time.Timer 源码分析

欧长坤
changkun.de

Go 夜读 SIG 小组 | 第 74 期
January 02, 2020

主要内容

- 调度器与调度循环
- Go 1.10 之前的 Timer 实现
- Go 1.10 的优化
- Go 1.14 的优化
- Timer 的启动与触发过程
- 读相关源码、实验
- 总结

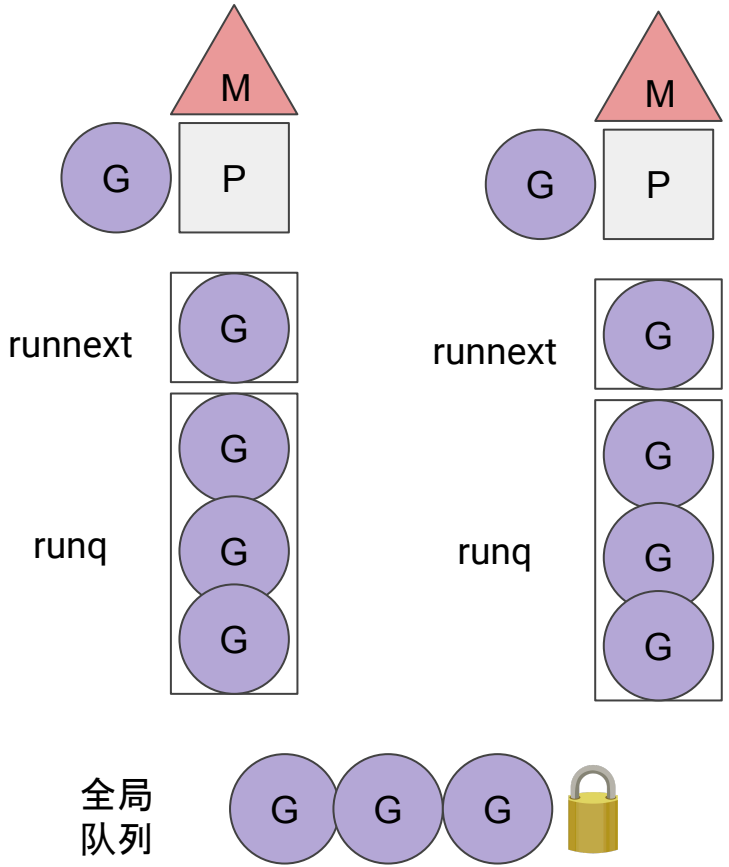
Go Scheduler in 5 Minutes

G: Goroutine
被调度的实体，即用户代码，在本地队列中不断的切换执行

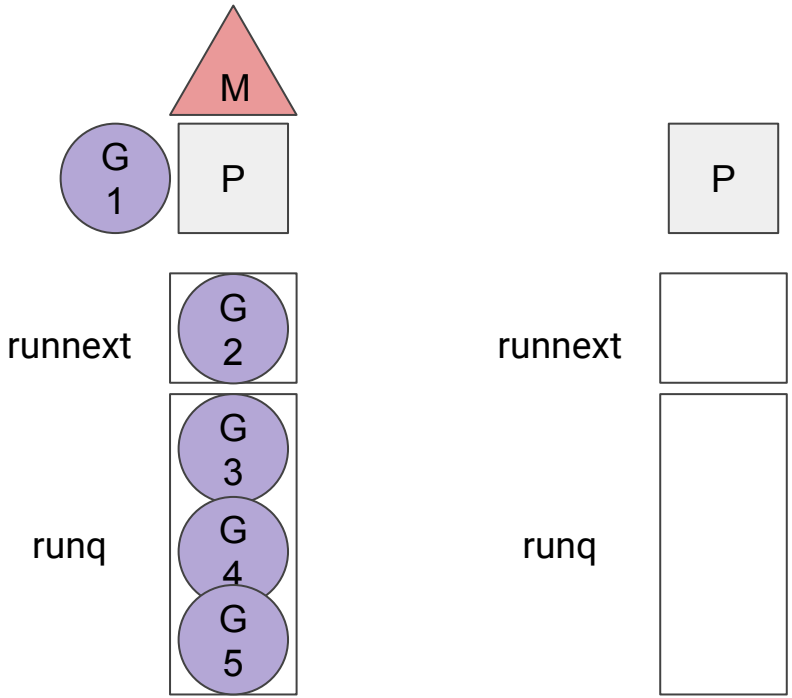
M: Machine
传统线程实体，即系统线程，负责代码的执行

P: Processor
处理器抽象，目的是实现串联 G 的本地队列，当 M 持有 P 时，访问的 G 不会出现数据竞争

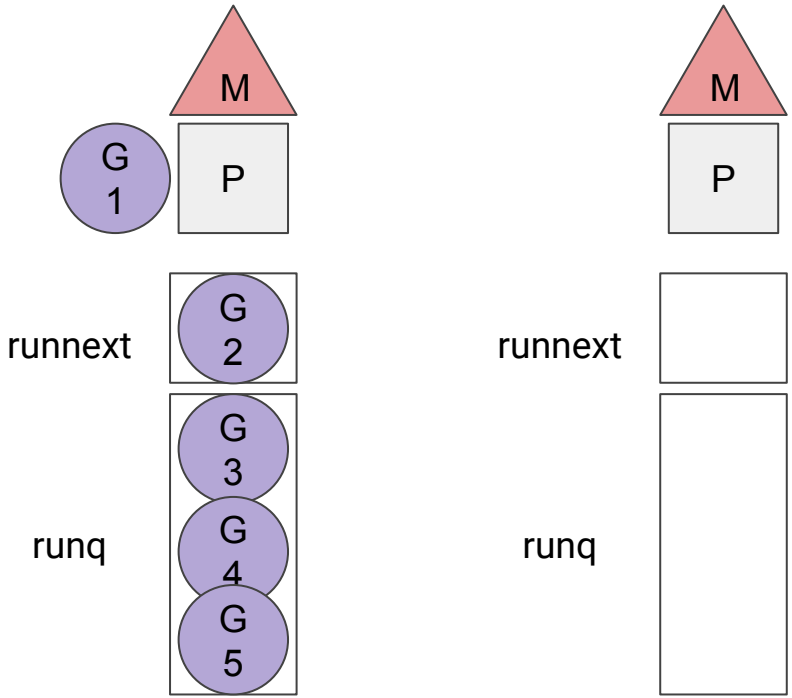
WSS：工作窃取调度
1. M 切换出 G 后，将其插入本地队列尾部
2. 本地队列为空则去全局队列偷
3. 然后才去其他 P 本地队列偷
4. 没有工作则休眠



