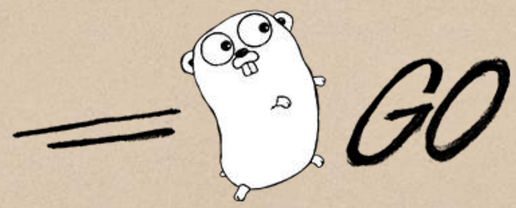
Golang 高级讲义

xiaorui.cc
github.com/rfyiamcool

v 0.3

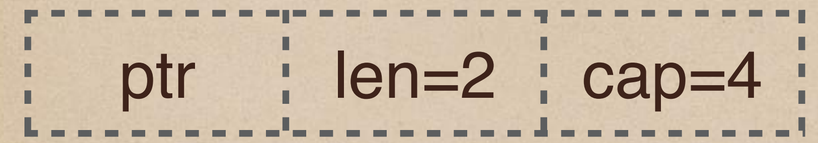
menu

- slice、map
- runtime.scheduler
- memory
- gc
- channel
- escape analysis



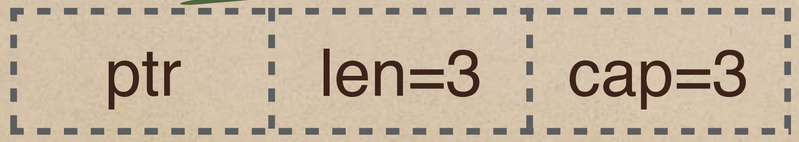
- netpoll
- timer
- sync lock
- sync map
- sync pool

slice



s1 := arr[1:3]

arr := [5]int{10, 20, 30, 40, 50}



s2 := arr[2:]

- ptr
  - 底层数组
- len
  - 占用个数
- cap
  - 容量大小

slice

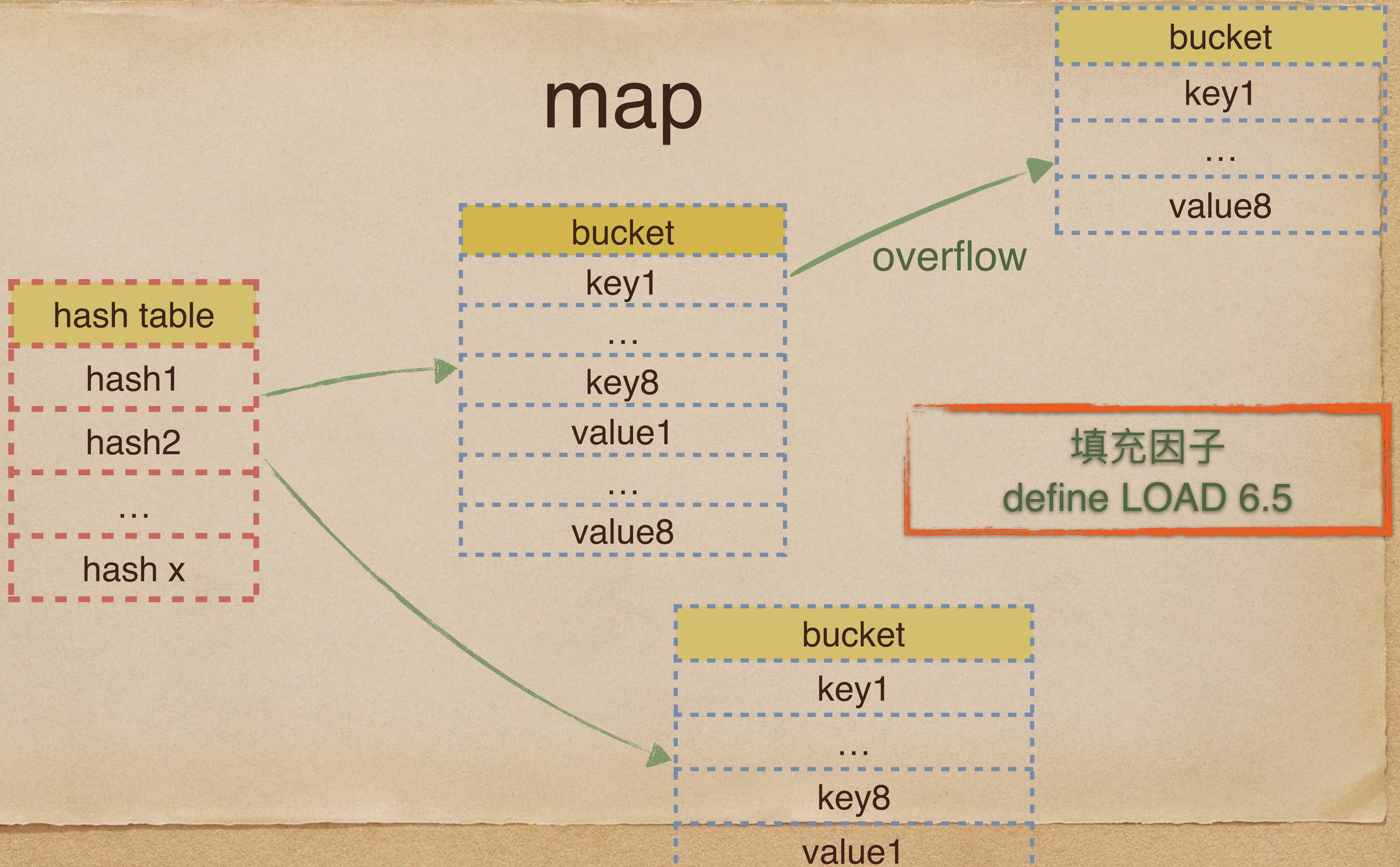
- 函数值传递只需24字节 (array地址 + len + cap)
- append
  - append会改变array的数据
  - 当cap不足时，会重新make一个新slice替换 ptr
- 函数传slice指针可解决ptr变更问题
- slice 线程不安全

map

```
struct Hmap
{
    uint8   B;     // 可以容纳2^B个项
    uint16  bucketsize;   // 每个桶的大小

    byte    *buckets;     // 2^B个Buckets的数组
    byte    *oldbuckets;  // 前一个buckets，只有当正在扩容时才不为空
};
```

```
struct Bucket
{
    uint8  tophash[BUCKETSIZE]; // hash值的高8位....低位从bucket的array定位到bucket
    Bucket *overflow;           // 溢出桶链表，如果有
    byte   data[1];             // BUCKETSIZE keys followed by BUCKETSIZE values
};
```

map

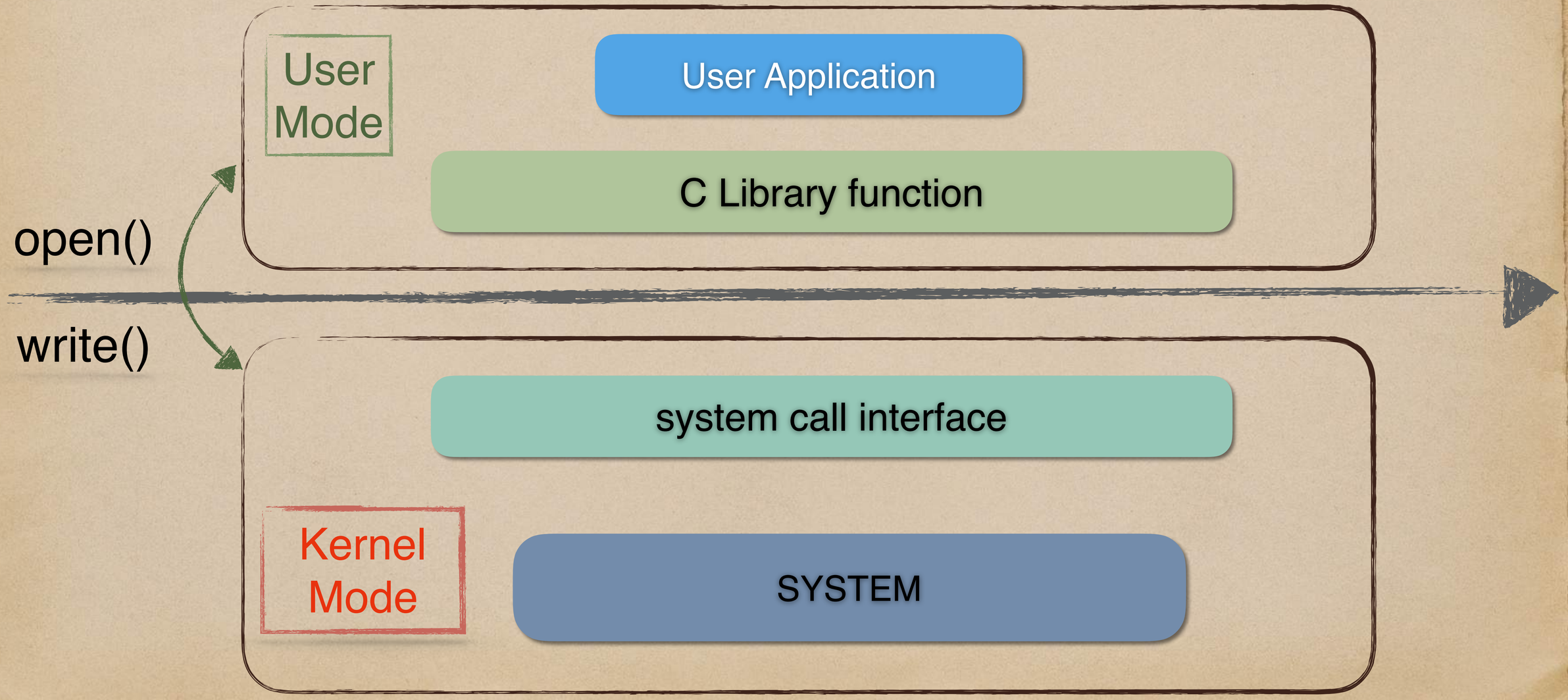

bucket
key1
...
value8
overflow
hash table
hash1
hash2
...
hash x
bucket
key1
...
key8
value1
...
value8
填充因子
define LOAD 6.5
bucket
key1
...
key8
value1

map

- map 可理解为引用
- map 内存释放问题
- go 1.10 value指针和值的性能
- map 线程不安全
  - sync.map
  - segment lock map

scheduler

syscall


User Mode
User Application
C Library function
open()
write()
system call interface
Kernel Mode
SYSTEM

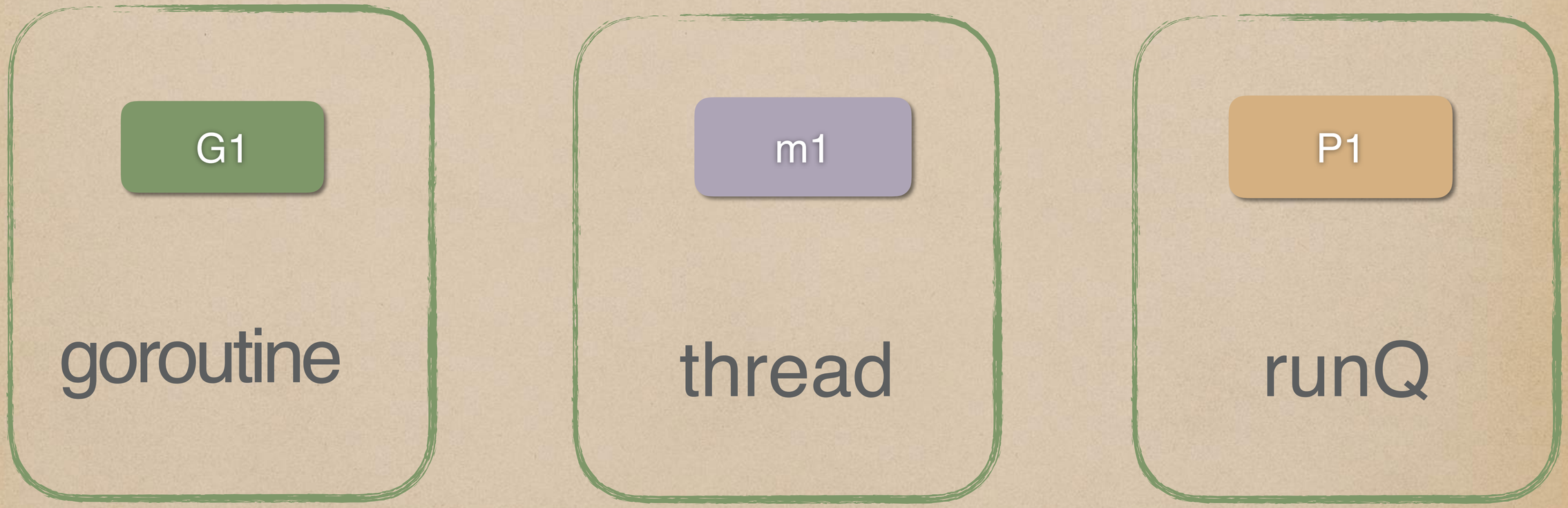
syscall

```
[pid 19154] <... read resumed> "\1\0\0\1\4<\0\0\2\3def\16refresh_master\nmai".
[pid 19150] <... pselect6 resumed> )     = 0 (Timeout)
[pid 19152] pselect6(0, NULL, NULL, NULL, {0, 3000}, 0 <unfinished ...>
[pid 19167] clock_gettime(CLOCK_MONOTONIC,  <unfinished ...>
[pid 19157] <... epoll_pwait resumed> {{EPOLLIN|EPOLLOUT|EPOLLRDHUP, {u32=1893
134576}}}, 128, 0, NULL) = 2
[pid 19157] read(3,  <unfinished ...>
[pid 19156] clock_gettime(CLOCK_MONOTONIC,  <unfinished ...>
[pid 19154] write(32, "\5\0\0\0\31\f\0\0\0", 9 <unfinished ...
[pid 19150] <... clock_gettime resumed> {935287, 674835167}) =
[pid 19167] futex(0xc420bda148, FUTEX_WAIT, 0, NULL <unfinishe
[pid 19157] <... read resumed> "HTTP/1.1 200\r\nDate: Thu, 03
```

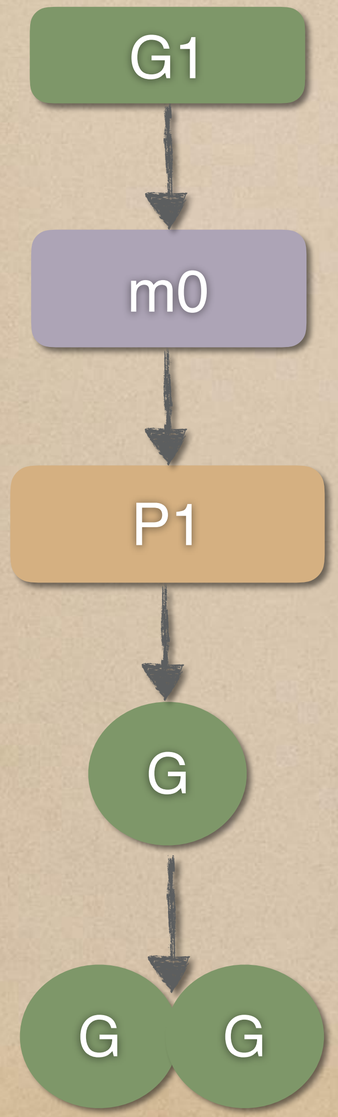
% time	seconds	usecs/call	calls	errors	syscall
84.79	30.827811	1230	25068	2507	futex
11.45	4.164287	185	22531		pselect6
1.89	0.687862	24	28721		epoll_pwait
1.16	0.422781	2	261155		clock_gettime
0.45	0.162315	4	36292		write
0.14	0.051648	2	31203	16372	read
0.03	0.010998	2750	4	2	restart_syscall
0.02	0.008774	9	986	986	connect
0.02	0.006912	3	2039		sched_yield
0.01	0.005180	5	1000		close
0.01	0.002445	2	986		socket
0.00	0.001707	2	985		getsockopt
0.00	0.001626	1	2018	6	epoll_ctl
0.00	0.001578	1	2083		setsockopt

