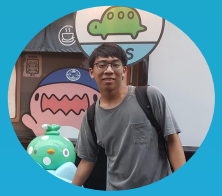
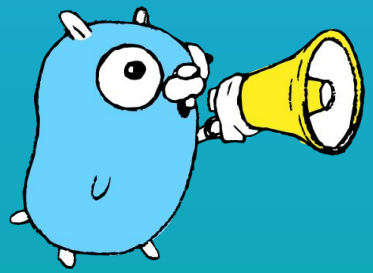
Taiwan, Taipei 2021

The life of heap memory



Gaston Qiu



Today's Agenda

Background Knowledge

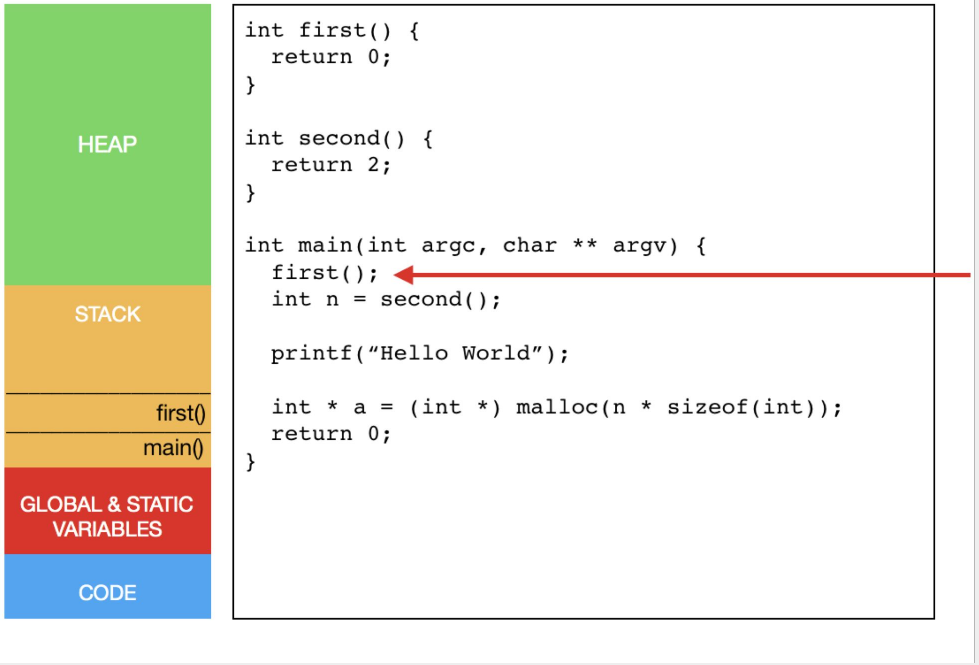
TCMalloc

Garbage collection

SECTION ONE

Background Knowledge

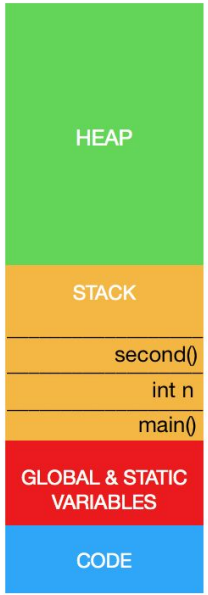
Heap and Stack Example



Stack is fixed size linear data struct and only handle the memory will be release after the function finished.

https://medium.com/@wireless.patrick/a-tale-of-memories-stack-and-heap-7528f49aea6e

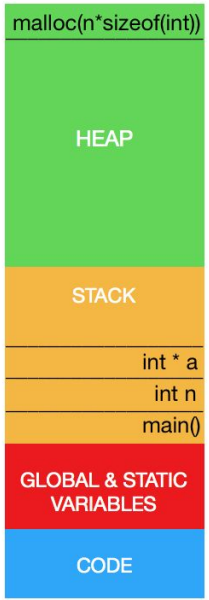
Heap and Stack Example




```
int first() {
  return 0;
}

int second() {
  return 2;
}

int main(int argc, char ** argv) {
  first();
  int n = second();

  printf("Hello World");

  int * a = (int *) malloc(n * sizeof(int));
  return 0;
}
```

Stack is fixed size linear data struct and only handle the memory will be release after the function finished.

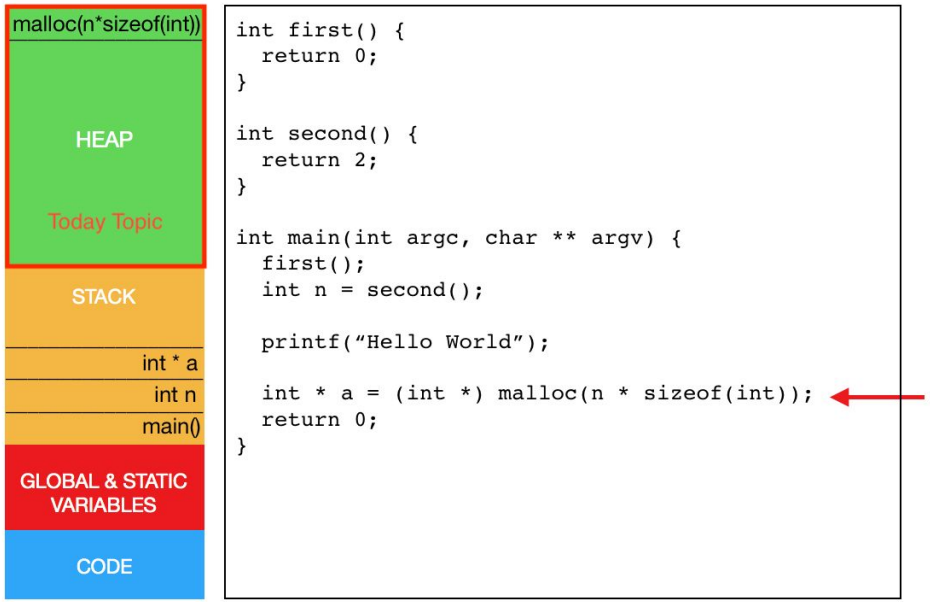
Heap and Stack Example




```
int first() {
  return 0;
}

int second() {
  return 2;
}

int main(int argc, char ** argv) {
  first();
  int n = second();

  printf("Hello World");

  int * a = (int *) malloc(n * sizeof(int));
  return 0;
}
```

Heap is the runtime determinate hierarchical data structure which we need to manage its life cycle.