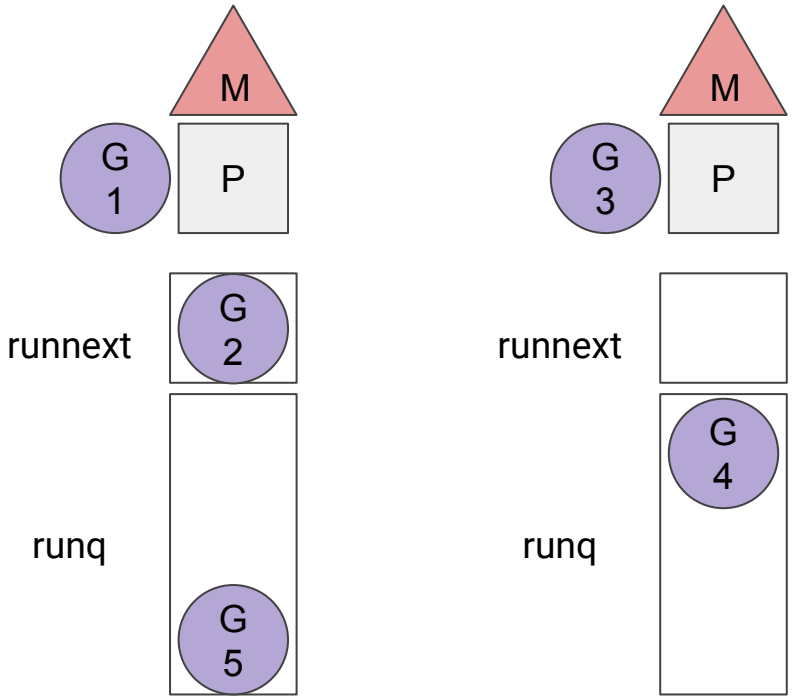
调度循环：go



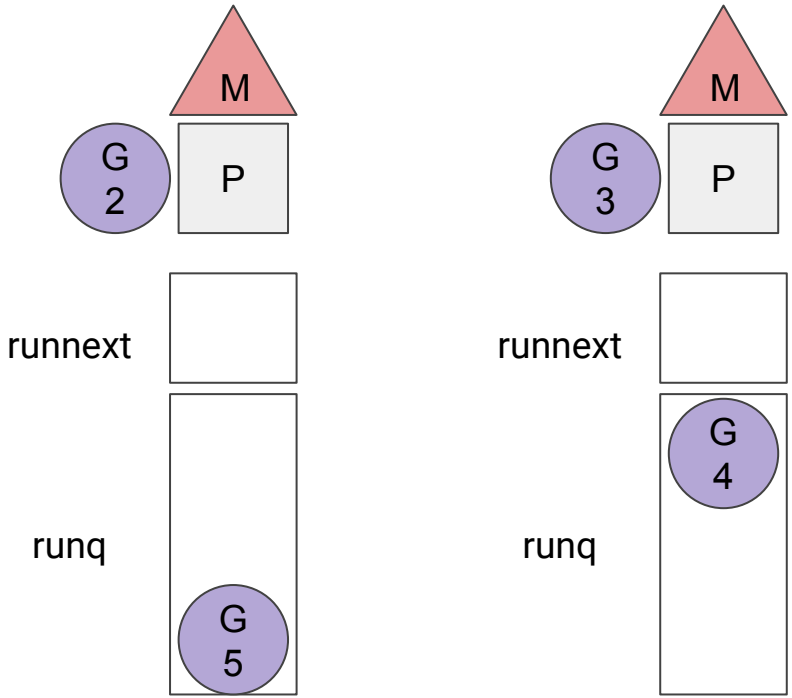
调度循环：go -> wakep



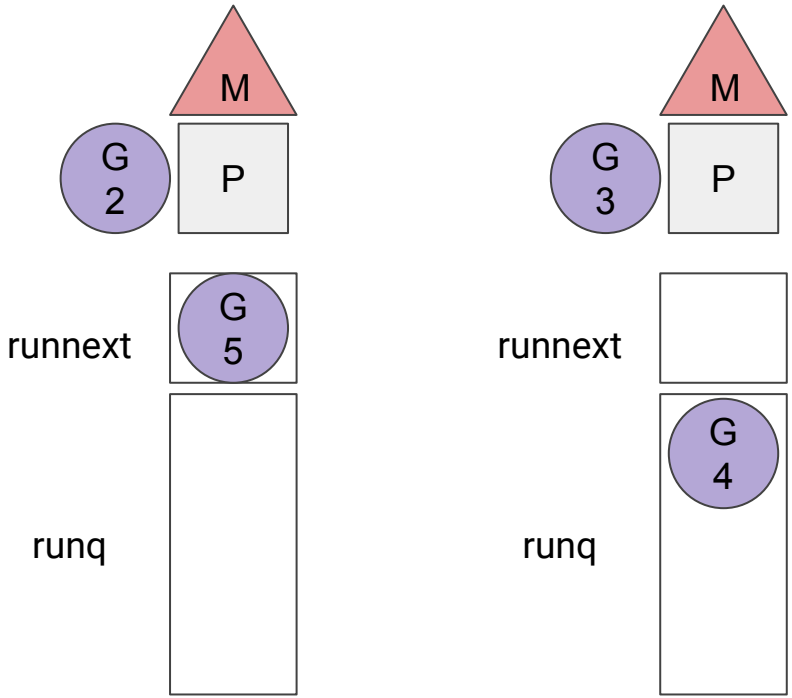
调度循环：go -> wakep -> schedule (findrunnable)



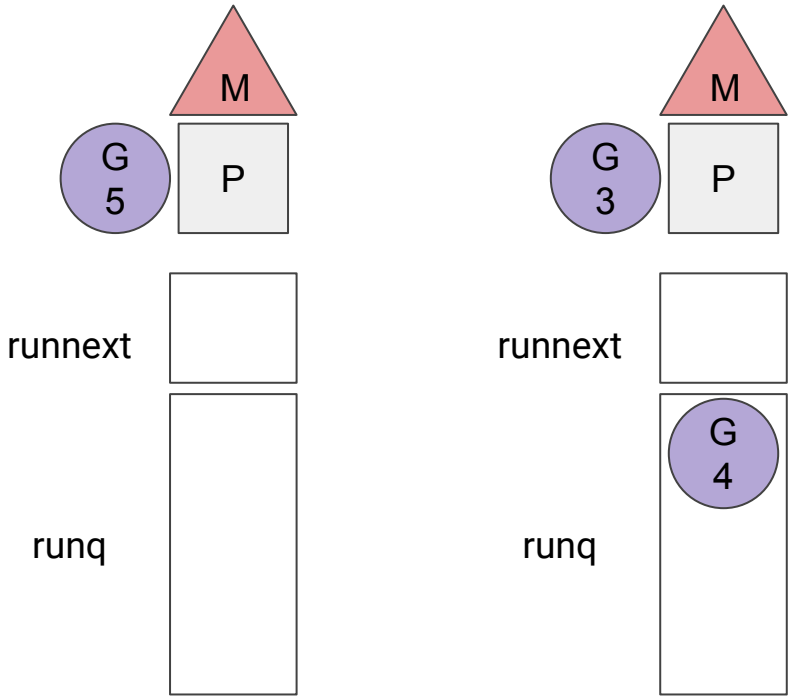
调度循环：go -> wakep -> schedule (findrunnable)



调度循环：go -> wakep -> schedule (findrunnable)



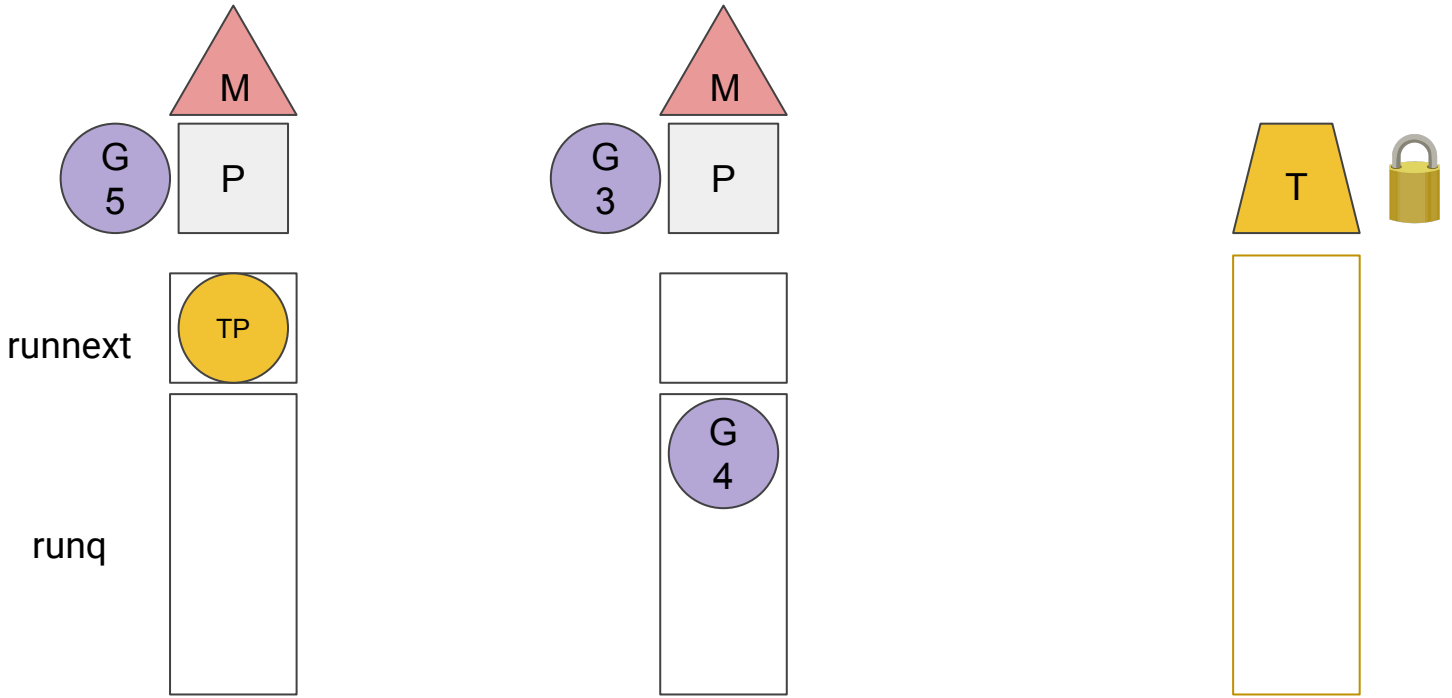
调度循环：go -> wakep -> schedule (findrunnable)