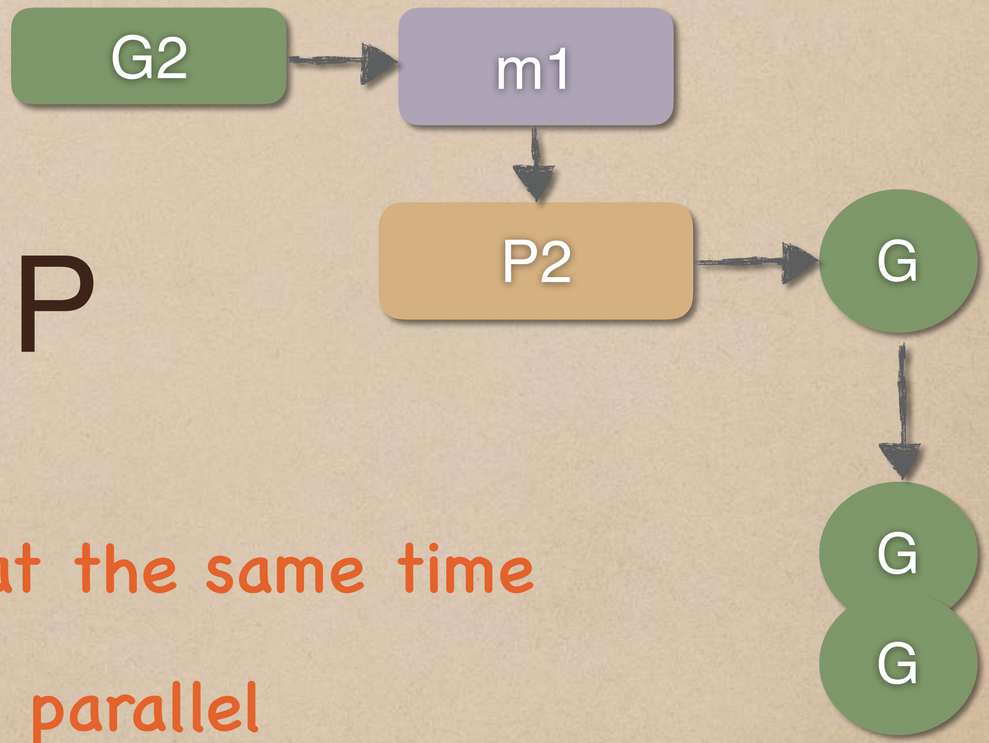
- how trap syscall
- when syscall ?

what PMG



scheduler





more P

- p link m only at the same time
- p count affect parallel

ext runtime struct

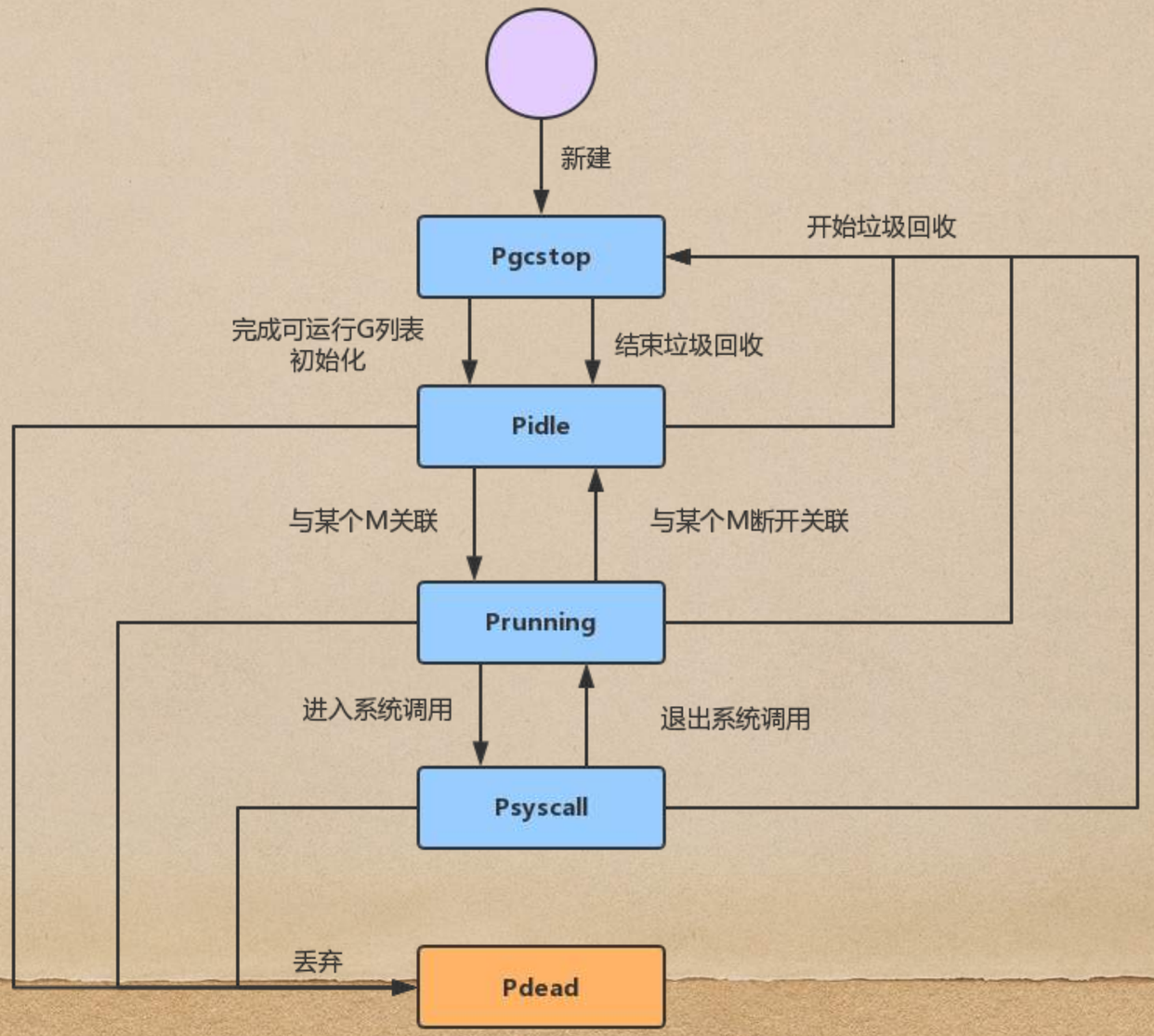
全局M列表	runtime.allm	存放所有M的列表
全局P列表	runtime.allp	存放所有P的列表
全局G列表	runtime.allg	存放所有G的列表
调度器空闲M列表	runtime.sched.midle	存放空闲M的列表
调度器空闲P列表	runtime.sched.pidle	存放空闲P的列表
调度器可运行G队列	runtime.shced.runq	存放可运行的G队列
P可运行的G队列	runq	存放当前P中可运行G
…	…	…

when to scheduler

- channel
- garbage collection
- blocking syscall like file or network IO
- time.Sleep
- more

P status

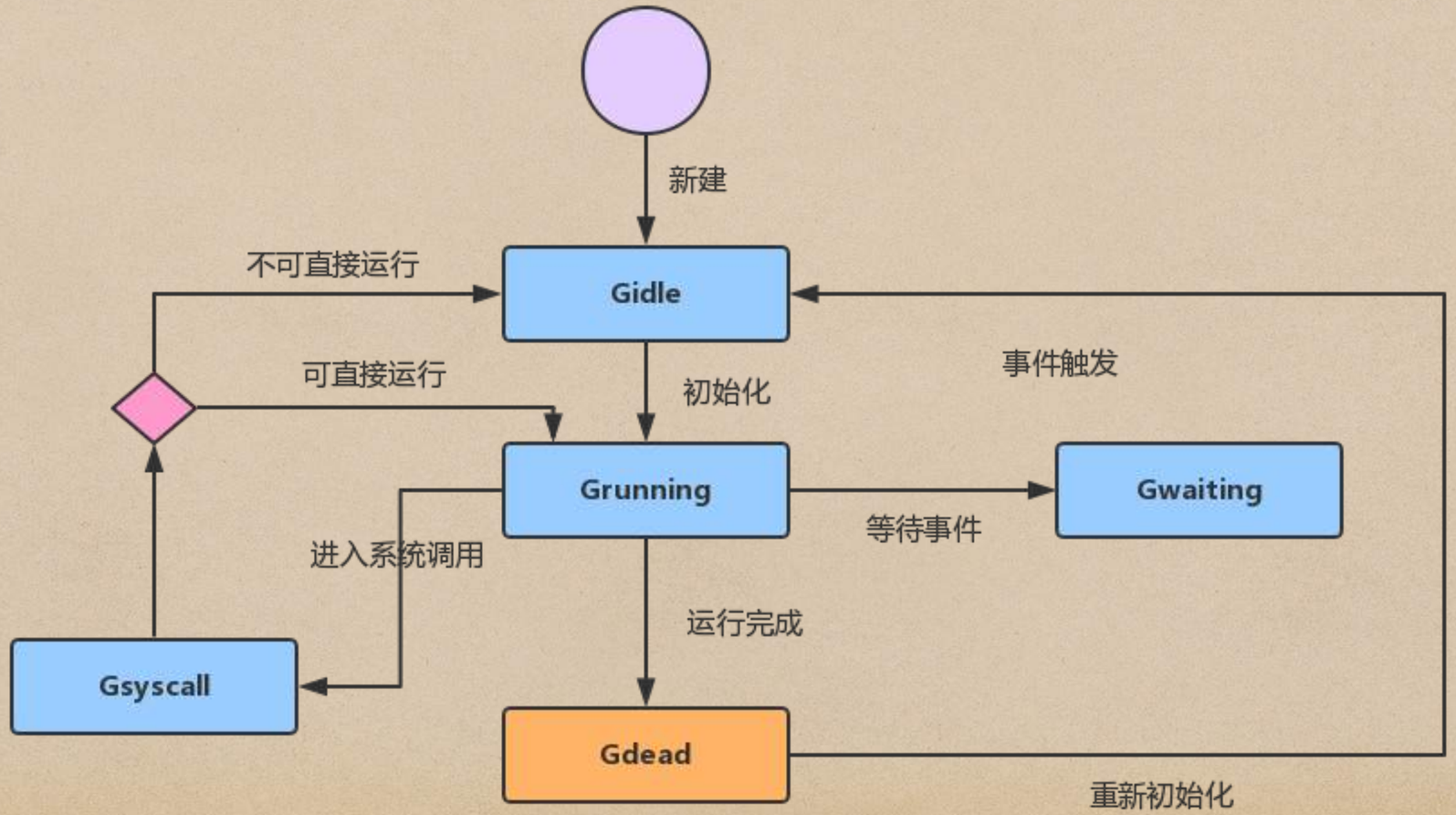
- Pgcstop
  - 正在进行垃圾回收
- Pidle
  - 没有跟M关联
- Prunning
  - 跟M关联
- Psyscall
  - 系统调用
- Pdead
  - 减少 GOMAXPROCS



M status

- 自旋中(spinning)
  - M正在从运行队列获取G，这时候M会拥有一个P
- 执行go代码中
  - M正在执行go代码, 这时候M会拥有一个P
- 执行原生代码中
  - M正在执行原生代码或者阻塞的syscall, 这时M并不拥有P
- 休眠中
  - M发现无待运行的G时会进入休眠, 并添加到空闲M链表中, 这时M并不拥有P

Goroutine status

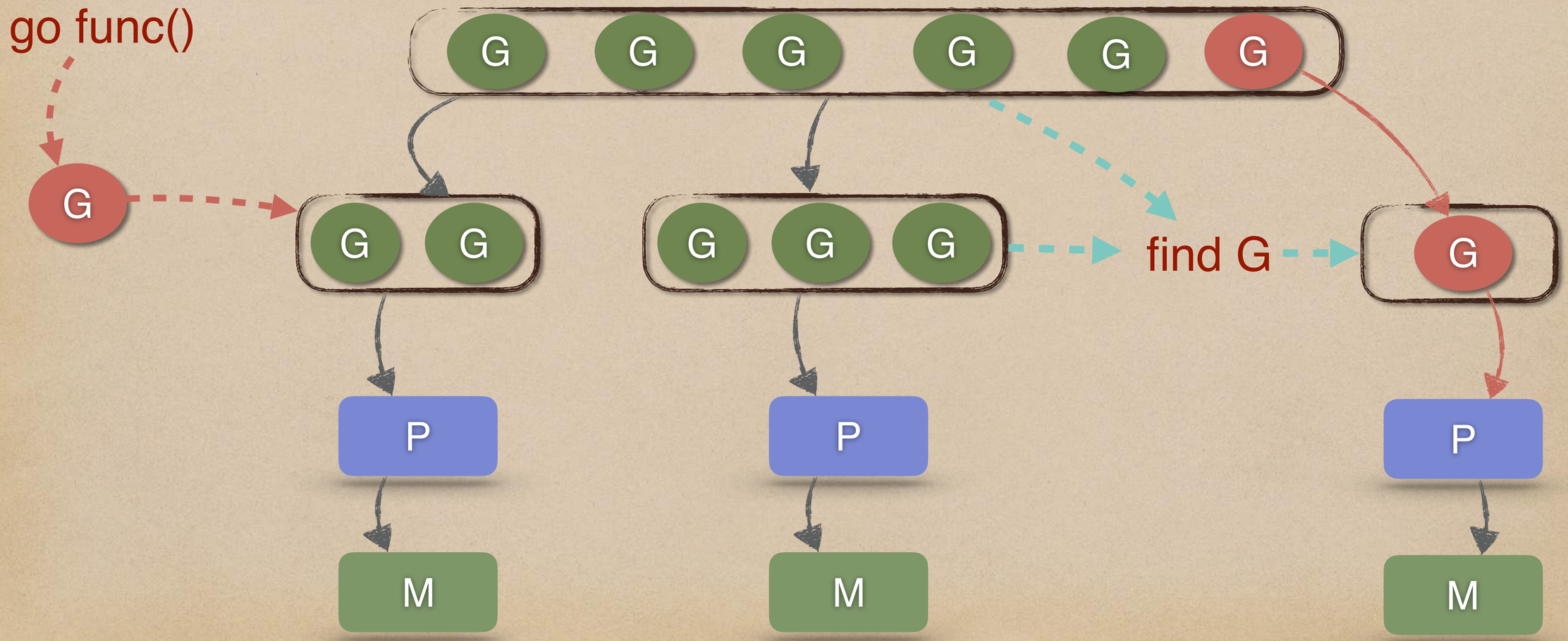


by xiaorui.cc

goroutine视角

- go, 产生新的g, 放到本地队列或全局队列
- gopark, g置为waiting状态, 等待显示goready唤醒, 在poller中用得较多
- goready, g置为runnable状态, 放入全局队列
- gosched, g显示调用runtime.Gosched让出资源, 置为runnable状态, 放入全局队列
- goexit, g执行完退出, g所属m切换到g0栈, 重新进入schedule
- g陷入syscall:
  - net io和部分file io, 没有事件则gopark;
  - 普通的阻塞系统调用返回时，m重新进入schedule