Heap and Stack Example



The memory allocation on heap is slow and handle its life cycle is difficult and tricky. So there is a lots of things we can optimize.

Escape Analysis

New an object on heap

```
func main() {
    num := getRandom()
    println(*num)
}

//go:noinline
func getRandom() *int {
    tmp := rand.Intn(100)
    return &tmp
}
```

Go decide the object should go to heap or stack through the escape analysis.

https://medium.com/a-journey-with-go/go-introduction-to-the-escape-analysis-f7610174e890

Garbage collection (Mark and Sweep)

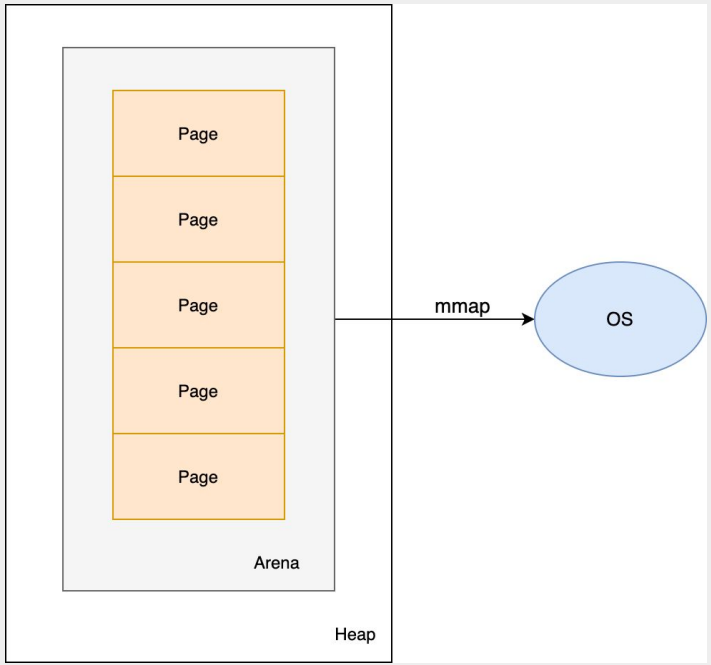
Free an allocated object

```
func main() {
    num := getRandom()
    println(*num)
    runtime.GC() // trigger gc
}

//go:noinline
func getRandom() *int {
    tmp := rand.Intn(100)
    return &tmp
}
```

The mechanism that decide the object should be released is called garbage collection. (In really life it will trigger automatically)

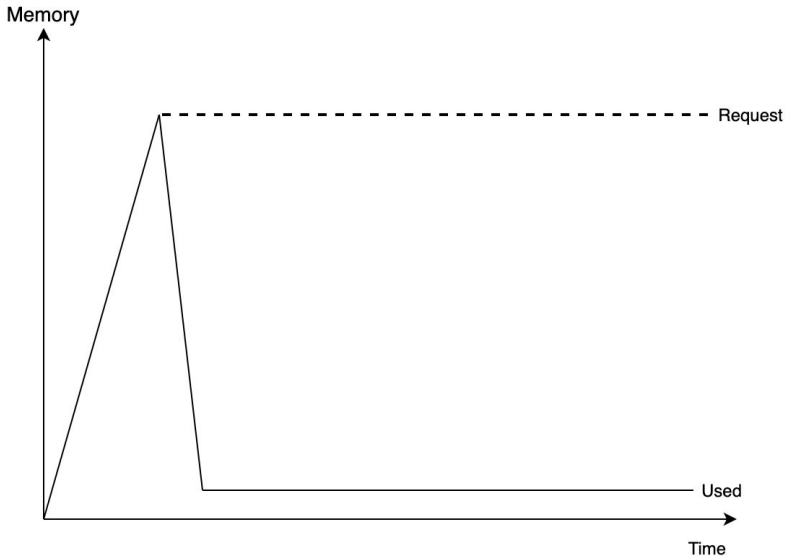
Heap struct in go

New an object on heap

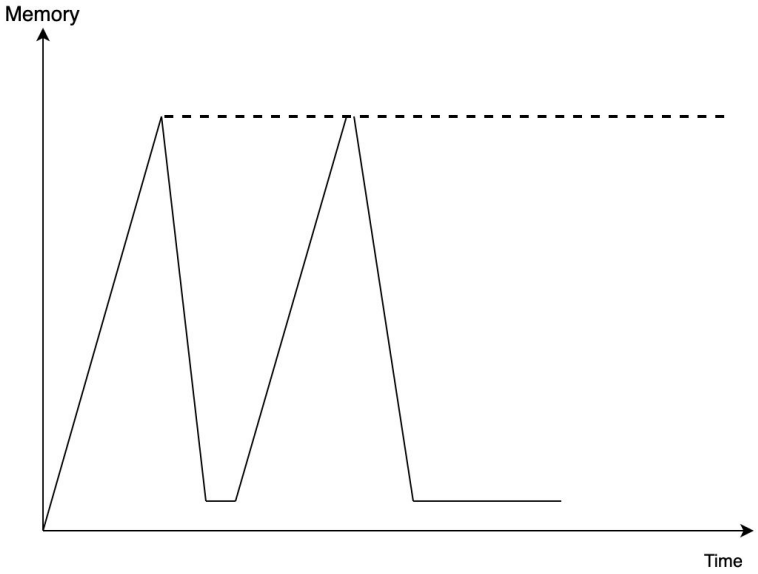


System calls (mmap) are very expensive. We pre-allocate an arena of memory.

Since we hold the memory, we need to release it.



Since we hold the memory, we need to release it.