G5: <-timer.C

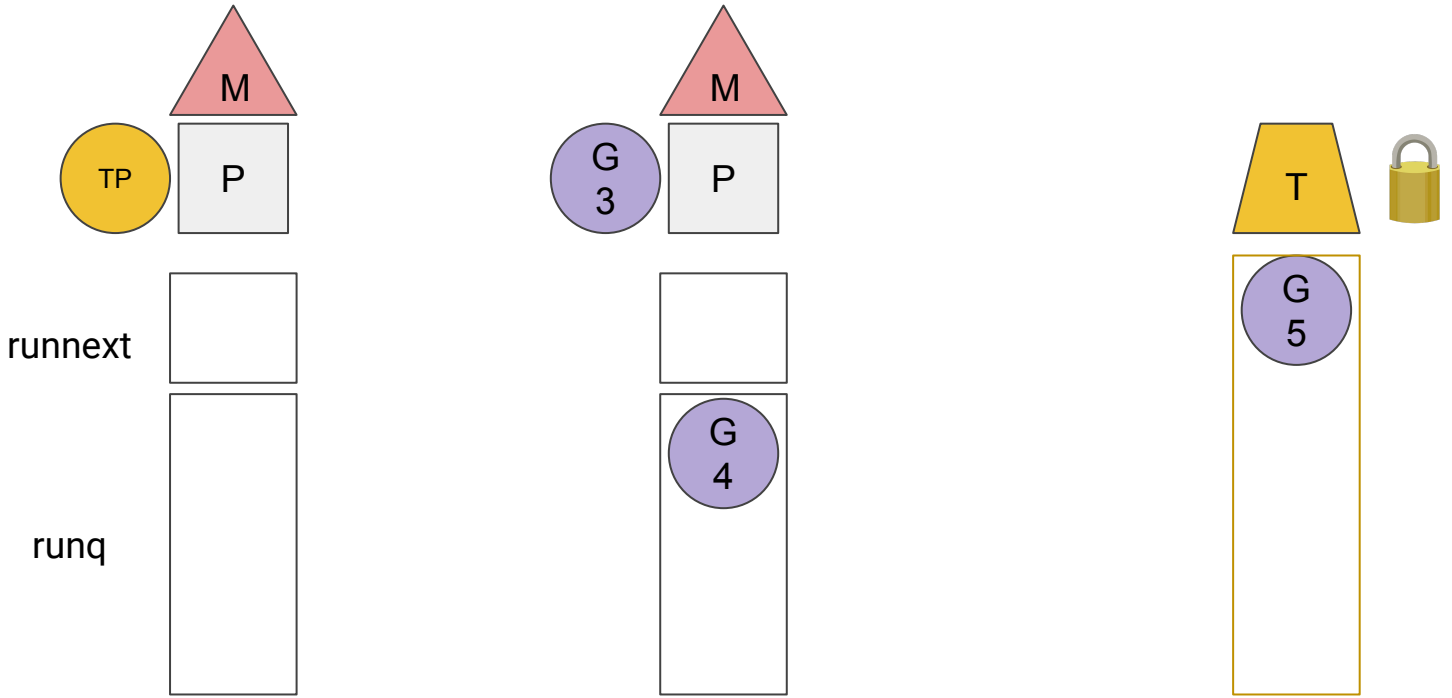
time.Timer: 开始休眠等待

< go 1.10



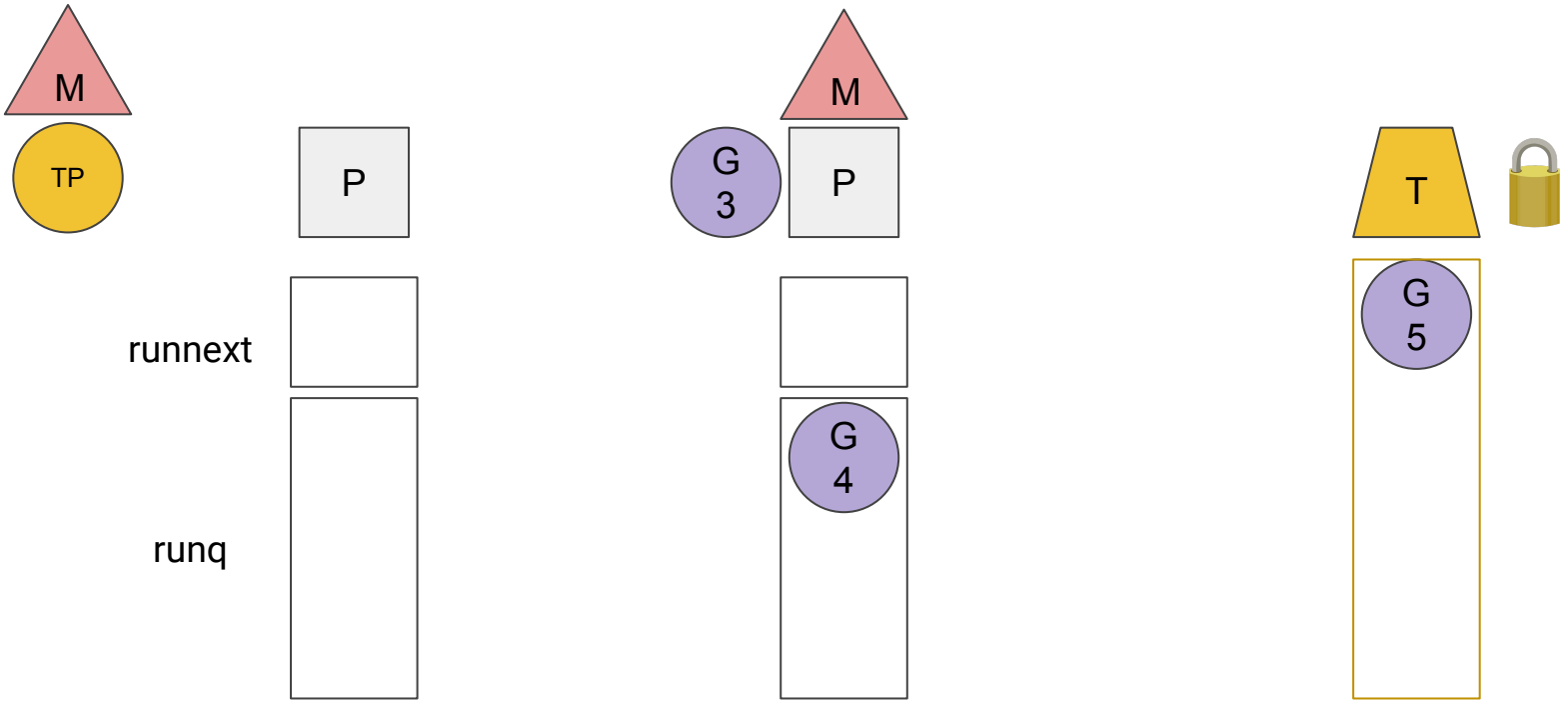
time.Timer: 开始休眠等待

< go 1.10



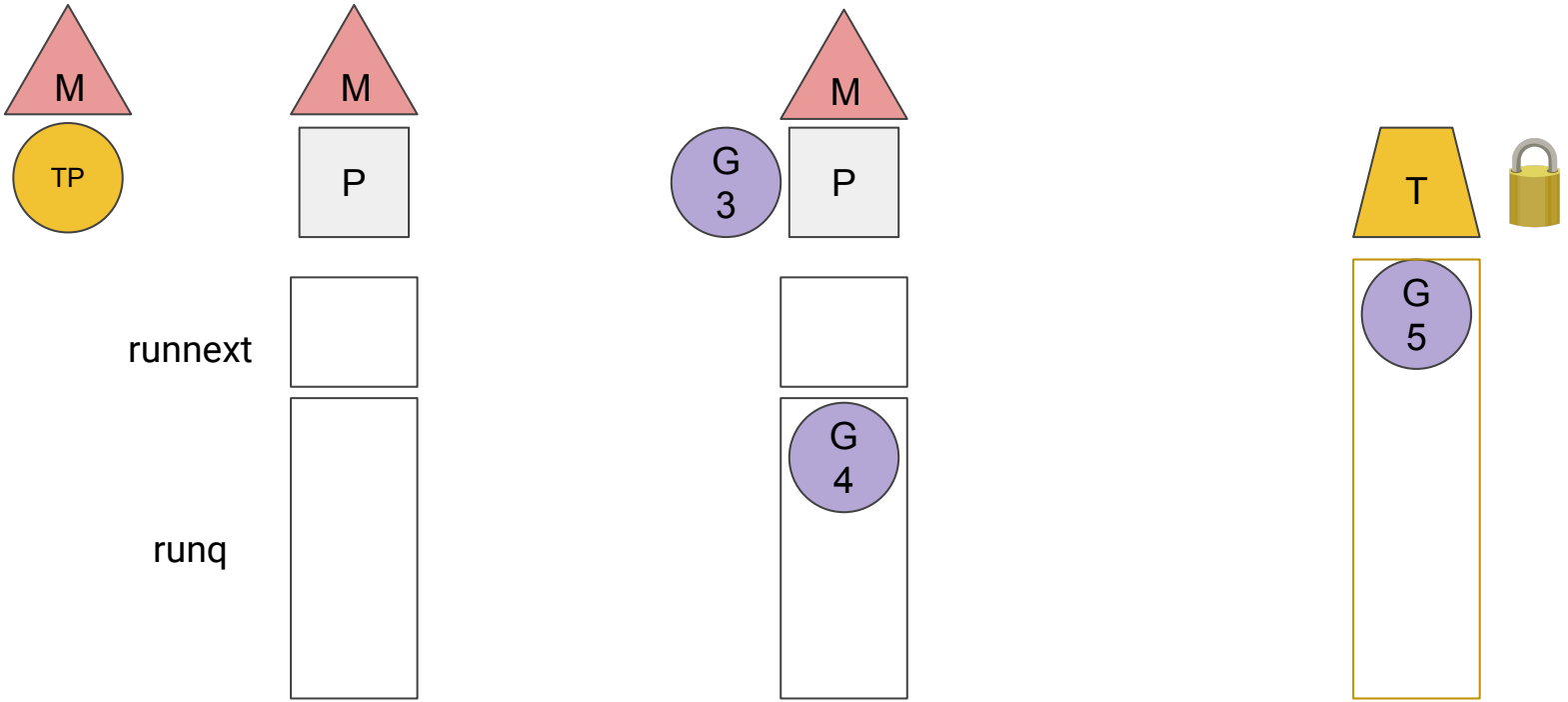
time.Timer: 开始休眠等待 -> handoffp

