本文基于go1.17.6

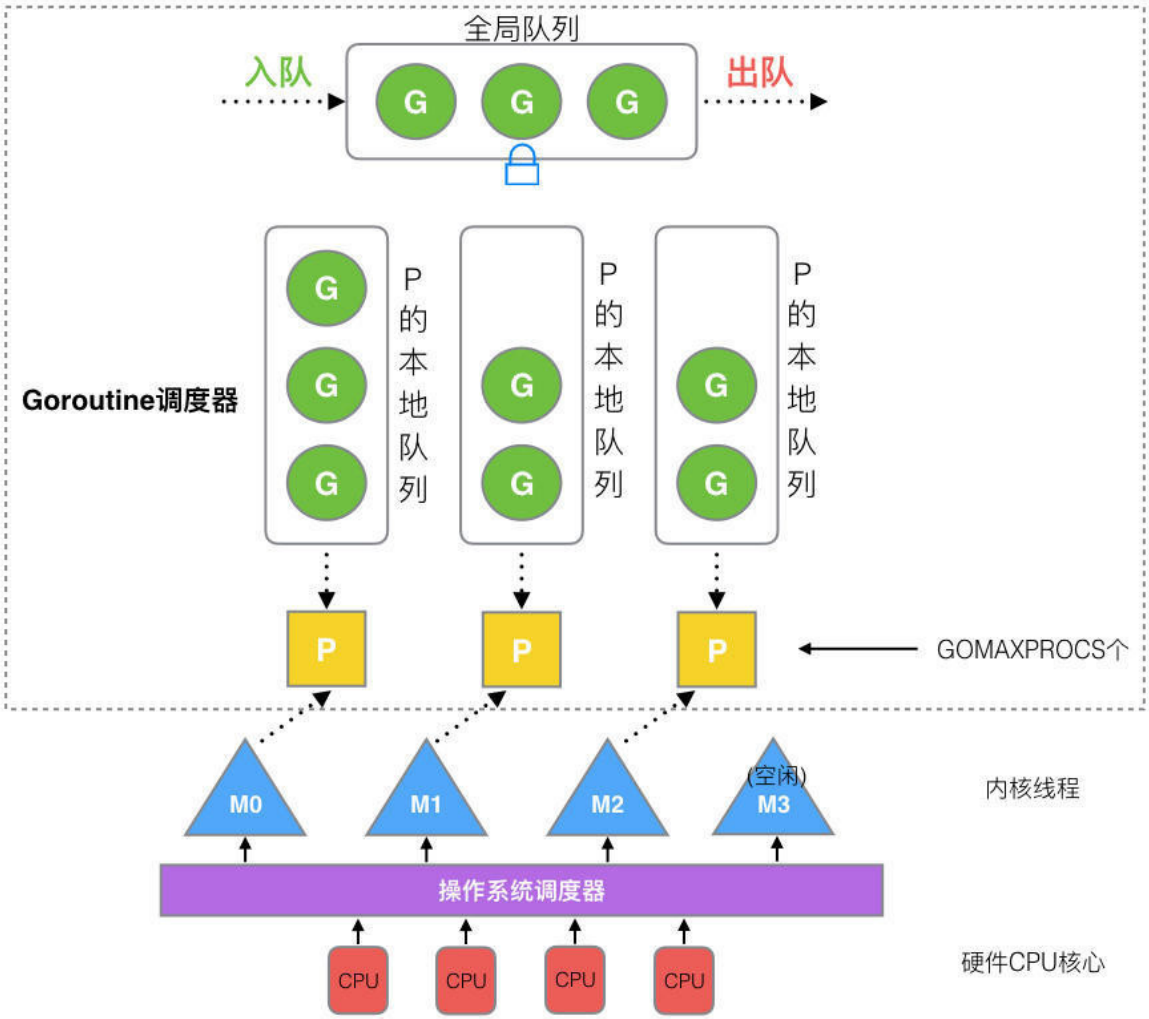
1.什么是协程?

协程，英文Coroutines，是一种基于线程之上，但又比线程更加轻量级的存在，这种由程序员自己写程序来管理的轻量级线程叫做『用户空间线程』， 具有对内核来说不可见的特性。

1.1 协程的特点

- 线程的切换由操作系统负责调度，协程由用户自己进行调度
- 线程的默认Stack大小是MB级别，而协程更轻量，接近KB级别。
- 在同一个线程上的多个协程，访问某些资源可以不需要锁
- 适用于被阻塞的，且需要大量并发的场景。