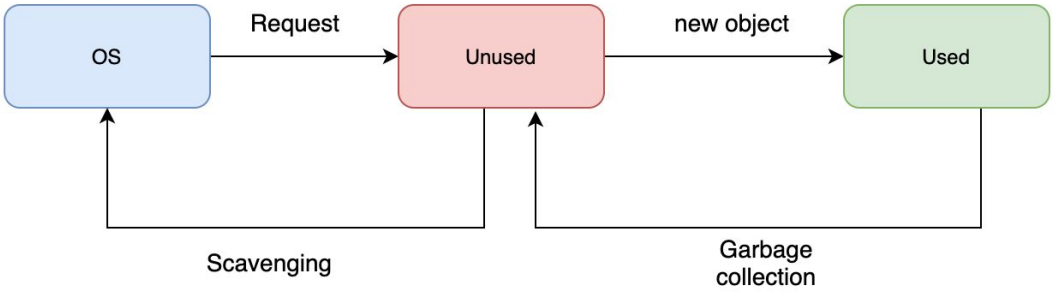
Garbage collection (Scavenging)

Release the memory to OS

```
func bgscavenge(c chan int)

func (p *pageAlloc) scavenge(
      nbytes uintptr,
      mayUnlock bool
) uintptr
```

After a memory is marked as unused. We should release it OS.

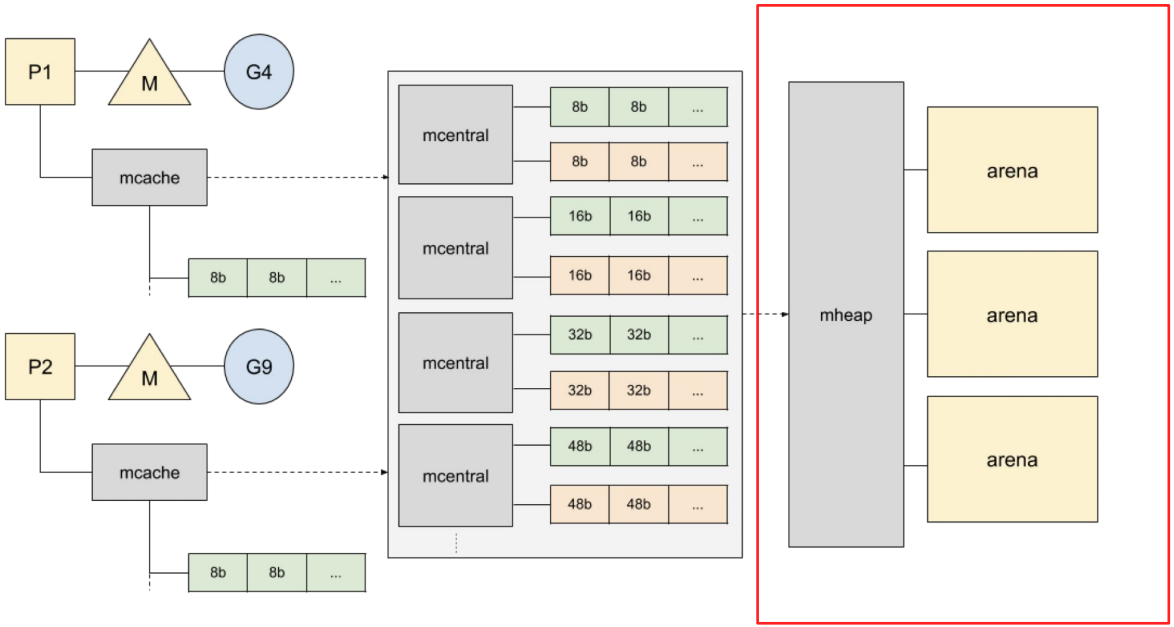
Memory life cycle



SECTION TWO

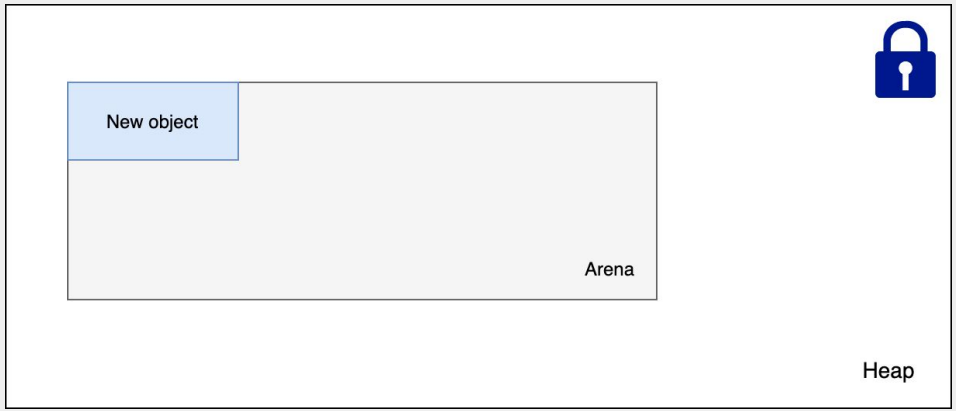
Thread-cache Malloc (TCMalloc)

Large allocation



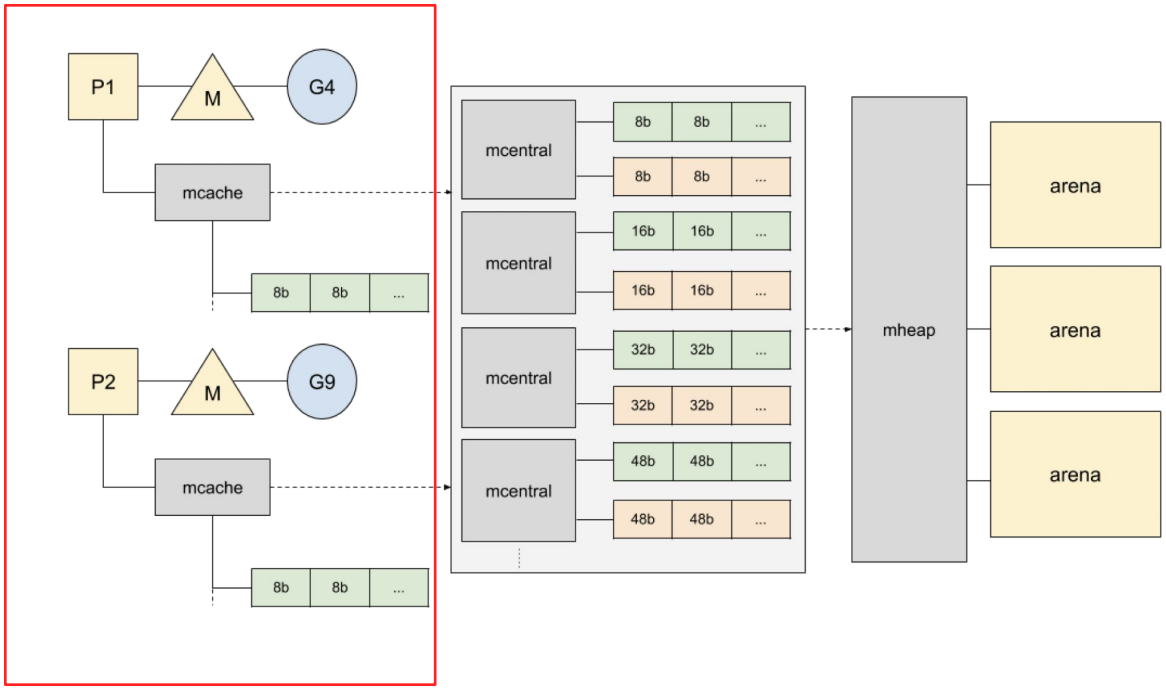
Ref:
https://medium.com/a-journey-with-go/go-memory-management-and-allocation-a7