work stealing

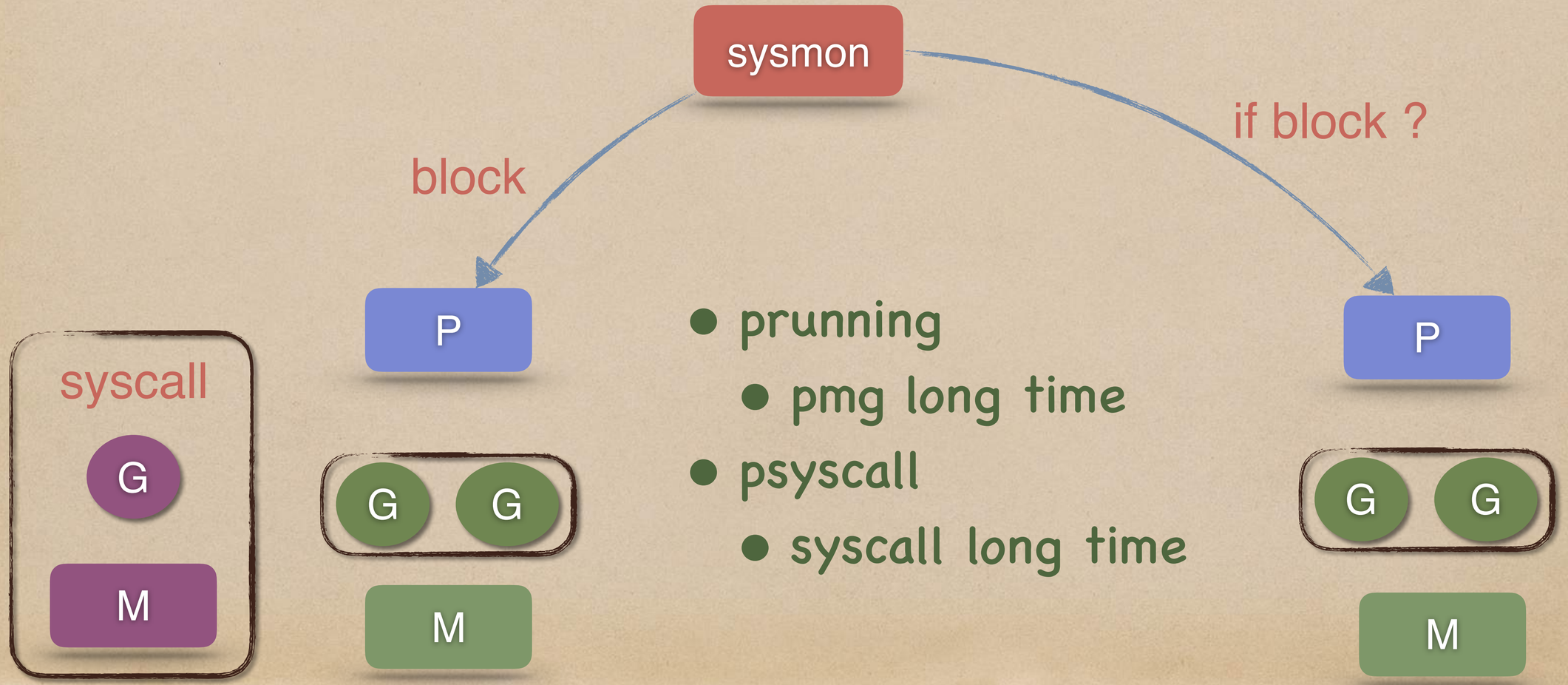


go work stealing

```go
func findrunnable() (gp *g, inheritTime bool) {
    _g_ := getg()
top:
    // 从本地runq获取G
    if gp, inheritTime := runqget(_g_.m.p.ptr()); gp != nil {
        return gp, inheritTime
    }
    // 从全局sched.runq中获取G
    if sched.runqsize != 0 {
        gp := globrunqget(_g_.m.p.ptr(), 0)
    }
    // 随机选择一个P 从它的runq中steal一部分任务G
    for i := 0; i < int(4*gomaxprocs); i++ {
        _p_ := allp[fastrand1()%uint32(gomaxprocs)]
        var gp *g
        if _p_ == _g_.m.p.ptr() {
            gp, _ = runqget(_p_)
        } else {
            stealRunNextG := i > 2*int(gomaxprocs)
            gp = runqsteal(_g_.m.p.ptr(), _p_, stealRunNextG)
        }
```

1. pop from local runq
2. pop from global runq
3. pop from netpoller
4. pop from other p' runq

抢占



- prunning
  - pmg long time
- psyscall
  - syscall long time

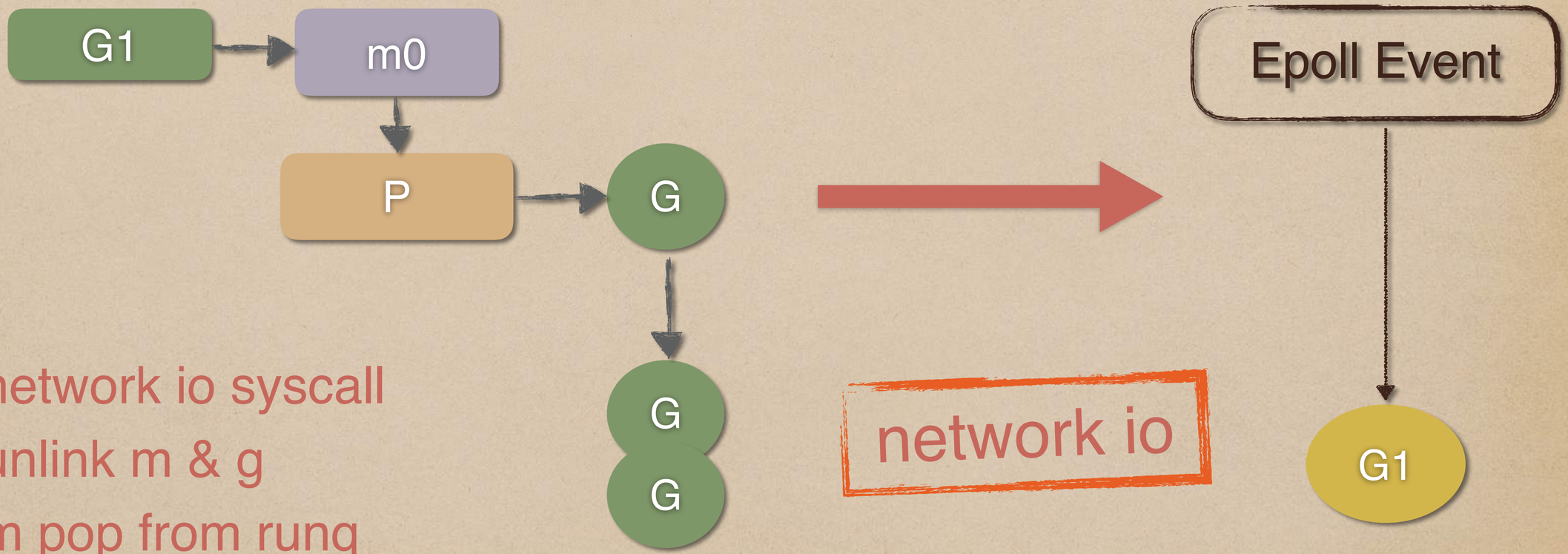
sysmon

循环调用存在于整个生命周期
min = 20us; max = 10ms

多功能
sysmon！

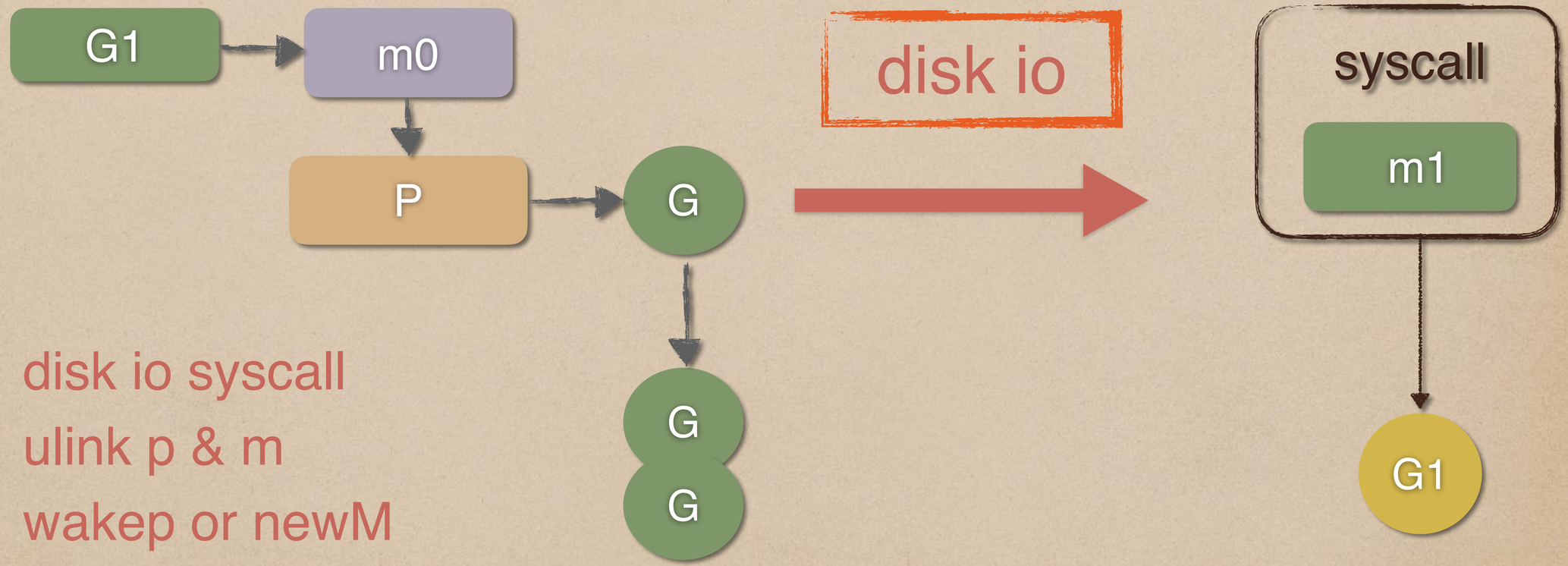
1. netpoll(获取fd事件)
2. retake(抢占)
3. forcegc(按时间强制执行gc)
4. scavenge heap(释放自由列表中多余的项减少内存占用)
5. more

unlink m & g case



1. network io syscall
2. unlink m & g
3. m pop from runq
4. m run g context

unlink p&m case



1. disk io syscall
2. ulink p & m
3. wakep or newM
4. link p & m

memory

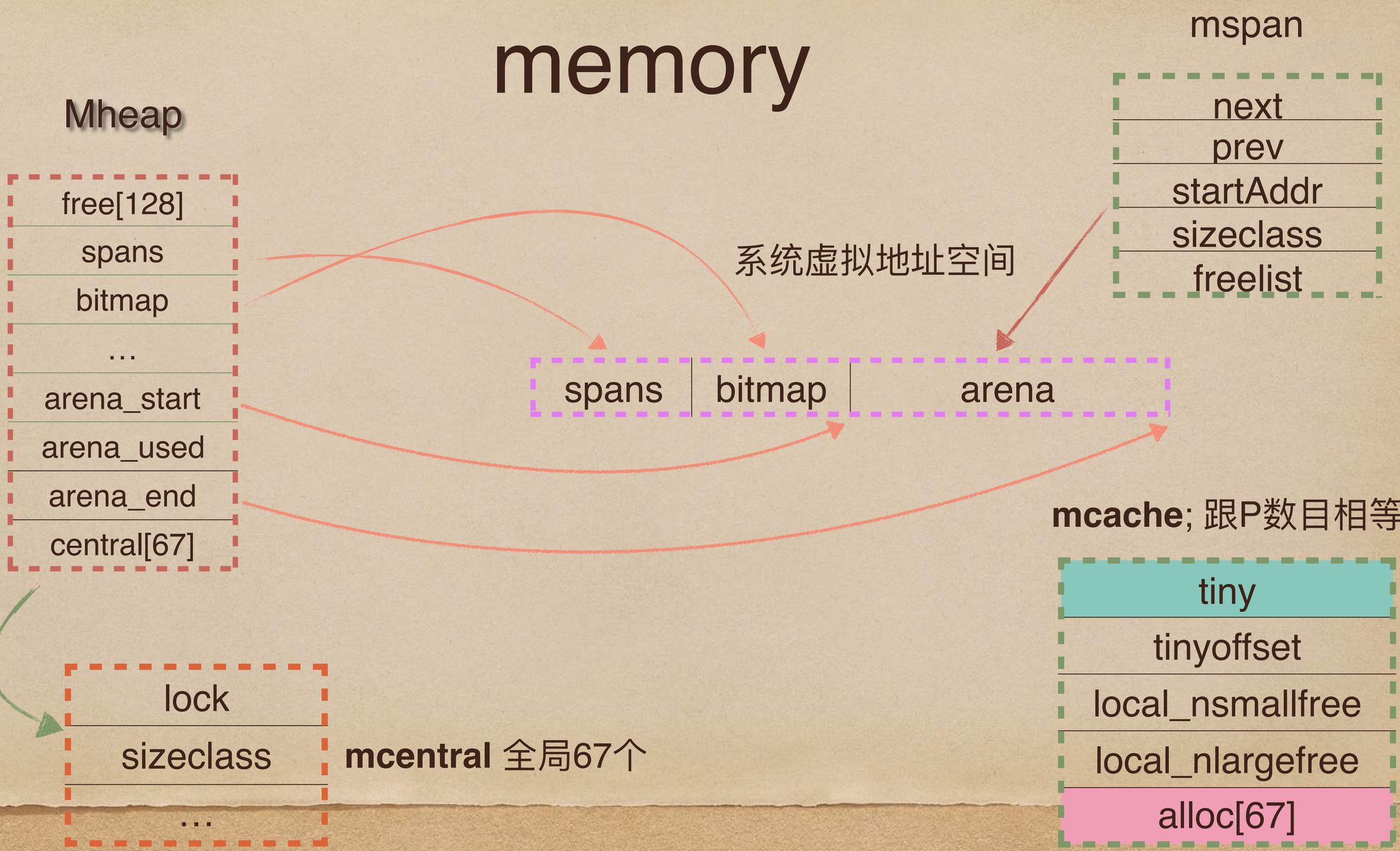
memory

- 基于tcmalloc构建的多级内存池
- mcache: per-P cache，可以认为是 local cache。
- mcentral: 全局 cache，mcache 不够用的时候向 mcentral 申请。
- mheap: 当 mcentral 也不够用的时候，通过 mheap 向操作系统申请。
- mspan: 内存分配的基本单位

memory

- mspan 默认 8k 起
- spans 512M
- bitmap 16G
- arena 512G

memory


Mheap
free[128]
spans
bitmap
…
arena_start
arena_used
arena_end
central[67]
系统虚拟地址空间
spans
bitmap
arena
mspan
next
prev
startAddr
sizeclass
freelist
mcache; 跟P数目相等
tiny
tinyoffset
local_nsmallfree
local_nlargefree
alloc[67]
lock
sizeclass
…
mcentral 全局67个