1.2 Golang的GMP模型

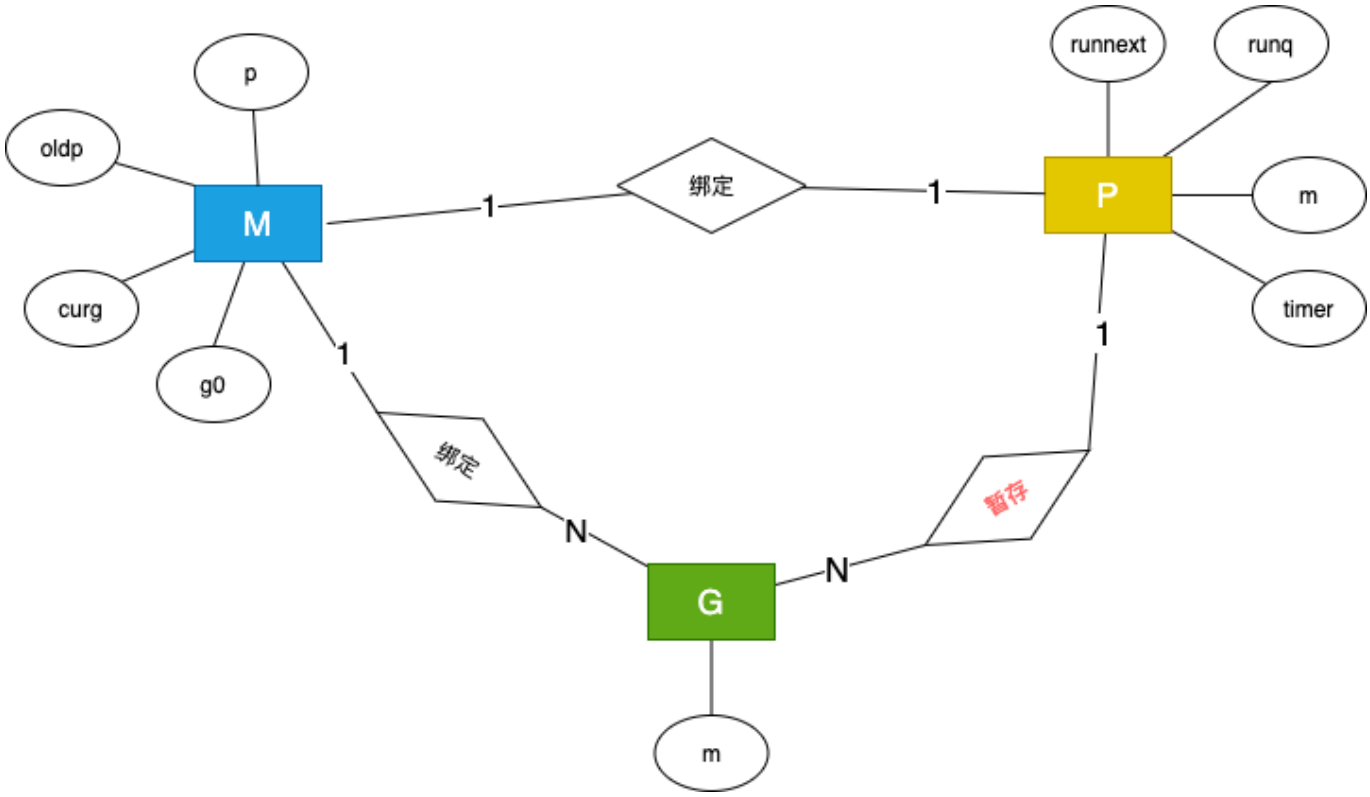


- CPU驱动线程上的任务执行
- 线程由操作系统内核进行调度，Goroutine由Golang运行时(runtime)进行调度
- P的 `local runnable queue`是无锁的，`global runnable queue`是有锁的
- P的 `local runnable queue`长度限制为256

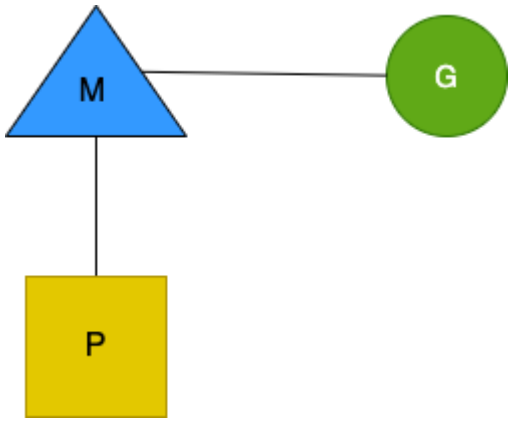
注意:

- 1. M和P是绑定关系
- 2. M和G是绑定关系
- 3. P只是暂存G，他们之间不是绑定关系

E-R图



简化后的E-R图



注意: 后面为了书写简单直接将

- `local runnable queue`表示为本地队列
- `global runnable queue`表示为全局队列

延伸

timer的四叉堆和内存分配器使用的mcache也是每个P一个

Q: 为什么默认情况下P的数量与CPU数量一致?