Large allocation

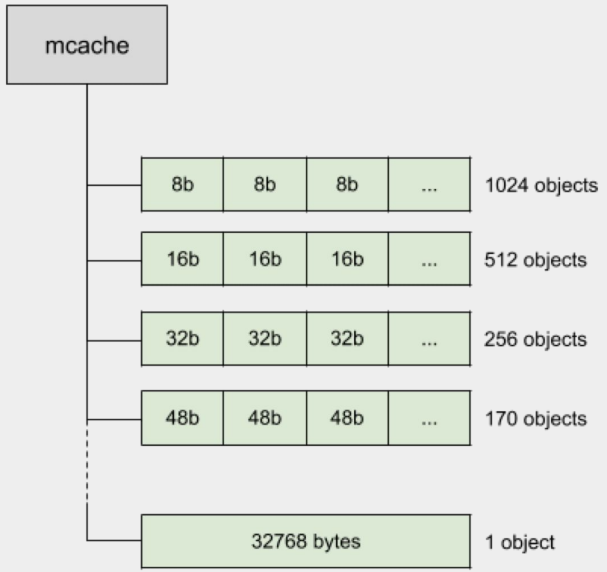


- Object size > 32KB
- Directly allocate on arena
- Lock required

Small allocation



Small allocation



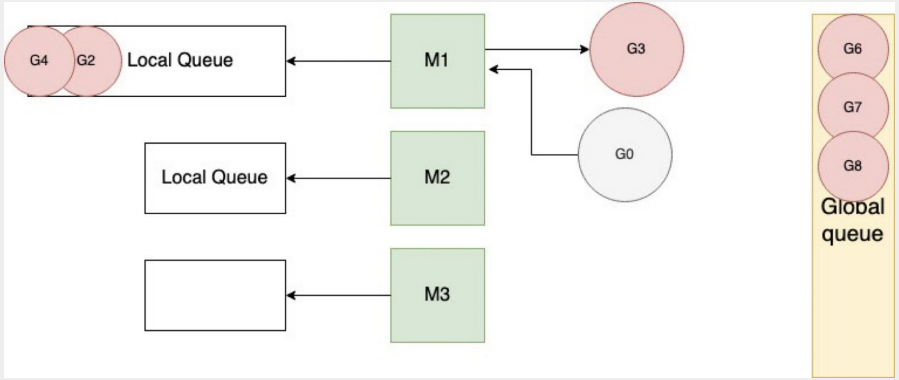
- Smallest object allocation (≤ 32KB)

- There is about 70 different classes of span from 8 bytes to 32KB.

- Round up to the class size

- No lock required in the small allocation.

Mcache benefit



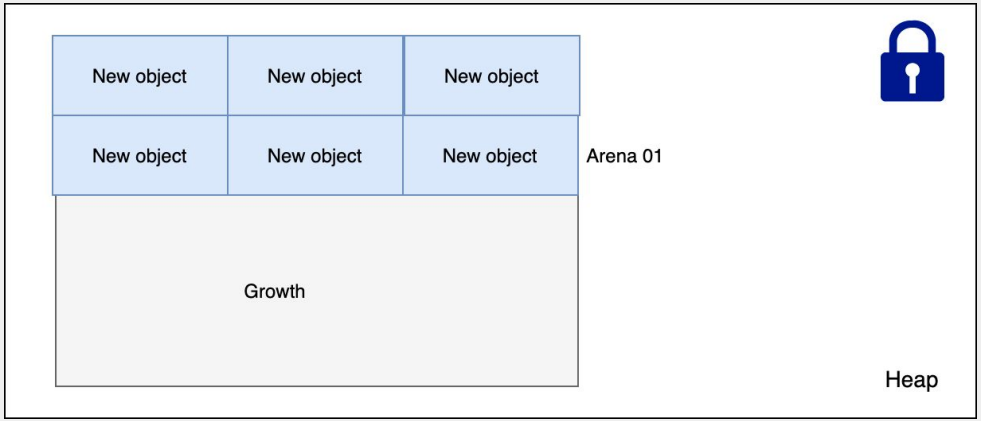
- Assuming most allocation is small object