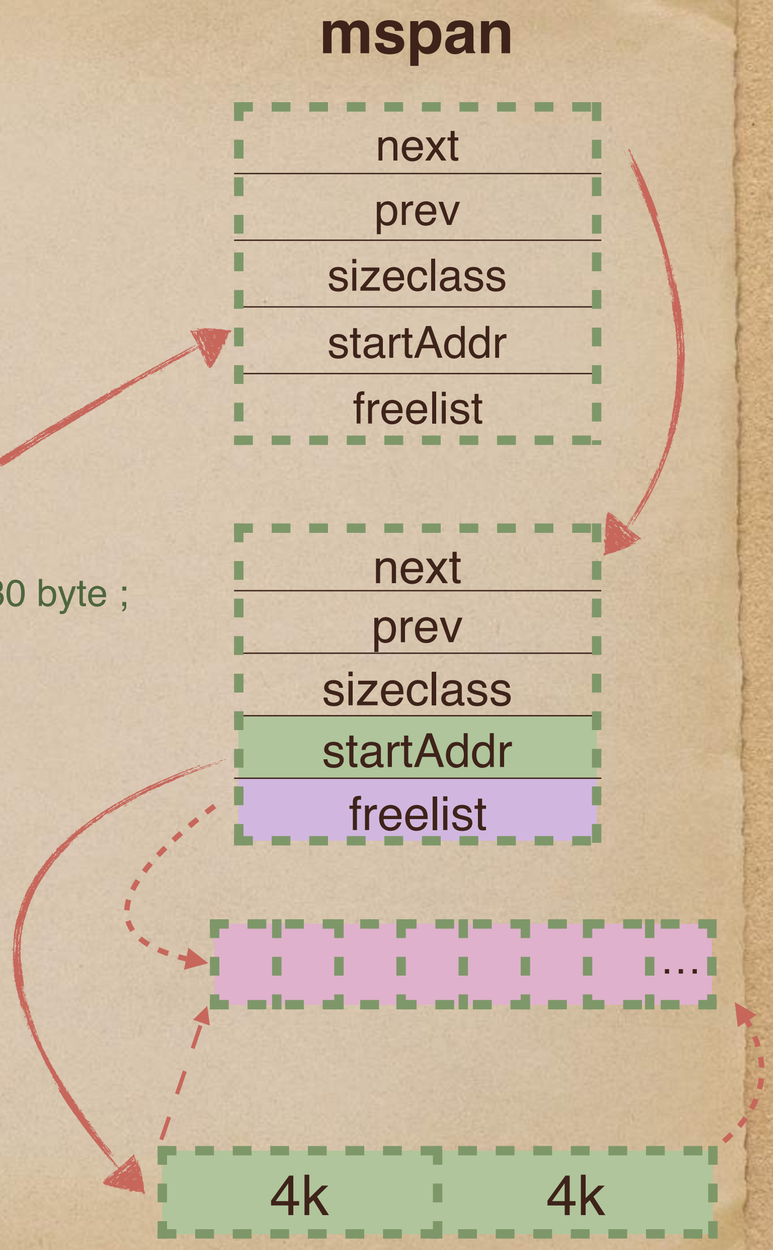
memory

alloc = [67]sizeclass
每行的sizeclass包含
一个mspan链表

mcache

```
// class  bytes/obj  bytes/span
…
//   2         16        8192
//   3         32        8192
//   4         48        8192
//   5         64        8192
…
//  23        448        8192
…
//  43       4096        8192
…
//  64      27264       81920
//  65      28672       57344
//  66      32768       32768
```

if malloc 30 byte ;



alloc memory rule

- 如果object size>32KB, 则直接使用mheap来分配空间；
- 如果object size<16Byte, 则通过mcache的tiny分配器来分配；
- 如果16Byte < object size < 32KB，首先尝试通过sizeclass对应的分配器分配；
  - 如果mcache没有空闲的span， 则向mcentral申请空闲块；
  - 如果mcentral也没空闲块，则向mheap申请并进行切分；
  - 如果mheap也没合适的span，则向系统申请新的内存空间。

object relase rule

- 如果object size>32KB, 直接将span返还给mheap的自由链；
- 如果object size<32KB, 查找object对应sizeclass， 归还到mcache自由链；
- 如果mcache自由链过长或内存过大，将部分span归还到mcentral；
- 如果某个范围的mspan都已经归还到mcentral，则将这部分mspan归还到mheap页堆；
- 而mheap会定时将释放的内存归还到系统；

gc

desc gc

- Golang 采用了标记清除的GC算法
- Golang的标记清除是一个三色标记法的实现，对于三色标记法，"三色"的概念可以简单的理解为:
  - 白色：还没有搜索过的对象（白色对象会被当成垃圾对象）
  - 灰色：正在搜索的对象
  - 黑色：搜索完成的对象（不会当成垃圾对象，不会被GC）

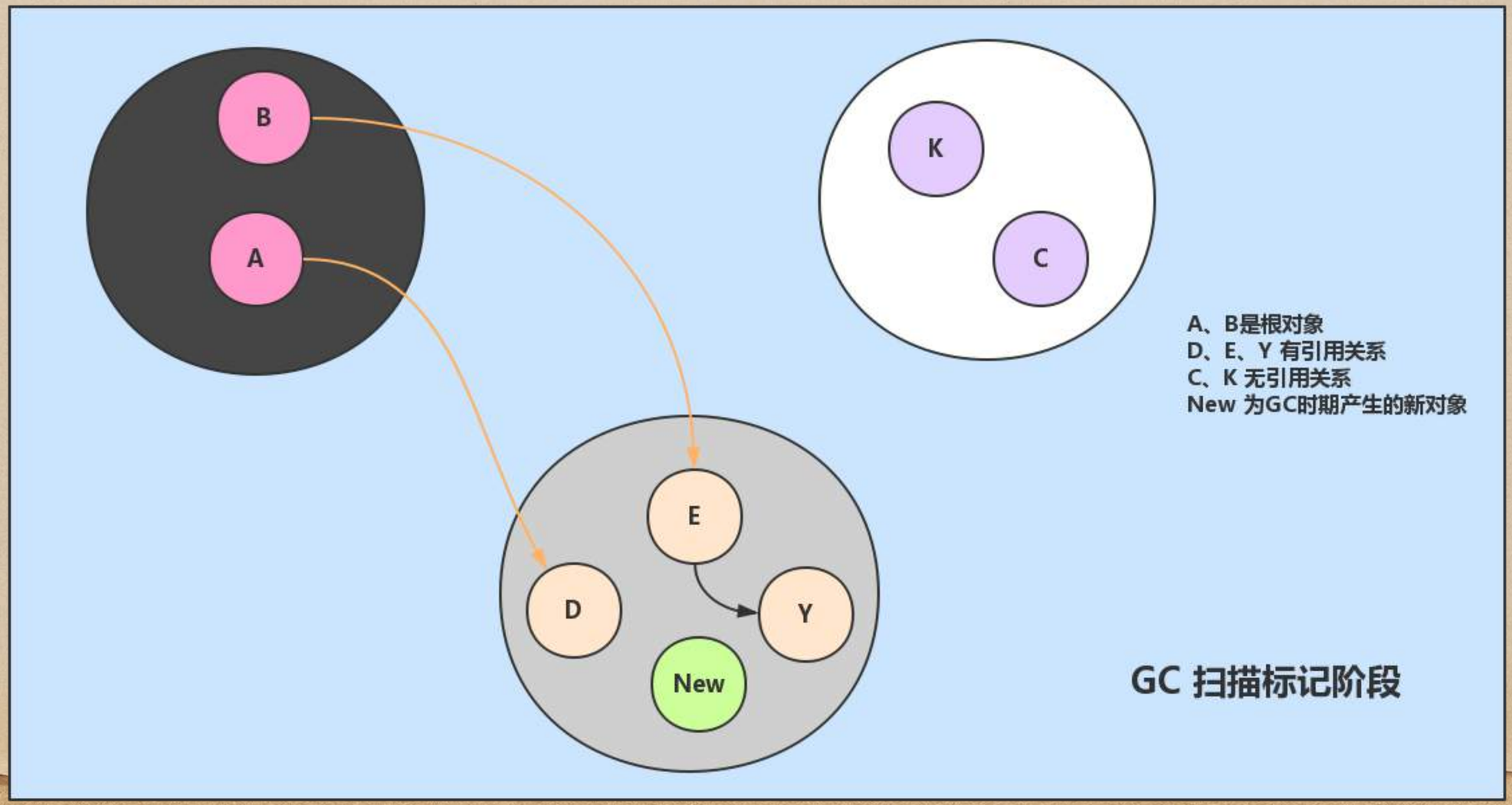
when trigger gc

- gcTriggerHeap
  - 当前分配的内存达到一定值就触发GC
    - default GCGO=100
- gcTriggerTime
  - 当一定时间没有执行过GC就触发GC
  - two minutes
- gcTriggerCycle
  - 要求启动新一轮的GC, 已启动则跳过, 手动触发GC的runtime.GC()会使用这个条件
- gcTriggerAlways
  - 强制触发GC

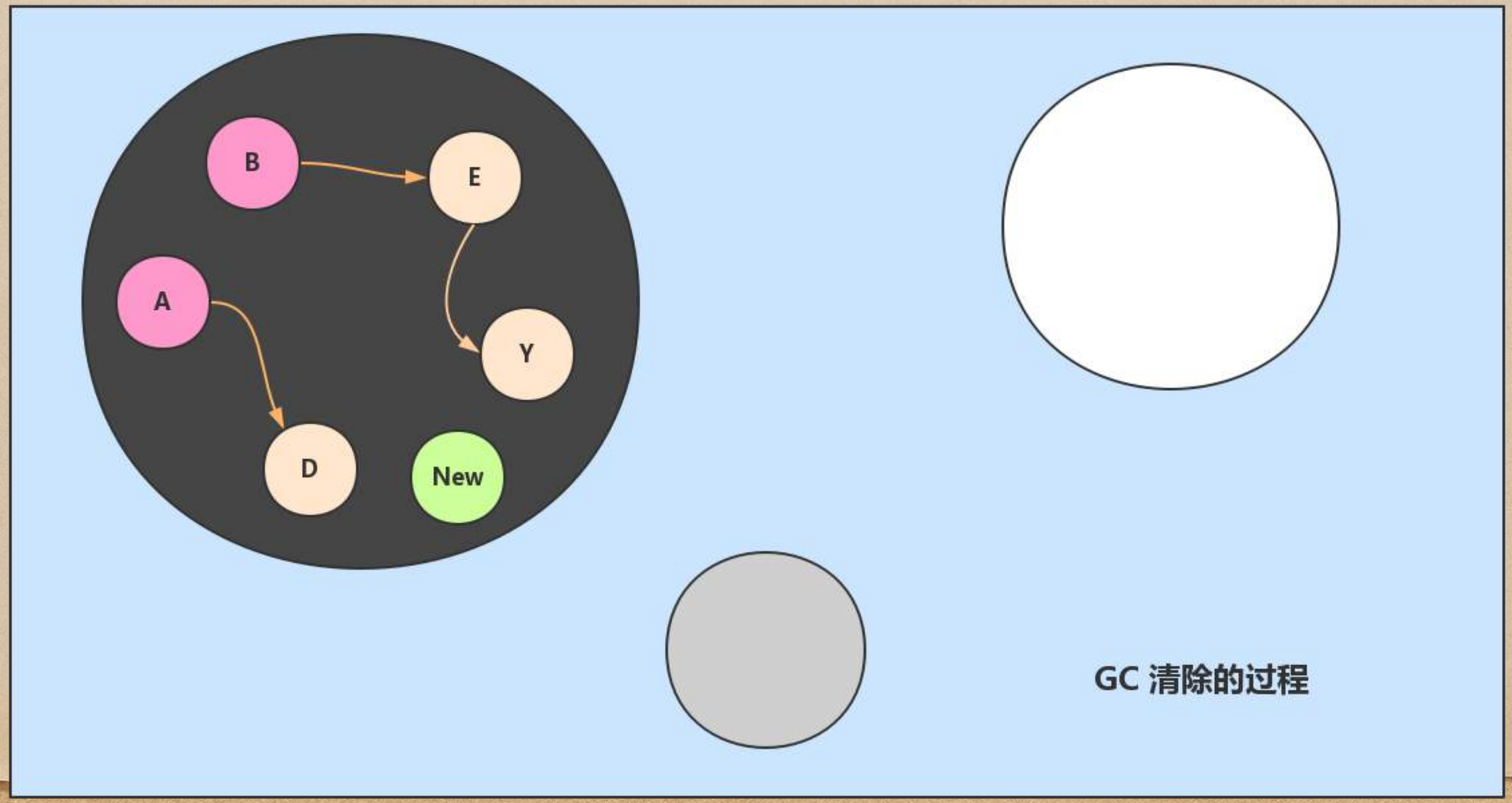
gc process

1. 首先创建三个集合：白、灰、黑。
2. 将所有对象放入白色集合中。
3. 然后从根节点开始遍历所有对象，把可追溯的对象从白色集合放入灰色集合。
4. 之后遍历灰色集合，将灰色对象引用的对象从白色集合放入灰色集合，之后将此灰色对象放入黑色集合。
5. 重复4直到灰色中无任何对象。
6. 通过写屏障检测对象有变化，重复以上操作。
7. 回收所有白色对象

