Netcracker
An NEC Company

Go Memory Management In Depth.

Billing&Rating DevTalks

1. Previously
2. Physical Memory
3. TCMalloc
4. Go Memory Structures
5. Garbage Collectors
6. Garbage Collector In Go
7. Tips

Calling convention

- All arguments passed on the stack
  - Offsets from FP
- Return arguments follow input arguments
  - Start of return arguments aligned to pointer size
- All registers are caller saved, except:
  - Stack pointer register
  - Zero register
  - G context pointer register
  - Frame pointer

Benchmark

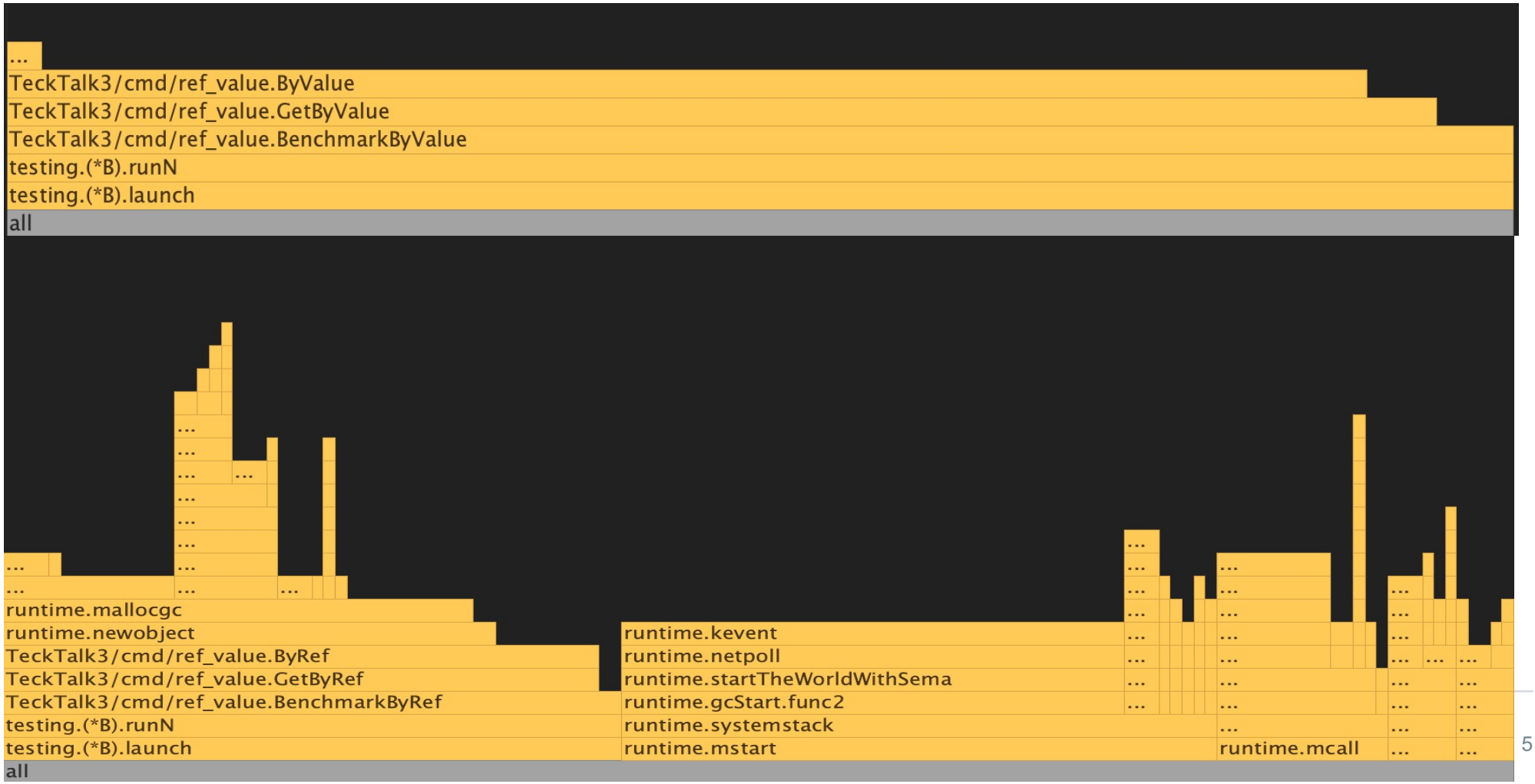
```go
//go:noinline
func ByRef(name string) *Account {
    acc := Account{
        name:    name,
        address: "some account address here",
        details: "some account details here",
        id:      12312312,
        zipCode: 143010,
    }

    return &acc
}
```

```go
//go:noinline
func ByValue(name string) Account {
    return Account{
        name:    name,
        address: "some account address here",
        details: "some account details here",
        id:      12312312,
        zipCode: 143010,
    }
}
```

```
⇒  go test -v -bench=. -benchmem
goos: darwin
goarch: amd64
pkg: TeckTalk3/cmd/ref_value
cpu: Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz
BenchmarkByRef
BenchmarkByRef-12       26352846                39.25 ns/op           64 B/op          1 allocs/op
BenchmarkByValue
BenchmarkByValue-12     154288635                7.856 ns/op           0 B/op          0 allocs/op
PASS
ok      TeckTalk3/cmd/ref_value 3.375s
```

CPU Profile

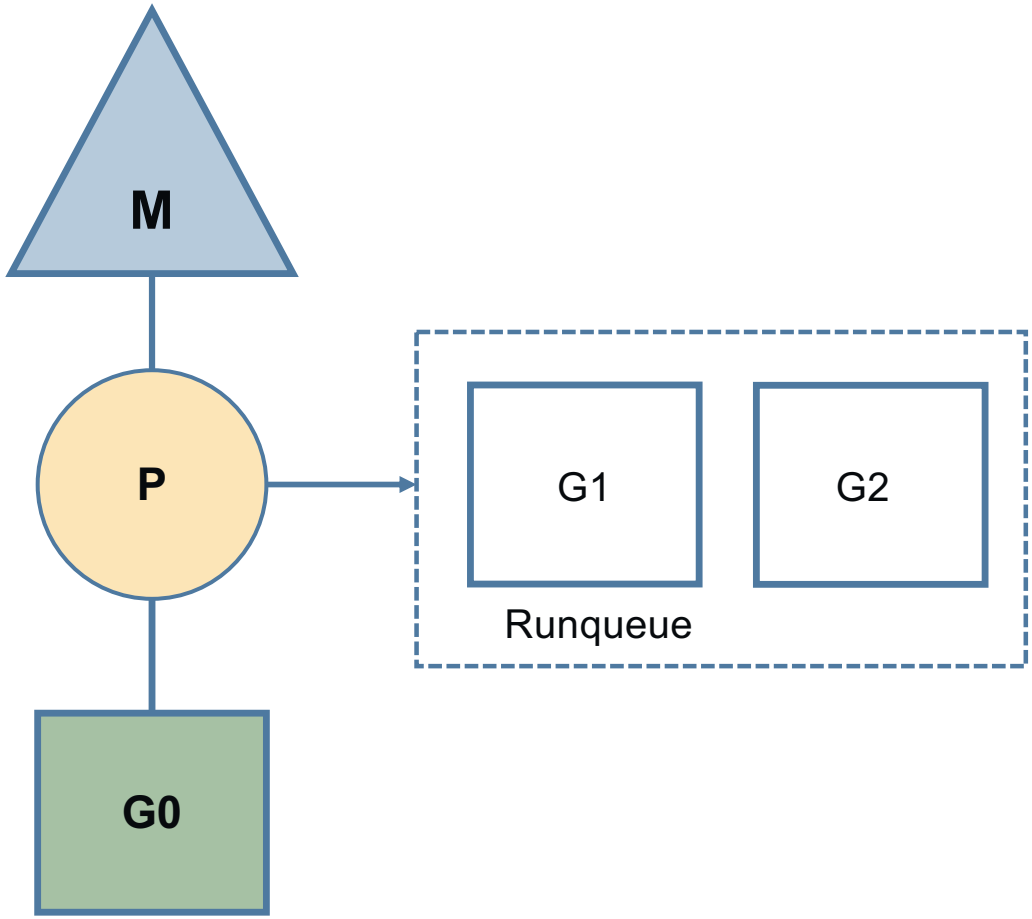

...
TeckTalk3/cmd/ref_value.ByValue
TeckTalk3/cmd/ref_value.GetByValue
TeckTalk3/cmd/ref_value.BenchmarkByValue
testing.(*B).runN
testing.(*B).launch
all
runtime.mallocgc
runtime.newobject
TeckTalk3/cmd/ref_value.ByRef
TeckTalk3/cmd/ref_value.GetByRef
TeckTalk3/cmd/ref_value.BenchmarkByRef
testing.(*B).runN
testing.(*B).launch
runtime.kevent
runtime.netpoll
runtime.startTheWorldWithSema
runtime.gcStart.func2
runtime.systemstack
runtime.mstart
runtime.mcall
all

Concurrency

- Multiple OS threads
- Multiple goroutines
- User interact only with gorounites

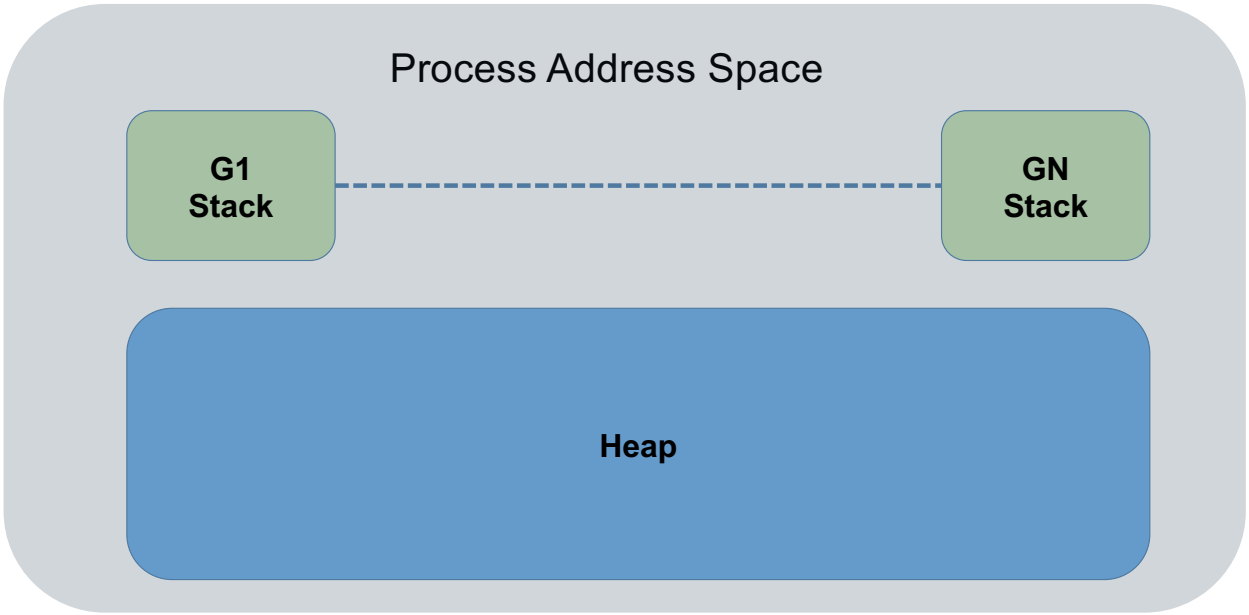
G M P

- M (machine) – OS thread
- P (processor) – Execution Context, holds runqueue
- G – Goroutine



The Go stack and heap

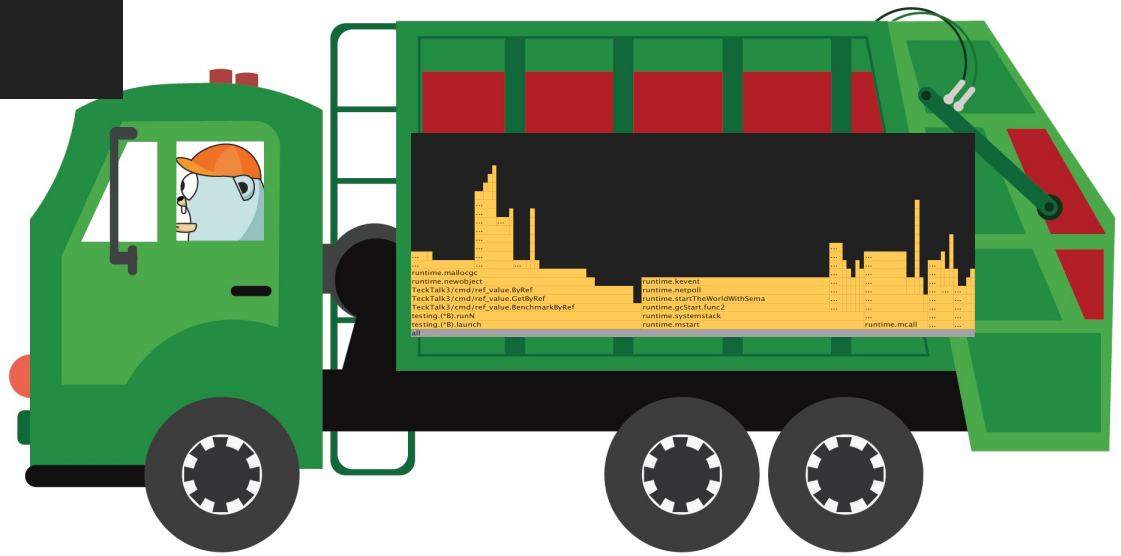
- Threads managed by the OS. Go use goroutines.
- Goroutines exist within user space - runtime, and not the OS, sets the rules of how stacks behave.

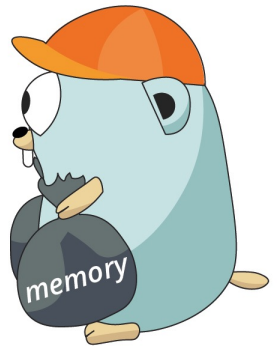


Magic Inside

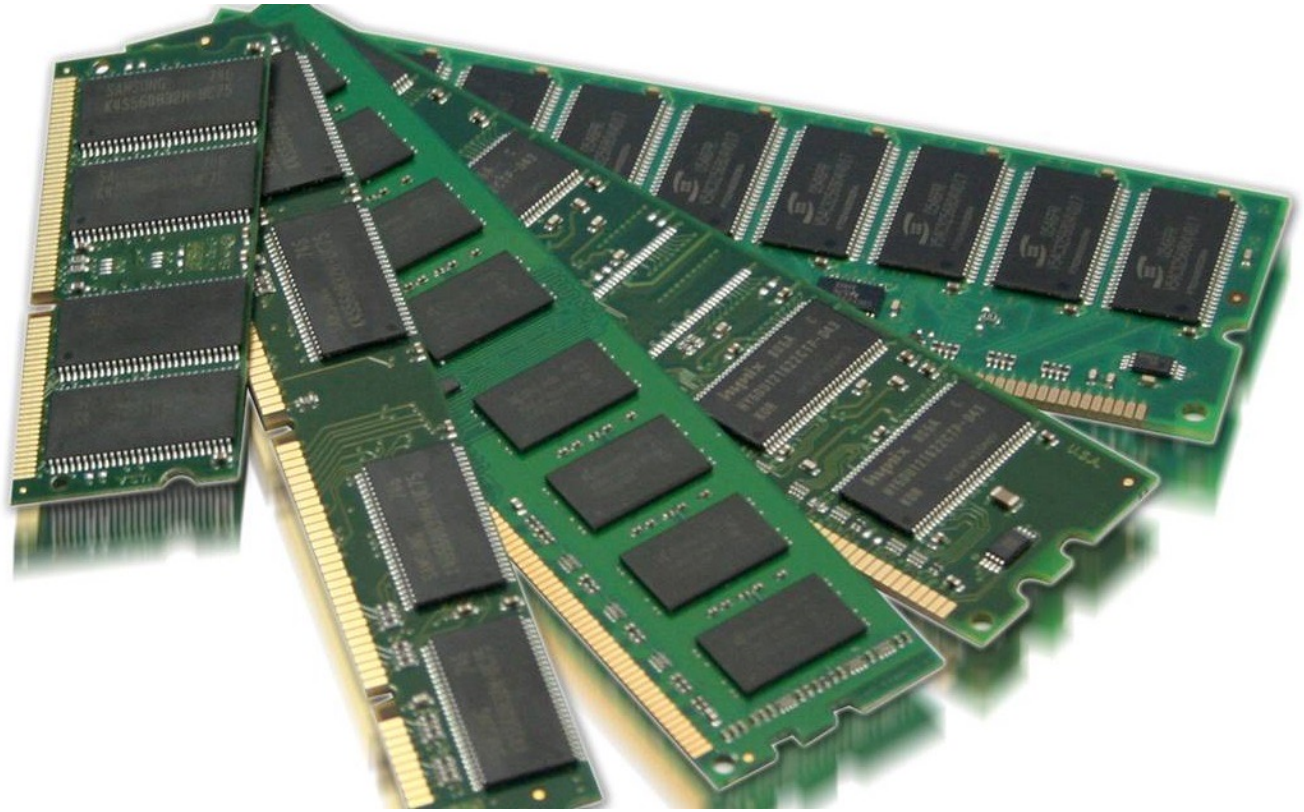
```go
//go:noinline
func ByRef(name string) *Account {
    acc := Account{
        name:    name,
        address: "some account address here",
        details: "some account details here",
        id:      12312312,
        zipCode: 143010,
    }

    return &acc
}
```