< go 1.10



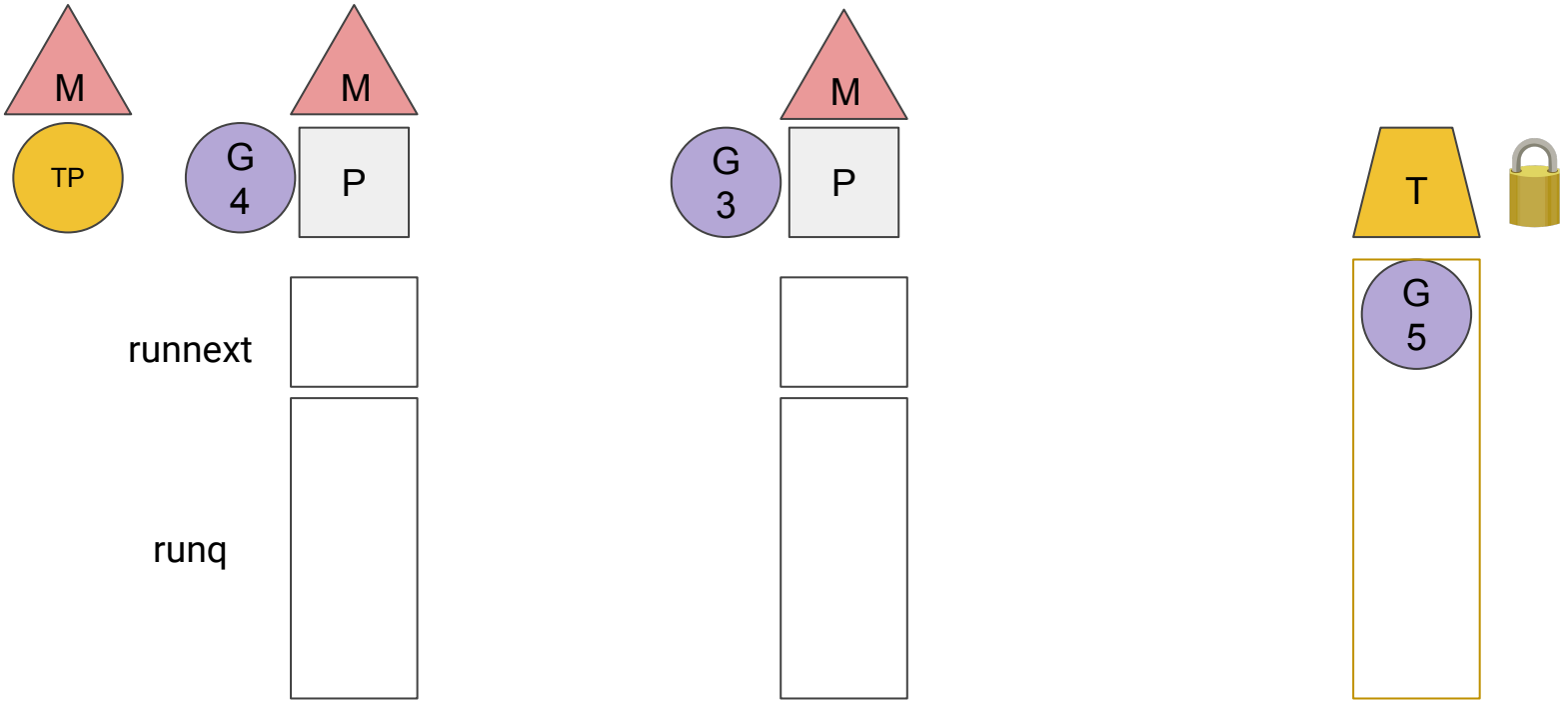
time.Timer: 开始休眠等待 -> handoffp -> wakep

< go 1.10



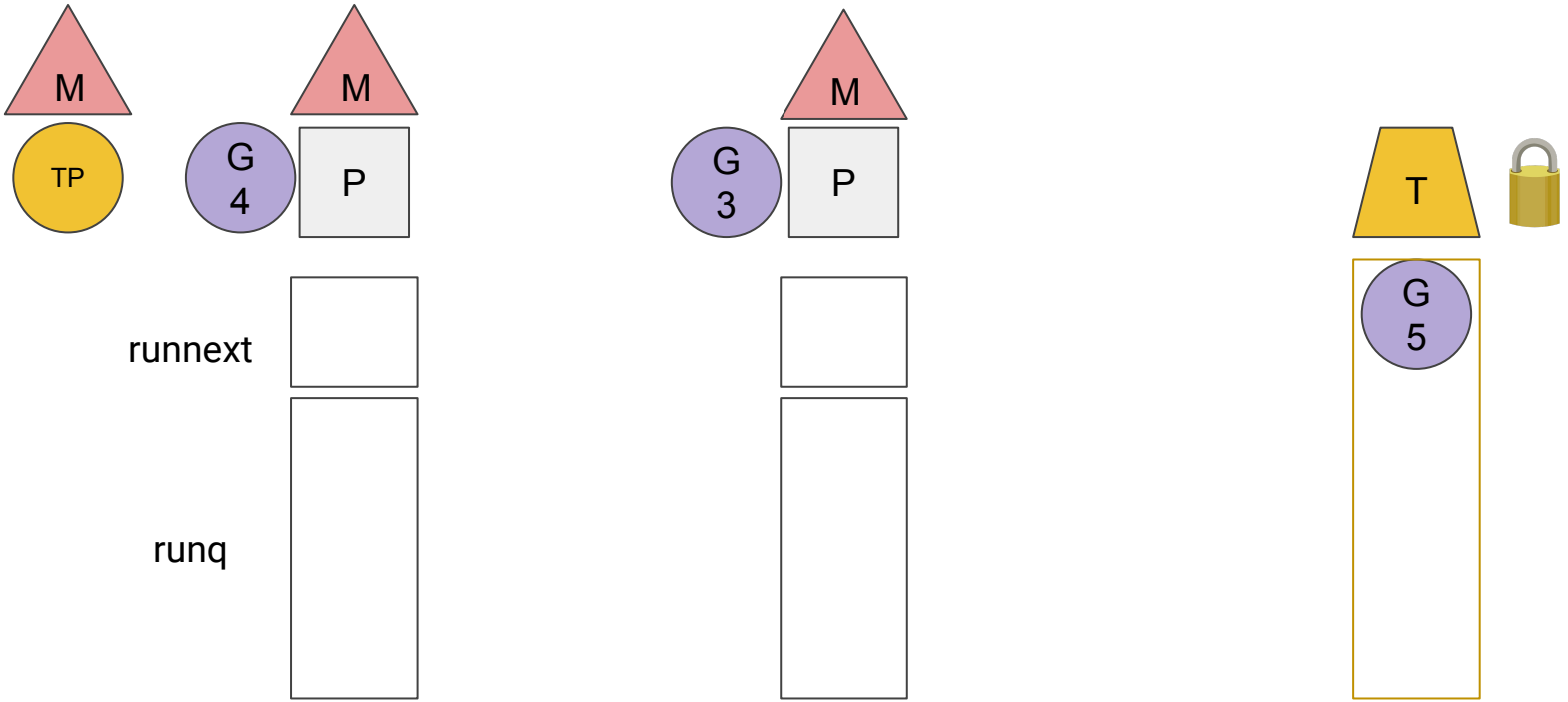
time.Timer: 开始休眠等待 -> handoffp -> wakep -> schedule (findrunnable)