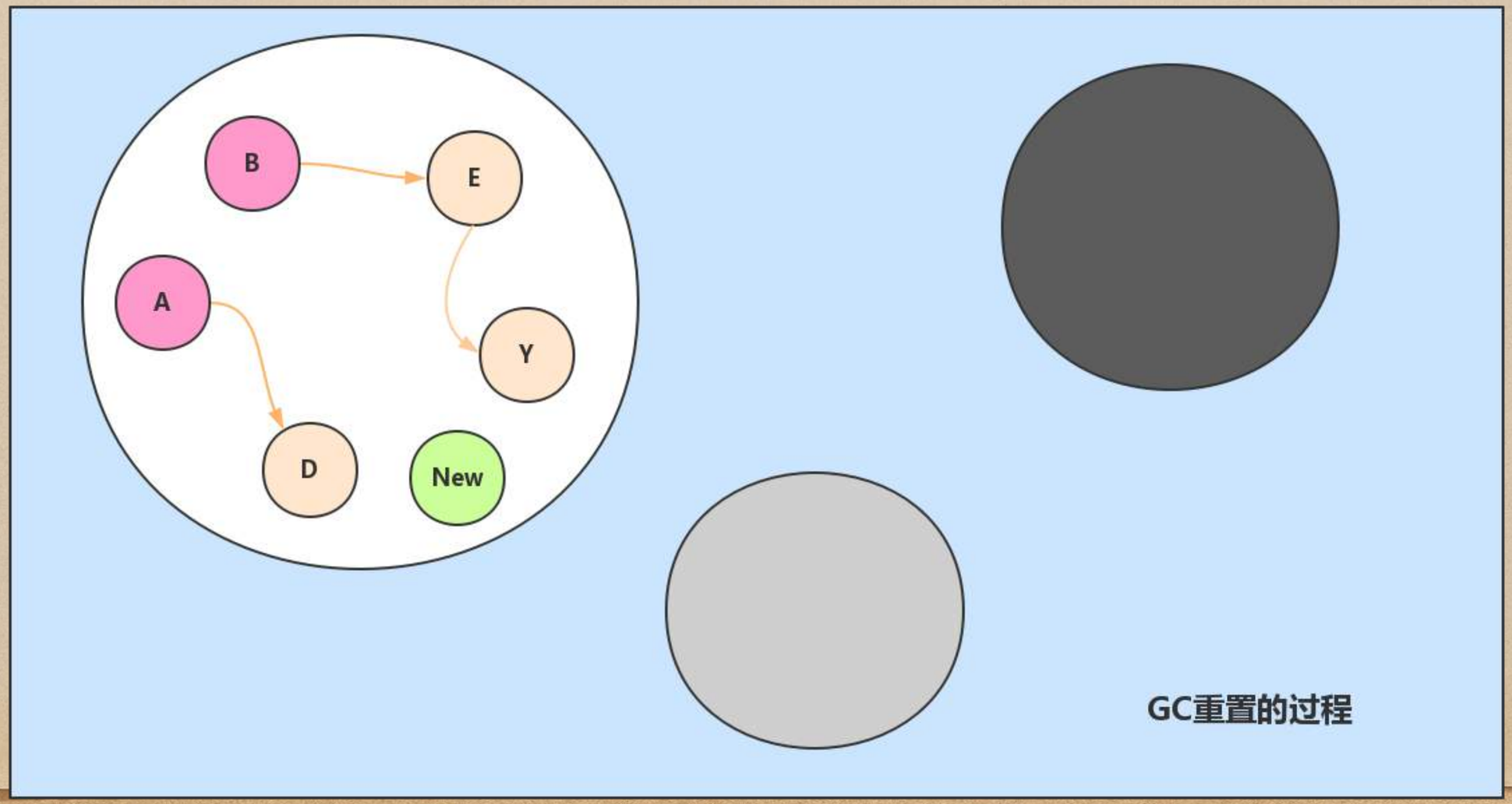
GC 扫描标记


B
A
K
C
E
D
Y
New
A、B是根对象
D、E、Y 有引用关系
C、K 无引用关系
New 为GC时期产生的新对象
GC 扫描标记阶段

GC 清除



GC 重置

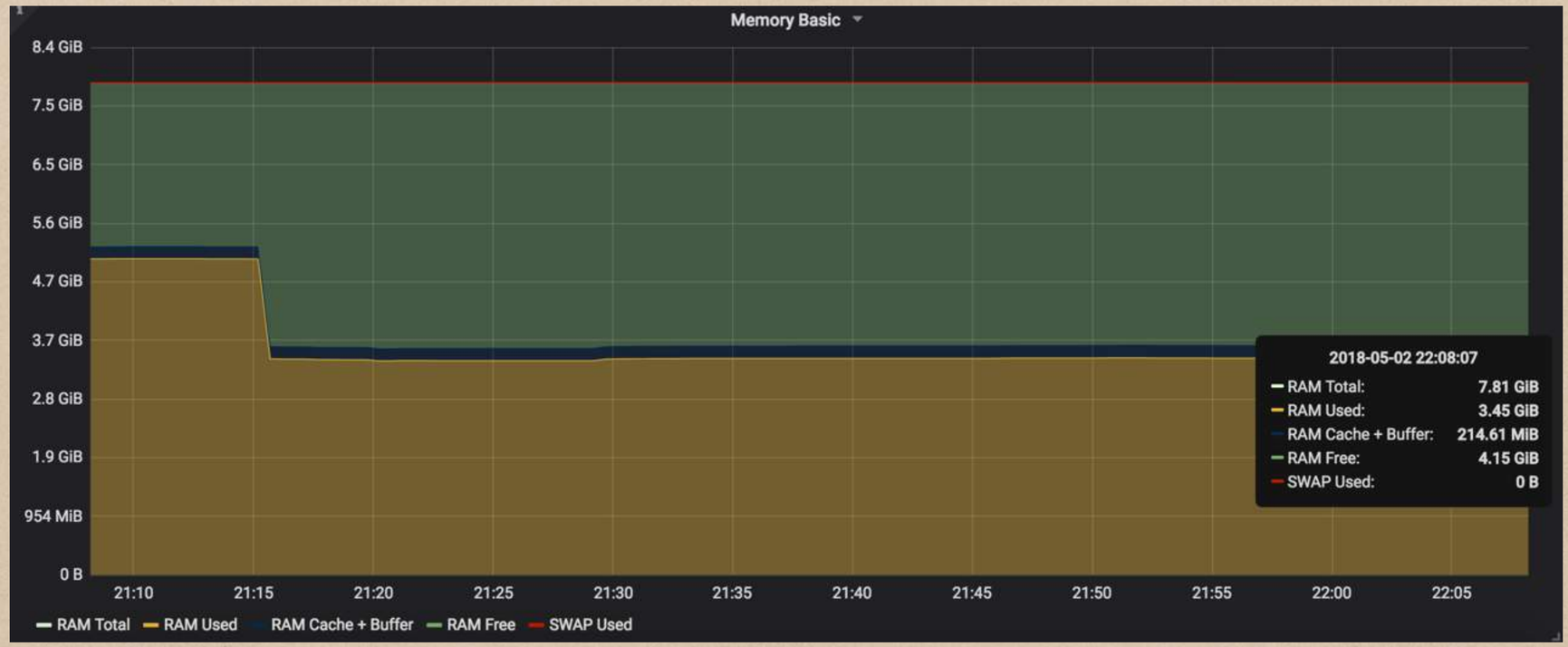


forcegc & return os

```
4219 // This is a variable for testing purposes. It normally doesn't change.
4220 var forcegcperiod int64 = 2 * 60 * 1e9
4221
4222 // Always runs without a P, so write barriers are not allowed.
4223 //
4224 //go:nowritebarrierrec
4225 func sysmon() {
4226     lock(&sched.lock)
4227     sched.nmsys++
4228     checkdead()
4229     unlock(&sched.lock)
4230
4231     // If a heap span goes unused for 5 minutes after a garbage collection,
4232     // we hand it back to the operating system.
4233     scavengelimit := int64(5 * 60 * 1e9)
4234
4235     if debug.scavenge > 0 {
4236         // Scavenge-a-lot for testing.
4237         forcegcperiod = 10 * 1e6
4238         scavengelimit = 20 * 1e6
4239     }
```

- Sysmon 会强制两分钟进行一次 GC
- 每5分钟会释放一些span, 归还操作系统

map的释放情况

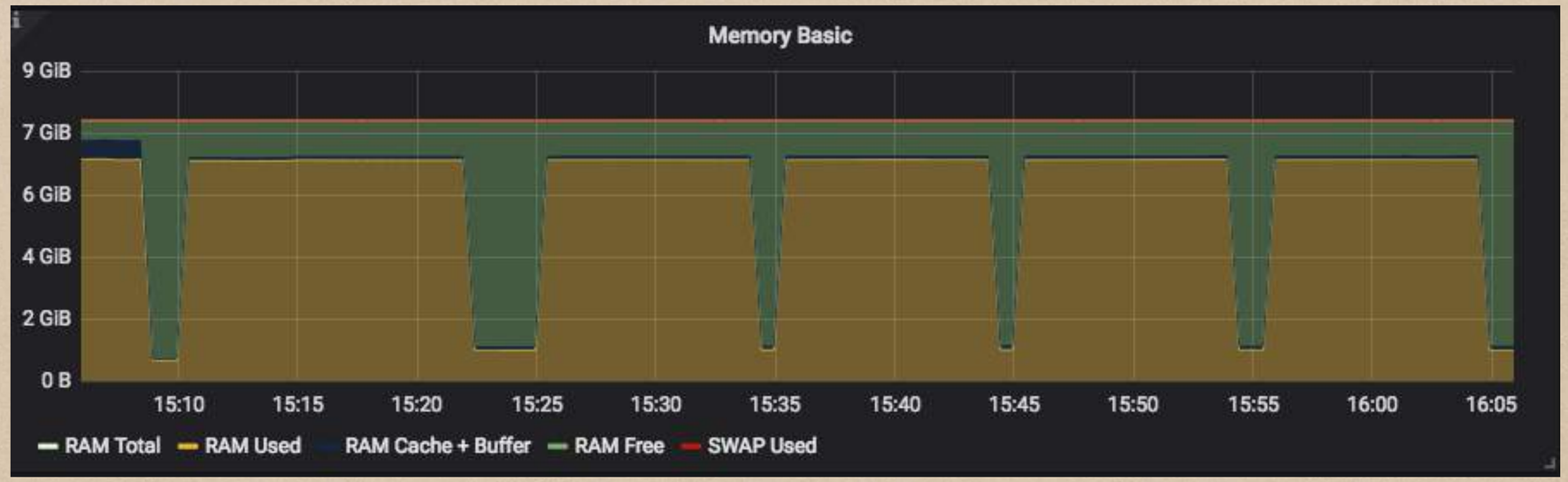


大量的使用big map下会出现对象被清楚，但释放不干净！

map = nil

debug.FreeOSMemory()

channel的释放情况



Channel被清空后，大约10分钟释放内存 .

debug.FreeOSMemory()

元素被gc清除后，什么时候释放内存归还 OS？

编码说 5分钟, 但事实不是这样…

手动触发 FreeOSMemory() 会更好的释放！

debug

GODEBUG=gctrace=1

```
gc 307 @1288.241s 1%: 0.28+73+0.082 ms clock, 0.57+83/42/0+0.16 ms cpu, 212->215->42 MB, 221 MB goal, 2 P
gc 308 @1288.673s 1%: 0.44+83+0.066 ms clock, 0.88+96/45/0+0.13 ms cpu, 202->204->40 MB, 210 MB goal, 2 P
gc 309 @1289.082s 1%: 0.20+70+0.046 ms clock, 0.40+83/40/0+0.093 ms cpu, 194->197->39 MB, 202 MB goal, 2 P
gc 310 @1289.495s 1%: 0.49+68+0.023 ms clock, 0.98+80/37/0+0.047 ms cpu, 190->192->37 MB, 198 MB goal, 2 P
gc 311 @1289.878s 1%: 0.39+61+0.091 ms clock, 0.79+81/30/0+0.18 ms cpu, 180->182->36 MB, 188 MB goal, 2 P
gc 312 @1290.254s 1%: 0.20+71+0.031 ms clock, 0.41+84/41/0+0.062 ms cpu, 175->177->36 MB, 182 MB goal, 2 P
gc 313 @1290.612s 1%: 0.18+69+0.055 ms clock, 0.37+83/39/0+0.11 ms cpu, 173->175->35 MB, 180 MB goal, 2 P
gc 314 @1290.960s 1%: 0.49+51+0.052 ms clock, 0.99+61/30/0+0.10 ms cpu, 172->174->31 MB, 179 MB goal, 2 P
GC forced
gc 12 @1324.050s 0%: 0.012+1.8+0.019 ms clock, 0.025+0/1.8/1.7+0.038 ms cpu, 2->2->2 MB, 16 MB goal, 2 P
```

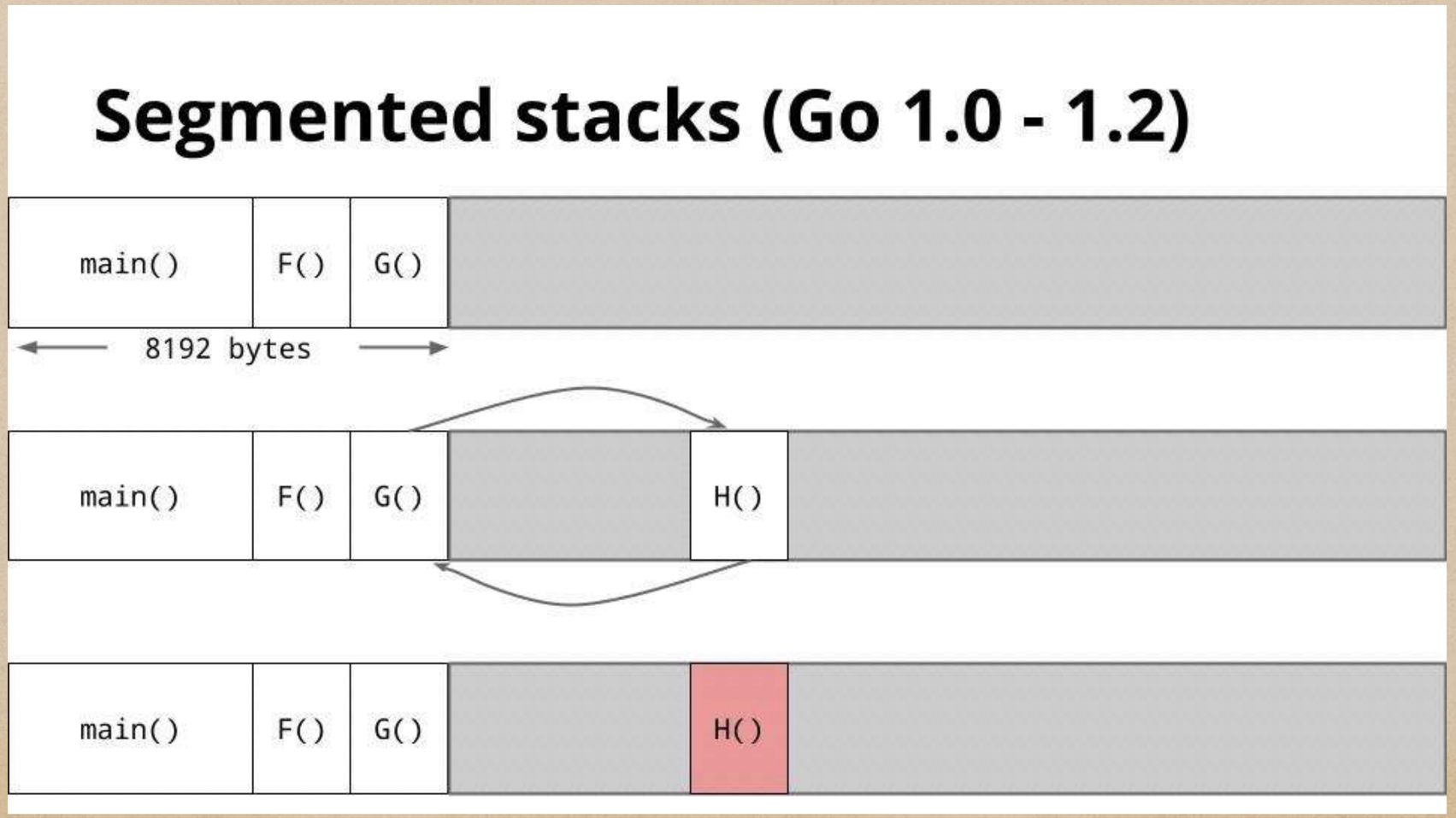
```
func printMem() {
    runtime.ReadMemStats(&mem)
    log.Println("mem.Sys: ", mem.Sys/1024/1024)
    log.Println("mem.Alloc: ", mem.Alloc/1024/1024)
    log.Println("mem.TotalAlloc: ", mem.TotalAlloc/1024/
    log.Println("mem.HeapAlloc: ", mem.HeapAlloc/1024
    log.Println("mem.HeapSys: ", mem.HeapSys/1024/1
    log.Println("mem.HeapObjects: ", mem.HeapObjects
}
```

```
2018/05/02 15:55:36 mem.Sys:  6121
2018/05/02 15:55:36 mem.Alloc:  15
2018/05/02 15:55:36 mem.HeapIdle:  5863
2018/05/02 15:55:36 mem.TotalAlloc:  5882
2018/05/02 15:55:36 mem.HeapAlloc:  15
2018/05/02 15:55:36 mem.HeapSys:  5879
2018/05/02 15:55:36 mem.HeapObjects:  144
```