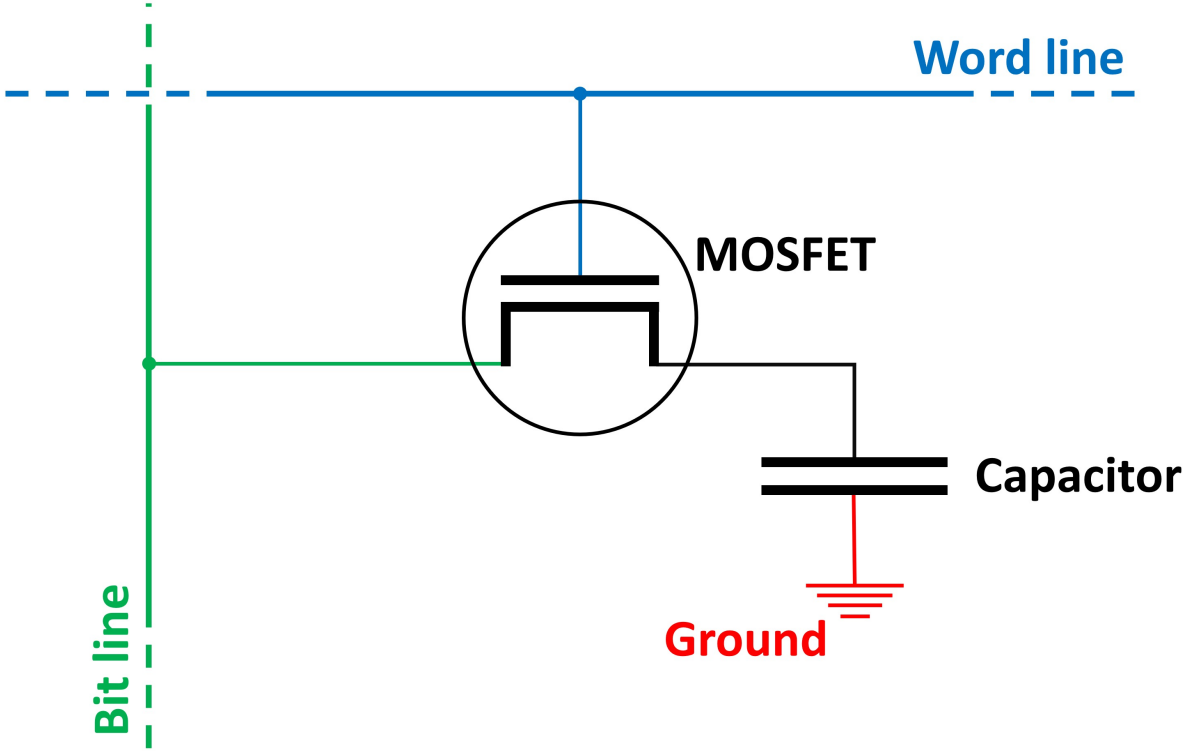




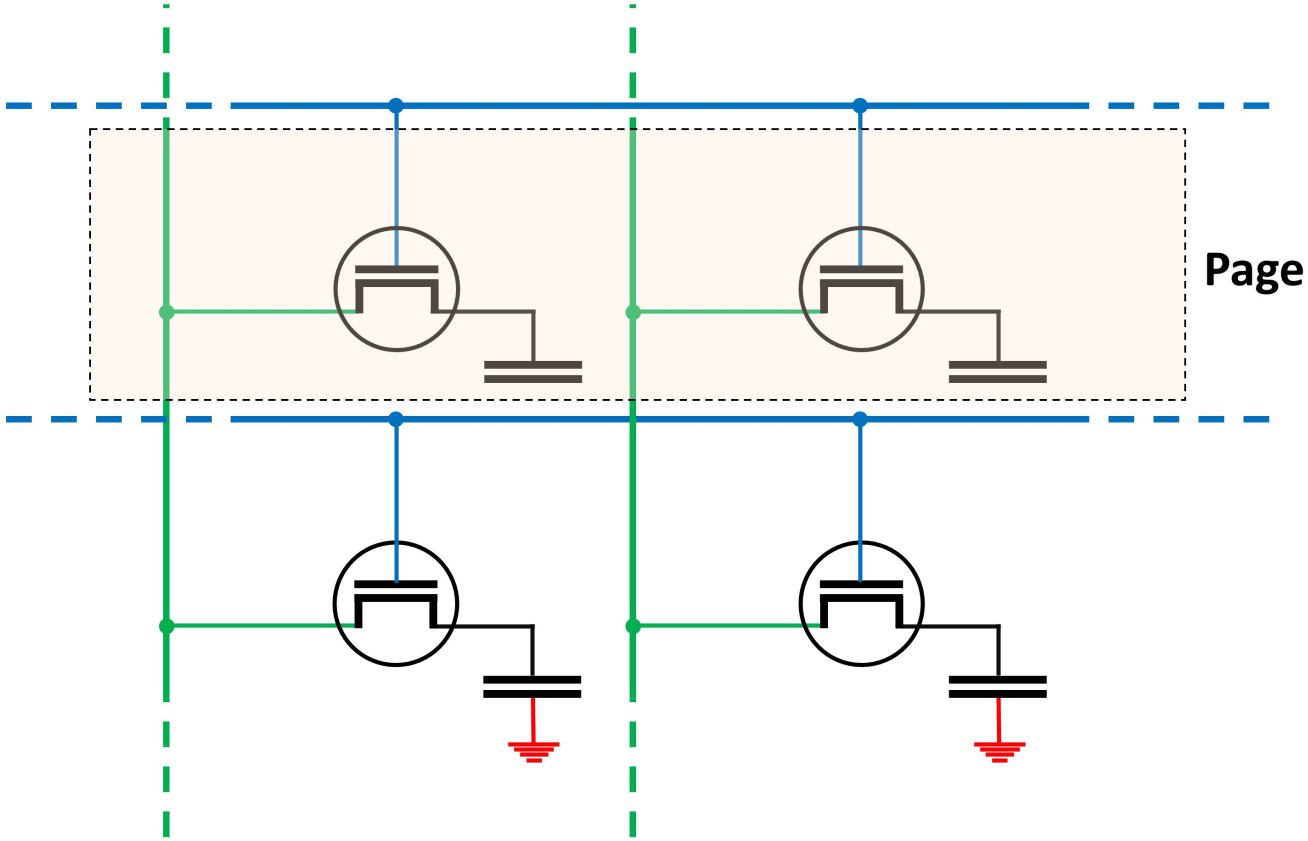
Physical Memory



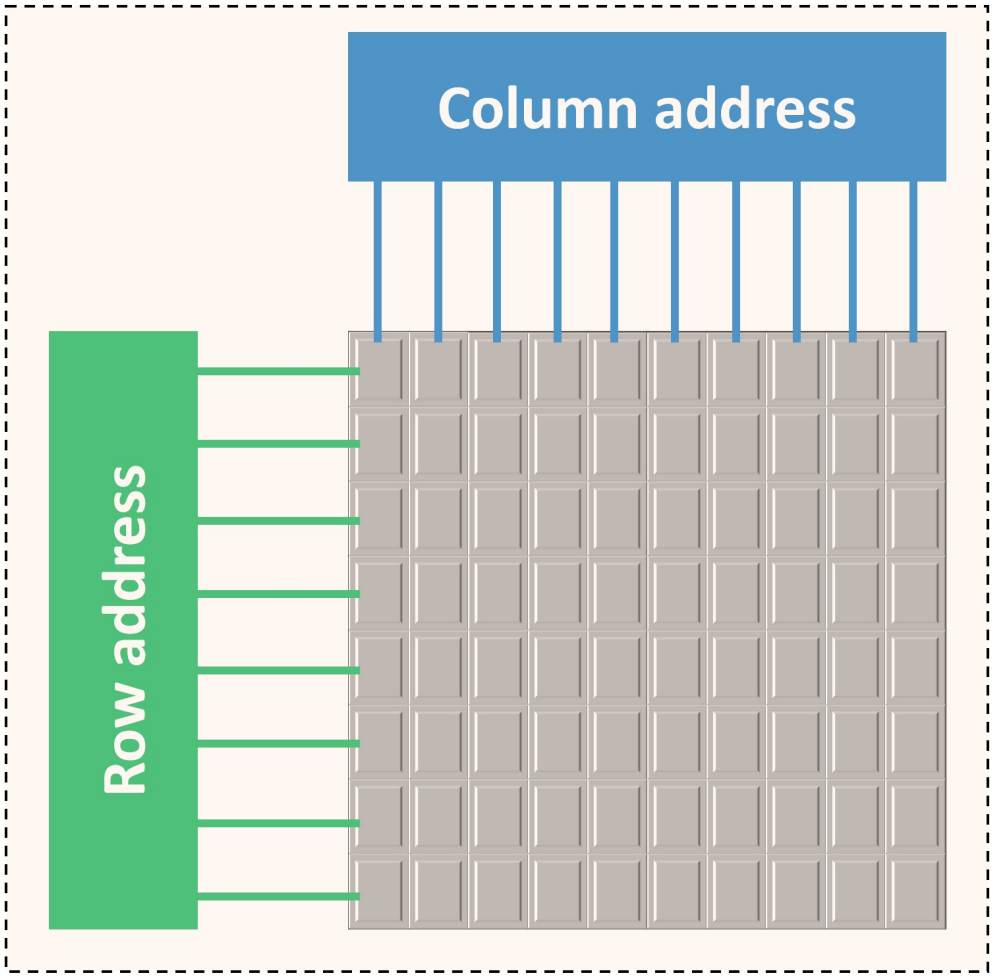
Data Cell



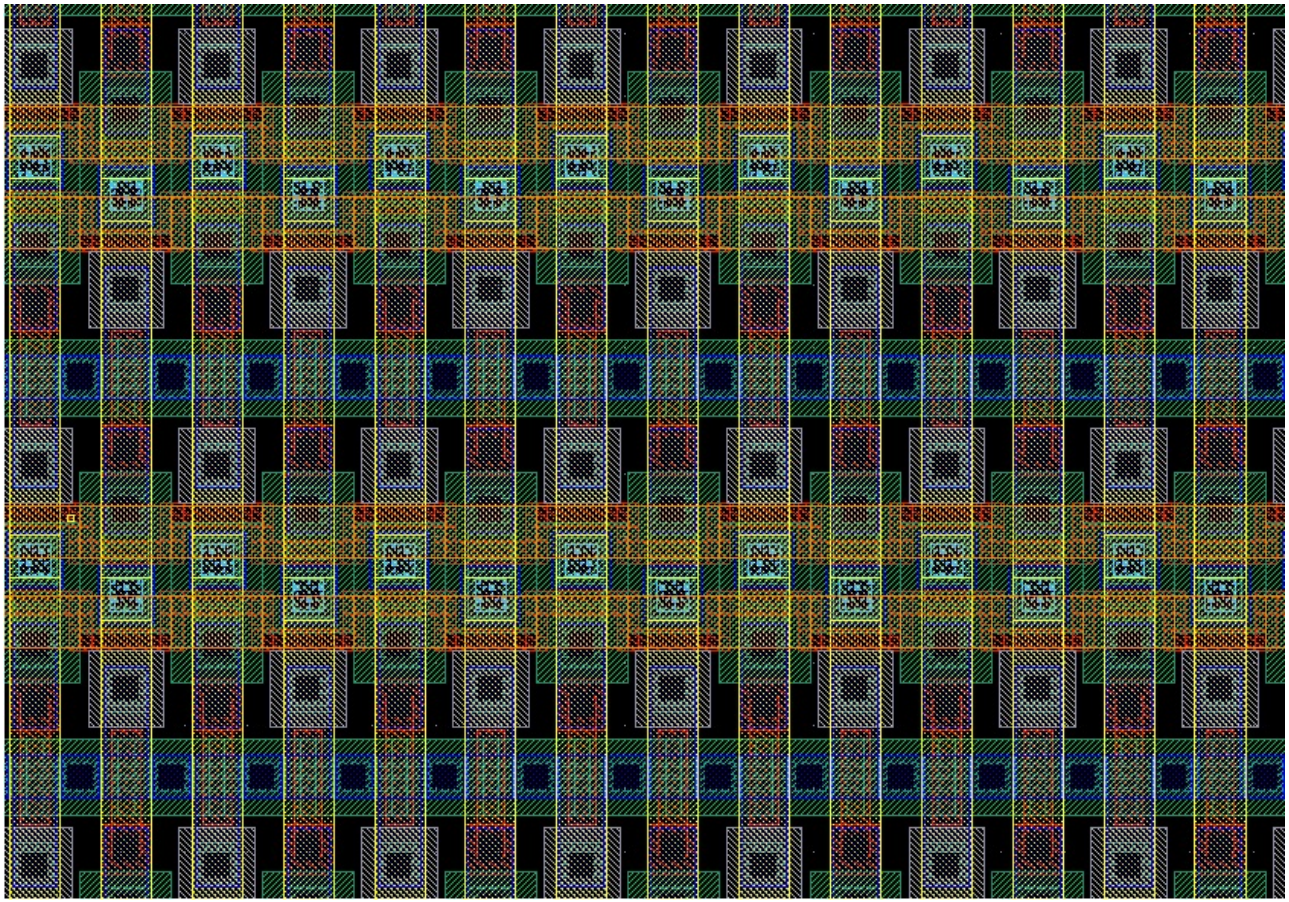
Page



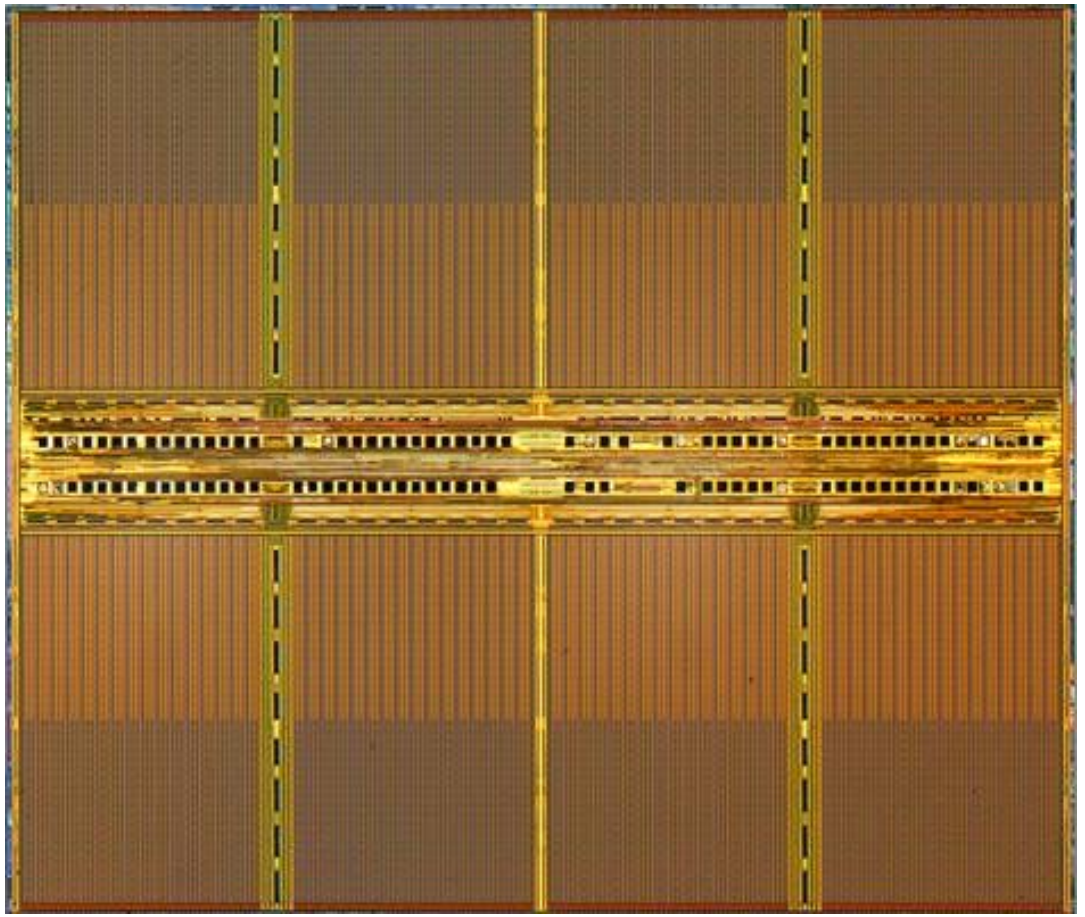
Bank



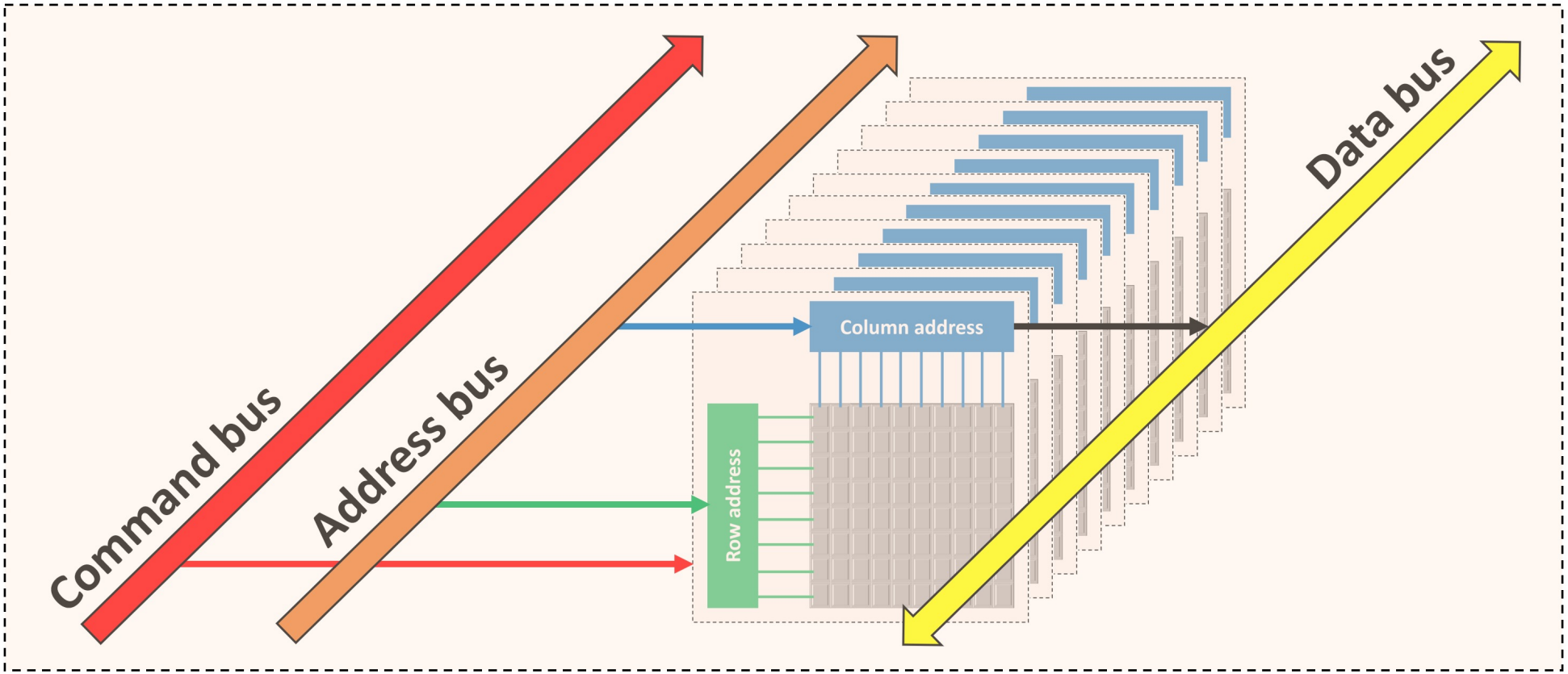
Layout of Memory Bank



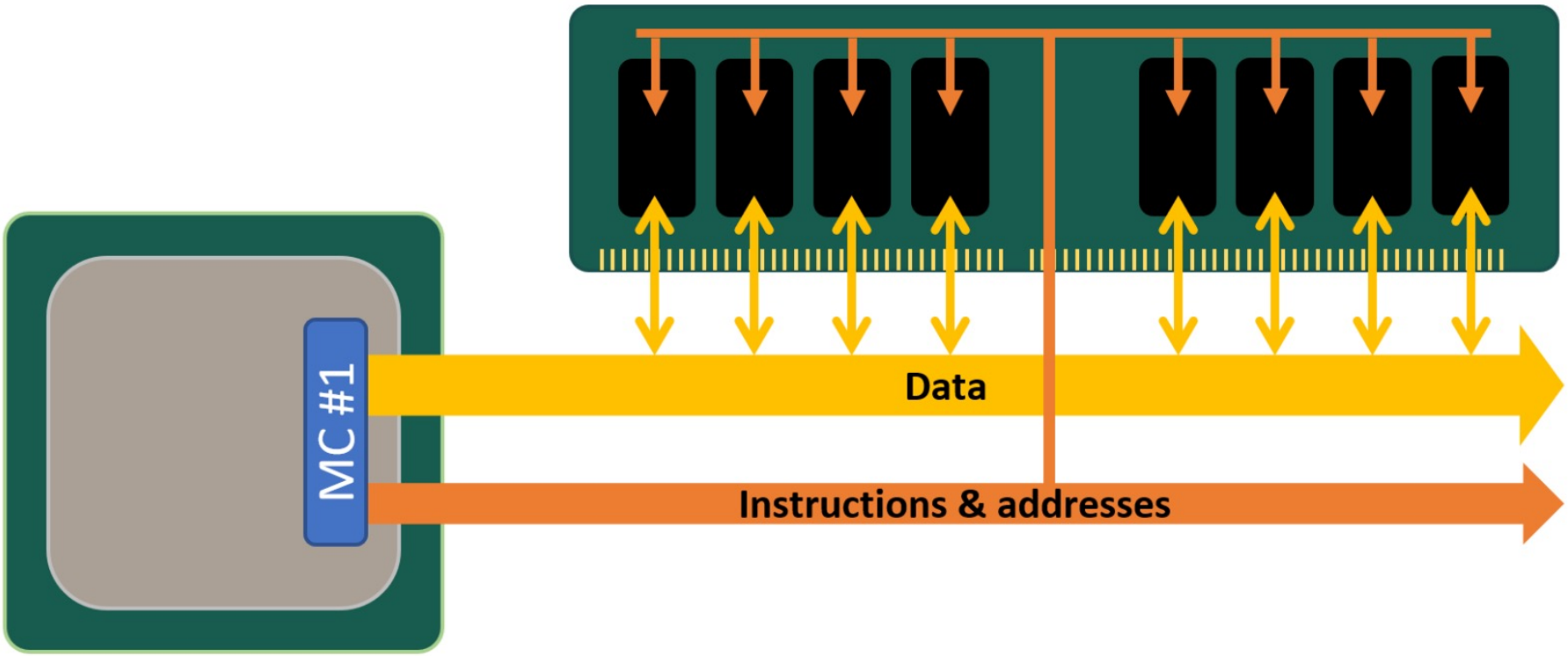
Samsung's DRAM cell



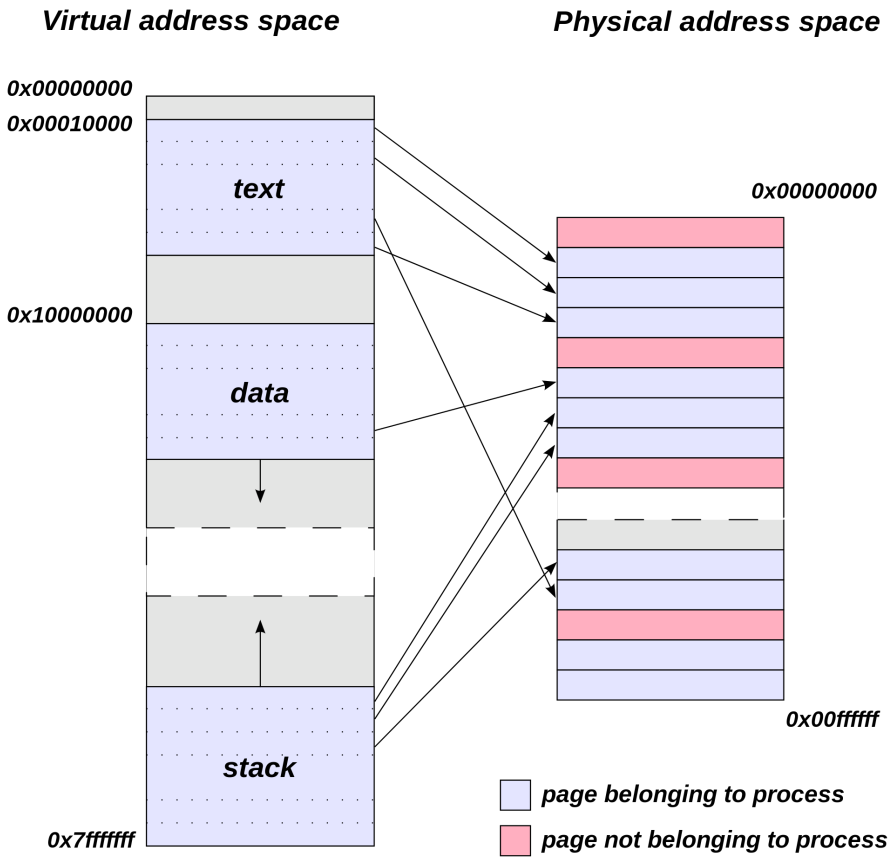
Module

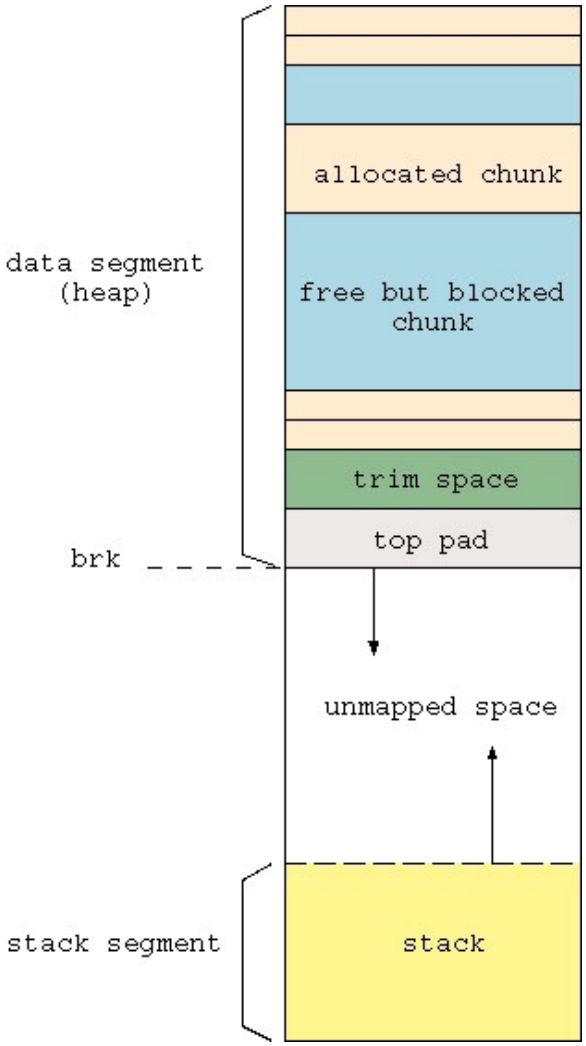


DRAM module

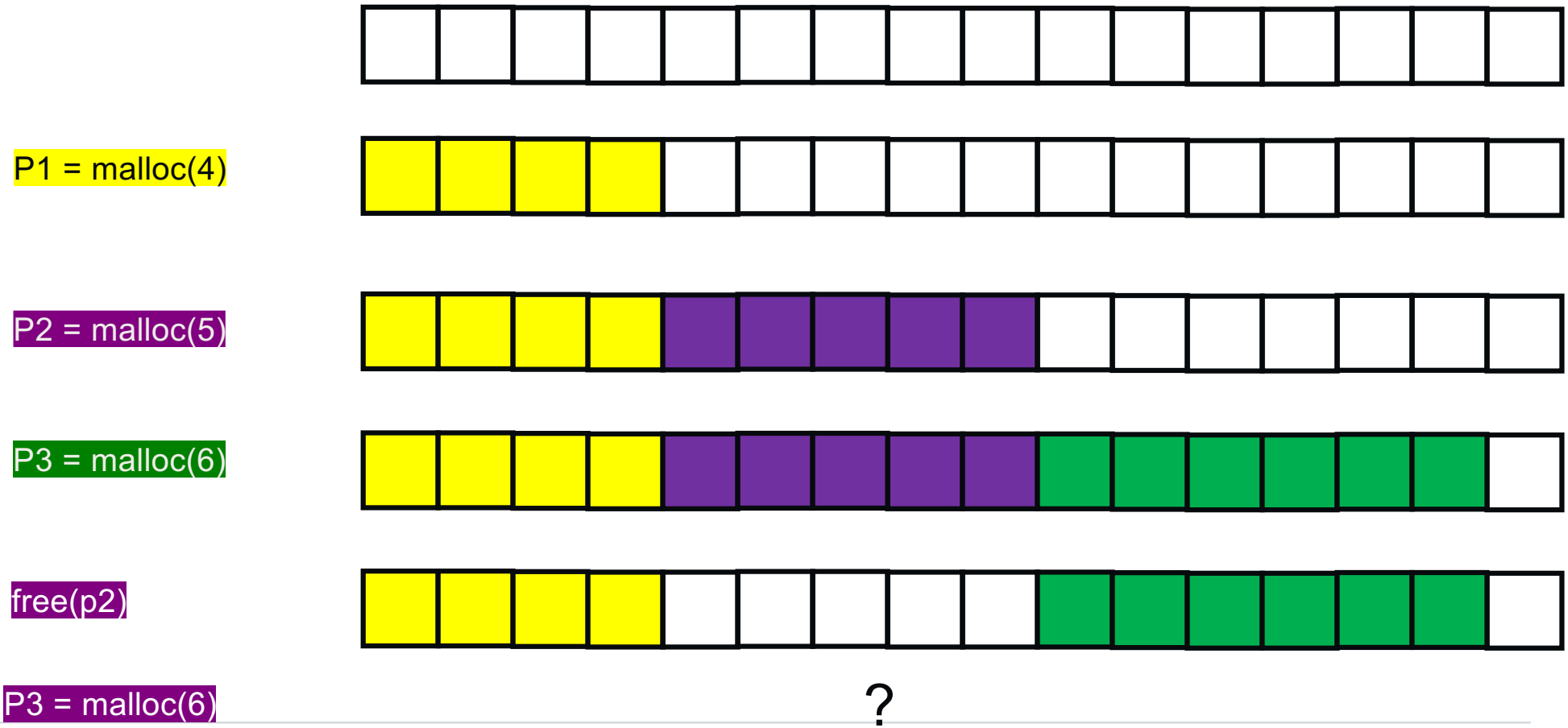