< go 1.10



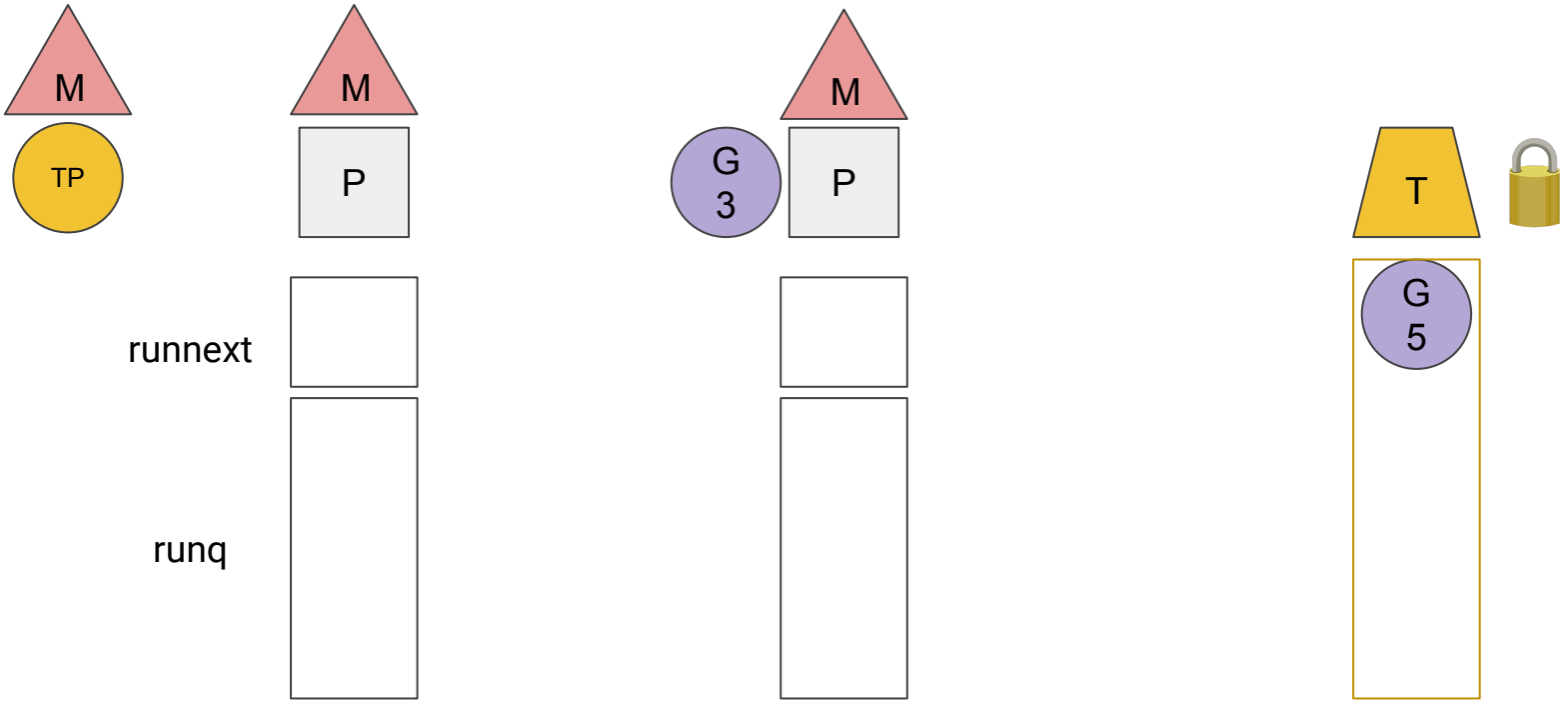
time.Timer: 开始休眠等待 -> handoffp -> wakep -> schedule (findrunnable)

< go 1.10



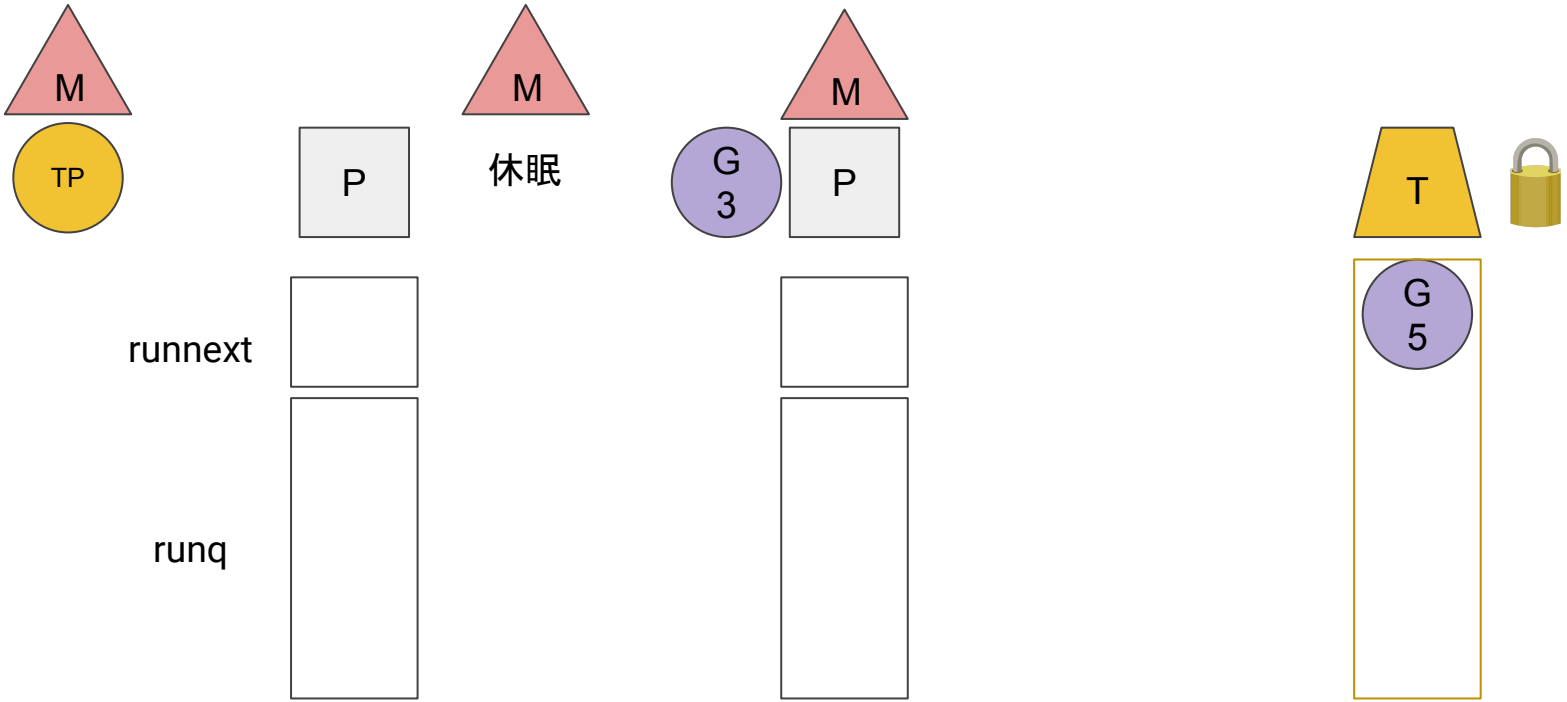
time.Timer: 开始休眠等待 -> handoffp -> wakep -> schedule (findrunnable)

< go 1.10



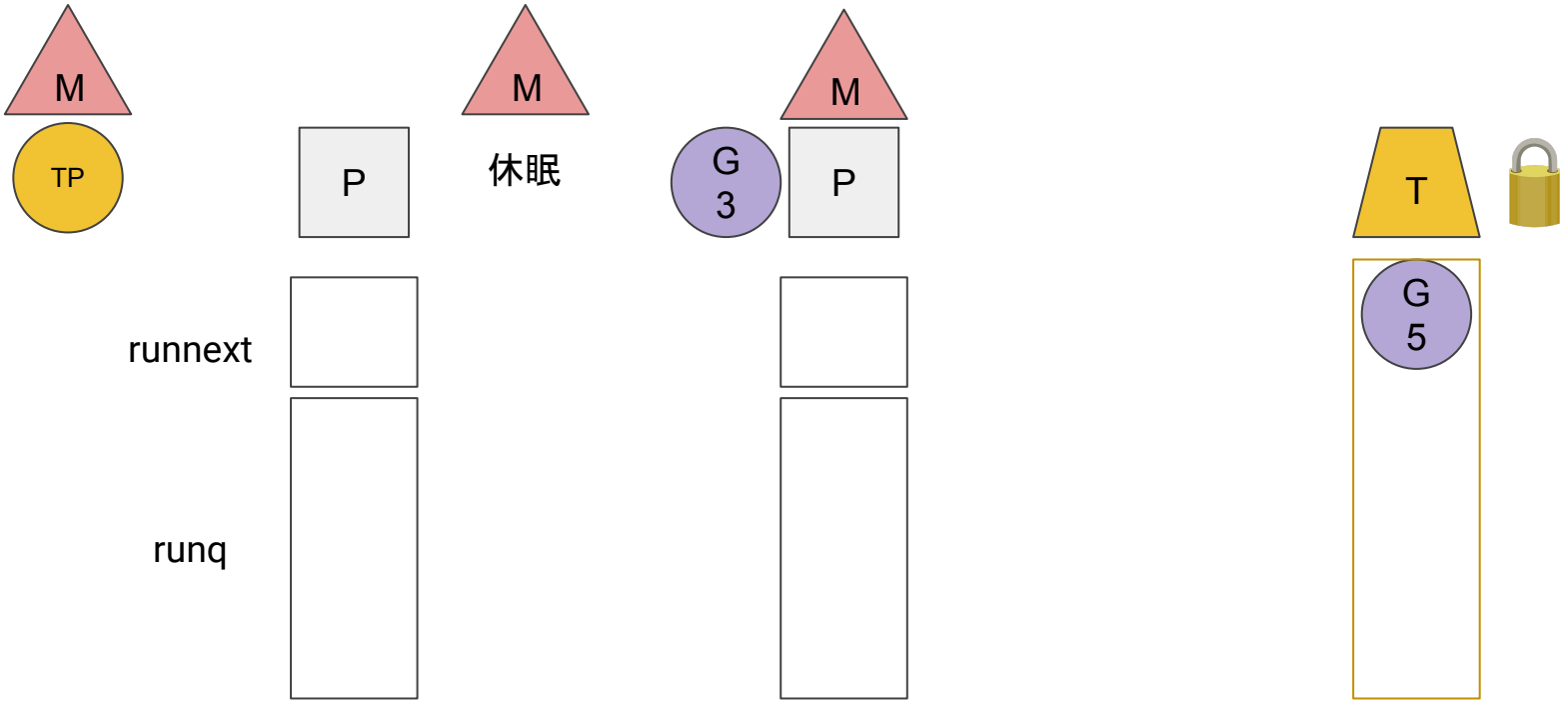
time.Timer: 开始休眠等待 -> handoffp -> wakep -> schedule (findrunnable) -> dropm