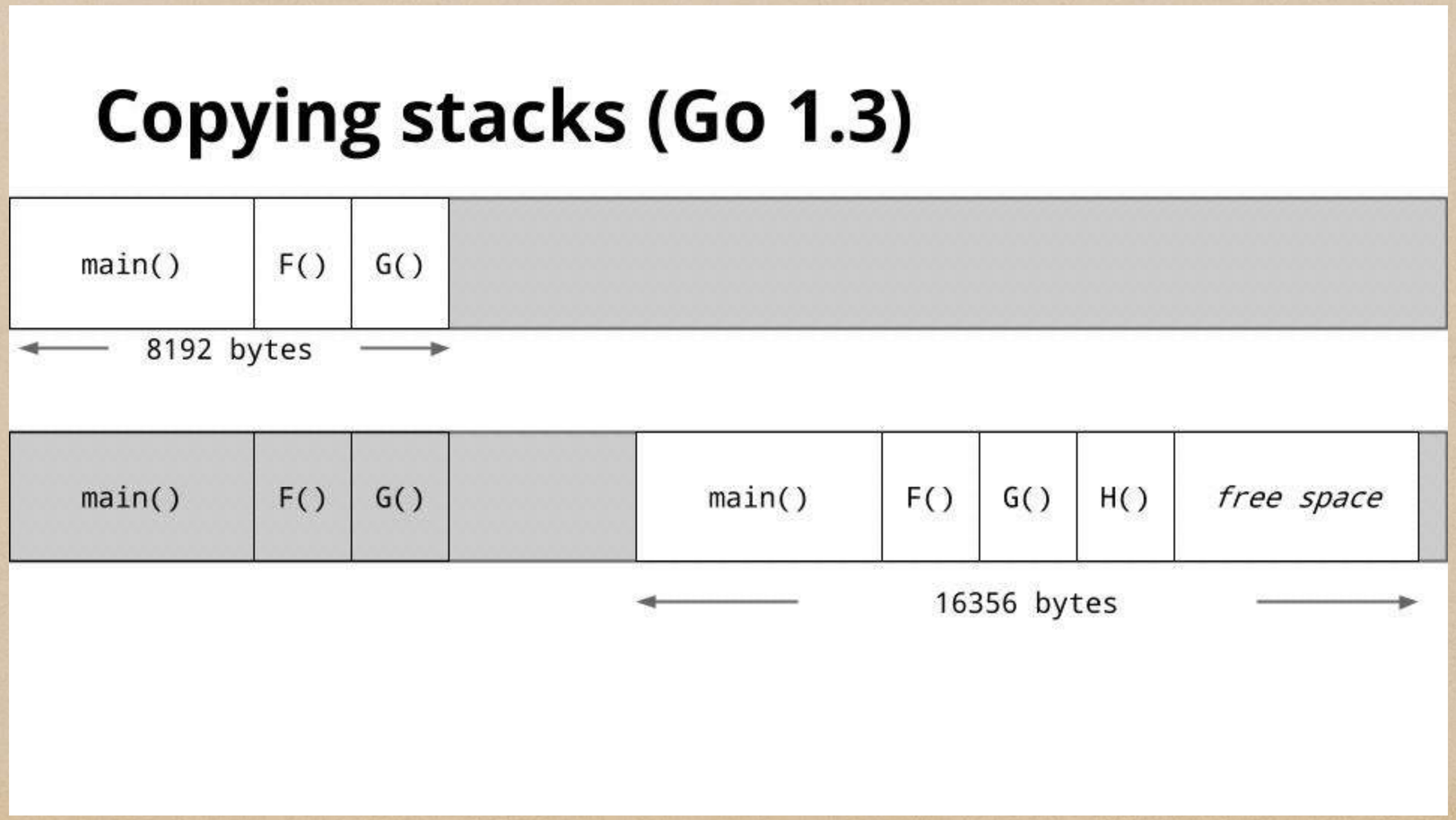
keyword

- 写屏障？
- 并发GC？
- heap 内存什么时候归还系统?
- golang1.9 gc的改进？

stack


Segmented stacks (Go 1.0 - 1.2)
main()
F()
G()
8192 bytes
main()
F()
G()
H()
main()
F()
G()
H()

stack


Copying stacks (Go 1.3)
main()
F()
G()
8192 bytes
main()
F()
G()
main()
F()
G()
H()
free space
16356 bytes

逃逸分析

Stack

Heap

逃逸分析

```go
package main

func foo() *int {
    x := 1
    return &x
}

func main() {
    x := foo()
    println(*x)
}
```

产生引用, 直接堆逃逸

```
0x0000 00000 (g.go:3)   TEXT    "".foo(SB), $24-8
0x0000 00000 (g.go:3)   MOVQ    (TLS), CX
0x0009 00009 (g.go:3)   CMPQ    SP, 16(CX)
0x000d 00013 (g.go:3)   JLS     72
0x000f 00015 (g.go:3)   SUBQ    $24, SP
0x0013 00019 (g.go:3)   MOVQ    BP, 16(SP)
0x0018 00024 (g.go:3)   LEAQ    16(SP), BP
0x001d 00029 (g.go:3)   FUNCDATA        $0, gclocals·2a5
0x001d 00029 (g.go:3)   FUNCDATA        $1, gclocals·33c
0x001d 00029 (g.go:4)   LEAQ    type.int(SB), AX
0x0024 00036 (g.go:4)   MOVQ    AX, (SP)
0x0028 00040 (g.go:4)   PCDATA  $0, $0
0x0028 00040 (g.go:4)   CALL    runtime.newobject(SB)
0x002d 00045 (g.go:4)   MOVQ    8(SP), AX
```

```
[ruifengyun@xiaorui ~ ]$ go tool compile -m g.go
g.go:3:6: can inline foo
g.go:8:6: can inline main
g.go:9:13: inlining call to foo
g.go:5:12: &x escapes to heap
g.go:4:5: moved to heap: x
g.go:9:13: main &x does not escape
```

逃逸分析

```go
package main

import (
    "fmt"
)

func main() {
    var a [1]int
    c := a[:]
    fmt.Println(c)
}
```

```
gyun@xiaorui ~ ]$ go tool compile -S k.go
STEXT size=183 args=0x0 locals=0x60
0x0000 00000 (k.go:7)   TEXT    "".main(SB), $96-0
0x0000 00000 (k.go:7)   MOVQ    (TLS), CX
0x0009 00009 (k.go:7)   CMPQ    SP, 16(CX)
0x000d 00013 (k.go:7)   JLS     173
0x0013 00019 (k.go:7)   SUBQ    $96, SP
0x0017 00023 (k.go:7)   MOVQ    BP, 88(SP)
0x001c 00028 (k.go:7)   LEAQ    88(SP), BP
0x0021 00033 (k.go:7)   FUNCDATA        $0, gclocals·69c1753bd5
0x0021 00033 (k.go:7)   FUNCDATA        $1, gclocals·57cc5e9a02
0x0021 00033 (k.go:8)   LEAQ    type.[1]int(SB), AX
0x0028 00040 (k.go:8)   MOVQ    AX, (SP)
0x002c 00044 (k.go:8)   PCDATA  $0, $0
0x002c 00044 (k.go:8)   CALL    runtime.newobject(SB)
0x0031 00049 (k.go:8)   MOVQ    8(SP), AX
0x0036 00054 (k.go:10)  MOVQ    AX, ""..autotmp_4+64(SP)
0x003b 00059 (k.go:10)  MOVQ    $1, ""..autotmp_4+72(SP)
0x0044 00068 (k.go:10)  MOVQ    $1, ""..autotmp_4+80(SP)
0x004d 00077 (k.go:10)  XORPS   X0, X0
```

```
[ruifengyun@xiaorui ~ ]$ go tool compile -m k.go
k.go:10:16: c escapes to heap
k.go:9:11: a escapes to heap
k.go:8:9: moved to heap: a
k.go:10:16: main ... argument does not escape
```

fmt引起的对象堆逃逸...

逃逸分析

所以;
不要肯定的认为 某个 对象在heap or stack上 .

- 分析汇编
  - go tool compile -S xxx.go
- 对象分布分析
  - go tool compile -m xxx.go

netpoller

```
type pollDesc struct {
    link *pollDesc
    lock    mutex
    fd      uintptr
    closing bool
    ...
}
```

func (fd *FD) Read() -->

func (pd *pollDesc) wait -->

func poll_runtime_pollWait -->

func netpollblock()

阻塞中 ...

```
func (fd *FD) Read(p []byte) (int, error) {
    if err := fd.pd.prepareRead(fd.isFile); err != nil {
        return 0, err
    }
    ...
    for {
        n, err := syscall.Read(fd.Sysfd, p)
        if err != nil {
            n = 0
            if err == syscall.EAGAIN && fd.pd.pollable() {
                if err = fd.pd.waitRead(fd.isFile); err == nil {
                    continue
                }
            }
        }
        err = fd.eofError(n, err)
        return n, err
    }
}
```

```
func netpollblock(pd *pollDesc, mode int32, waitio bool) bool {
    for {
        old := *gpp
        if old == pdReady {
            *gpp = 0
            return true
        }
        if old != 0 {
            throw("runtime: double wait")
        }
        ...
    }
        if waitio || netpollcheckerr(pd, mode) == 0 {
        gopark(netpollblockcommit, unsafe.Pointer(gpp), "IO wait", trac
    }
    ...
    return old == pdReady
}
```

netpoller

```
    //判断获取最后一次从网络I/O轮循查找G的时间
    if lastpoll != 0 && lastpoll+10*1000*1000 < now {
    //更新最后一次查询G时间，为了下一次做判断。
    atomic.Cas64(&sched.lastpoll, uint64(lastpoll), uint64(now))
    //从网络轮询器查找已经就绪的，这里是非阻塞读取。
    gp := netpoll(false)

    if gp != nil {
        incidlelocked(-1)
        //找到后注入到调度器下面的可获取的G队列
        injectglist(gp)
        incidlelocked(1)
    }
}
```

sysmon

```
func netpoll(block bool) *g {
    if epfd == -1 {
        return nil
    }
    // ...
    n := epollwait(epfd, &events[0], int32(len(events)), waitms)
    if n < 0 {
        if n != -_EINTR {
            println("runtime: epollwait on fd", epfd, "failed with"
            throw("epollwait failed")
        }
        goto retry
    }
    var gp guintptr
    for i := int32(0); i < n; i++ {
        ev := &events[i]
        if ev.events == 0 {
            continue
        }
        var mode int32
        if ev.events&(_EPOLLIN|_EPOLLRDHUP|_EPOLLHUP|_EPOLLERR) !=
            mode += 'r'
        }
        if ev.events&(_EPOLLOUT|_EPOLLHUP|_EPOLLERR) != 0 {
            mode += 'w'
```

唤醒中 ...

go timer

timer.After

timer.NewTimer

timer.Tick

```
c := make(chan Time, 1)
t := &Ticker{ // &Timer
    C: c,
    r: runtimeTimer{
        when:   when(d),
        period: int64(d),
        f:      sendTime,
        arg:    c,
    },
}
startTimer(&t.r)
return t
```

```
func sendTime(c interface{}, seq uintptr) {
    select {
    case c.(chan Time) <- Now():
    default:
    }
}
```

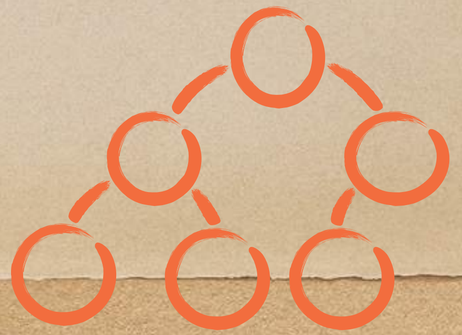
```
func startTimer(t *timer) {
    ...
    addtimer(t)
}

func addtimer(t *timer) {
    lock(&timers.lock)
    addtimerLocked(t)
    unlock(&timers.lock)
}

func addtimerLocked(t *timer) {
    ...
    // 添加任务
    timers.t = append(timers.t, t)
    siftupTimer(t.i)
    ...
    if !timers.created {
        timers.created = true
        go timerproc() // 异步任务，只初始化一次
    }
```

go timer

```go
func timerproc(tb *timersBucket) {
    tb.gp = getg()
    for {
        lock(&tb.lock)
        tb.sleeping = false
        now := nanotime()
        delta := int64(-1)
        for {
            if len(tb.t) == 0 { //无timer的情况
                delta = -1
                break
            }
            t := tb.t[0] //拿到堆顶的timer
            delta = t.when - now
            if delta > 0 { // 所有timer的时间都没有到期
                break
            }
            if t.period > 0 { // t[0] 是ticker类型，调整其到期时间并
                // leave in heap but adjust next time to fire
                t.when += t.period * (1 + -delta/t.period)
                siftdownTimer(tb.t, 0)
            } else {
                // Timer类型的定时器是单次的，所以这里需要将其从堆里面删
                last := len(tb.t) - 1
                if last > 0 {
                    tb.t[0] = tb.t[last]
                    tb.t[0].i = 0
                }
                tb.t[last] = nil
```

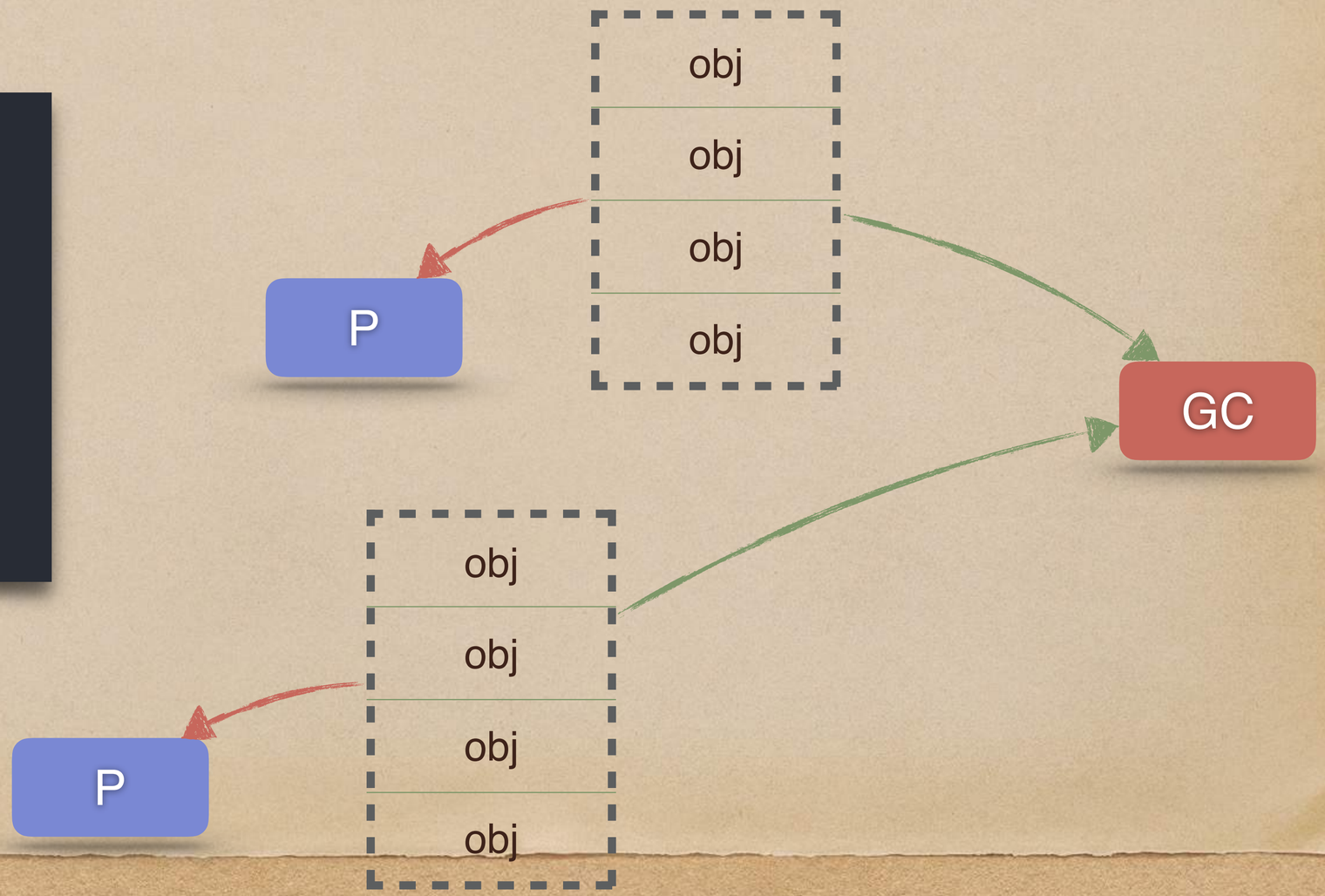
```go
        f := t.f
        arg := t.arg
        seq := t.seq
        unlock(&tb.lock)
        if raceenabled {
            raceacquire(unsafe.Pointer(t))
        }
        f(arg, seq) // 调用定时器处理函数
        lock(&tb.lock)
    }
    if delta < 0 || faketime > 0 {
        // No timers left - put goroutine to sleep.
        tb.rescheduling = true
        goparkunlock(&tb.lock, "timer goroutine (idle)", traceEv
        continue
    }
    // 每次插入定时任务都会修改轮询器的最小的等待时间
    tb.sleeping = true
    tb.sleepUntil = now + delta
    noteclear(&tb.waitnote) // 清理timer轮询器
    unlock(&tb.lock)
    notetsleepg(&tb.waitnote, delta) // 重置timer轮询器
```

