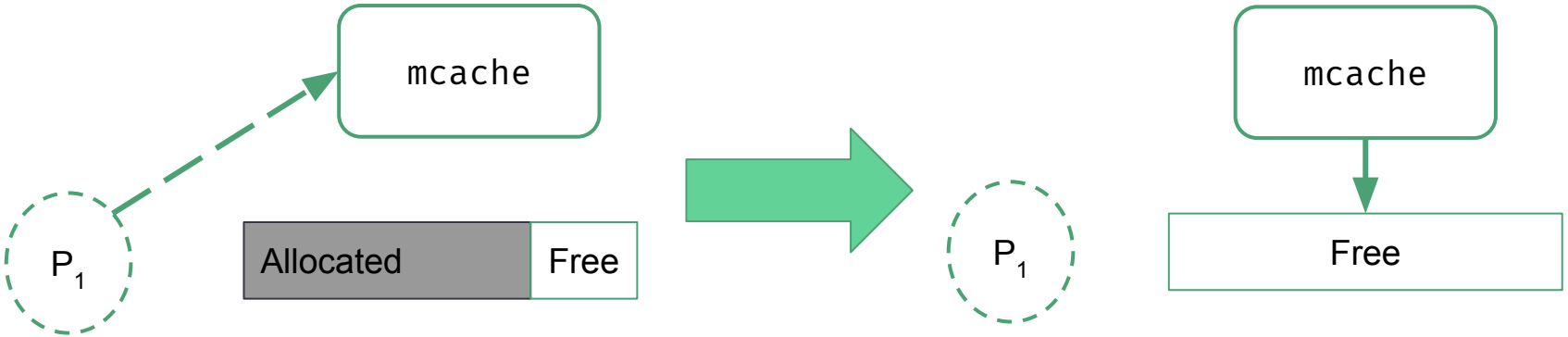
The main targets of tiny allocator are **small strings** and standalone escaping variables. On a json benchmark the allocator reduces number of allocations by ~12% and reduces **heap size by ~20%**.

Go’s Allocator - Tiny Allocations



- Each **P** keeps a 64-bytes **object** allocated from a span
- Each tiny allocation appends a **subobject**

Go’s Allocator - Tiny Allocations



- Grab a new object from the `mcache` ≃ small allocation
- Eventually, GC will deallocate the old object

Go’s Allocator - Releasing memory to the OS

- Runtime periodically releases memory to the OS
- Releases spans that were swept more than **5 minutes ago**
- In Linux, uses the `madvise(2)` syscall

```
madvise(addr, size, _MADV_DONTNEED)
```

```go
stats := runtime.MemStats{}
runtime.ReadMemStats(&stats)
```

```go
type MemStats struct {
    ...
    // Heap memory statistics.
    HeapAlloc uint64
    HeapSys uint64
    HeapIdle uint64
    HeapInuse uint64
    HeapReleased uint64
    HeapObjects uint64
    ...
}
```

References

1. http://goog-perftools.sourceforge.net/doc/tcmalloc.html
2. https://www.ardanlabs.com/blog/2017/05/language-mechanics-on-stacks-and-pointers.html
3. https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/
4. Lec 10 | MIT 6.172 - https://www.youtube.com/watch?v=p0bc1f6ULxw
5. https://faculty.washington.edu/aragon/pubs/rst89.pdf
6. http://man7.org/linux/man-pages/man2/mmap.2.html
7. http://man7.org/linux/man-pages/man2/madvise.2.html
8. https://nostarch.com/tlpi

Thanks!

andrestc.com

@andresantostc