< go 1.10



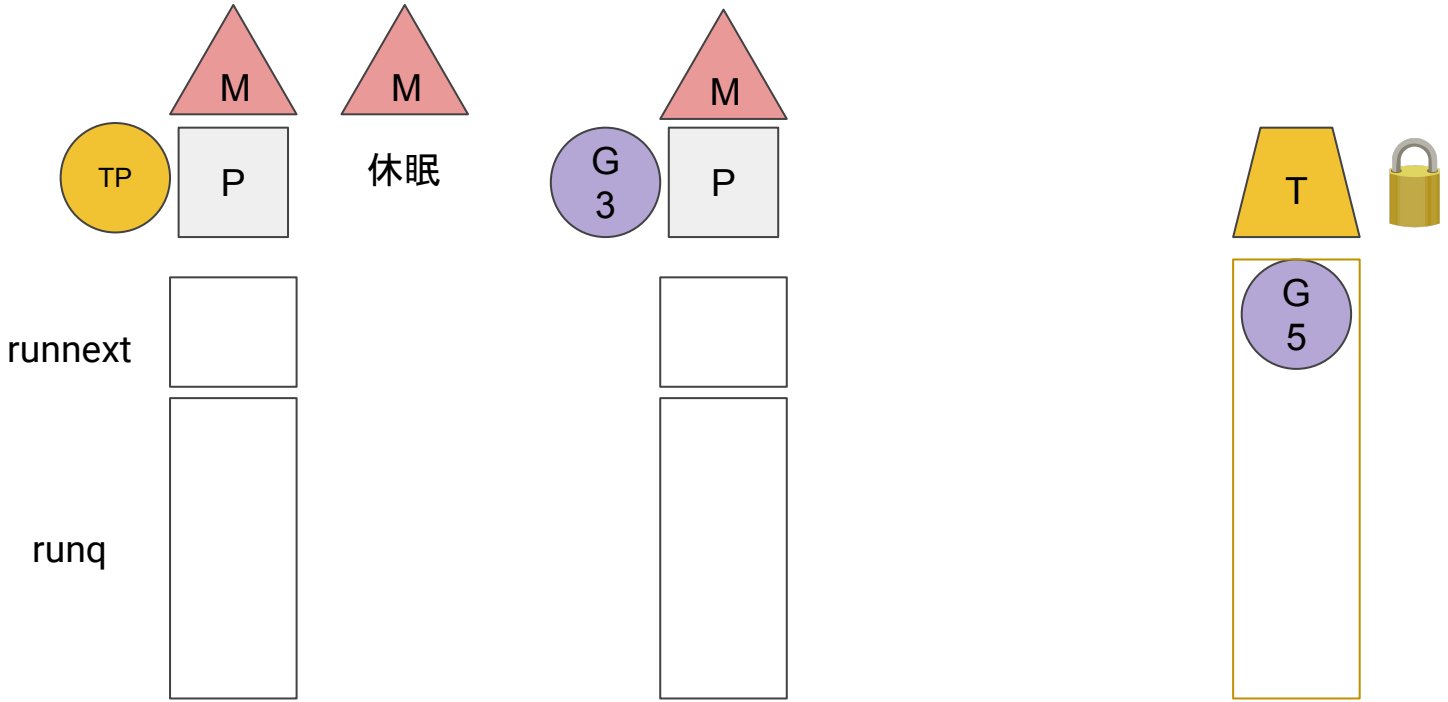
time.Timer: 开始唤醒

< go 1.10



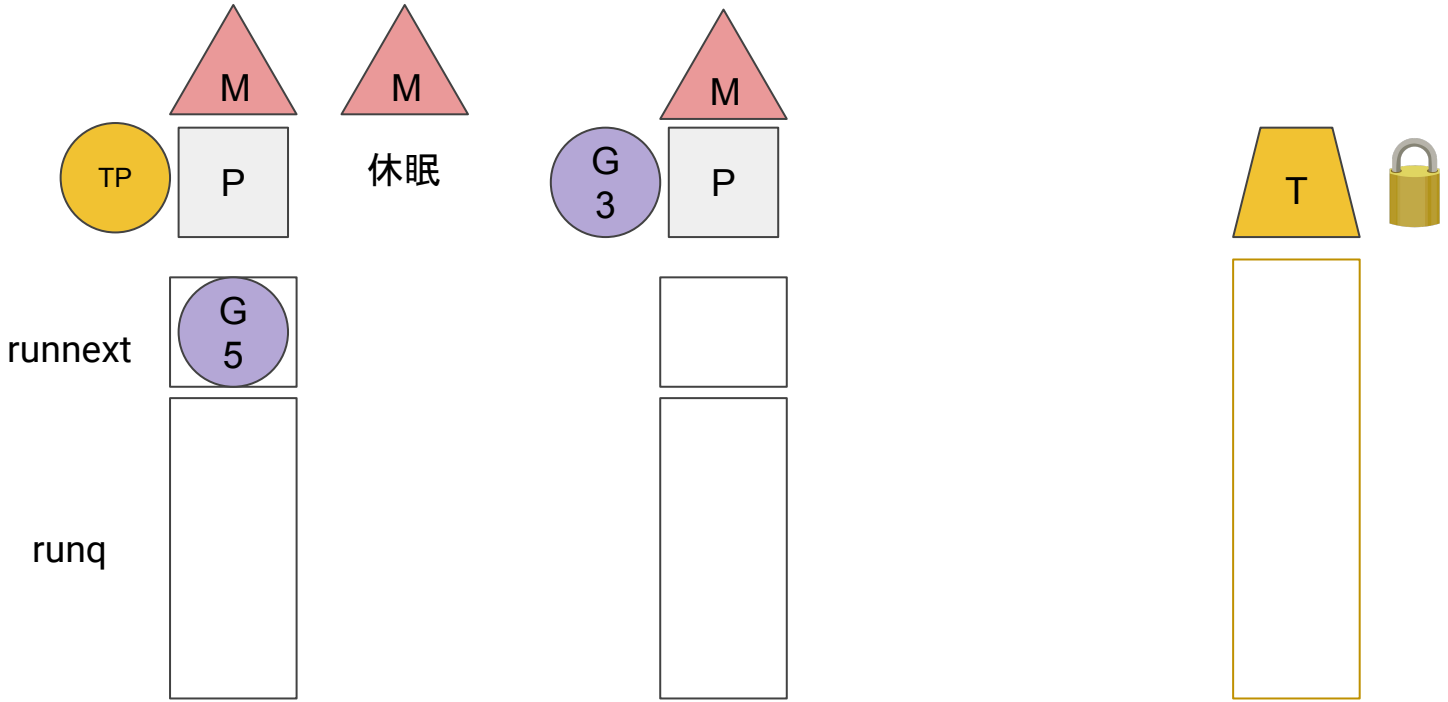
time.Timer: 开始唤醒 -> acquirep

< go 1.10



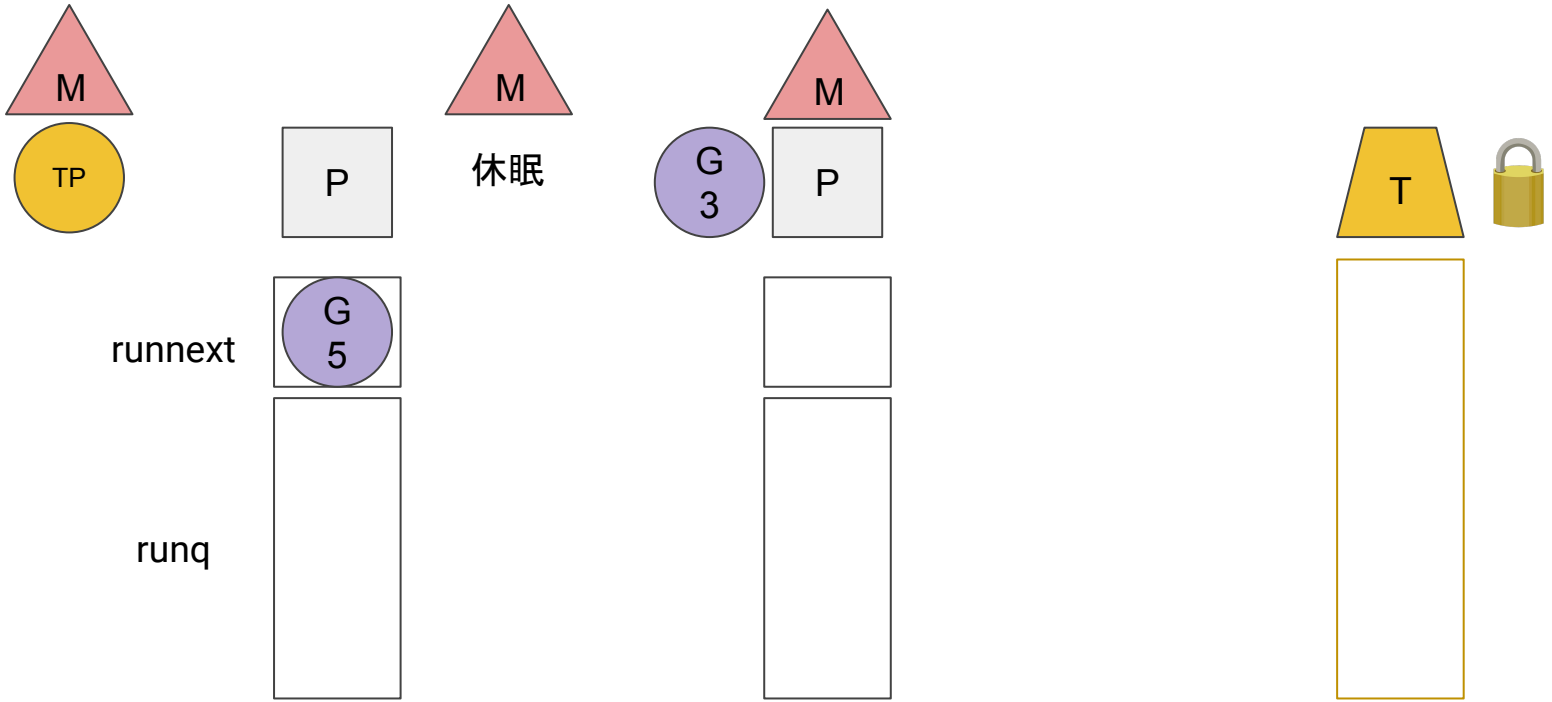