小顶堆



sync.pool

```
var (
    gbuffer = &sync.Pool{
        New: func() interface{} {
            return &bytes.Buffer{}
        },
    }
)

func writeBuffer() {
    buf := gbuffer.Get().(*bytes.Buffer)
    defer gbuffer.Put(buf)

}
```



sync.pool

- 使用slice做缓冲对象存储
- mutex 保护 shared slice
- shared会被偷走, private 留个本

```go
// Put adds x to the pool.
func (p *Pool) Put(x interface{}) {
    if x == nil {
        return
    }
    ...
    l := p.pin()
    if l.private == nil {
        l.private = x
        x = nil
    }
    if x != nil {
        l.Lock()
        l.shared = append(l.shared, x)
        l.Unlock()
```

```go
// Local per-P Pool appendix.
type poolLocalInternal struct {
    private interface{}   // Can be used only by t
    shared  []interface{} // Can be used by any P.
    Mutex                 // Protects shared.
}
```

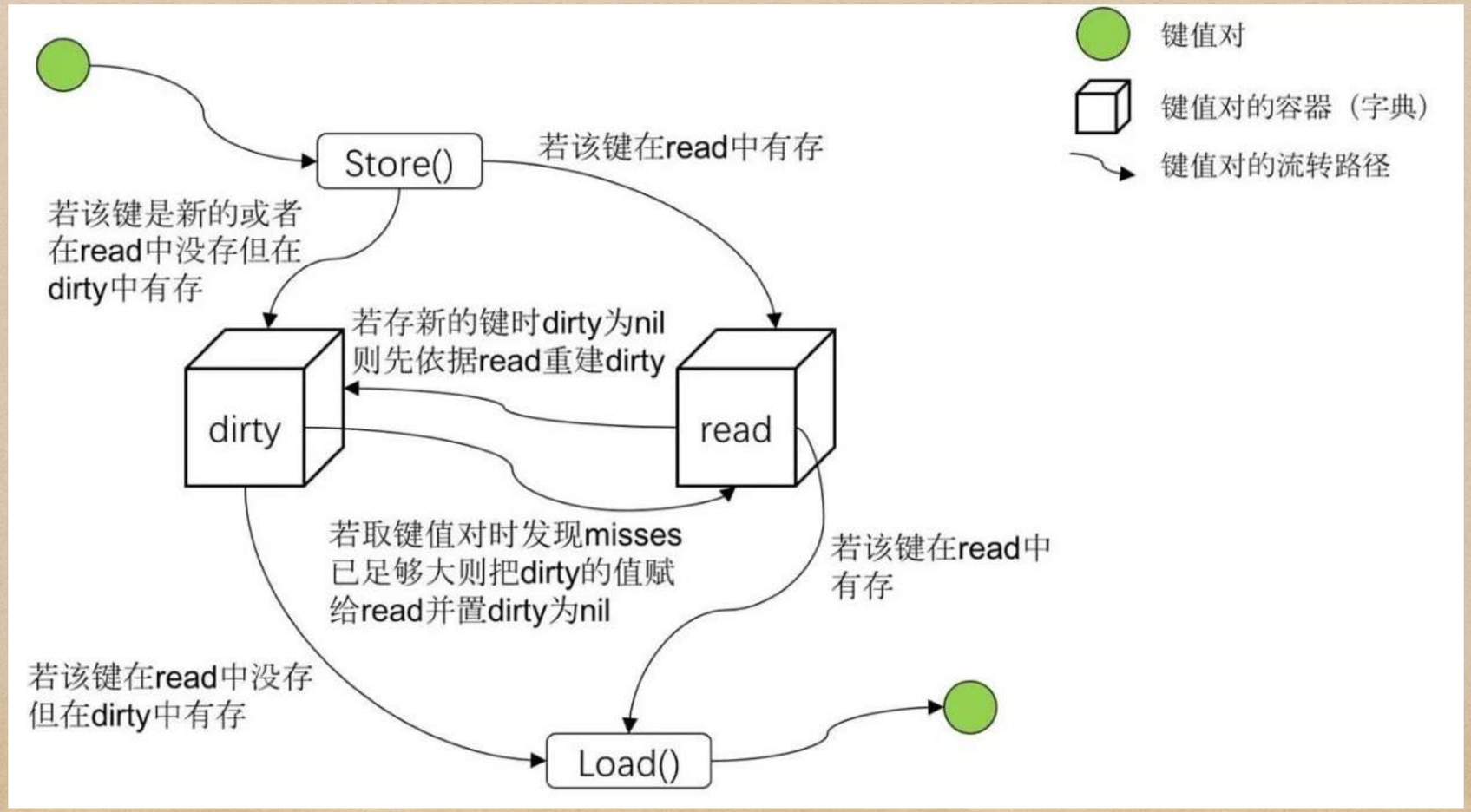
```go
func (p *Pool) pin() *poolLocal {
    pid := runtime_procPin()
    s := atomic.LoadUintptr(&p.localSize) // load-acquire
    l := p.local                          // load-consume
    if uintptr(pid) < s {
        return indexLocal(l, pid)
    }
    return p.pinSlow()
}
```

```go
func (p *Pool) Get() interface{} {
    ...
    l := p.pin()
    x := l.private
    l.private = nil
    if x == nil {
        l.Lock()
        last := len(l.shared) - 1
        if last >= 0 {
            x = l.shared[last]
            l.shared = l.shared[:last]
        }
        l.Unlock()
        if x == nil {
            x = p.getSlow()
        }
    }
    ...
    if x == nil && p.New != nil {
        x = p.New()
    }
    return x
```

sync.map

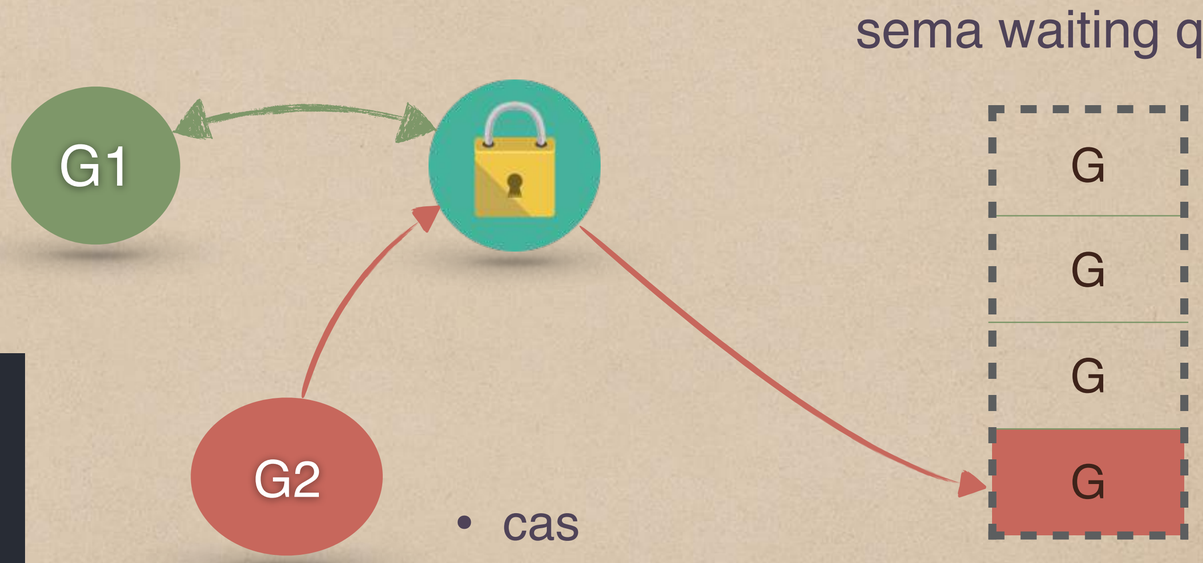
```
type Map struct {
    // 当涉及到dirty数据的操作的时候，需要使用这个锁
    mu Mutex

    // 一个只读的数据结构，因为只读，所以不会有读写冲突。
    read atomic.Value // readOnly

    // 对于dirty的操作需要加锁，因为对它的操作可能会有读写竞争。
    dirty map[interface{}]*entry

    // 当misses累积到 dirty的长度的时候
    //就会将dirty提升为read,避免从dirty中miss太多次。因为操作dirty需要加锁。
    misses int
}
```

sync.map


键值对
键值对的容器（字典）
键值对的流转路径
Store()
若该键在read中有存
若该键是新的或者
在read中没存但在
dirty中有存
若存新的键时dirty为nil
则先依据read重建dirty
dirty
read
若取键值对时发现misses
已足够大则把dirty的值赋
给read并置dirty为nil
若该键在read中
有存
若该键在read中没存
但在dirty中有存
Load()

sync.Mutex

```go
// A Mutex must not be copied after first use.
type Mutex struct {
    state int32
    sema  uint32
}

const (
    mutexLocked = 1 << iota // mutex is locked
    mutexWoken
    mutexStarving
    mutexWaiterShift = iota
```

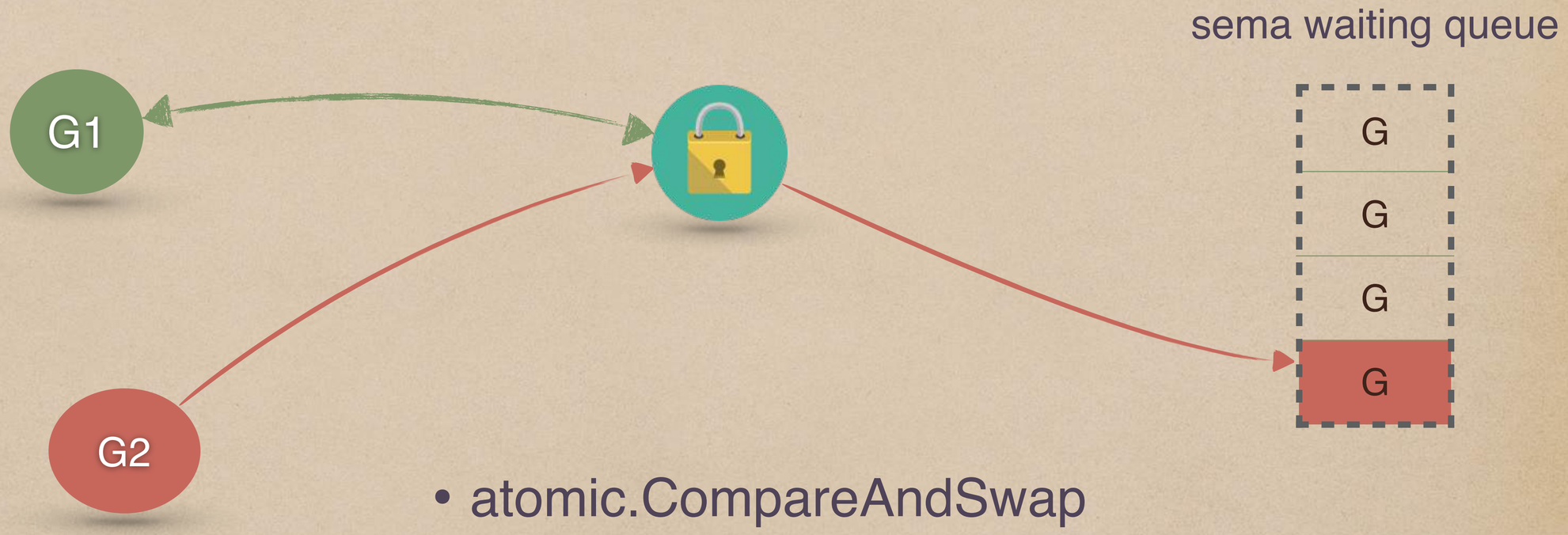


- cas
- 适当减少syscall
- 通过runtime调度唤醒

```go
}
if atomic.CompareAndSwapInt32(&m.state, old, new) {
    if old&(mutexLocked|mutexStarving) == 0 {
        break // locked the mutex with CAS
    }
    // If we were already waiting before, queue at the front of the queue.
    queueLifo := waitStartTime != 0
    if waitStartTime == 0 {
        waitStartTime = runtime_nanotime()
    }
    runtime_SemacquireMutex(&m.sema, queueLifo)
    starving = starving || runtime_nanotime()-waitStartTime > starvationThresholdNs
    old = m.state
    if old&mutexStarving != 0 {
        // If this goroutine was woken and mutex is in starvation mode,
        // ownership was handed off to us but mutex is in somewhat
        // inconsistent state: mutexLocked is not set and we are still
        // accounted as waiter. Fix that.
        if old&(mutexLocked|mutexWoken) != 0 || old>>mutexWaiterShift == 0 {
            throw("sync: inconsistent mutex state")
        }
        delta := int32(mutexLocked - 1<<mutexWaiterShift)
        if !starving || old>>mutexWaiterShift == 1 {
```

```go
// Grab the right to wake someone.
new = (old - 1<<mutexWaiterShift) | mutexWoken
if atomic.CompareAndSwapInt32(&m.state, old, new) {
    runtime_Semrelease(&m.sema, false)
    return
}
old = m.state
```

sync.Mutex



- atomic.CompareAndSwap
- 适当减少syscall
- 通过runtime调度唤醒

```go
// A Mutex must not be copied after first use.
type Mutex struct {
    state int32
    sema  uint32
}

const (
    mutexLocked = 1 << iota // mutex is locked
    mutexWoken
    mutexStarving
    mutexWaiterShift = iota
```

- 双检查锁机制
- 使用互斥量保护wait queue

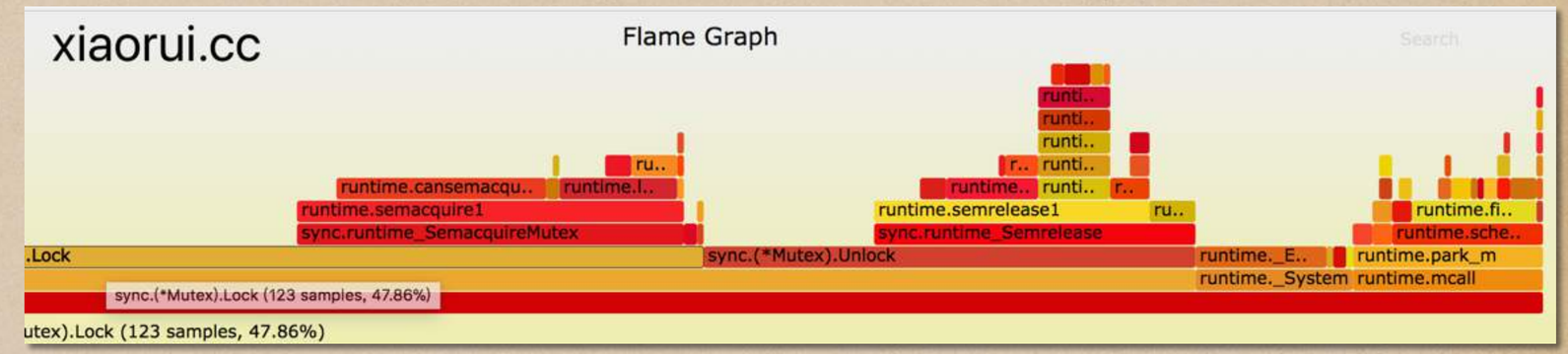
```go
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        // 拿锁成功
        return
    }

    for {
        old := m.state
        new := old | mutexLocked
        if old&mutexLocked != 0 {
            if runtime_canSpin(iter) {
                if !awoke && old&mutexWoken == 0 && old>>mutexWa
                    atomic.CompareAndSwapInt32(&m.state, old, ol
                }
            }
        ...
        if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&mutexLocked == 0 {
                break // 拿锁成功
            }
            // 拿锁失败
            runtime_Semacquire(&m.sema)
```

```go
func semacquire(addr *uint32, profile bool) {
    ...
    for {
        lock(&root.lock)
        // Add ourselves to nwait to disable "easy case" in semreleas
        atomic.Xadd(&root.nwait, 1)
        // Check cansemacquire to avoid missed wakeup.
        if cansemacquire(addr) {
            atomic.Xadd(&root.nwait, -1)
            unlock(&root.lock)
            break
        }

        root.queue(addr, s)
        goparkunlock(&root.lock, "semacquire", traceEvGoBlockSync, 4)
        if cansemacquire(addr) {
            break
        }
```

```go
    }
    if atomic.CompareAndSwapInt32(&m.state, old, new) {
        if old&(mutexLocked|mutexStarving) == 0 {
            break // locked the mutex with CAS
        }
        // If we were already waiting before, queue at the front of the queue.
        queueLifo := waitStartTime != 0
        if waitStartTime == 0 {
            waitStartTime = runtime_nanotime()
        }
        runtime_SemacquireMutex(&m.sema, queueLifo)
        starving = starving || runtime_nanotime()-waitStartTime > starvationThresholdNs
        old = m.state
        if old&mutexStarving != 0 {
            // If this goroutine was woken and mutex is in starvation mode,
            // ownership was handed off to us but mutex is in somewhat
            // inconsistent state: mutexLocked is not set and we are still
            // accounted as waiter. Fix that.
            if old&(mutexLocked|mutexWoken) != 0 || old>>mutexWaiterShift == 0 {
                throw("sync: inconsistent mutex state")
            }
            delta := int32(mutexLocked - 1<<mutexWaiterShift)
            if !starving || old>>mutexWaiterShift == 1 {
```

sync.Mutex

```go
for i := 0; i < 200; i++ {
    wg.Add(1)
    go func(wg *sync.WaitGroup, i int) {
        for i := 0; i < 50000; i++ {
            lock.Lock()
            counter += 1
            lock.Unlock()
        }
        wg.Done()
    }(&wg, i)
}
```