- The mcache design align with golang m:n scheduler

- Better locality

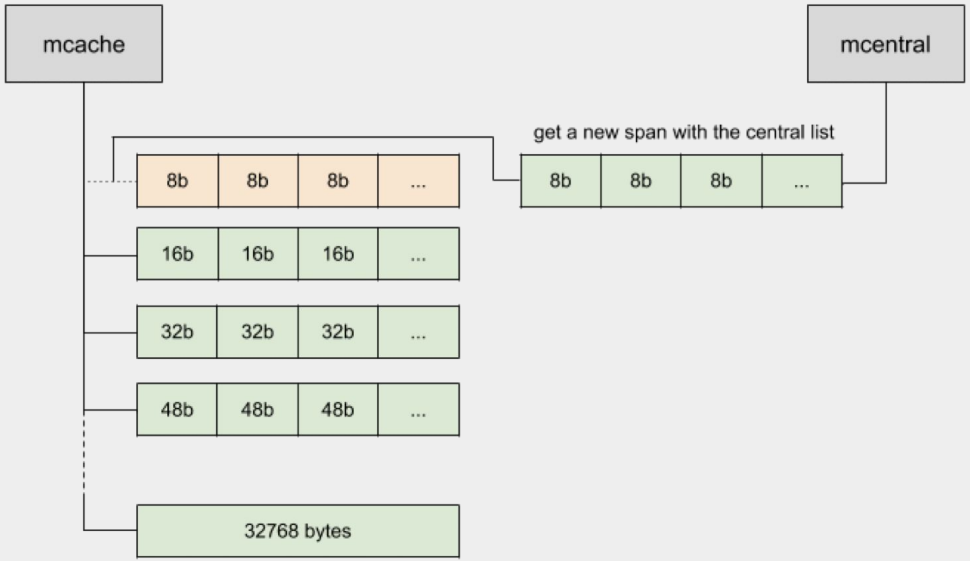
- Less fragmentation

The arena has no space (large allocation)



- Ask more pages from OS.

The mcache has no space (small allocation)

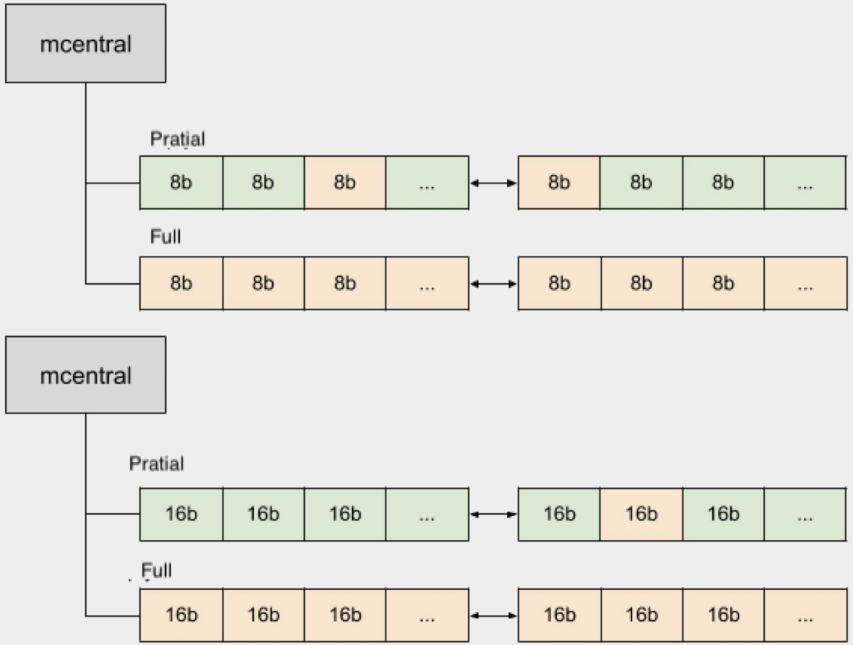


- Ask a new span from mcentral.

- All the mcaches share the same mcentral.

- The lock is required when ask a new span from mcentral.

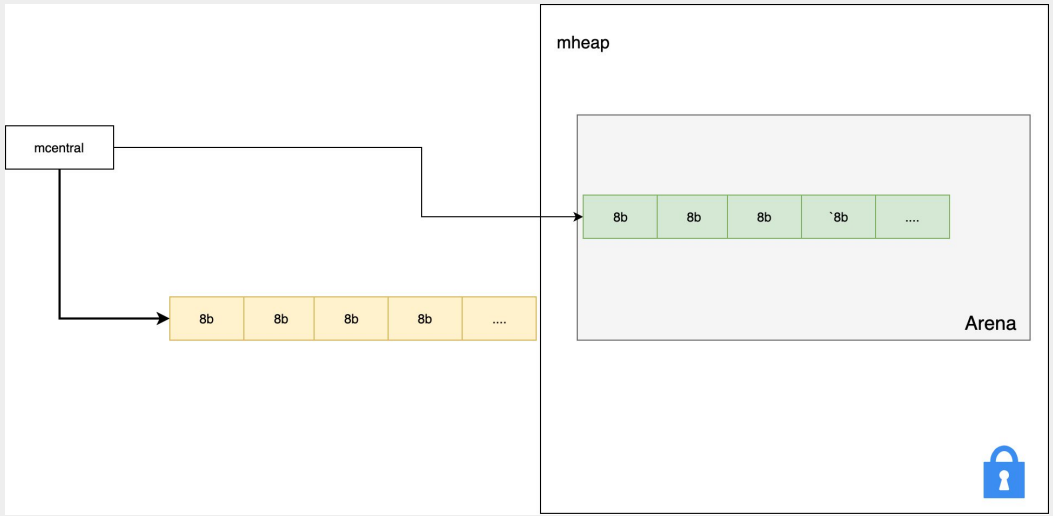
mcentral



- Partial: Partial of the span is in-used.

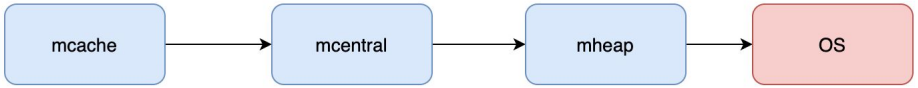
- Full: The span is full.

The central list has no space (small allocation)



- When mcentral has no space, we can allocate a new span on heap.