time.Timer: 开始唤醒 -> acquirep

< go 1.10



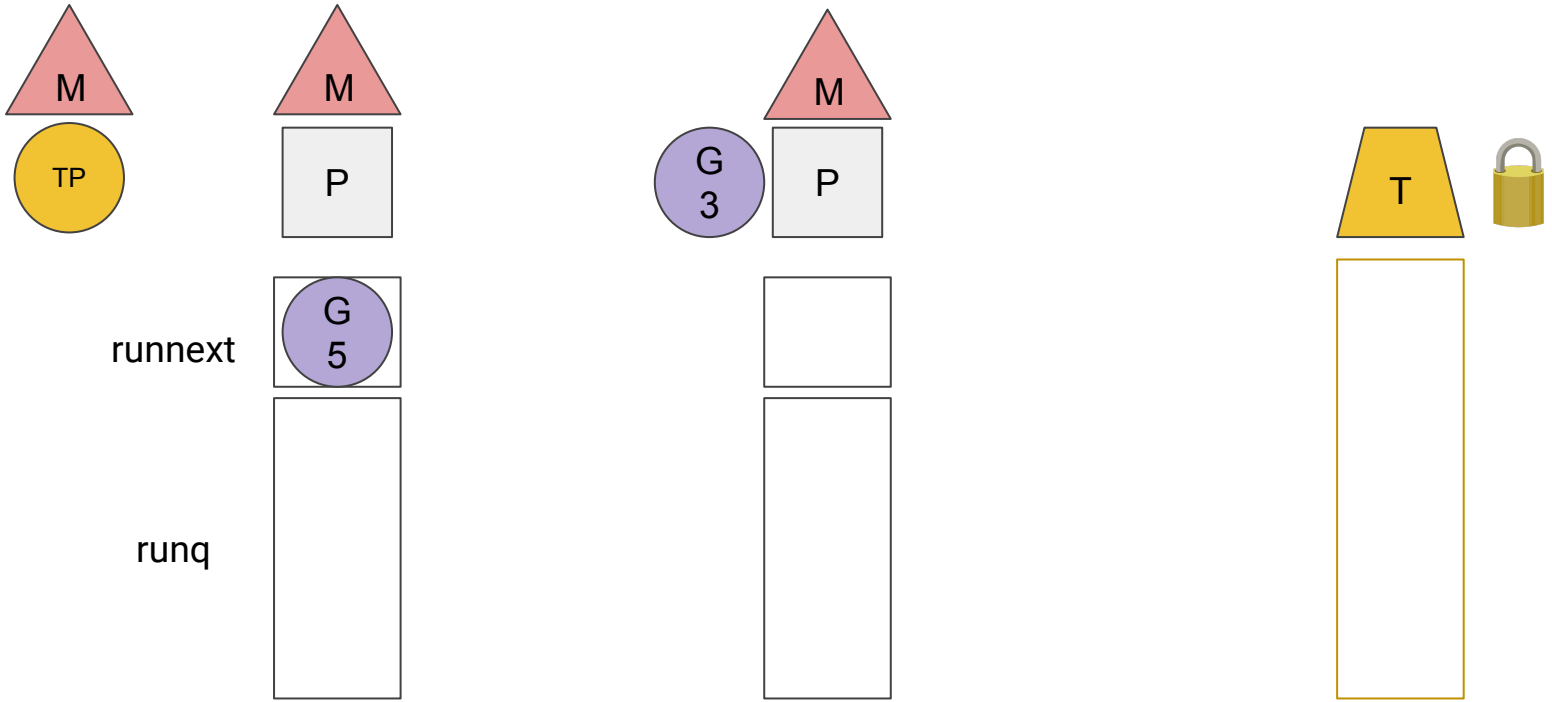
time.Timer: 开始唤醒 -> acquirep -> handoffp

< go 1.10



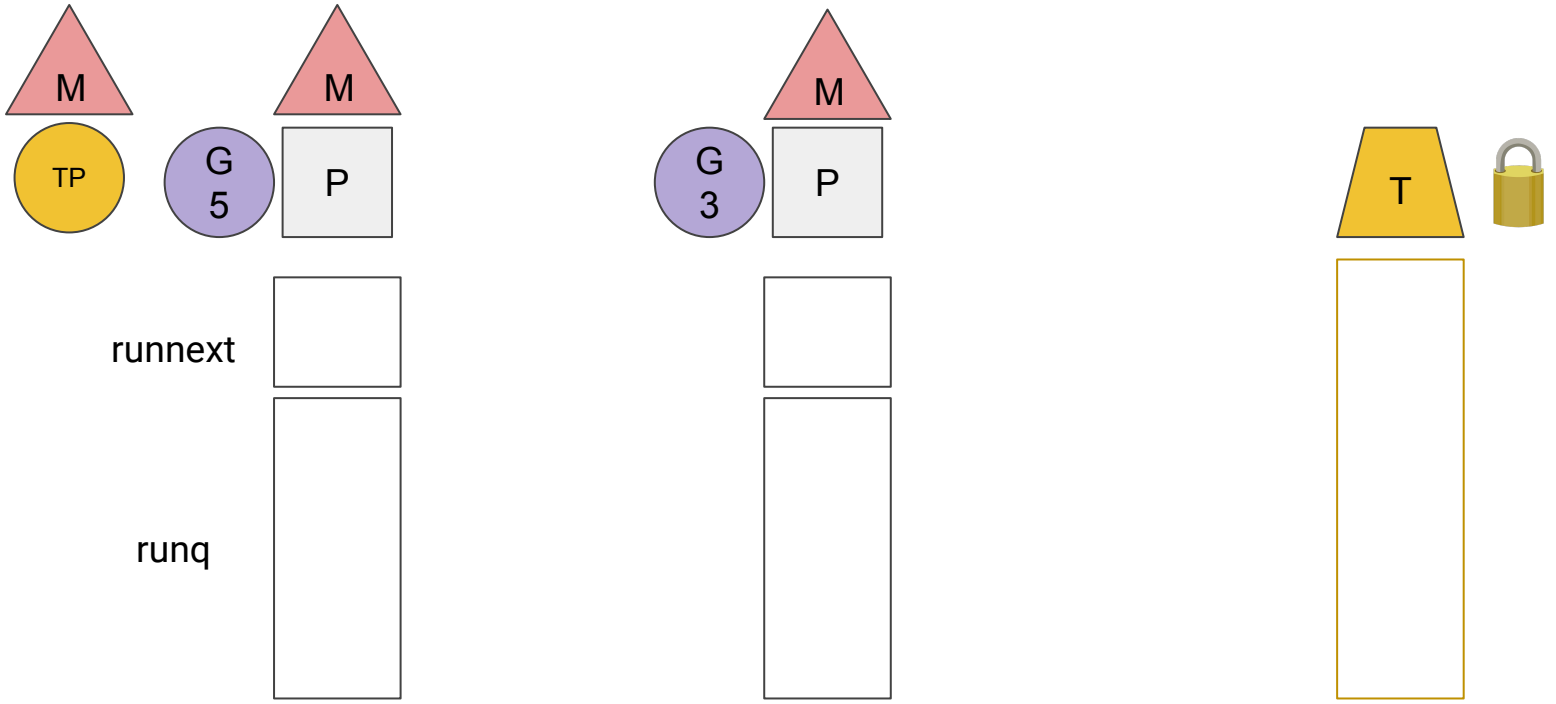
time.Timer: 开始唤醒 -> acquirep -> handoffp -> wakep