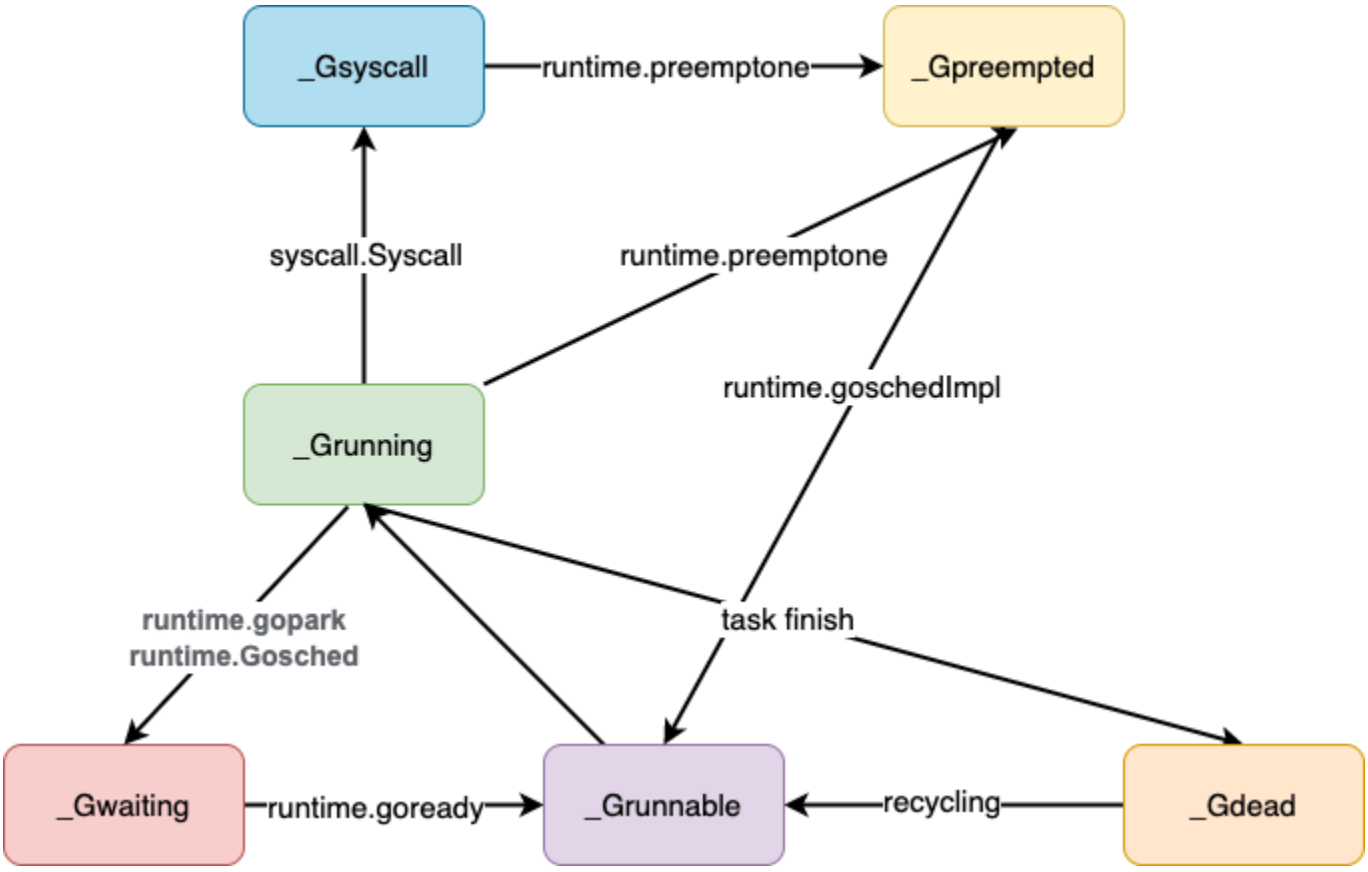
A:

这样可以避免把CPU时间浪费在上线文切换上

1.3 协程和线程的资源消耗对比

类别	栈内存	上下文切换	备注
Thread	1MB	1us	内存占用使用的是Java线程的默认栈大小
Goroutine	4KB	0.2us	内存占用使用的是Linux+x86下的栈大小

2. 常见的Goroutine 让出/调度/抢占场景



2.1 协程被创建

- 1)P的本地队列有剩余空间时，放入P的本地队列
- 2)P的本地队列没有剩余空间时，将本地队列的一部分Goroutine以及待加入的Goroutine添加到全局队列

2.2 主动让出

2.2.1 使用runtime.Gosched

gosched.go

```
package main
```

```
import (
    "fmt"
    "runtime"
)

func main() {
    runtime.GOMAXPROCS(1)
    for i := 0; i < 100; i++ {
        fmt.Println("Goroutine1:", i)
        //time.Sleep(500 * time.Millisecond)
        if i == 5 {
            go func() {
                for i := 0; i < 100; i++ {
                    fmt.Println("Goroutine2:", i)
                    //time.Sleep(500 * time.Millisecond)
                }
            }()
            runtime.Gosched()
        }
    }
}
```

具体执行流程

```
1.Gosched() -> 2.mcall(fn func(*g)) -> 3.gosched_m(gp *g) ->
4.goschedImpl(gp *g) ->  dropg()
                         globrunqput()
                         schedule()
```

step2 mcall(fn func(*g))

mcall函数是通过汇编实现的， 在 asm_amd64.s 中

- 1)保存当前goroutine的状态(PC/SP)到g->sched中，方便下次调度；
- 2)切换到m->g0的栈；
- 3)然后g0的堆栈上调用fn；

注意：每个m上都有一个自己g0，仅用于调度，不指向任何可执行的函数

> mcall returns to the original goroutine g later, when g has been rescheduled. fn must not return at all; typically it ends by calling schedule, to let the m run other goroutines.

step4 goschedImpl(gp *g)

- 1)修改Goroutine的状态 _Grunning -> _Grunnable
- 2)dropg() 将G和M解绑
- 3)globrunqput(gp) 将G放入全局runnable队列
- 4)schedule() 进行一轮调度，寻找一个runnable的G，并执行它，**函数不会返回**

step4-4 schedule()

```
schedule() -> 1.findrunnable()
              2.execute() ->gogo()
```

step4-4 schedule() --> findrunnable()

- 1）从同一个P的本地runnable队列中
- 2）从全局的runnable队列中
- 3）从网络轮训器(netpoll)中，是否有事件就绪的G
- 4）通过runtime.runqsteal从其它P的本地runnable队列偷取一半G放入本地runnable队列，并取出一个用来执行

`gogo()` 汇编实现，用于恢复现场(PC/SP)，运行上一步找到的新的可运行的G

Q:为什么一定要单独设置一个g0来执行goschedImpl(gp *g)