Request more memory



SECTION Three

Garbage collection

Mark and Sweep

Trigger garbage collection

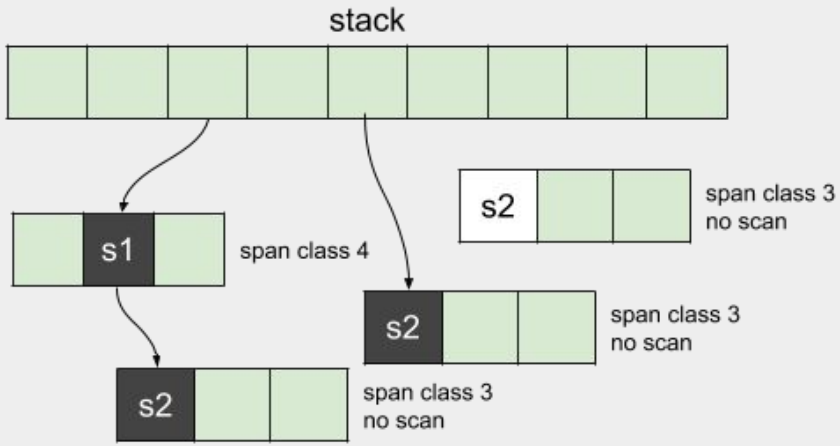
- 2 * GC percentage / 100 * current memory size (In default GC percentage = 100)

- Every few minutes

https://www.ardanlabs.com/blog/2018/12/garbage-collection-in-go-part1-semantics.html

Mark and Sweep

Mark phase (tri-color mark)



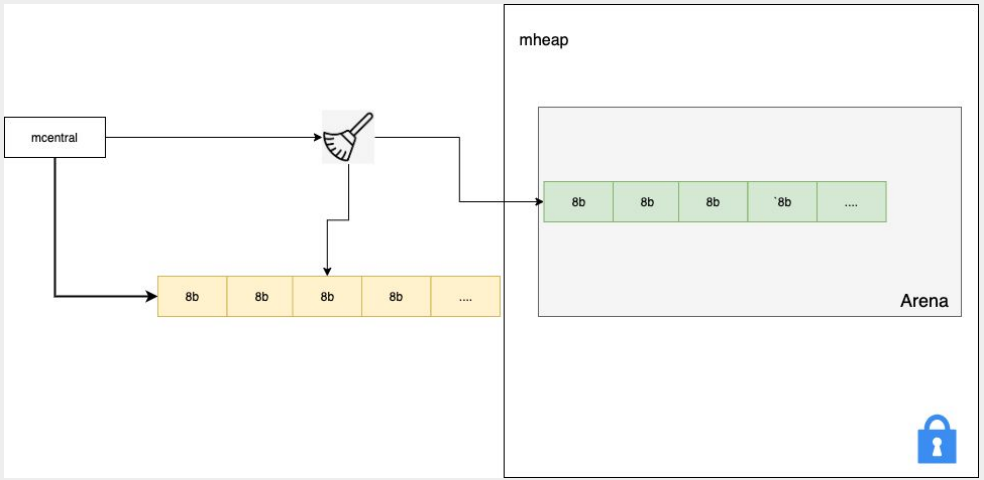
- Scan the heap and check the object is reference or not.

- Black: In-used, White: Unused

https://medium.com/a-journey-with-go/go-how-does-the-garbage-collector-mark-the-memory-72cfc12c6976

Mark and Sweep

Sweep phase

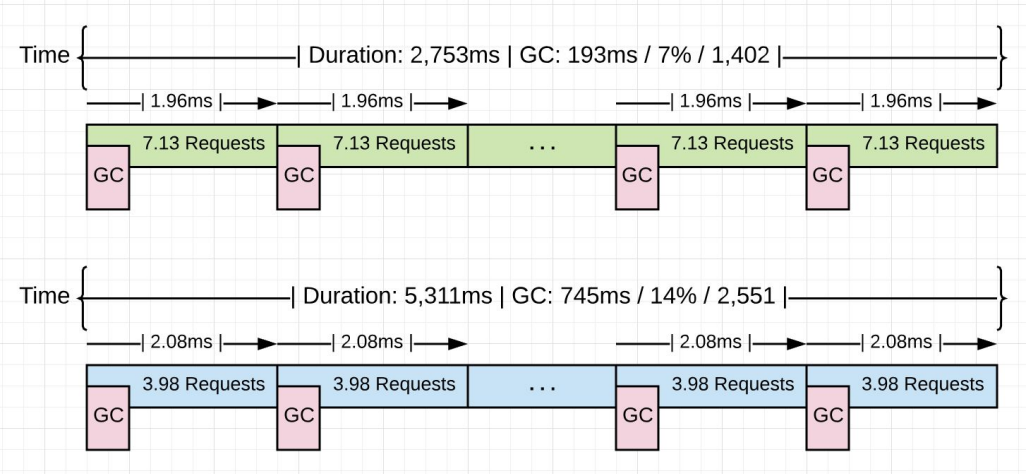


- Sweep concurrently in a background goroutine.

- When the goroutine needs another space on heap, it first attempts to reclaim that much memory by sweeping. (Before ask more from mheap or mcentral)

Mark and Sweep

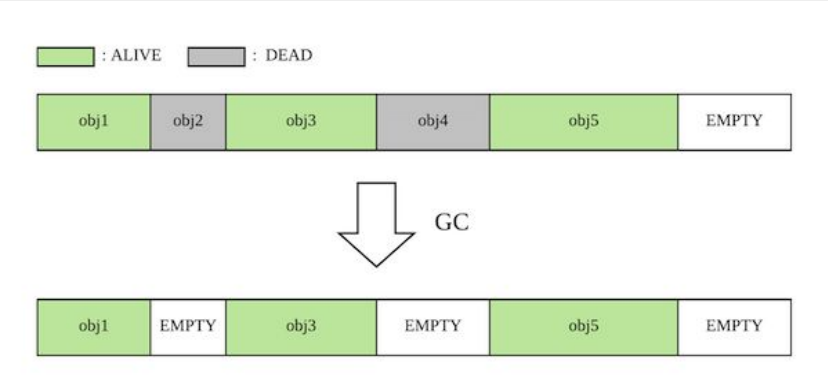
Pacing



- Pacing calculate how fast the next garbage collection trigger.

https://www.ardanlabs.com/blog/2018/12/garbage-collection-in-go-part1-semantics.html