Virtual Address Space And Physical Address Space

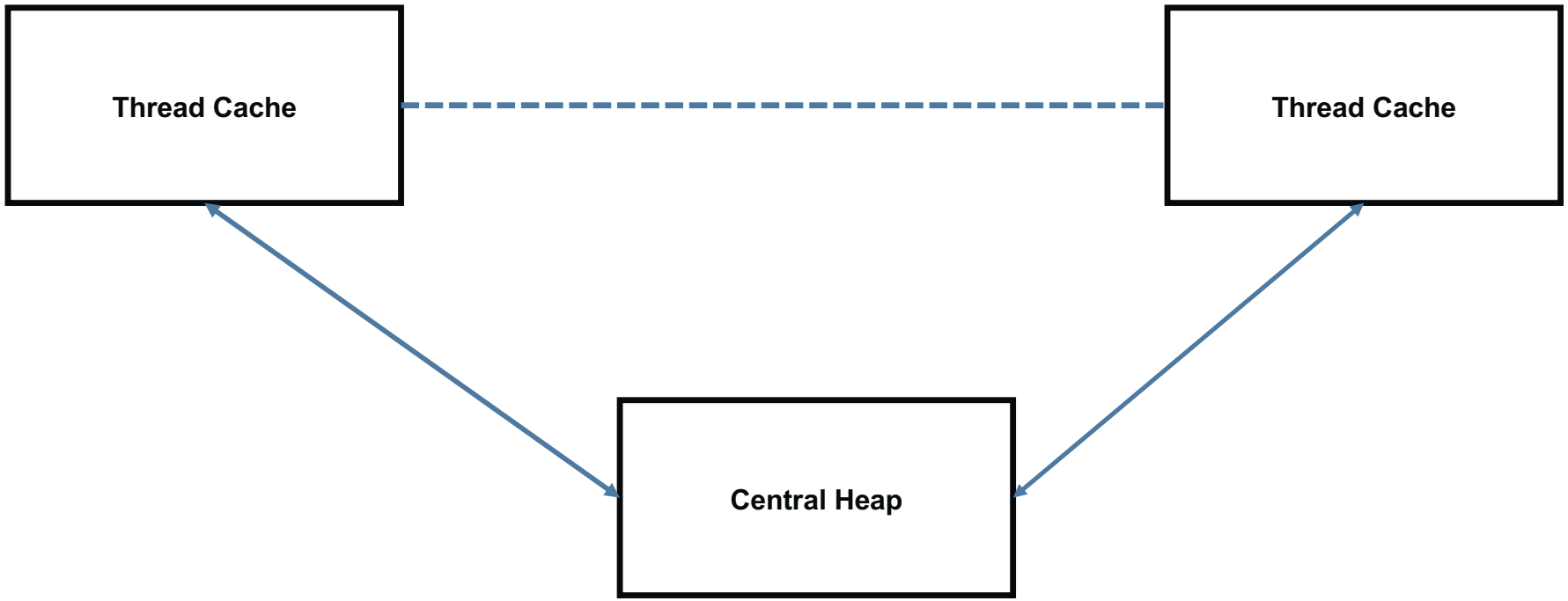


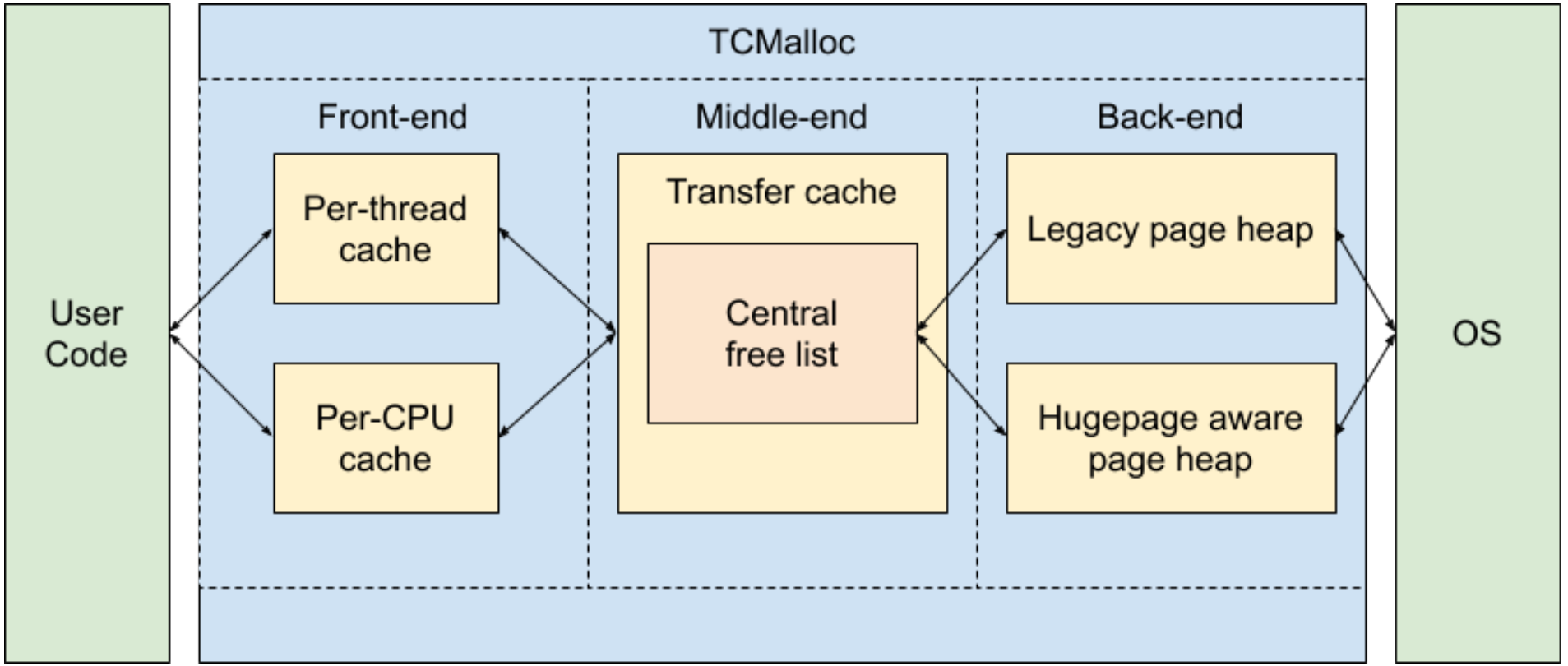


Allocations

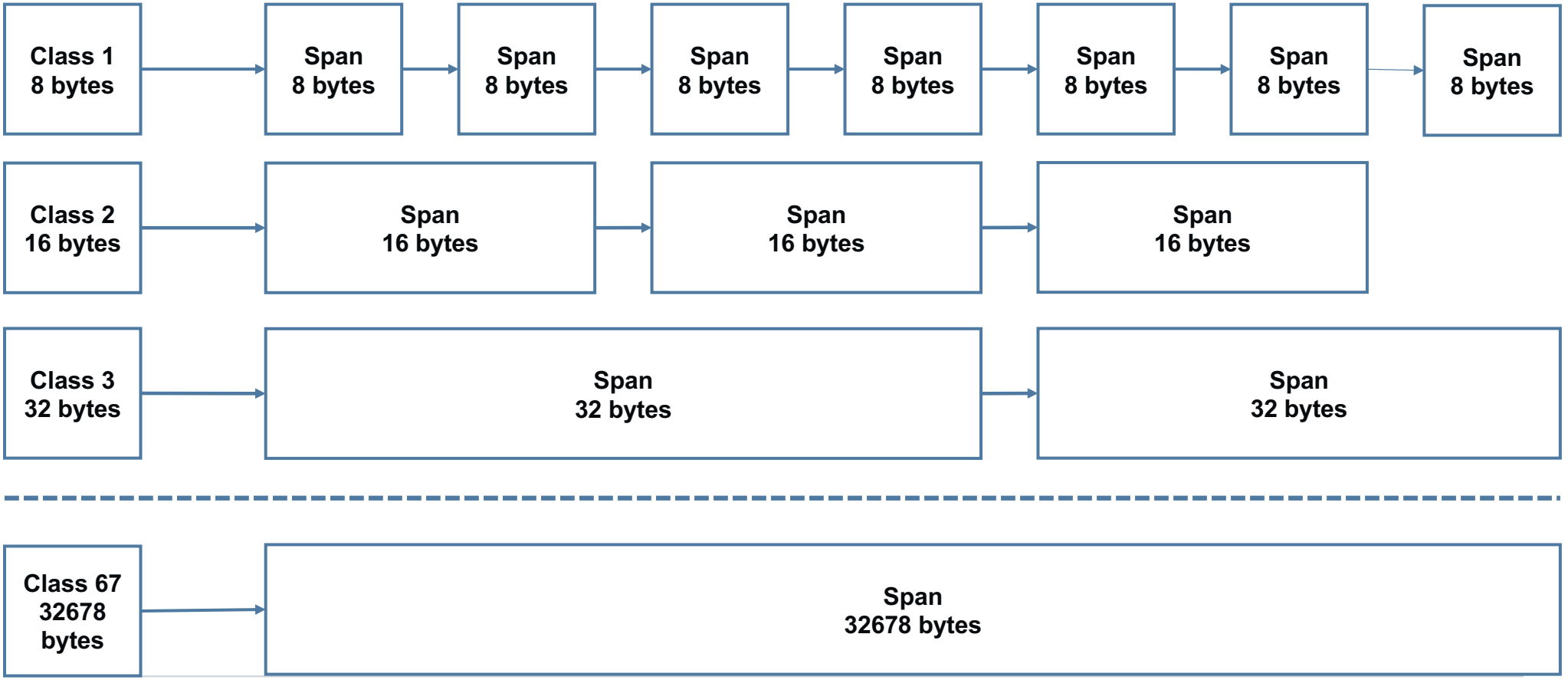


TCMalloc

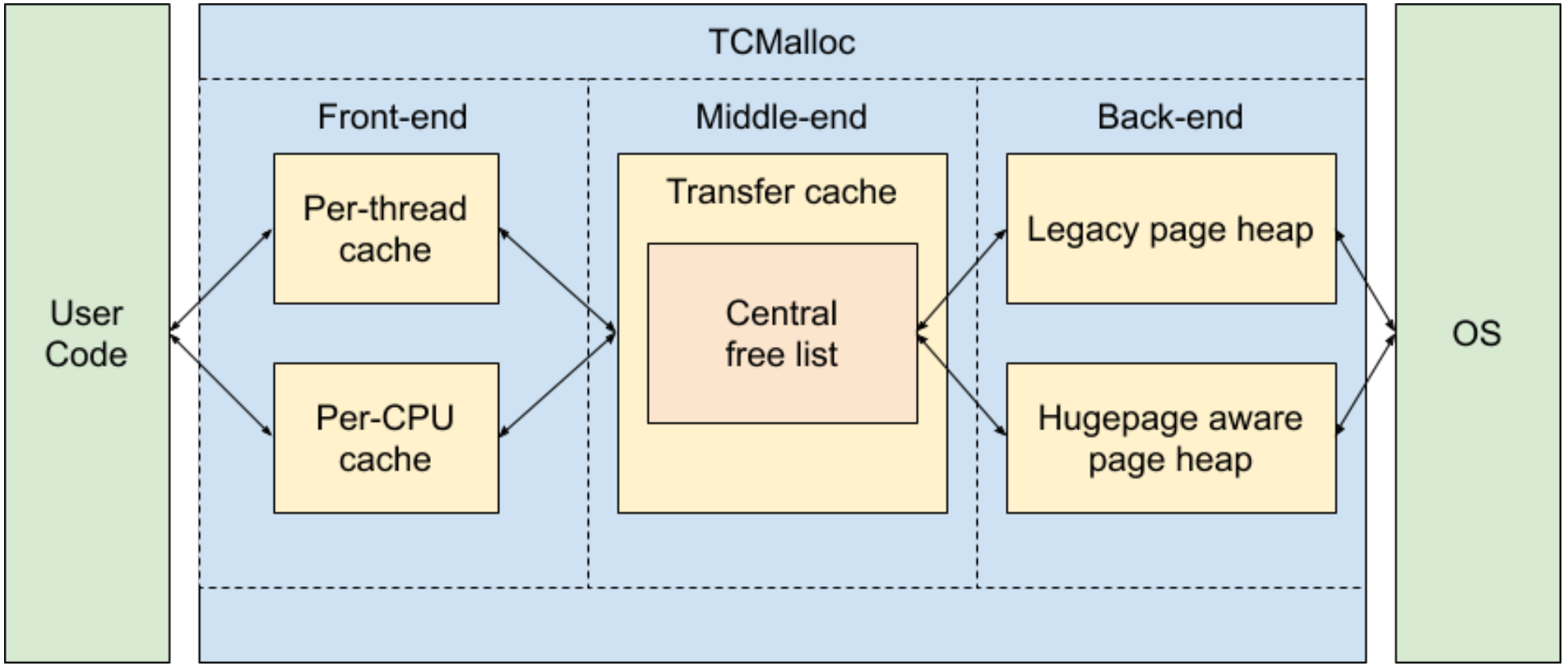




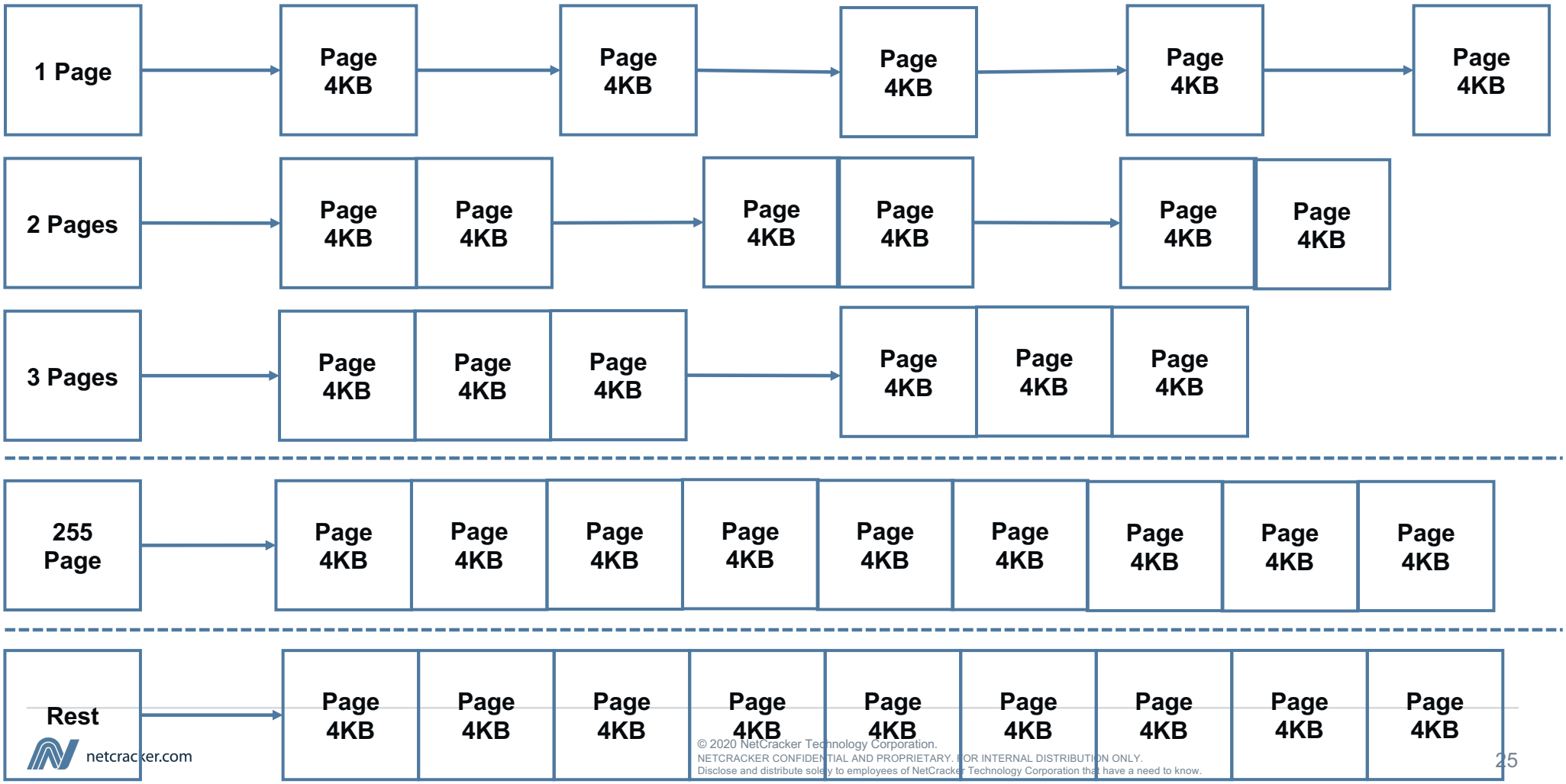
Thread Cache - Frontend



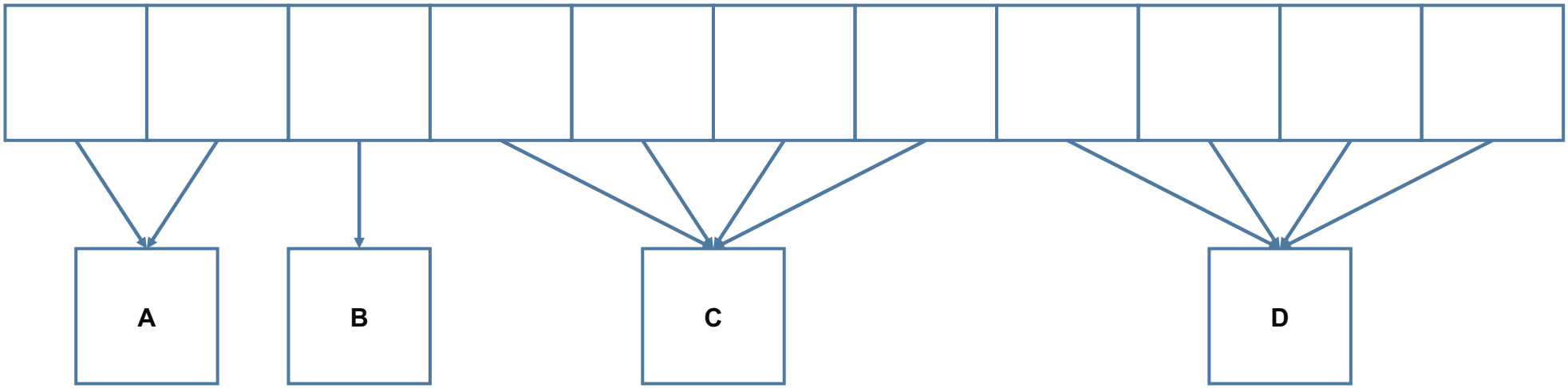
Middle End And Back End

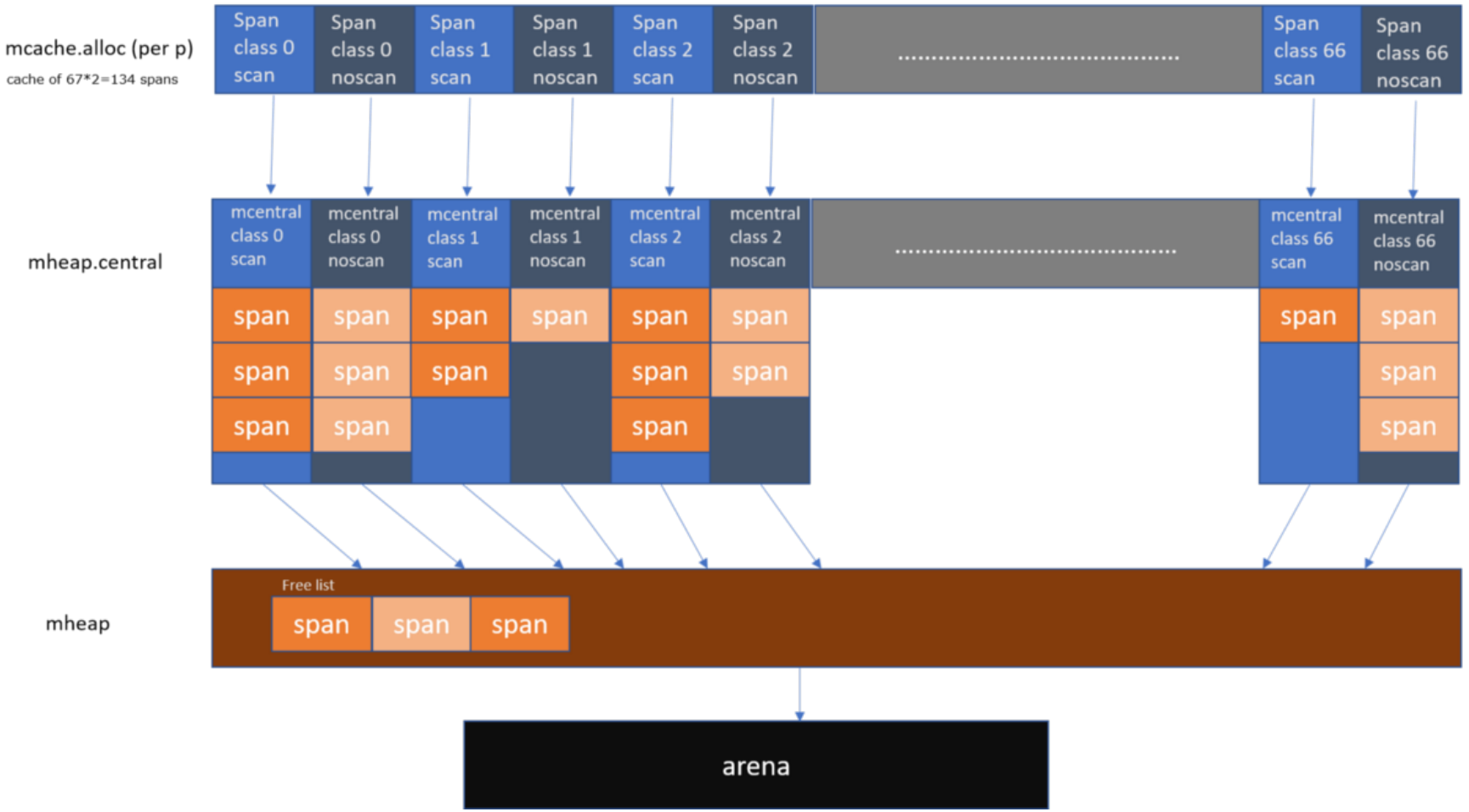


TCMalloc Backend

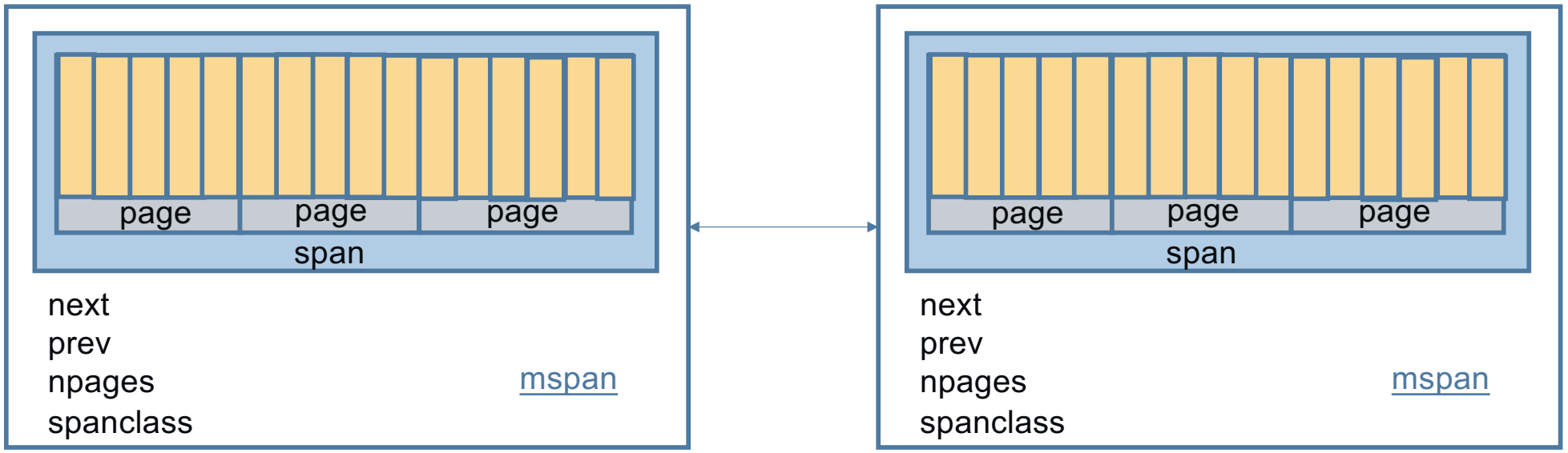


Spans

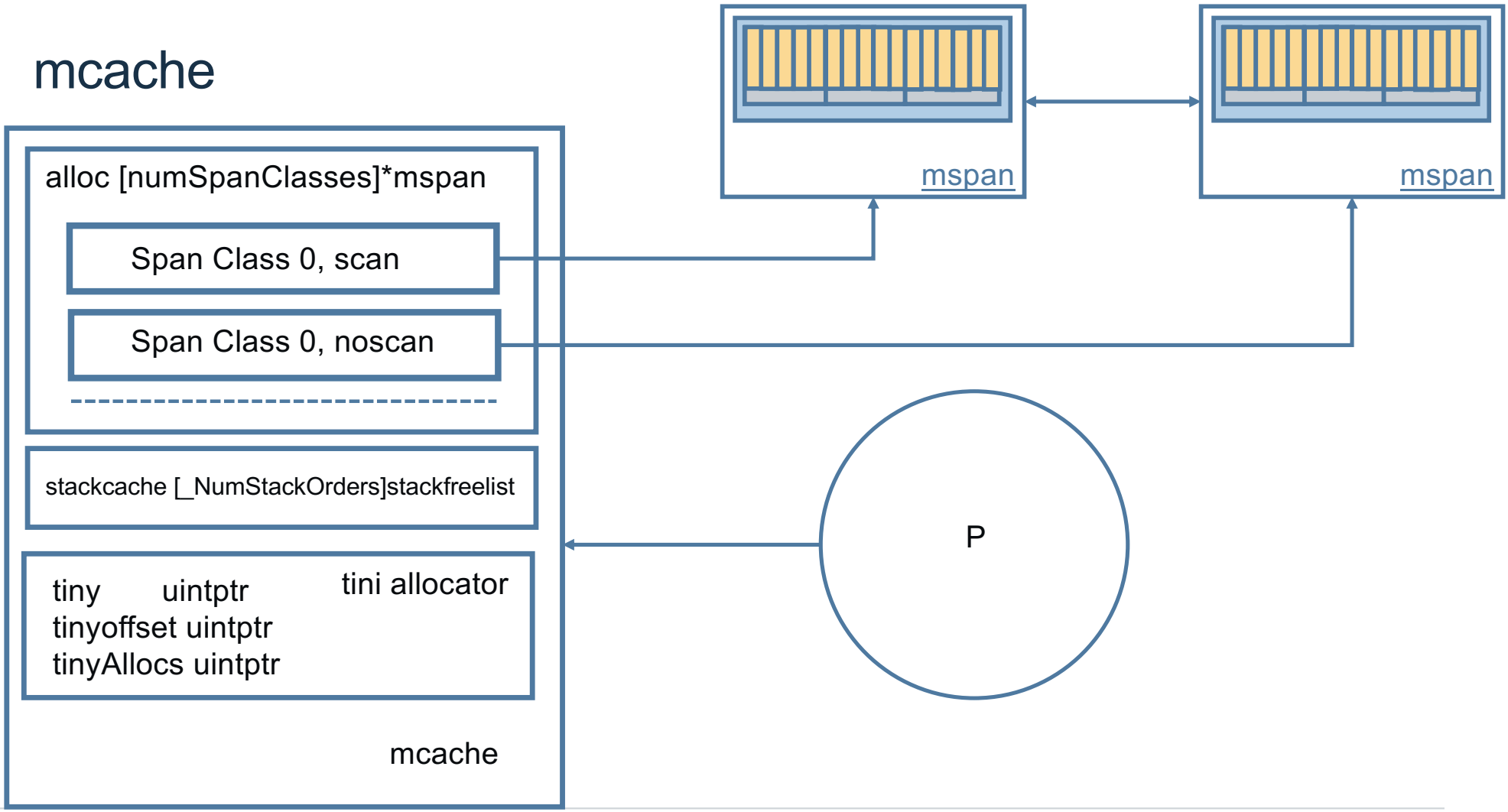


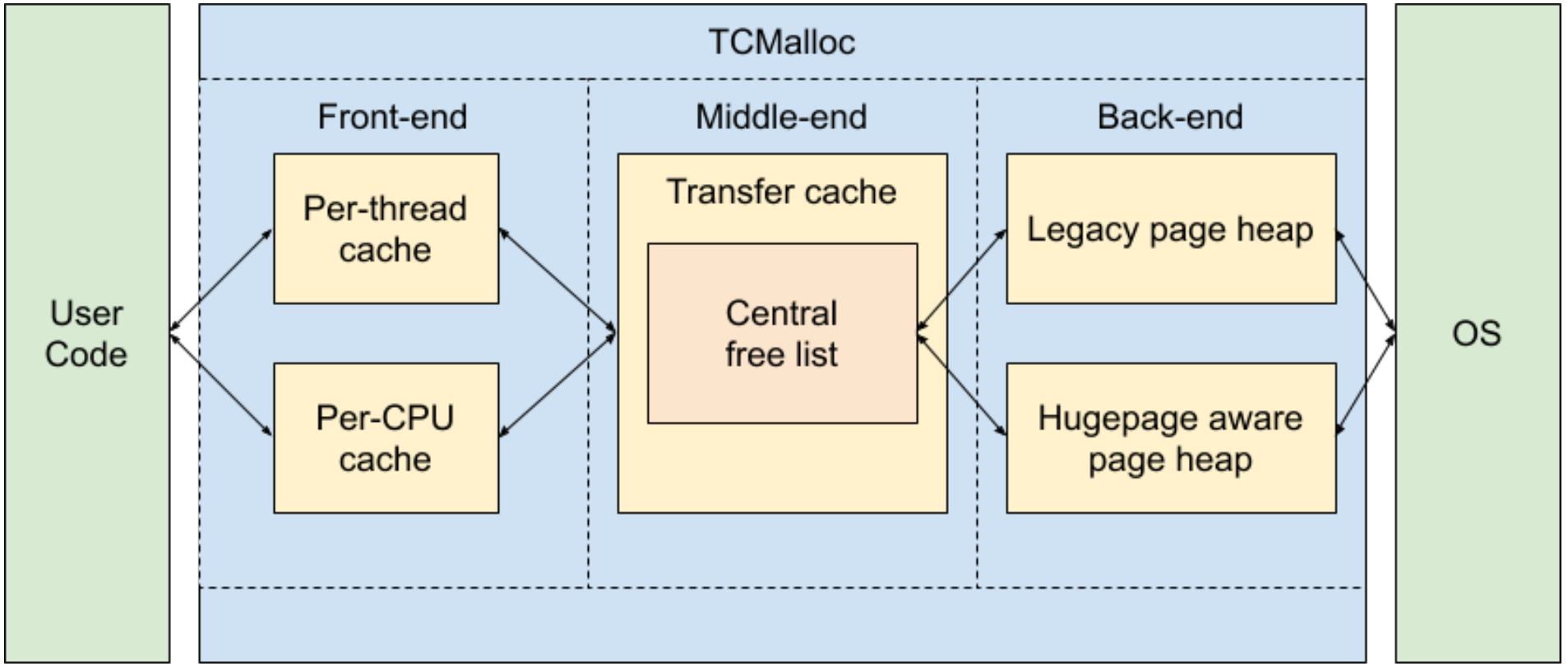