A:schedule()会将G的stack搞乱

2.2.2 任务执行完毕

```
goexit1() -> mcall(fn func(*g)) -> goexit0(gp *g)
```

```
// goexit continuation on g0.
func goexit0(gp *g) {
    _g_ := getg()
    casgstatus(gp, _Grunning, _Gdead)
    gp.m = nil
    dropg()
    gfput(_g_.m.p.ptr(), gp)
    schedule()
}
```

- 1）修改G的状态 _Grunning -> _Gdead
- 2）解除G和M的绑定关系
- 3）将G放入P的空闲G的链表(gfree list)
- 4）触发一轮调度

2.3 抢占式调度

抢占式调度是由守护进程 sysmon() 触发的 sysmon()是一个特殊的m，它不需要和P进行绑定。

```
func sysmon() {
    for{
```

```
            // 1. 运行计时器
            // 2. 检查网络轮询器(netpoll)
            // 3. 触发抢占式调度
            // 4. 触发GC
        }
    }
```

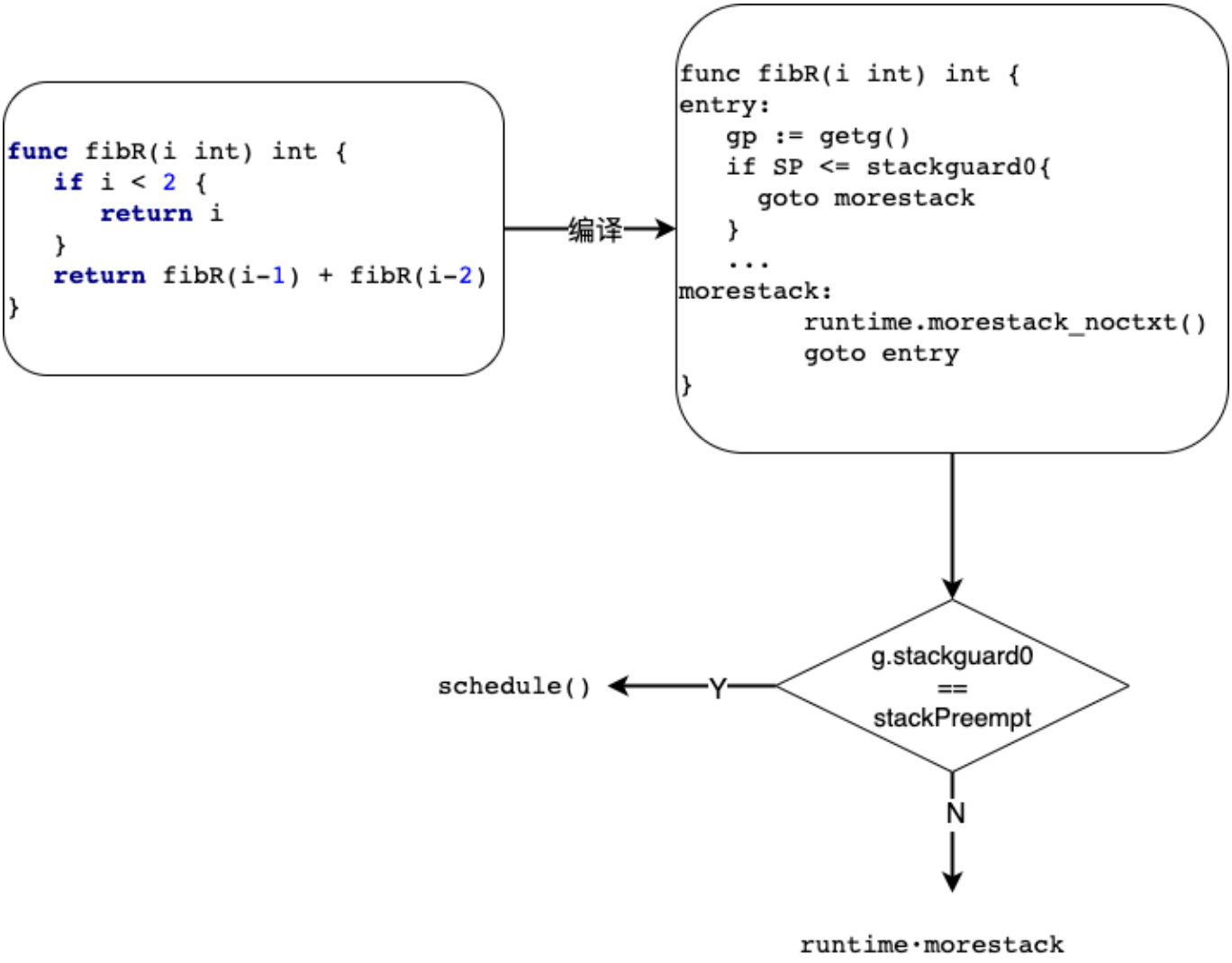
最大时间片是10ms

推荐观看 幼麟实验室的视频

1. 深度探索Go语言：抢占式调度
2. 深度探索Go语言：抢占式调度(2)

2.3.1 基于协作的抢占式调度

依赖栈增长监测代码




```
sysmon() -> retake() -> preemptone()
```

```
type g struct {
    stackguard0 uintptr // offset known to liblink
    preempt     bool // preemption signal, duplicates stackguard0 =
stackpreempt
}
```