Heap fragmentation

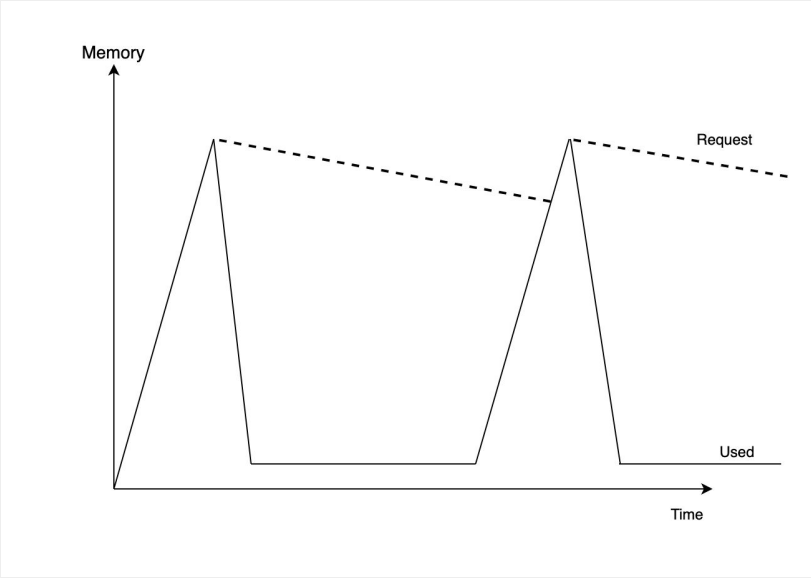


- The heap memory will be fragment because of GC.

- The fragmentation reduce the efficiency.

Ref: https://engineering.linecorp.com/en/blog/go-gc/

Scavenging

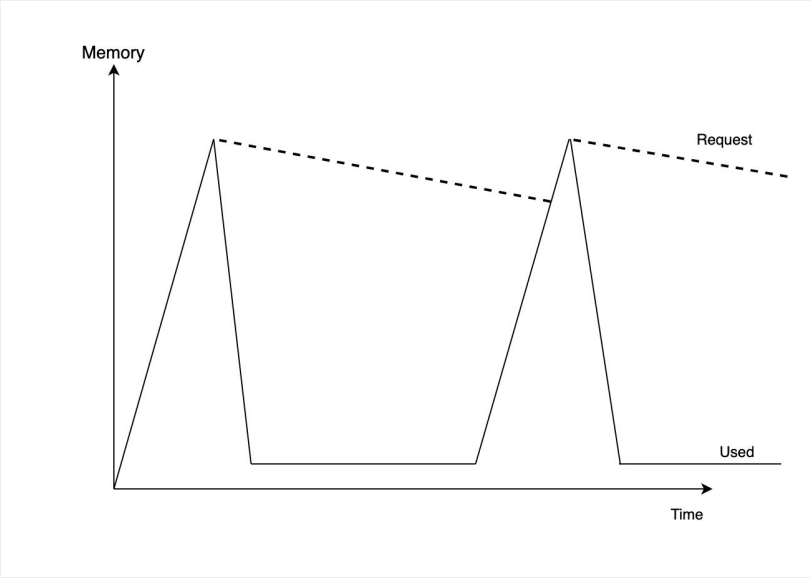


- Free and release the physical page backing mapping memory to OS.

- Reduce the page level fragmentation and Resident set size.

Scavenging

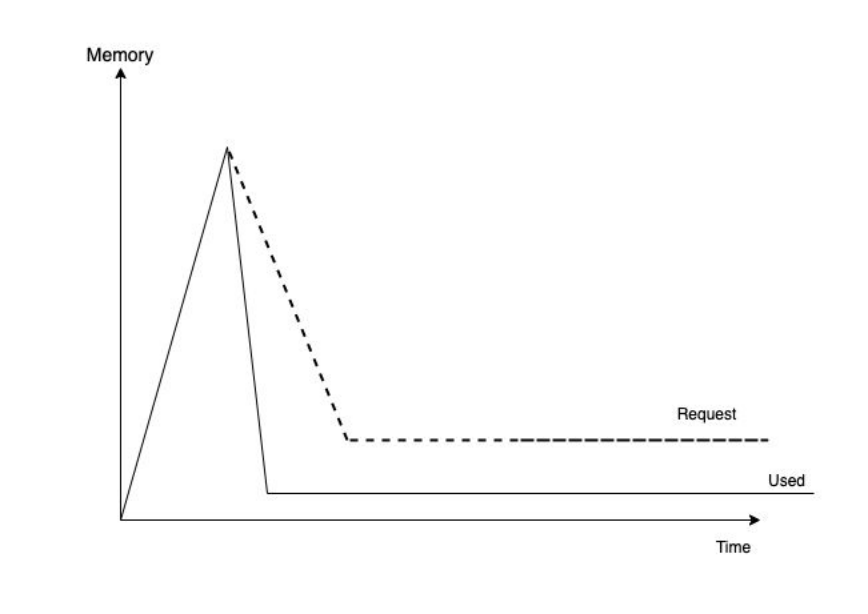
Trigger scavenging



- Background scavenging
- Heap growth

Scavenging

How many memory we should retain

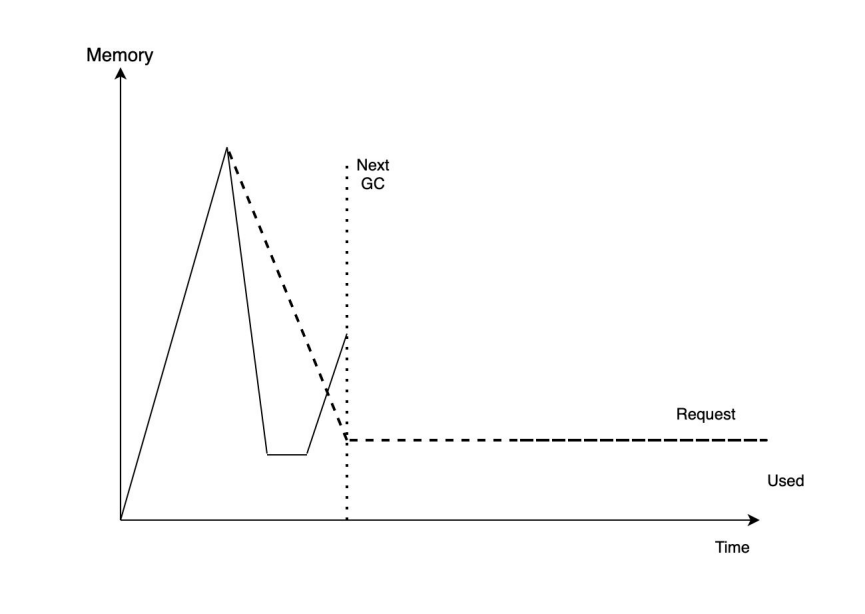


C * (next_gc_goal/last_next_gc_goal) * last_heap_inuse

(C = 1.1 in 1.16 version)