mspans

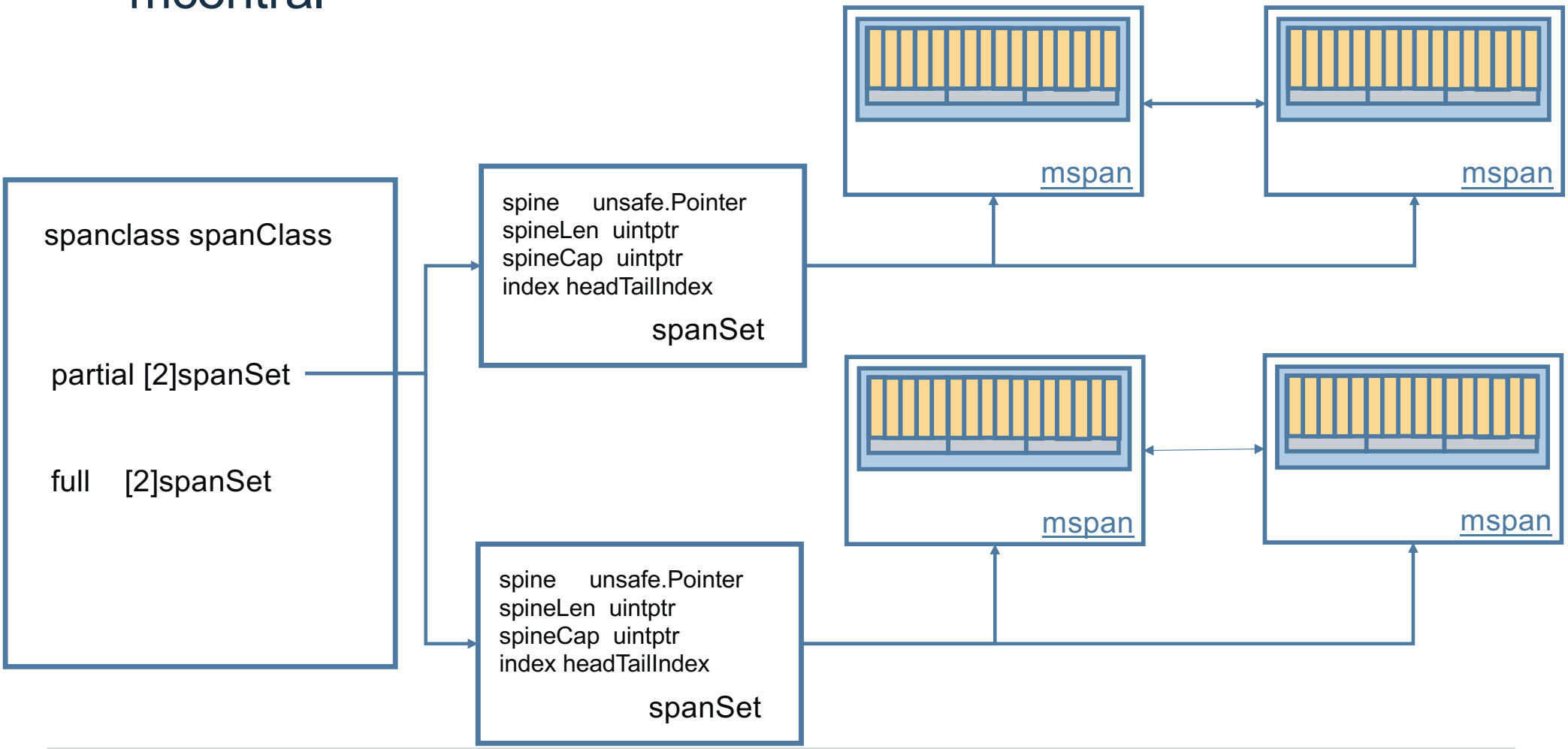


mcache

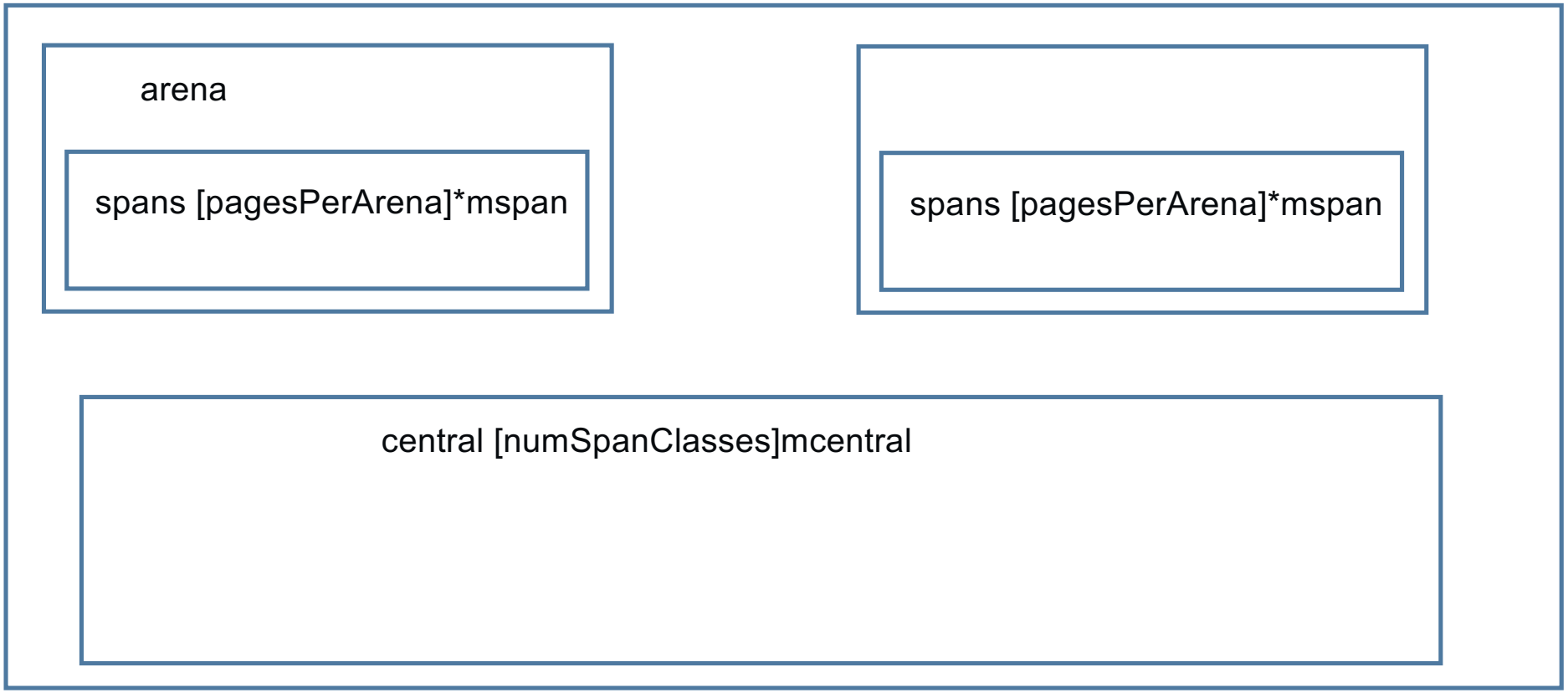




mcentral



mheap



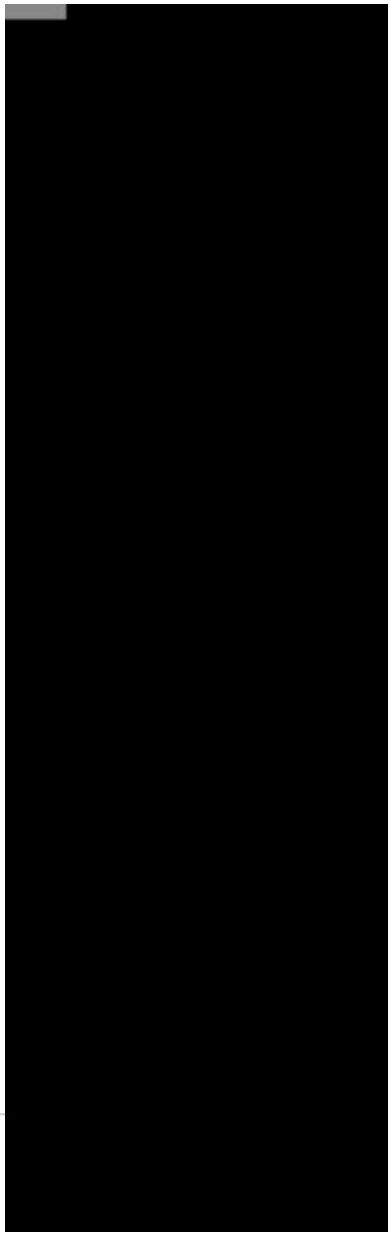
Memory allocation

- Large then 32 KB -> mheap
- Less then 16 KB -> mcache -> tiny allocator
- 16-32 KB -> calculate size class -> mcache -> mcentral -> arena -> OS

GC Types

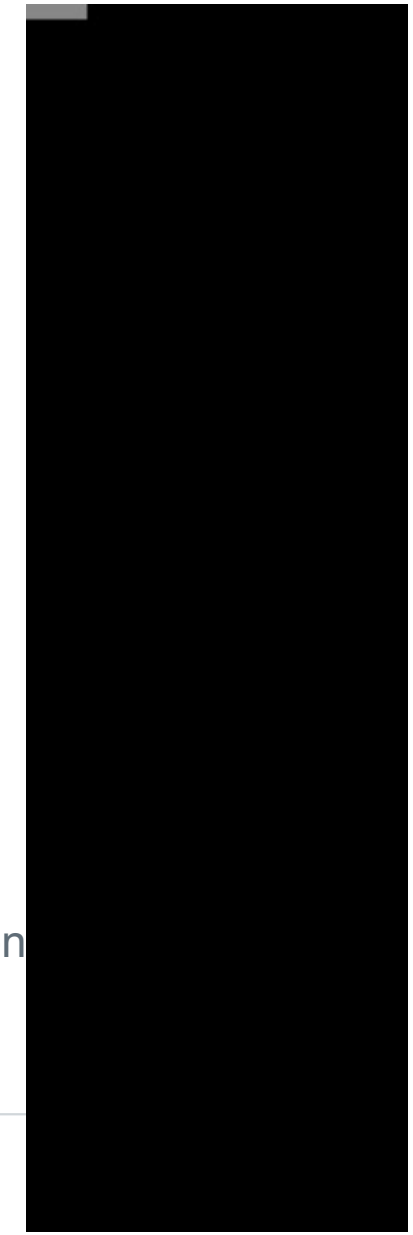
Free At Exit, aka There's Plenty of RAM

- simplest possible system
- best concurrency (only allocator)
- reasonable for small programs