2.3.2 基于信号的抢占式调度

发出信号

```
sysmon() -> retake() -> preemptone() -> signalM(mp, sigPreempt)
```

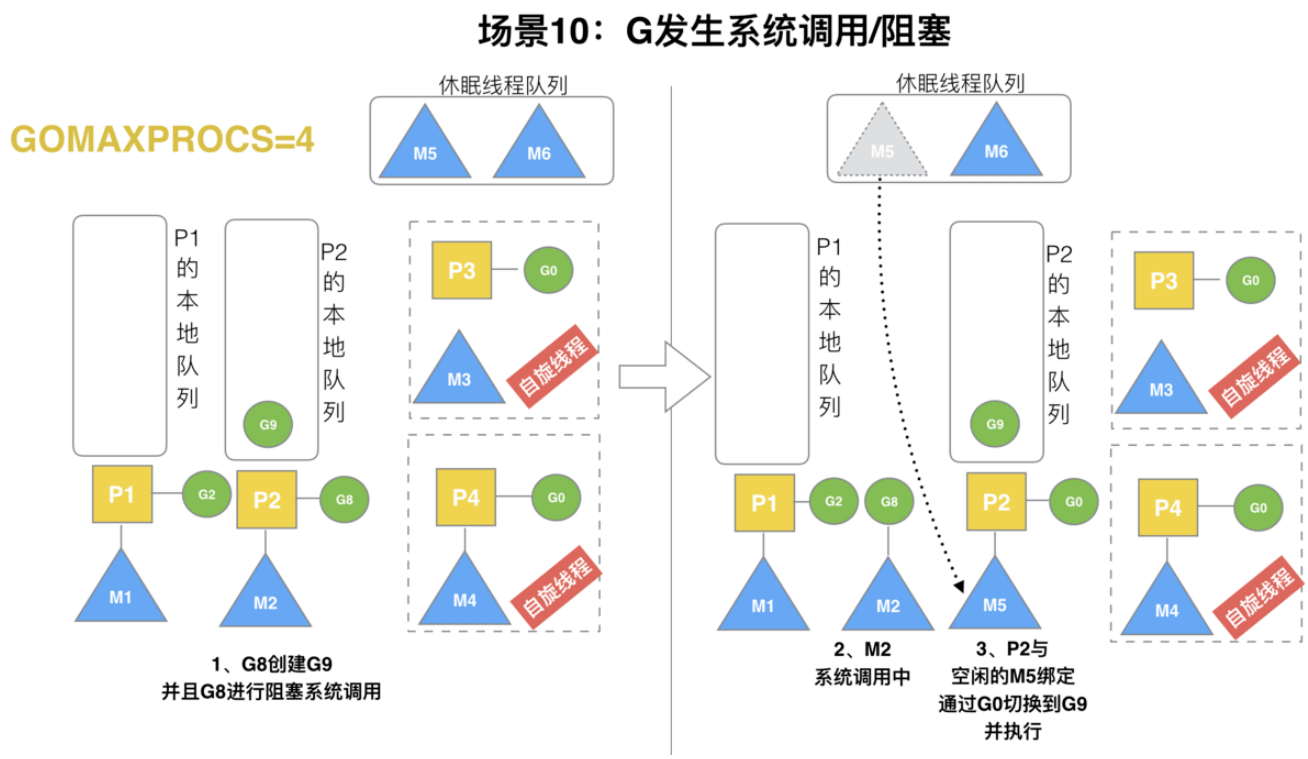
```
func preemptone(_p_ *p) bool {
    mp := _p_.m.ptr()
    if mp == nil || mp == getg().m {
        return false
    }
    gp := mp.curg
    if gp == nil || gp == mp.g0 {
        return false
    }

    gp.preempt = true

    // Every call in a goroutine checks for stack overflow by
    // comparing the current stack pointer to gp->stackguard0.
    // Setting gp->stackguard0 to StackPreempt folds
    // preemption into the normal stack overflow check.
    gp.stackguard0 = stackPreempt

    // Request an async preemption of this P.
    if preemptMSupported && debug.asyncpreemptoff == 0 {
        _p_.preempt = true
        preemptM(mp)
    }

    return true
}
```

接收到信号

```
sighandler() -> doSigPreempt() -> asyncPreempt()->  globalrunqput()
```

asyncPreempt由汇编实现 preempt_amd64.s

2.4 hand off p



场景读写磁盘文件

涉及syscall.Syscall()

注意:

- 由于handeroff机制的存在，读写磁盘文件的过程中，如果IO的压力过大可能会导致大量的系统线程被创建
- 在Golang中(Linux平台)，读写网络连接是非阻塞的边缘触发
- 在Golang中(Linux平台)，读写磁盘文件是阻塞式系统调用

```
File.Read() -> poll.FD.Read() -> syscall.Read() -> syscall.Syscall()
```

poll/fd_unix.go

```
// Read implements io.Reader.
func (fd *FD) Read(p []byte) (int, error) {
    ...
    for {
        n, err := ignoringEINTRIO(syscall.Read, fd.Sysfd, p) // 执行syscall
        if err != nil {
            n = 0
            if err == syscall.EAGAIN && fd.pd.pollable() {
                if err = fd.pd.waitRead(fd.isFile); err == nil { // gopark
                    continue
                }
```

```
            }
        }
        err = fd.eofError(n, err)
        return n, err
    }
}
```

```
func read(fd int, p []byte) (n int, err error) {
    var _p0 unsafe.Pointer
    if len(p) > 0 {
        _p0 = unsafe.Pointer(&p[0])
    } else {
        _p0 = unsafe.Pointer(&_zero)
    }
    r0, _, e1 := Syscall(SYS_READ, uintptr(fd), uintptr(_p0),
uintptr(len(p)))
    n = int(r0)
    if e1 != 0 {
        err = errnoErr(e1)
    }
    return
}
```