Mutex也不廉价

% time	seconds	usecs/call	calls	errors	syscall
64.94	80.950334	237	341599	136495	futex
34.67	43.213808	100	432249		pselect6
0.39	0.489443	1	887958		clock_gettime
0.00	0.000033	11	3		rt_sigprocmask
0.00	0.000000	0	6		write

sync.Mutex

- sync.Mutex 锁性能不是问题
  - noDeferLock 轻易干到 5000w/s
  - DeferLock 也可以到 1700w/s
- 锁竞争才是最大问题

```
In [3]: count = 100000000

In [4]: one_cost = 19.1

In [5]: '%.3f'%(float(count)/(count * one_cost /1000/1000/1000))
Out[5]: '52356020.942'
```

```
goos: darwin
goarch: amd64
BenchmarkDeferLock-4            20000000                59.4 ns/op             0 B/op          0 allocs/op
BenchmarkNoDeferLock-4          100000000               19.1 ns/op             0 B/op          0 allocs/op
BenchmarkMultiNoDeferLock-4       1000000             1099 ns/op               0 B/op          0 allocs/op
PASS
ok      _/Users/ruifengyun/aaa  4.295s
```

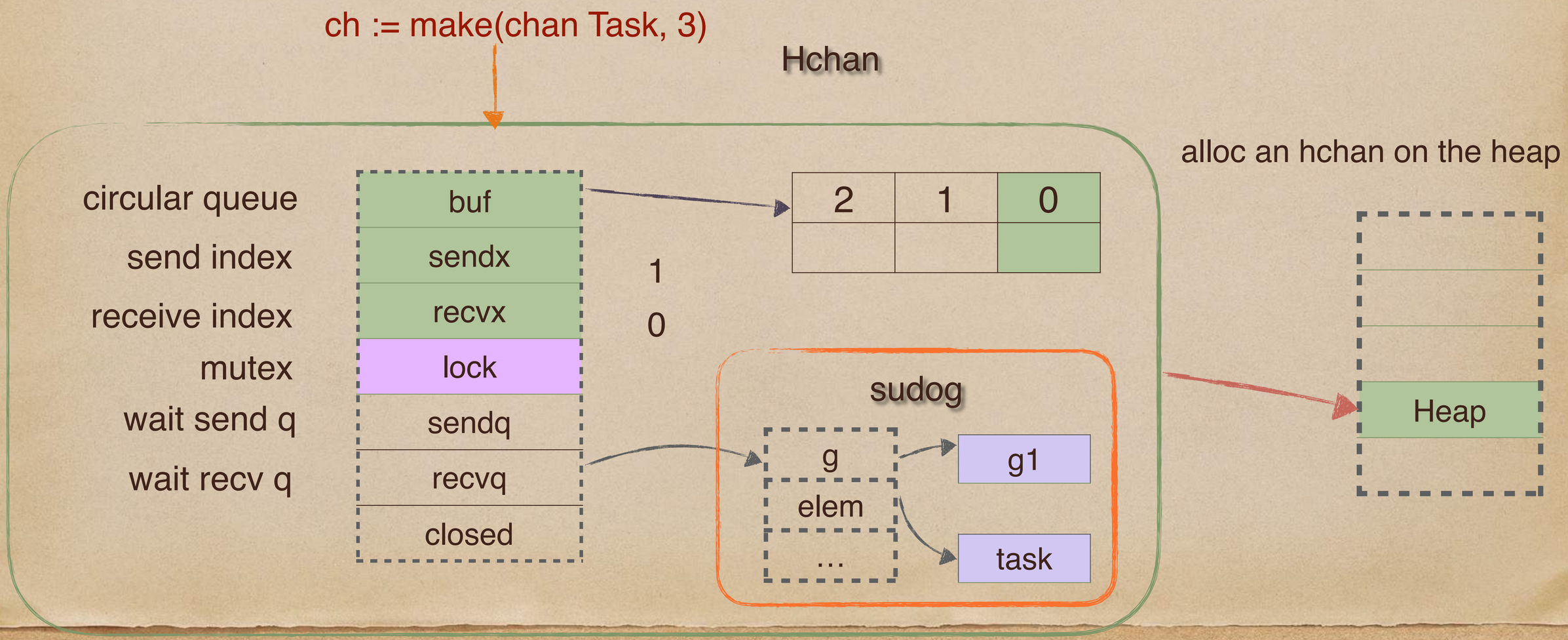
1099/10 = 109ns

open 10个Goroutine

思考？

- 为什么已经有cas指令，还在syscall futex？
- golang 底层都谁在调用futex锁？
- 是否可以适当规避futex的使用？

channel设计


ch := make(chan Task, 3)
Hchan
circular queue
send index
receive index
mutex
wait send q
wait recv q
buf
sendx
recvx
lock
sendq
recvq
closed
1
0
2
1
0
sudog
g
elem
...
g1
task
alloc an hchan on the heap
Heap

send full channel?

G1

ch <- task, but chan is full

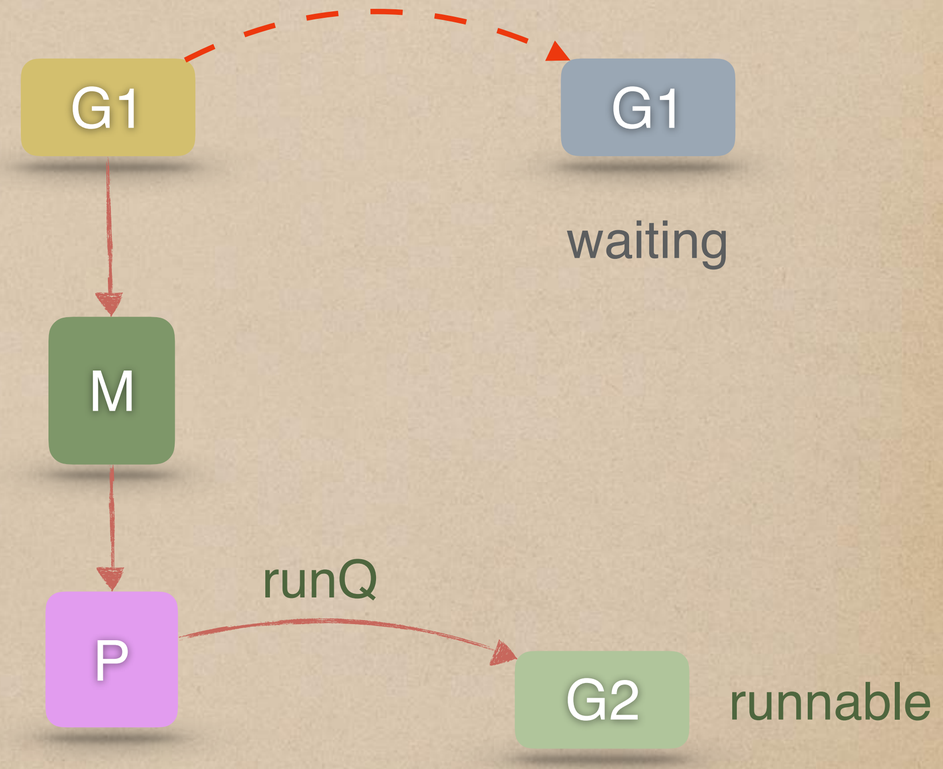
is block, so gopark()

calls into the scheduler

set G1 to waiting

unlink G1, M

schedule a runnable G
from the P runqueue



resuming goroutines

G2

task := <- chan, then sender can run

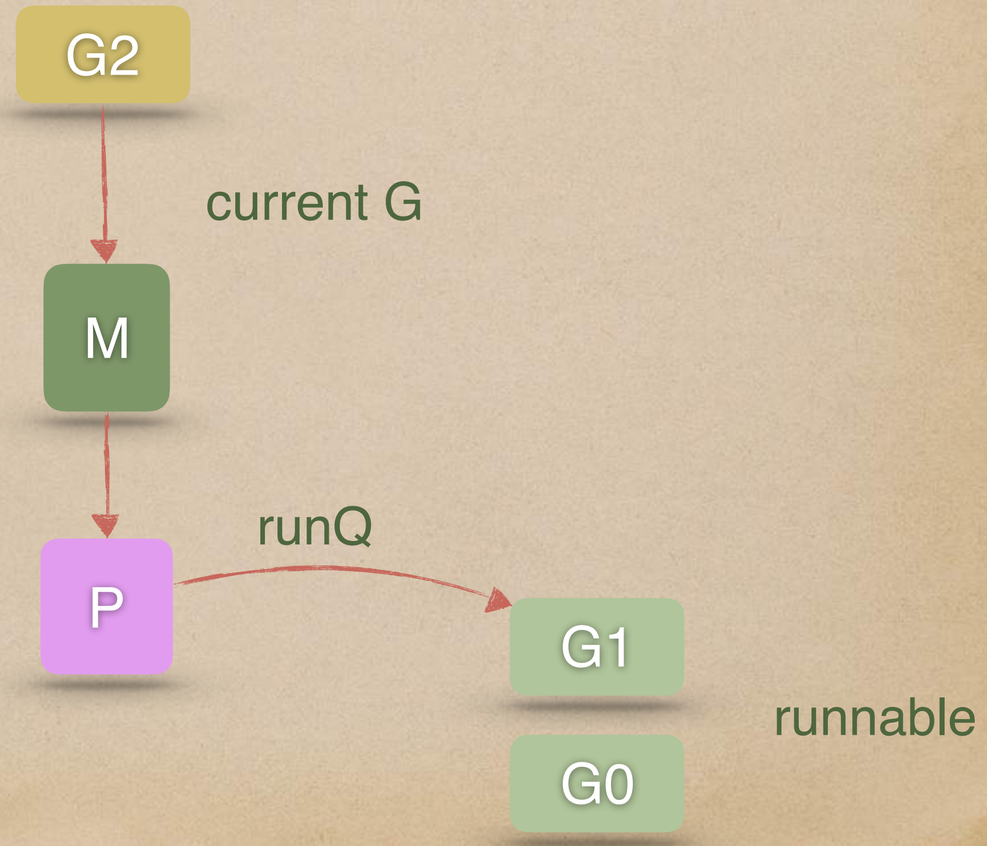
goready(G1)

calls into the scheduler

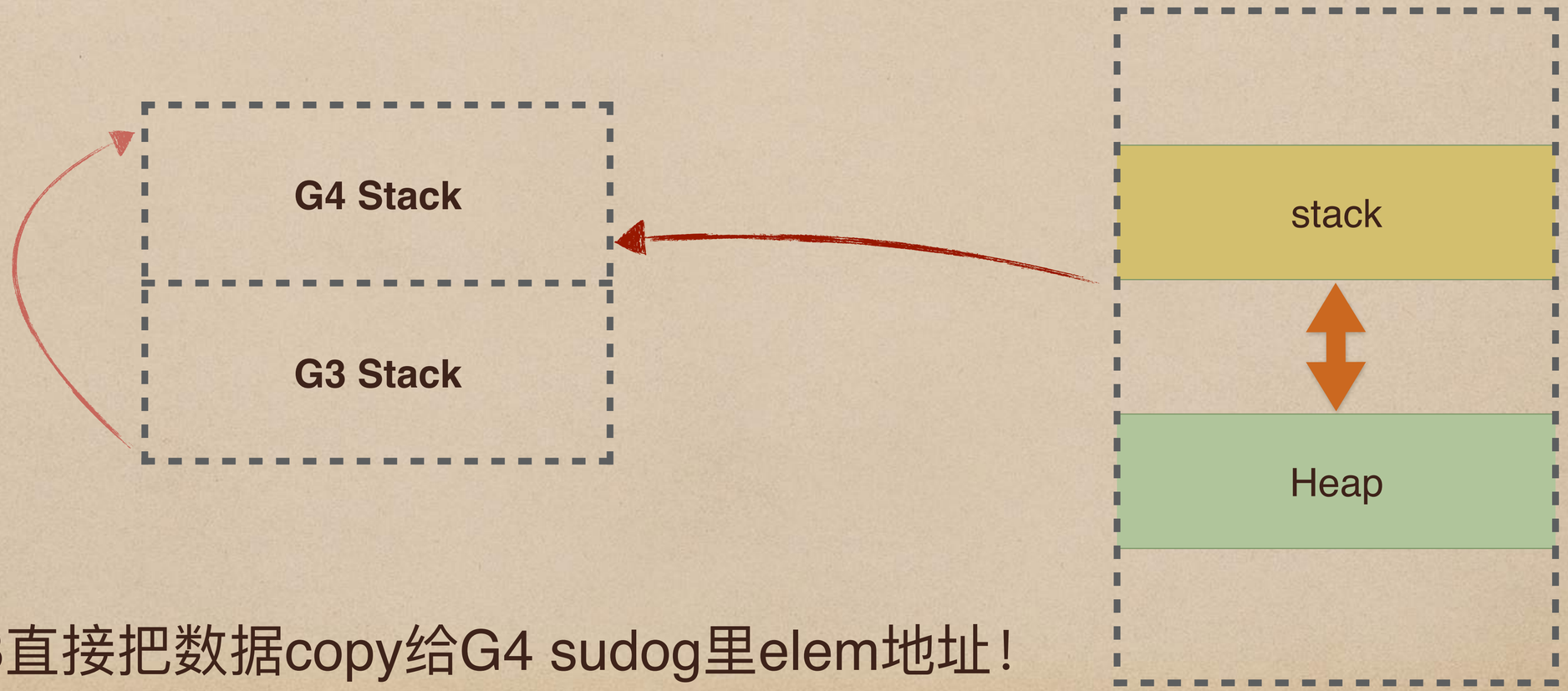
set G1 to runnable

puts it on runqueue

return to G2



direct write



G3直接把数据copy给G4 sudog里elem地址！

tools

- go tool pprof
- go tool trace
- go-torch
- go test -bench=. -benchmem ...

link

- https://github.com/qyuhen
- https://www.youtube.com/watch?v=C1EtfDnsdDs
- https://github.com/golang/go/tree/master/src
- https://github.com/golang/go/issues

“Q & A”

–rfyiamcool