- definition of "small" increases as technology improves
- no destructors



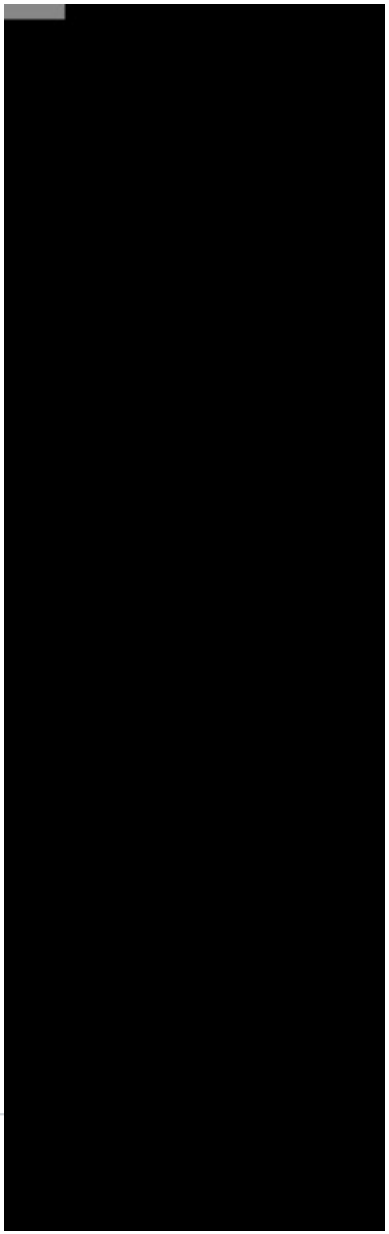
Reference Counting

- synchronous - destructors useful
- accidentally ammortized (but long pauses possible)
- simple concurrency
- possible to retrofit
- expensive in cpu
- needs extra word per object to hold counts
- no cycles
- complicated api and/or leaky abstraction
- expensive allocator (fragmentation, locality)
- threading problems (mutating ref counts - no read-only data!, destructor runs on random thread)
- iOS, file systems