Syscall和RawSyscall的源码

```
1  //Syscall
2  TEXT ·Syscall(SB),NOSPLIT,$0-56
3      CALL    runtime·entersyscall(SB)
4      MOVQ    a1+8(FP), DI
5      MOVQ    a2+16(FP), SI
6      MOVQ    a3+24(FP), DX
7      MOVQ    $0, R10
8      MOVQ    $0, R8
9      MOVQ    $0, R9
10     MOVQ    trap+0(FP), AX  // syscall entry
11     SYSCALL
12     CMPQ    AX, $0xfffffffffffff001
13     JLS ok
14     MOVQ    $-1, r1+32(FP)
15     MOVQ    $0, r2+40(FP)
16     NEGQ    AX
17     MOVQ    AX, err+48(FP)
18     CALL    runtime·exitsyscall(SB)
19     RET
20 ok:
21     MOVQ    AX, r1+32(FP)
22     MOVQ    DX, r2+40(FP)
23     MOVQ    $0, err+48(FP)
24     CALL    runtime·exitsyscall(SB)
25     RET
```

2.4.1 entersyscall()

1. 设置_g_.m.locks++，禁止g被强占
2. 设置_g_.stackguard0 = stackPreempt，禁止调用任何会导致栈增长/分裂的函数
3. 保存现场，在 syscall 之后会依据这些数据恢复现场
4. 更新G的状态为_Gsyscall
5. 释放局部调度器P：解绑P与M的关系；
6. 更新P状态为_Psyscall
7. *g*.m.locks–解除禁止强占。

进入系统调用的goroutine会阻塞，导致内核M会阻塞。此时P会被剥离掉， 所以P可以继续去获取其余的空闲M执行其余的goroutine。

2.4.2 阻塞式系统调用长期运行将会导致的流程

```
sysmon() -> retake() -> handoffp()
```

如果P的本地队列不为空，handoffp()会尝试获取一个M来运行P M有2种来源:

- 1）`midle`: idle m's waiting for work
- 2）newm() 创建一个新M

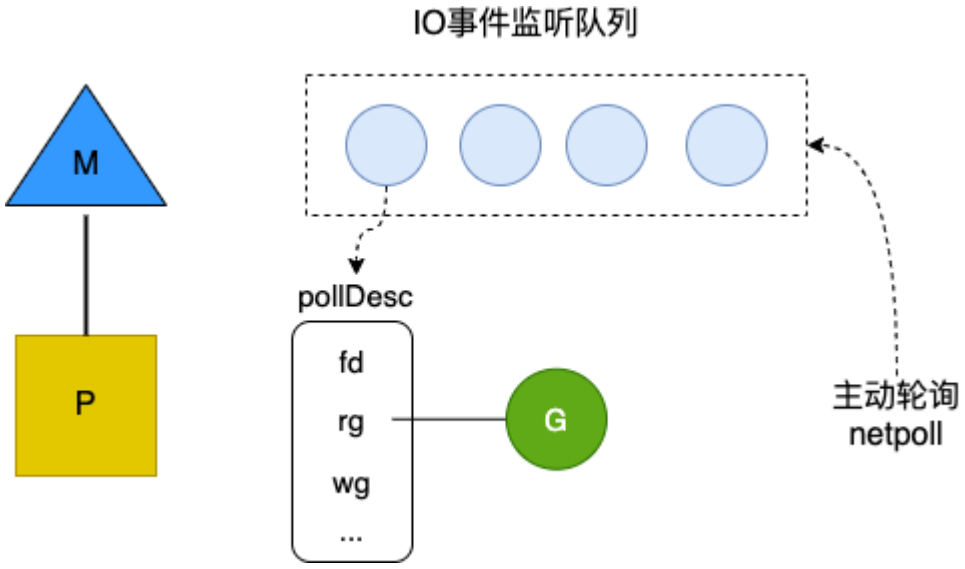
2.4.3 exitsyscall()

1. 设置 *g*.m.locks++ 禁止强占
2. 调用 exitsyscallfast() 快速退出系统调用
   2.1. Try to re-acquire the last P，如果成功就直接接return;
   2.2. Try to get any other idle P from allIdleP list;
   2.3. 没有获取到空闲的P
3. 如果快速获取到了P：
   3.1. 更新G 的状态是_Grunning
   3.2. 与G绑定的M会在退出系统调用之后继续执行
4. 没有获取到空闲的P：
   4.1. 调用mcall()函数切换到g0的栈空间；
   4.2. 调用exitsyscall0函数：
   4.2.1. 更新G 的状态是_Grunning
   4.2.2. 调用dropg()：解除当前g与M的绑定关系；
   4.2.3. 调用globrunqput将G插入global queue的队尾，
   4.2.4. 调用stopm()释放M，将M加入全局的idel M列表，这个调用会阻塞，知道获取到可用的P。
   4.2.5. 如果4.2.4中阻塞结束，M获取到了可用的P，会调用schedule()函数，执行一次新的调度。

2.5 系统调用

以netpoll为例，linux操作系统下，netpoll基于epoll实现的

```
#include <sys/epoll.h>
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int
timeout);
```