Scavenging

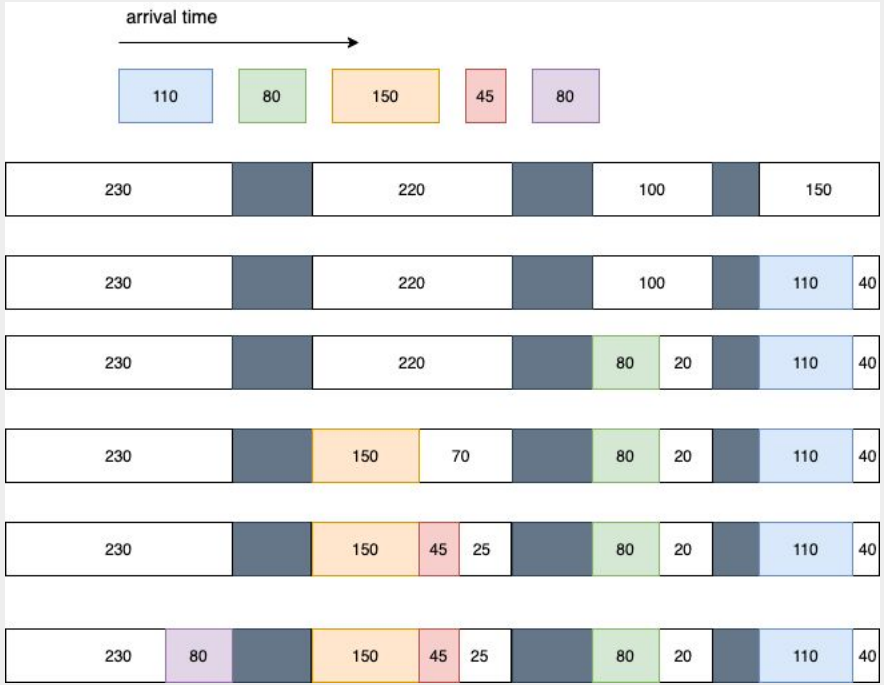
At what rate is memory scavenged



The goal is achieving the goal before the next collection trigger. (We record the interval between each trigger in pacing)

Scavenging

How to handle new allocation on heap

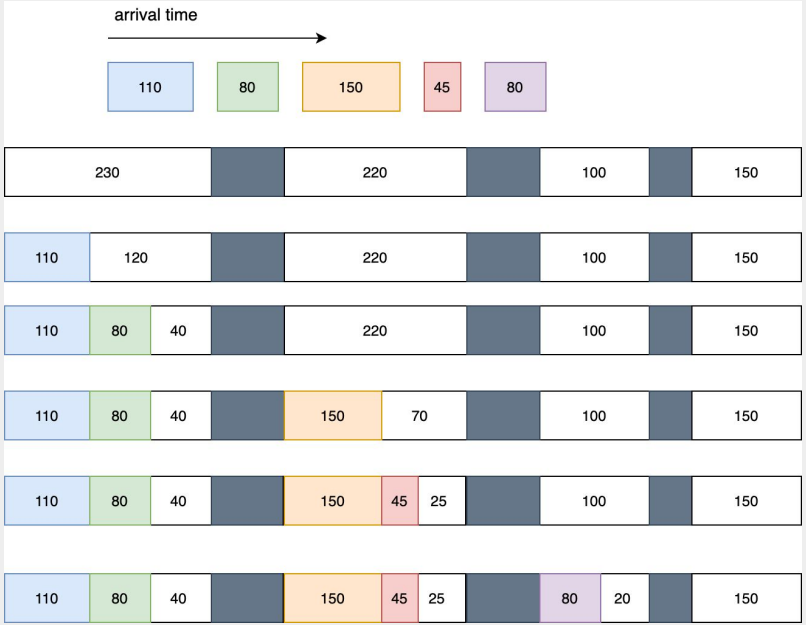


- Best-fit: the memory allocated at the closest fragmentation.

- This should provide the less fragmentation.

Scavenging

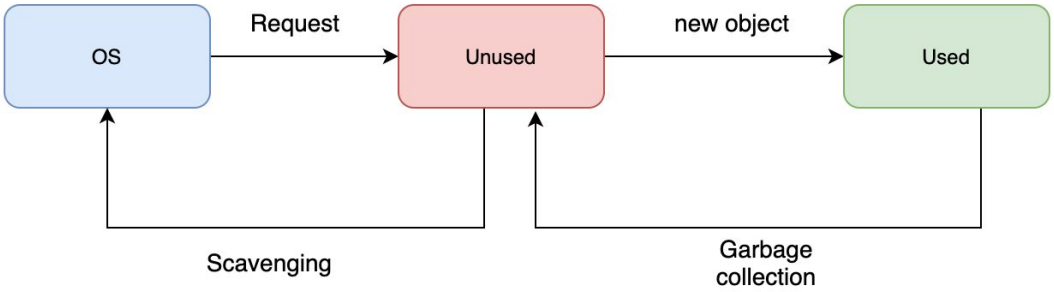
How heap handle new allocation



- First-fit: the partition is allocated which is first sufficient.

- Scavenging from the highest base addresses first.

Recap


OS
Request
Unused
new object
Used
Scavenging
Garbage
collection

Reference:

- Smarter Scavenging:
  https://go.googlesource.com/proposal/+/master/design/30333-smarter-scavenging.md
- Deep understanding go memory allocation:
  https://programmer.group/deep-understanding-go-memory-allocation.html
- Evolving the Go Memory Manager's RAM and CPU Efficiency: https://youtu.be/S_1YfTfuWmo
- Tcmalloc: https://google.github.io/tcmalloc/design.html
- GC in go: https://slides.com/jalex-chang/gc-in-go

Thank you