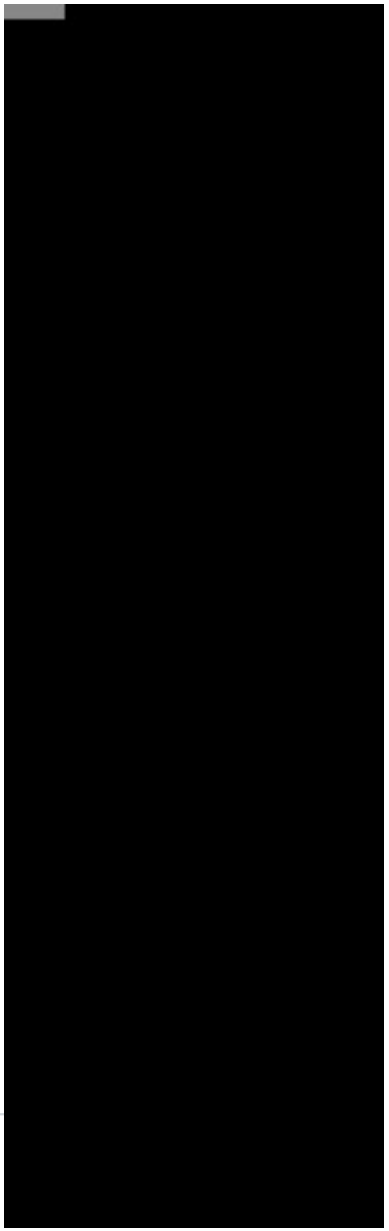
Mark-Sweep Collector

- traversal required
- simple
- destructors easy, but delayed
- conservative option (traversal can be approximated)
- possible to retrofit
- asynch
- expensive allocator (fragmentation, locality)
- complicated concurrency (multi-color)
- Lua, Flash, Ruby, Go
- extensions:
- deferred free
- mark tables (examine multiple objects at once)



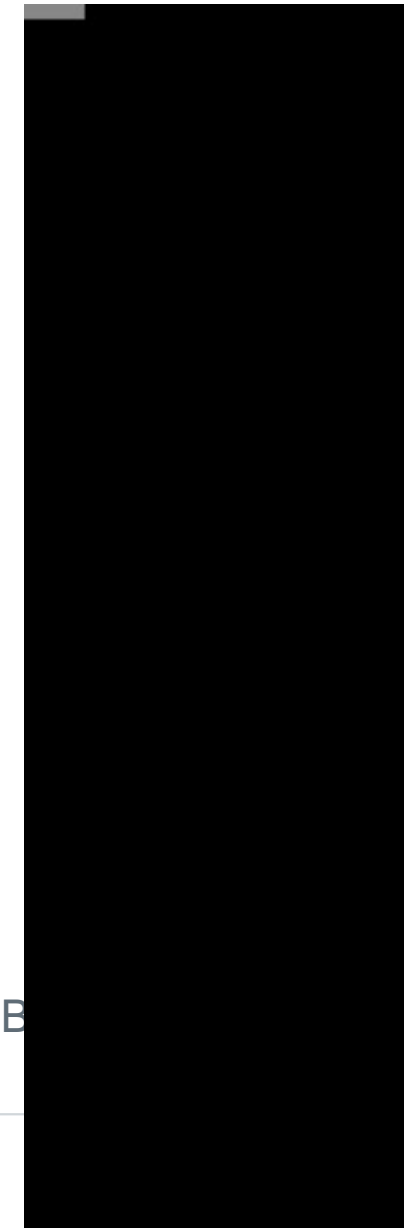
Mark-Compact Collector

- precise traversal required
- simple
- minimum total memory usage
- super cheap allocator ("bump" allocator with only a few instructions)
- moving objects difficult to retrofit
- needs 3 passes, but can trade memory for performance
- very complicated concurrency (multi-color, barriers)
- general algorithm that can make other problems easier
- example: fair random row selection

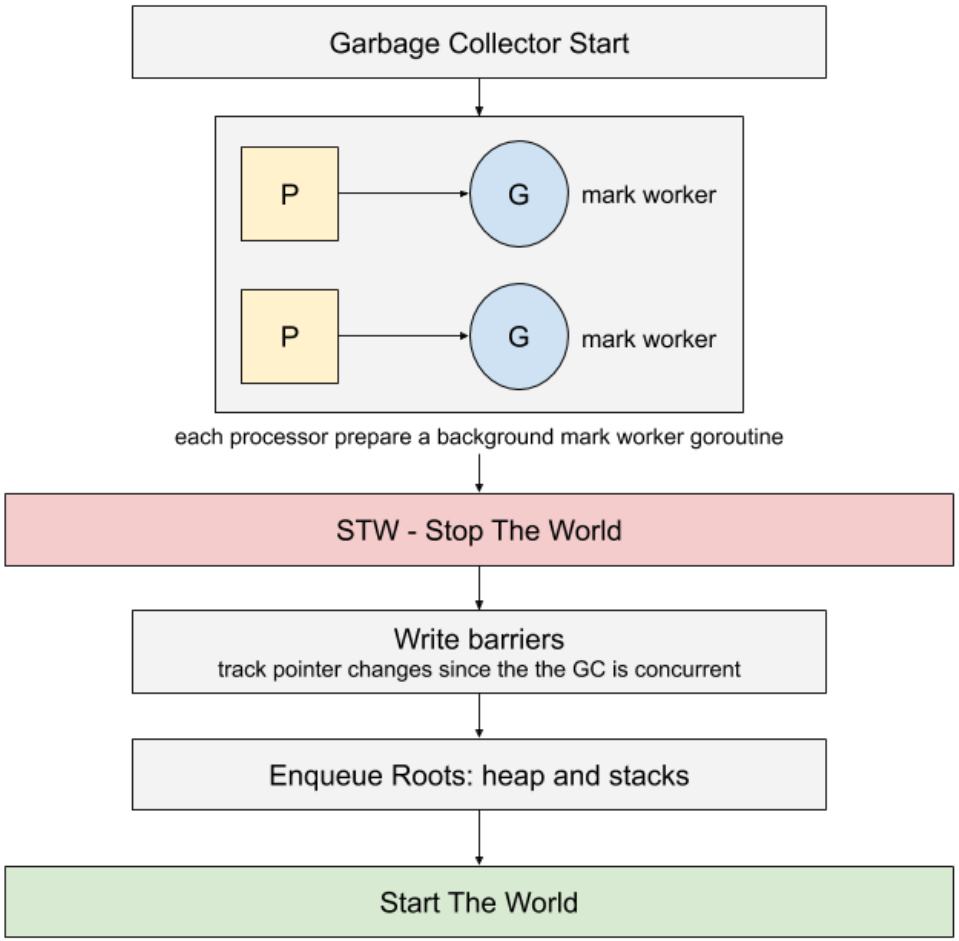


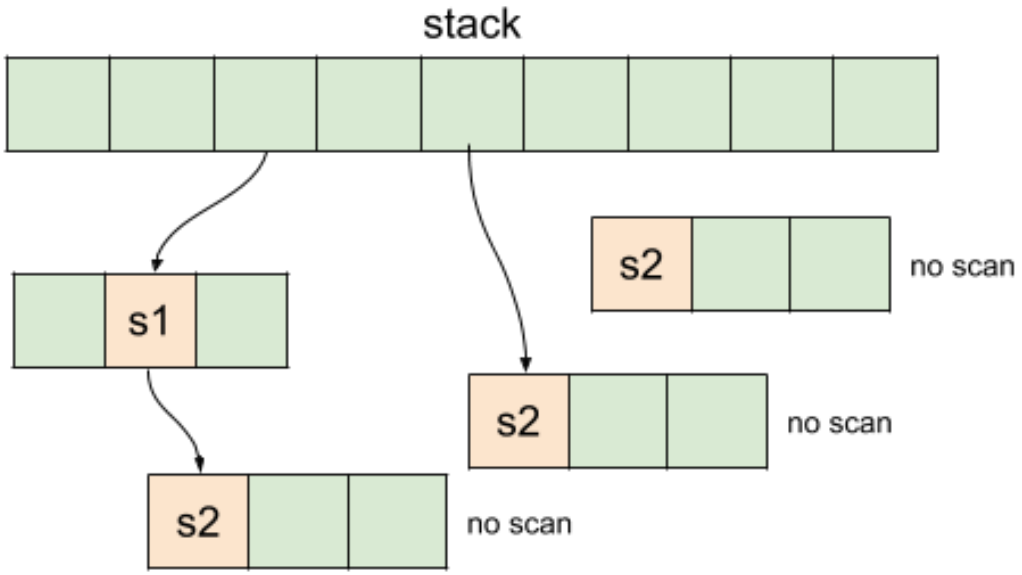
Copying Collector

- precise traversal required
- simplest
- super cheap allocator
- work proportional to live data (garbage doesn't matter!)
- semi-spaces
- no destructors - finalize should not be used
- very complicated concurrency (multi-color, barriers)
- moving objects difficult to retrofit
- common degenerate case: per-transaction pool, delete when done
- most useful gc algorithm whenever you have little live data
- non-memory example: web session storage deletion on Amazon SimpleDB