2.5.1 让出流程

```
pollDesc.waitRead() -> runtime.poll_runtime_pollWait() -> runtime.gopark()
-> mcall(fn func(*g)) -> park_m(gp *g)
```

gopark()主要流程

- 1）g切换到g0
- 2）修改G的状态 _Grunning -> _Gwaiting
- 3）dropg() 解除G和M的绑定关系
- 4）schedule() 触发一轮调度

2.5.2 放回 主动触发netpoll()

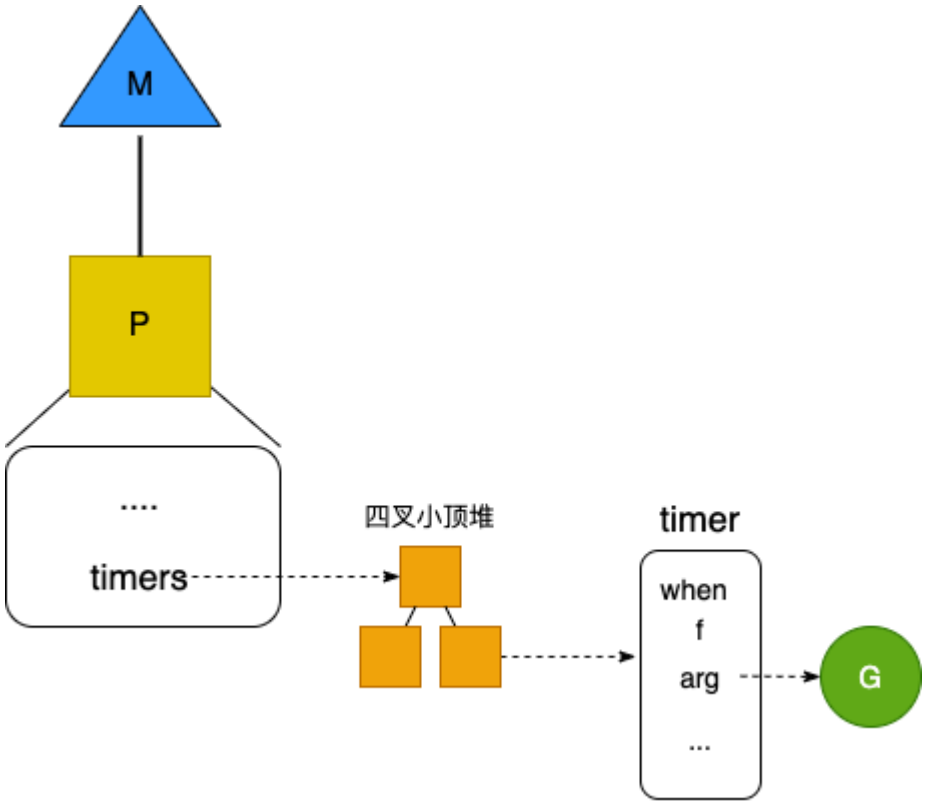
```
findrunnable()       -> netpoll() ->  injectglist() ->
globrunqputbatch()/runqputbatch()
pollWork()
startTheWorldWithSema
sysmon()
```

注意: netpoll函数是非阻塞的

2.6 定时器

场景

```
time.Sleep(1 * time.Second)
```



2.6.1 让出

```
time.Sleep() -> runtime.timeSleep() -> mcall() -> park_m()
-> resetForSleep() -> resettimer() -> doaddtimer()
```

2.6.2 唤醒

```
checkTimers() -> runtimer() -> runOneTimer() -> goready() -> systemstack()
-> ready()
```

goready()主要流程

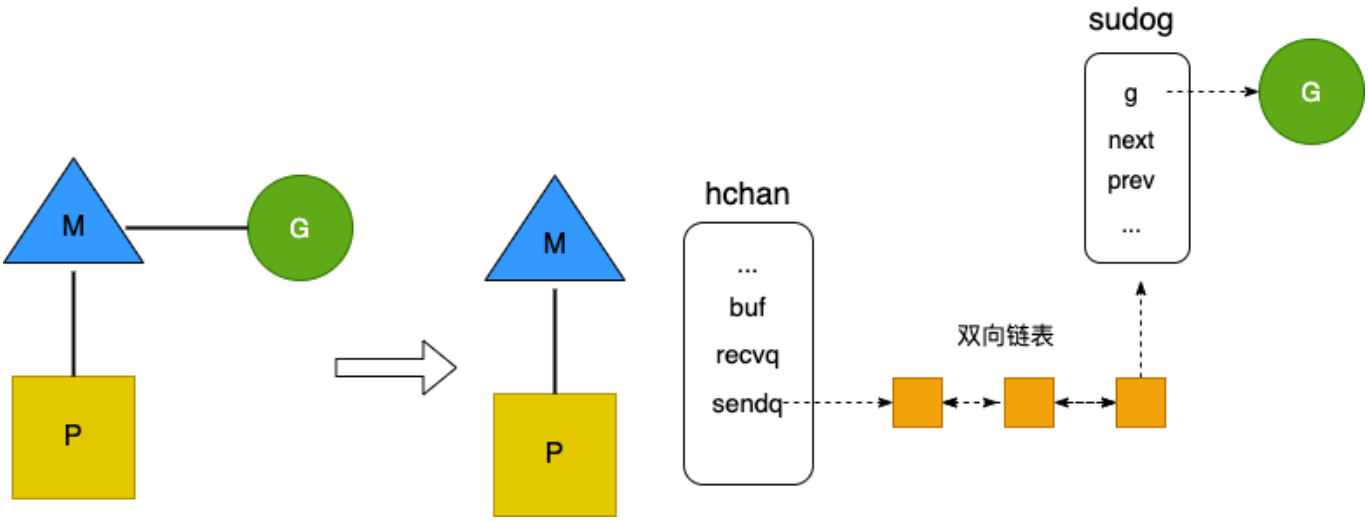
- 1）切换到g0的栈空间(不严谨)
- 2）修改Goroutine的状态 _Gwaiting -> _Grunnable
- 3）runqput() 把Goroutine放入P的本地队列的头部
- 4）如果M的数量不足，尝试创建一些P