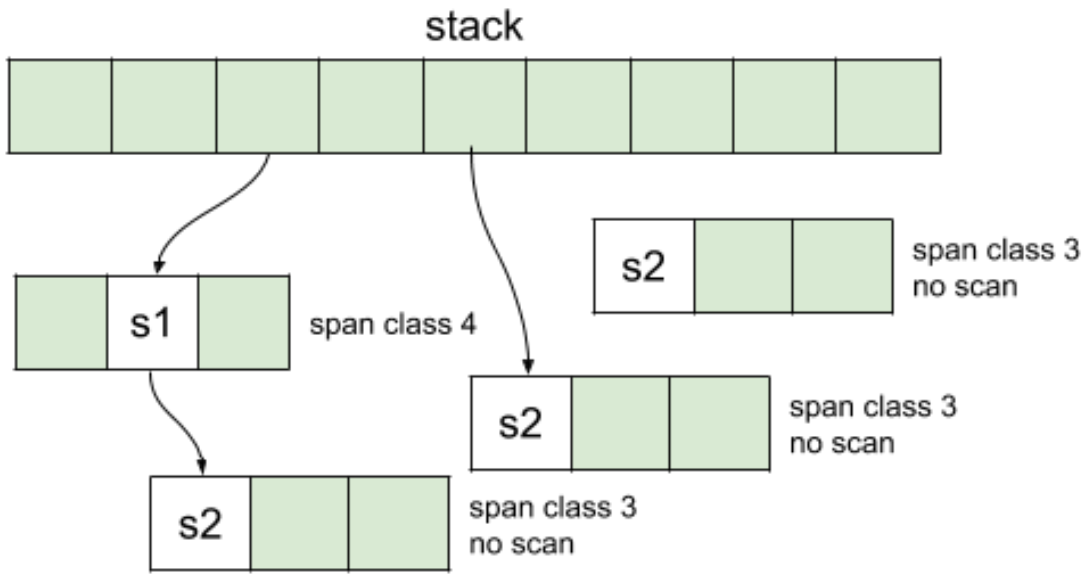


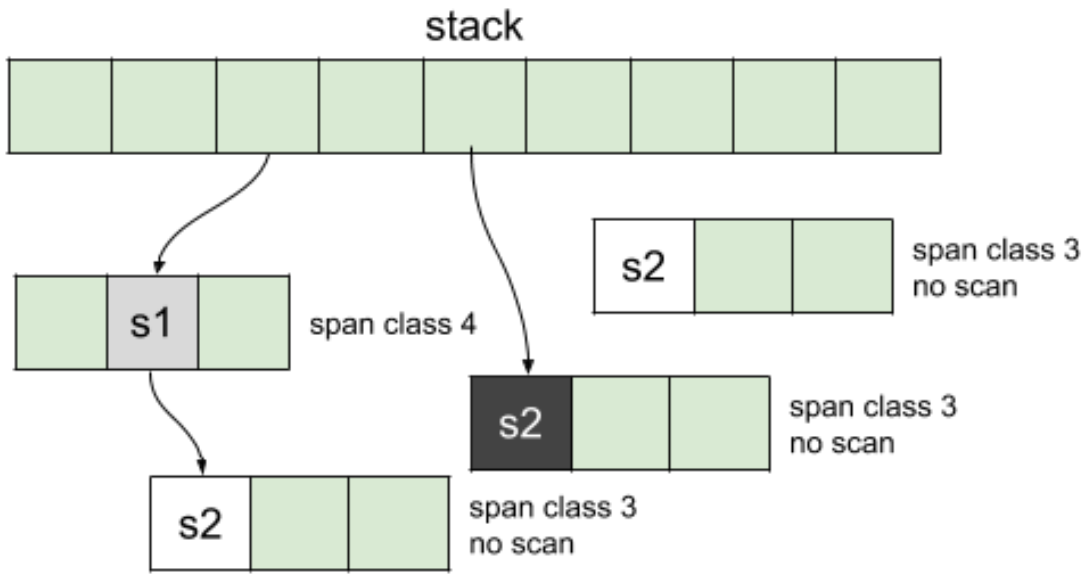
Collector Behavior

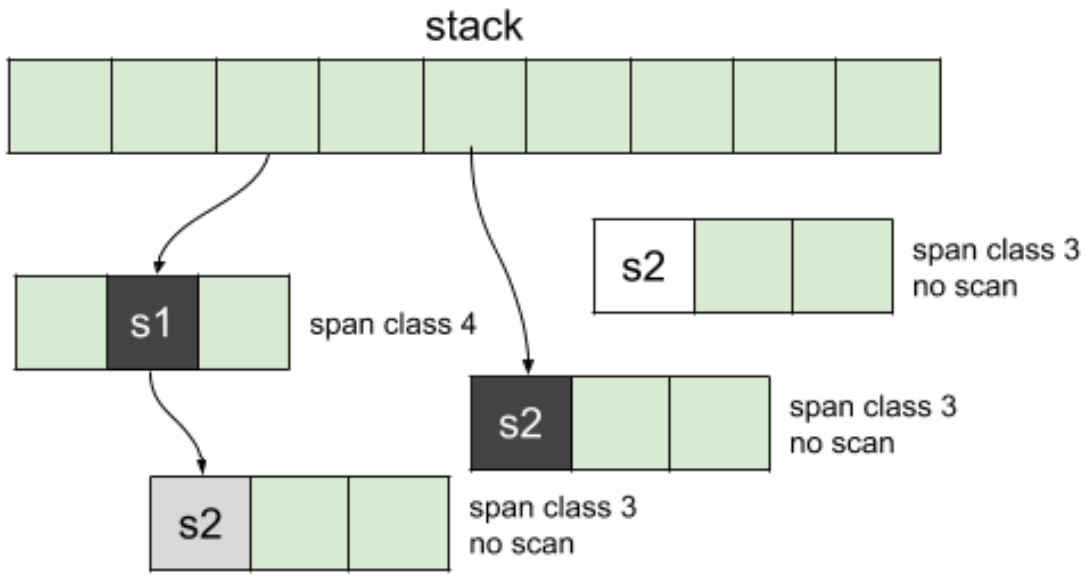


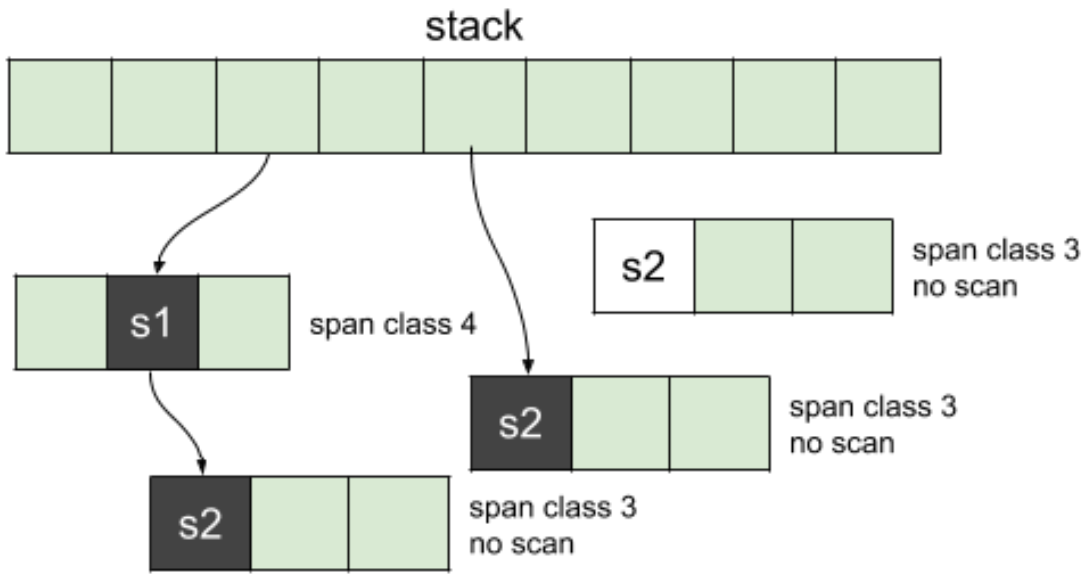


Coloring

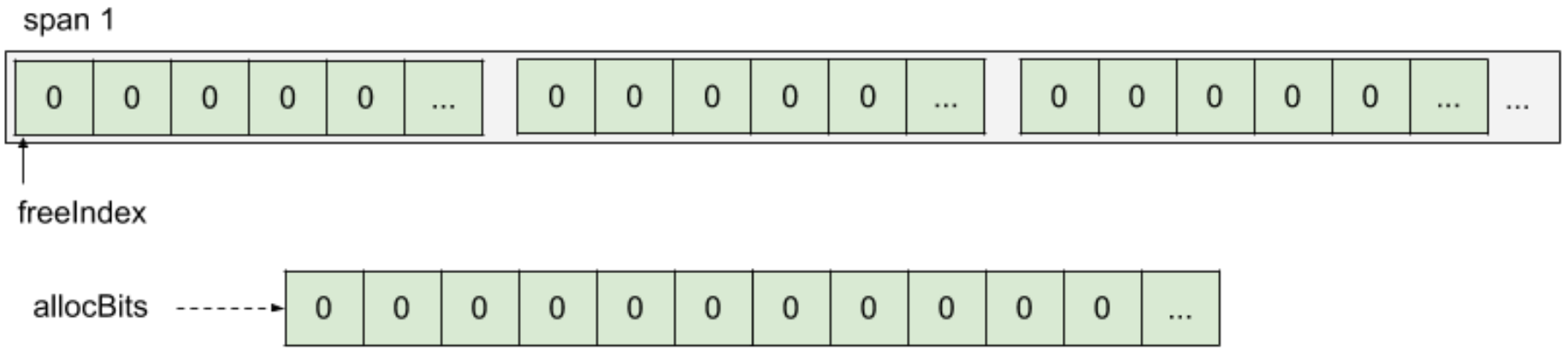


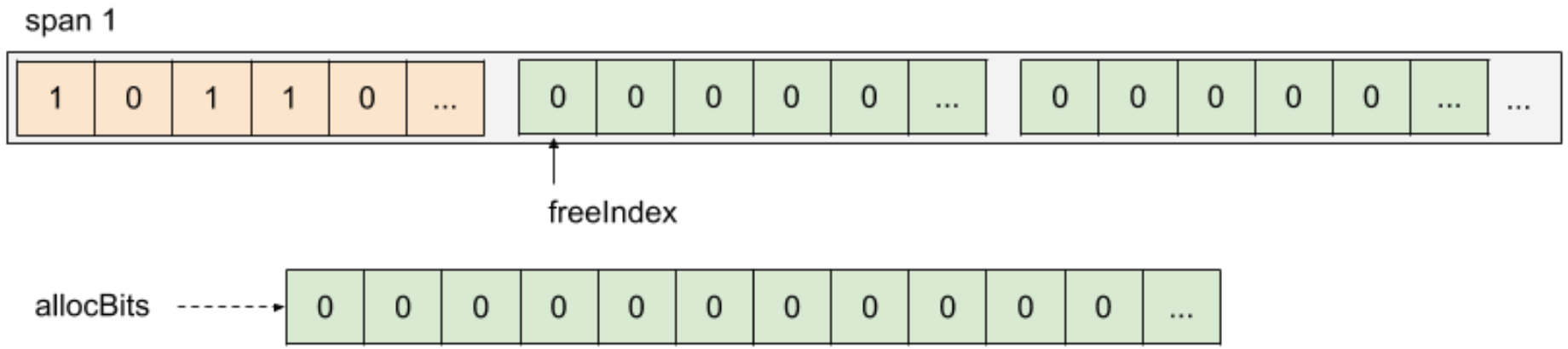


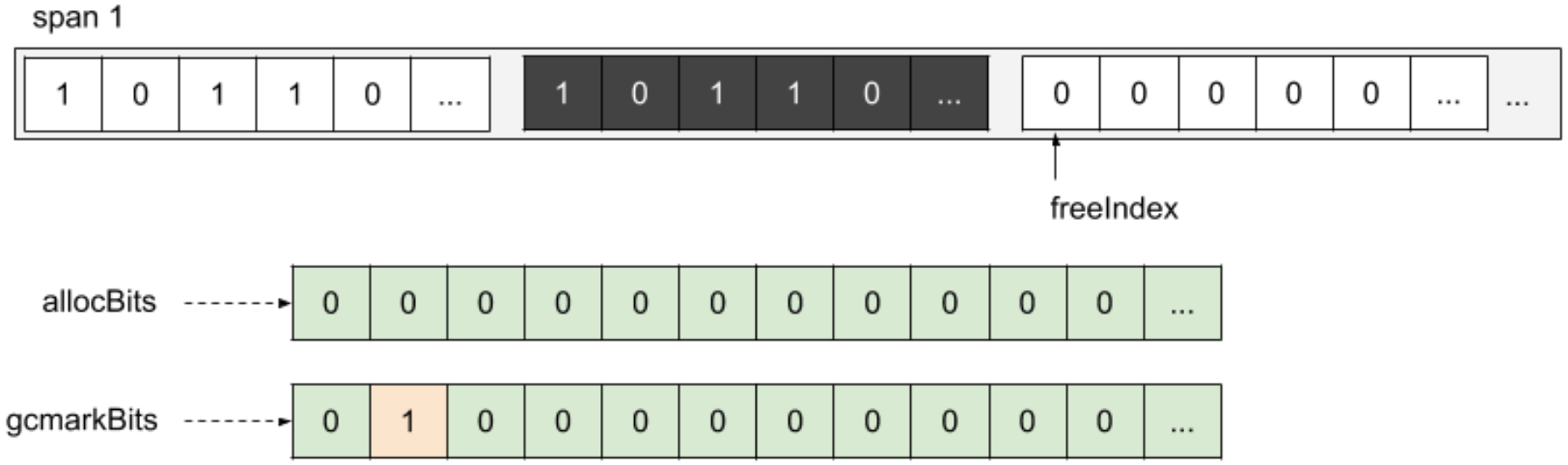


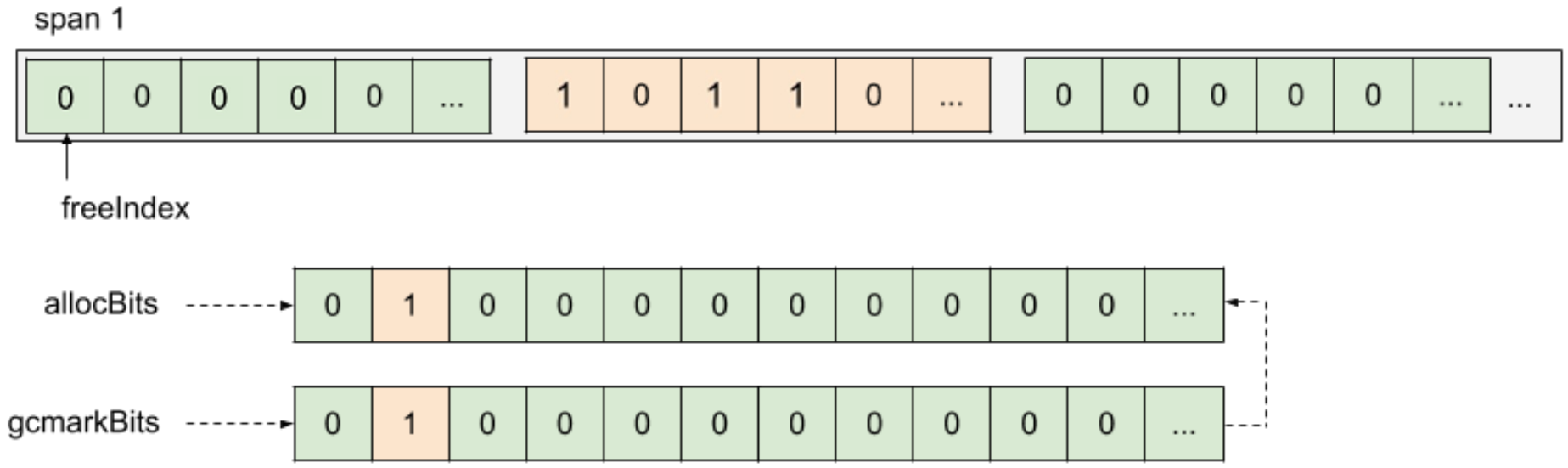


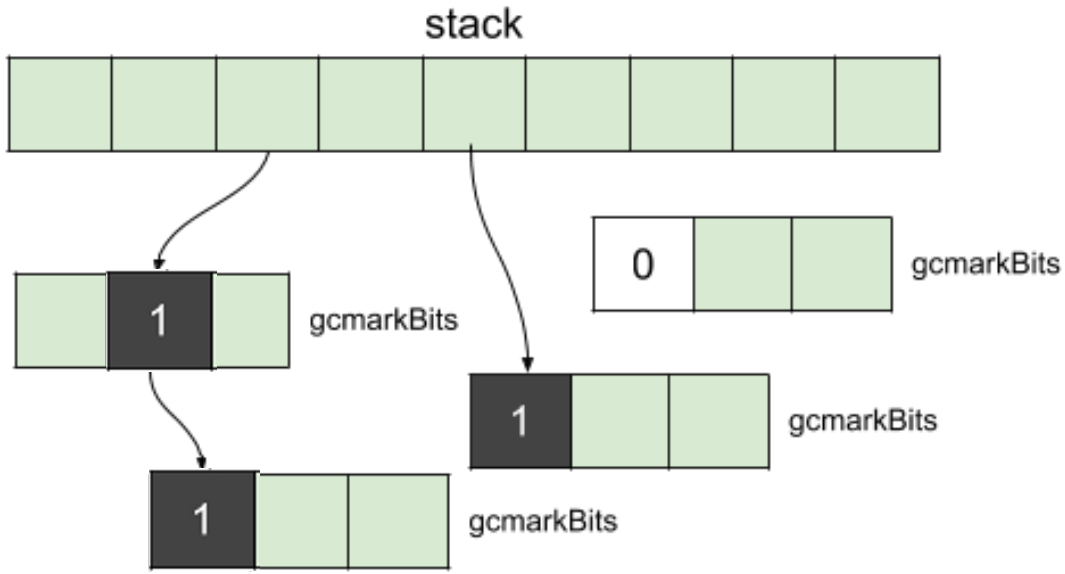
Bitmaps



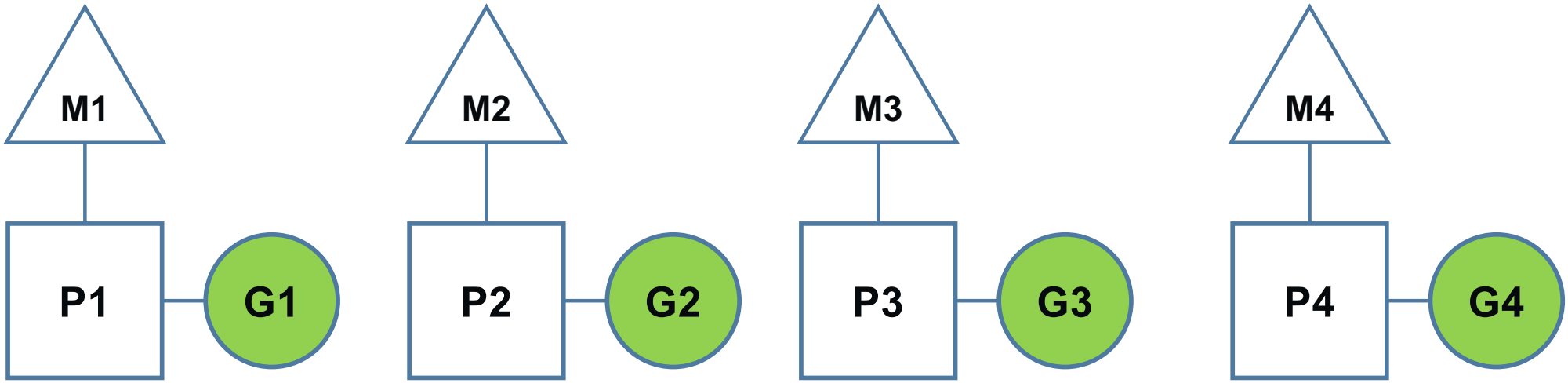




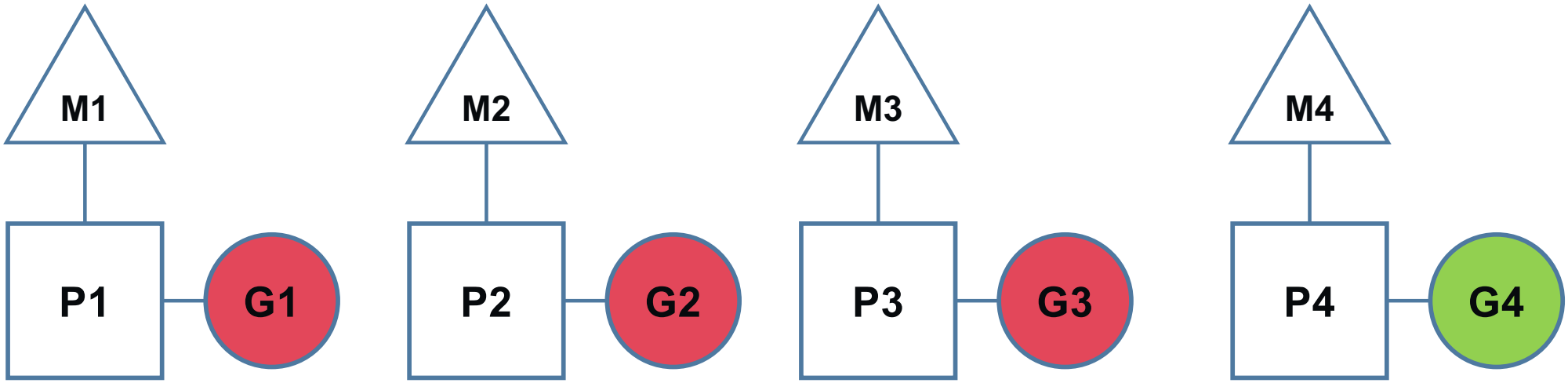




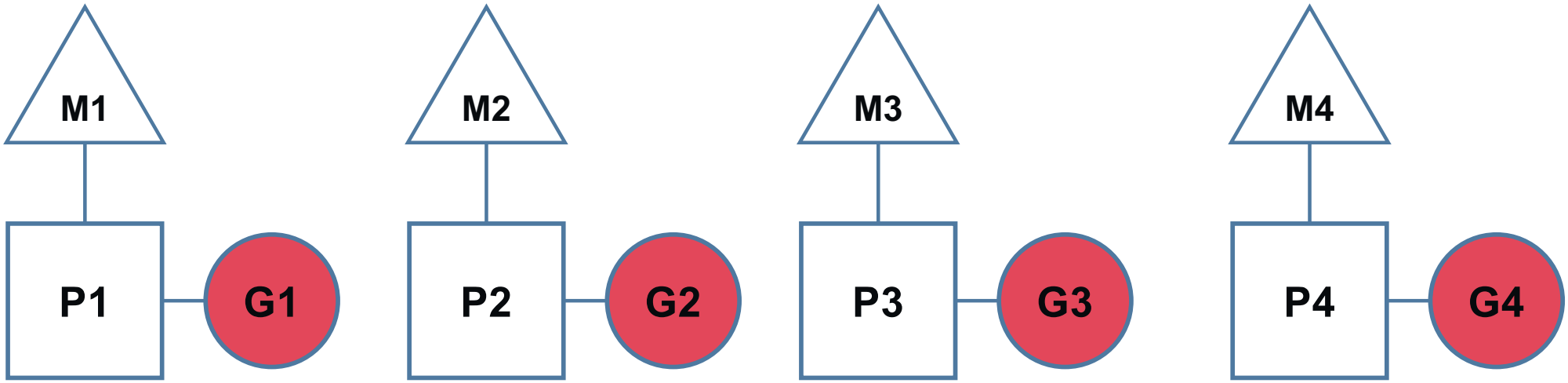
Mark Setup - STW



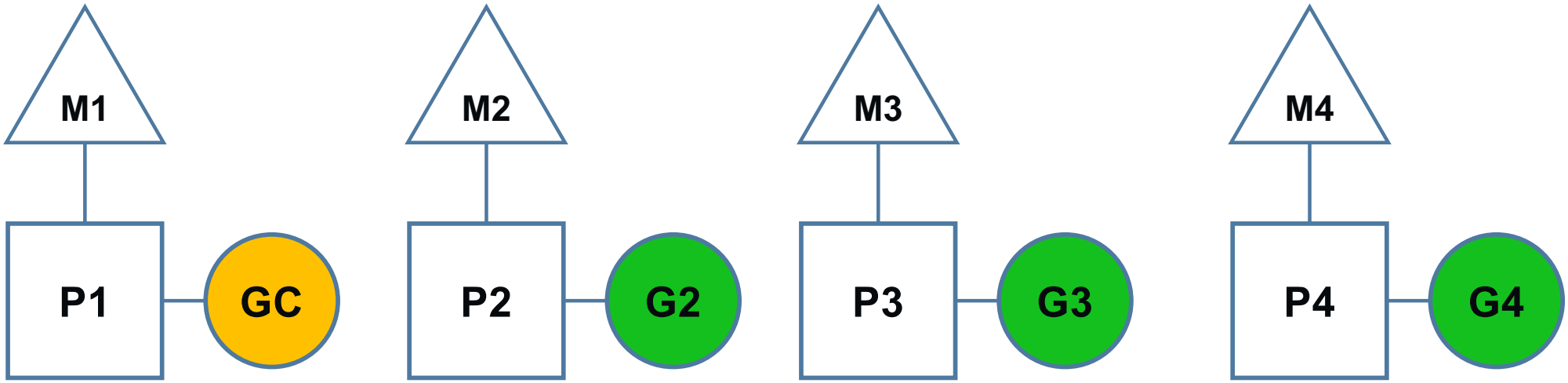
Mark Setup - STW



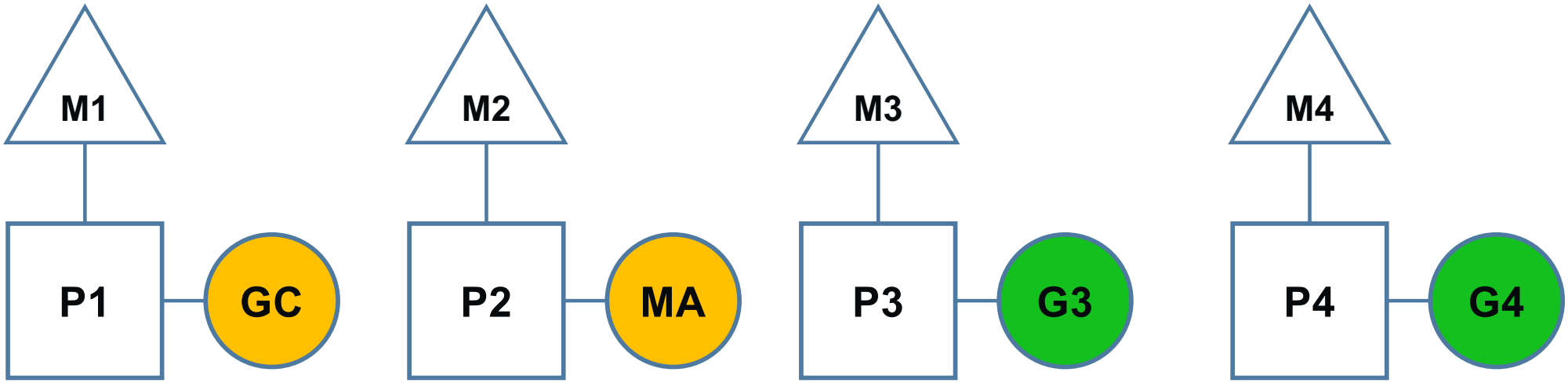
Mark Setup - STW



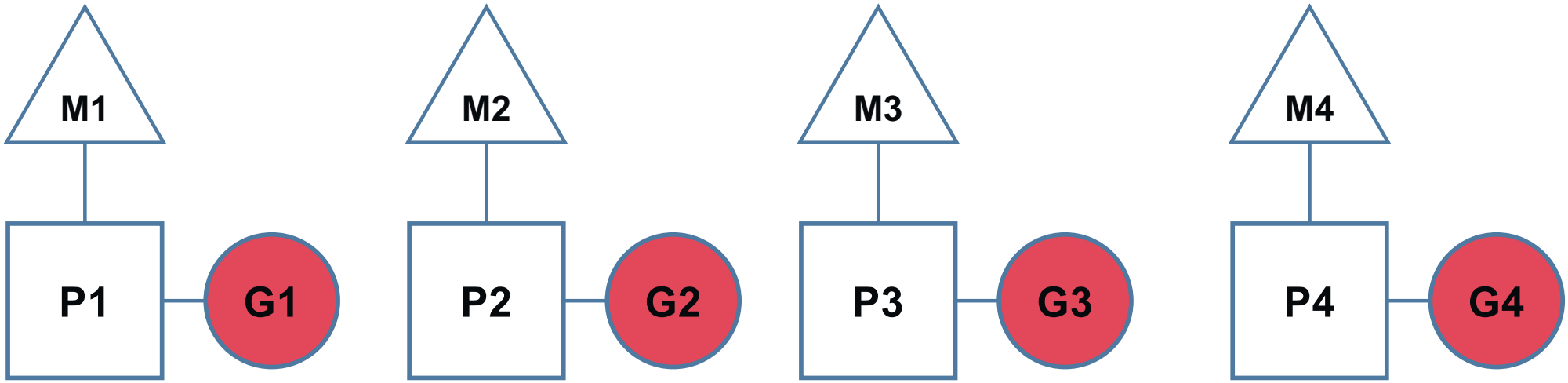
Marking - Concurrent

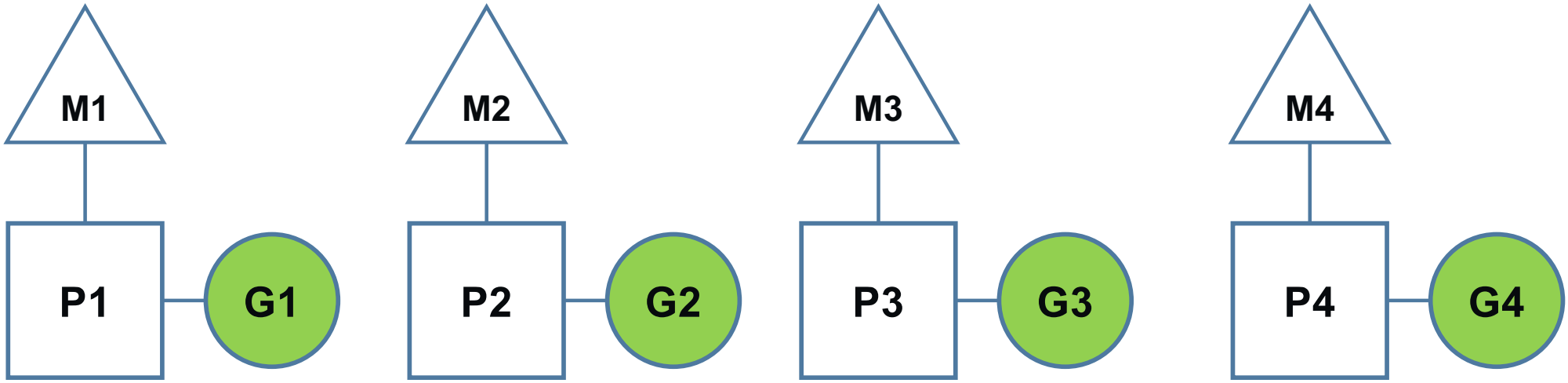


Mark Assist



Mark Termination - STW

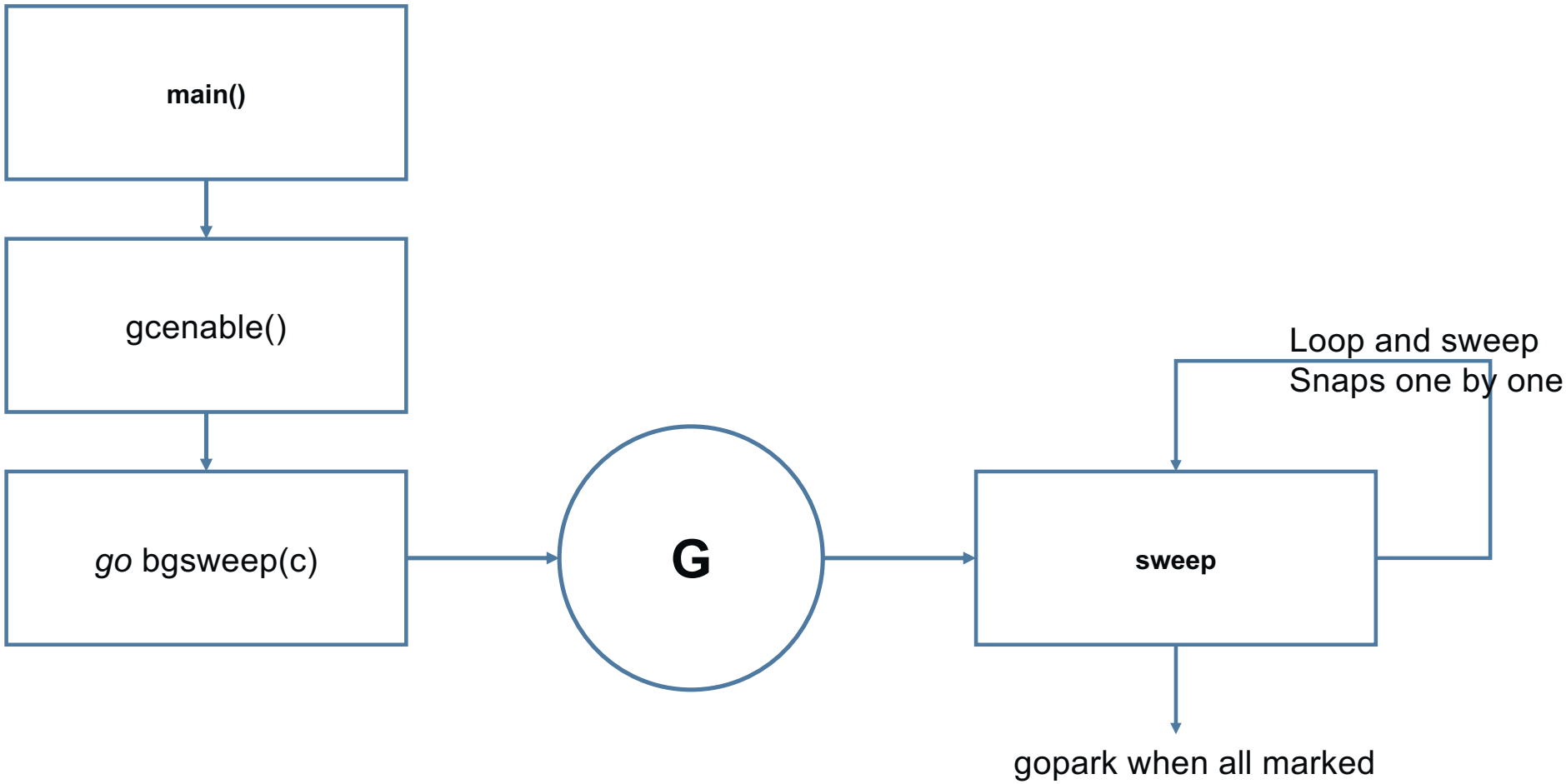




Sweeping - Concurrent

- Go provides two ways to sweep the memory:
- with a worker waiting in background that sweeps the spans one by one
- on the fly when the allocation requires a span

Sweep



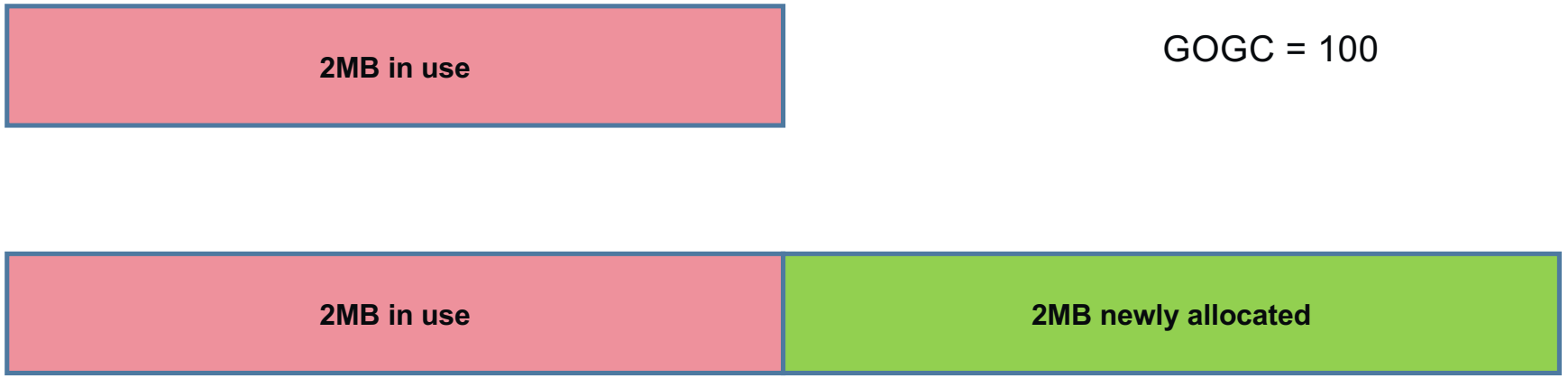
Collision with Collection Cycles

Timing

GC Percentage

HeapGoal = HeapLive * (1 + GOGC/100)

GOGC = 100



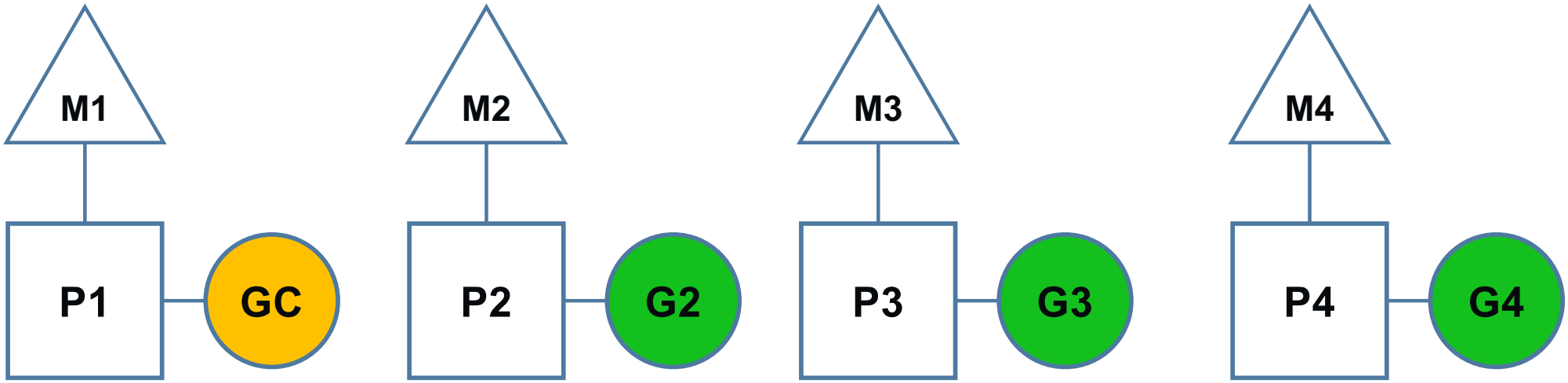
GC Trace

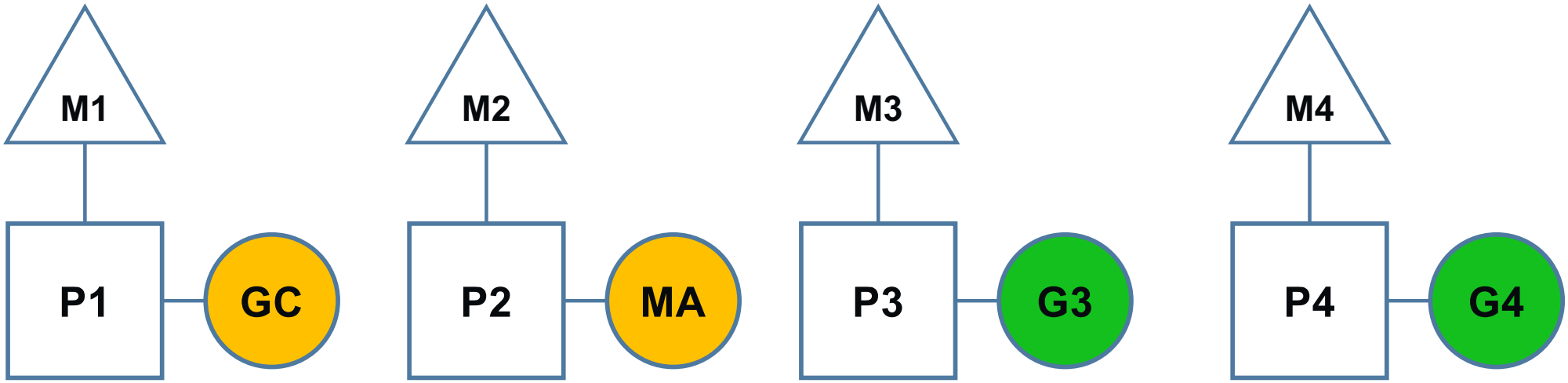
- GODEBUG=gctrace=1

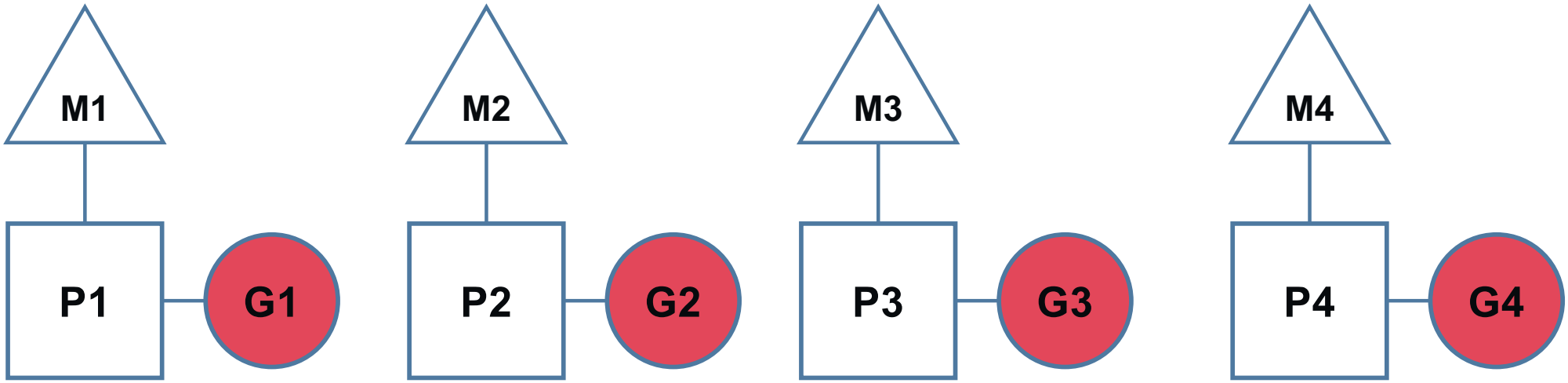
```
GODEBUG=gctrace=1 ./project > /dev/null
gc 3 @3.182s 0%: 0.015+0.59+0.096 ms clock, 0.19+0.10/1.3/3.0+1.1 ms cpu, 4->4->2 MB, 5 MB goal, 12 P
.
.
.
gc 2553 @8.452s 14%: 0.004+0.33+0.051 ms clock, 0.056+0.12/0.56/0.94+0.61 ms cpu, 4->4->2 MB, 5 MB goal, 12 P
```

Pacing

Collector Latency Costs







Less allocations on the heap

Less marking work

Shorter GC cycle

Values vs Pointers

- Prefer coping to passing pointers
- Avoid pointers in you types
- Check standard library for hidden pointers

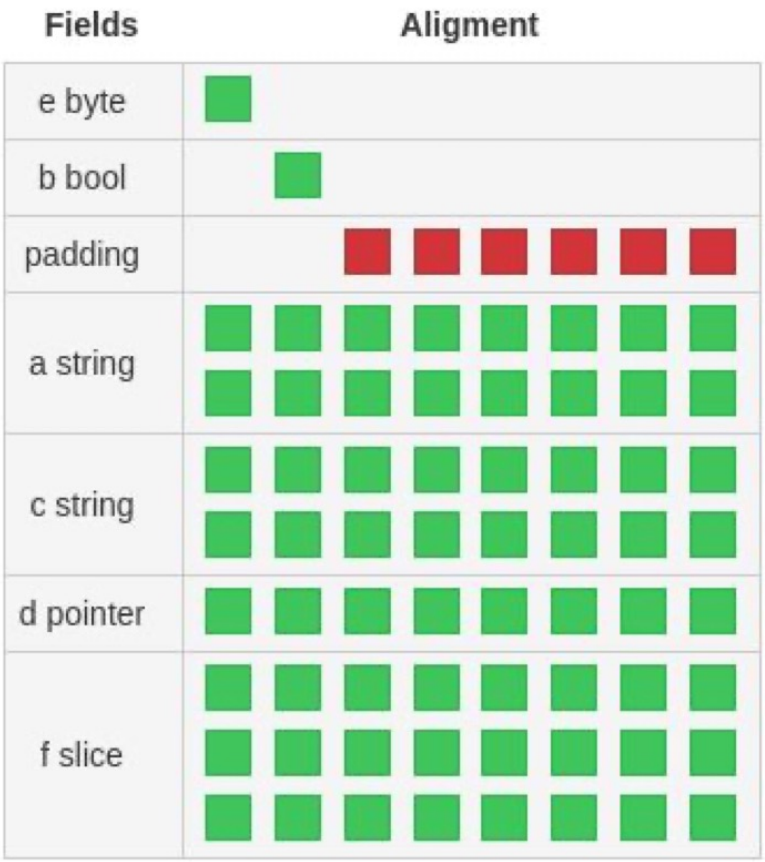
```go
type Time struct {
    // wall and ext encode the wall time seconds, wall time nanoseconds,
    // and optional monotonic clock reading in nanoseconds.
    //
    // From high to low bit position, wall encodes a 1-bit flag (hasMonotonic),
    // a 33-bit seconds field, and a 30-bit wall time nanoseconds field.
    // The nanoseconds field is in the range [0, 999999999].
    // If the hasMonotonic bit is 0, then the 33-bit field must be zero
    // and the full signed 64-bit wall seconds since Jan 1 year 1 is stored in ext.
    // If the hasMonotonic bit is 1, then the 33-bit field holds a 33-bit
    // unsigned wall seconds since Jan 1 year 1885, and ext holds a
    // signed 64-bit monotonic clock reading, nanoseconds since process start.
    wall uint64
    ext  int64

    // loc specifies the Location that should be used to
    // determine the minute, hour, month, day, and year
    // that correspond to this Time.
    // The nil location means UTC.
    // All UTC times are represented with loc==nil, never loc==&utcLoc.
    loc *Location
}
```

Struct Layout

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

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



Reuse Memory

- Reuse objects
- sync.Pool

Q&A


Netcracker
An NEC Company