注意: mcall()和systemstack()是对应的

2.7 Channel

场景

```
ch := make(chan int)
ch <- 15
```



2.7.1 写入channel并阻塞

```
chansend1() -> chansend() ->  hchan.sendq.enqueue() -> gopark()
```

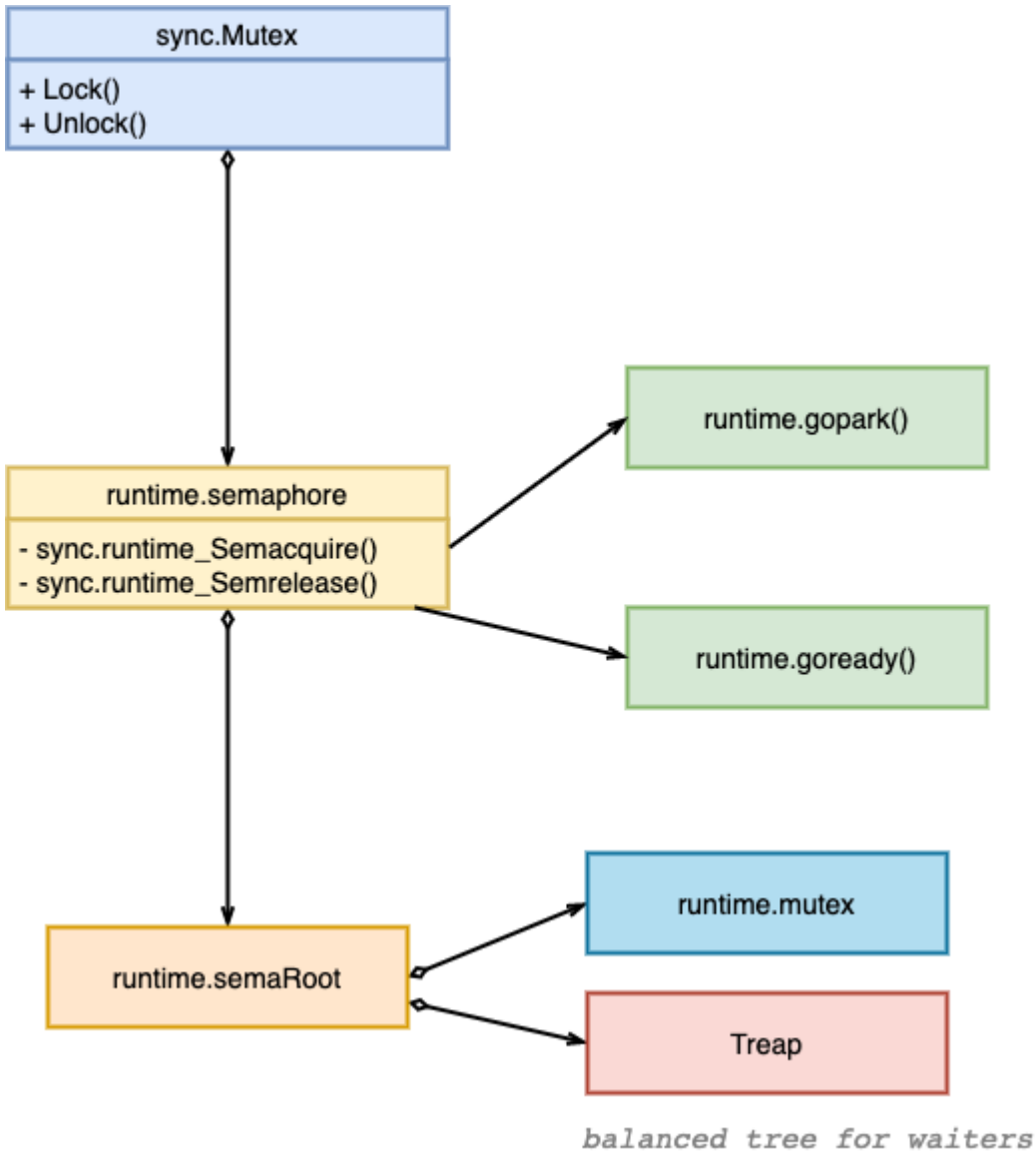
2.7.2 就绪

```
chanrecv1() -> chanrecv() ->  hchan.sendq.dequeue() -> recv() -> goready()
```

2.8 同步原语

互斥锁与2.6、2.7的情况非常类似，只是G会存储在Treap中

- Treap是一种平衡二叉树
- semaRoot 是全局唯一的



3.参考资料

1.g0-特殊的goroutine 2.golang syscall原理 3.Linux中的EAGAIN含义 4.Golang-gopark函数和goready函数原理分析 5.幼麟实验室-协程让出、抢占、监控和调度 6.Golang 调度器 GMP 原理与调度全分析 7.time.Sleep(1)后发生了什么 8.mcall systemstack等汇编函数