< go 1.10



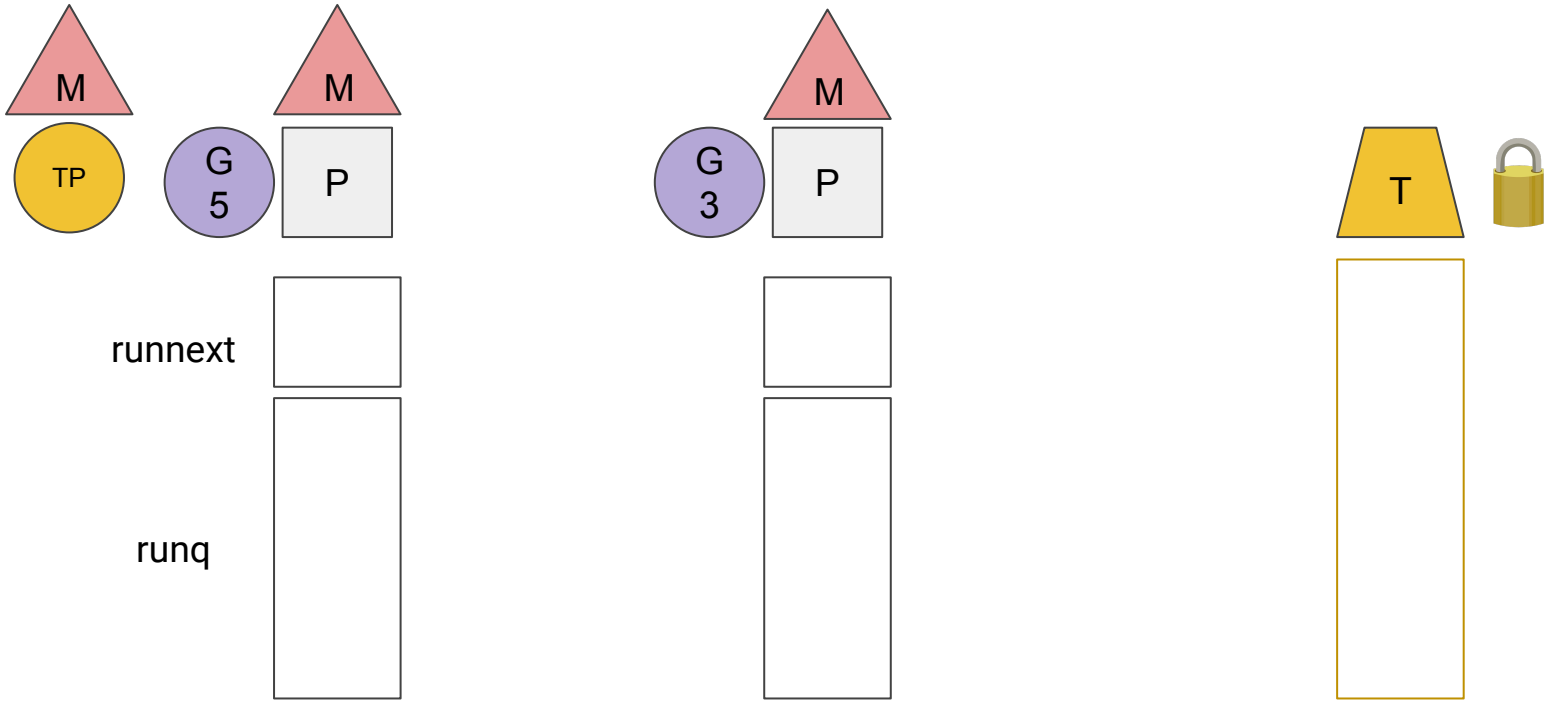
time.Timer: 开始唤醒 -> acquirep -> handoffp -> wakep -> schedule (findrunnable)

< go 1.10



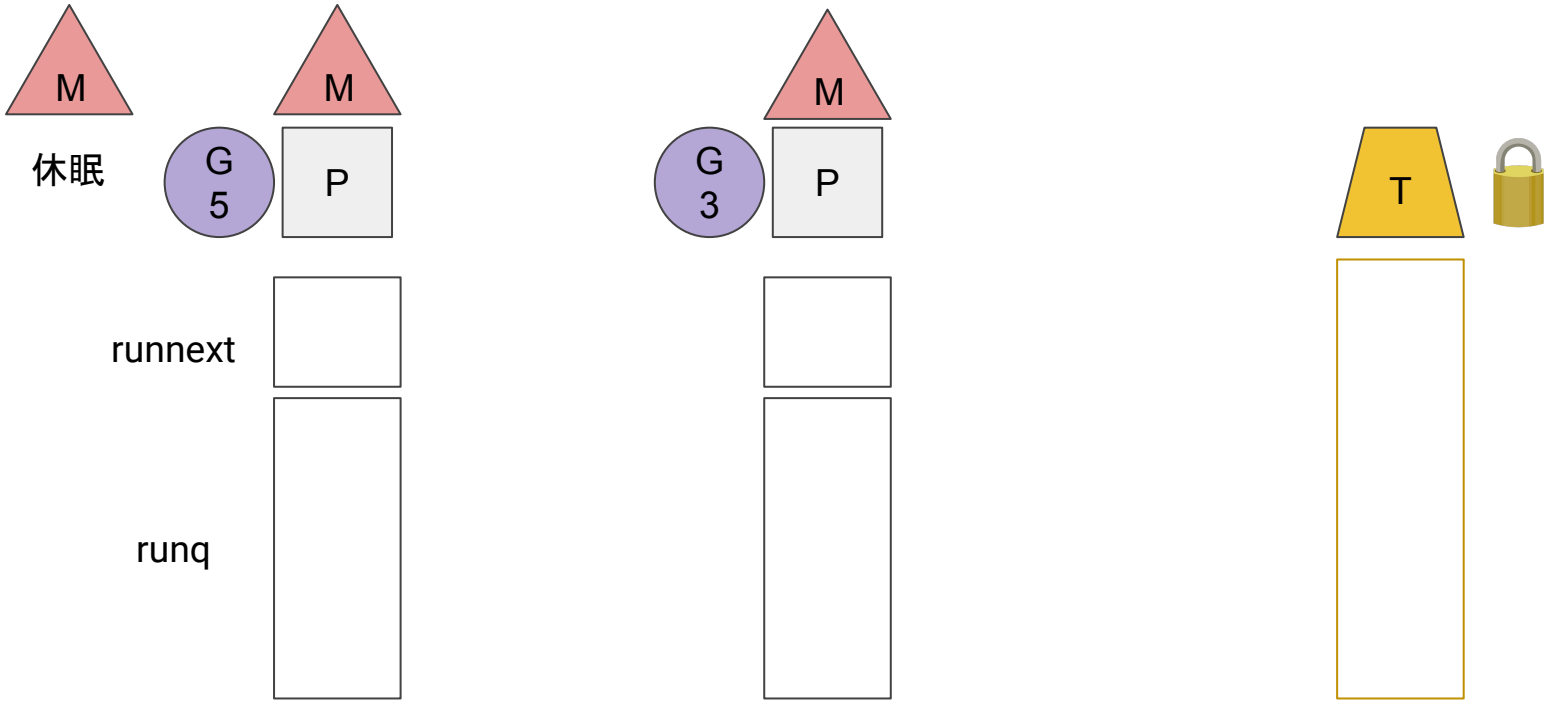
time.Timer: 开始唤醒 -> acquirep -> handoffp -> wakep -> schedule (findrunnable)

< go 1.10



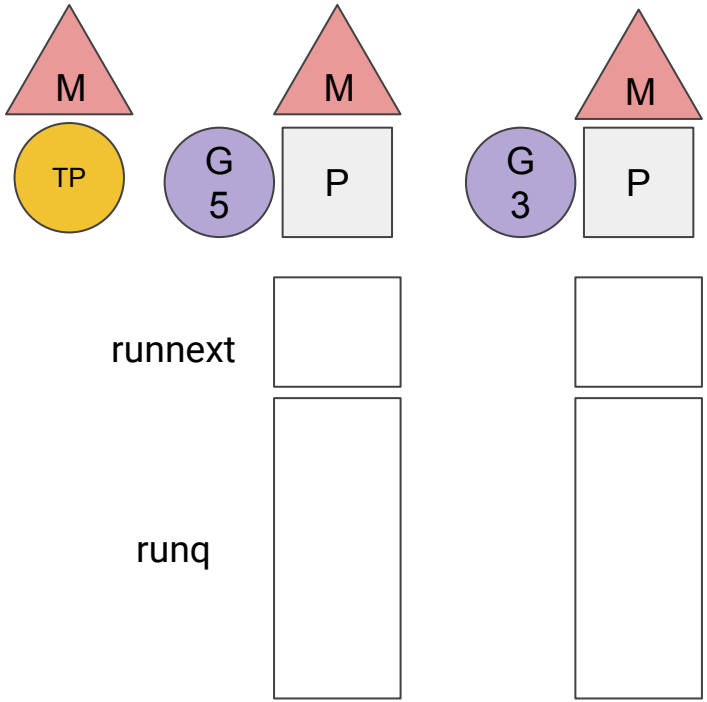
time.Timer: 开始唤醒 -> acquirep -> handoffp -> wakep -> schedule (findrunnable)

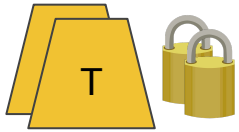
< go 1.10



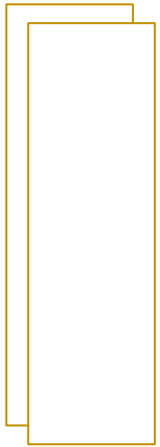
Go 1.10 优化

Go 1.10 的优化: per-P timer 堆





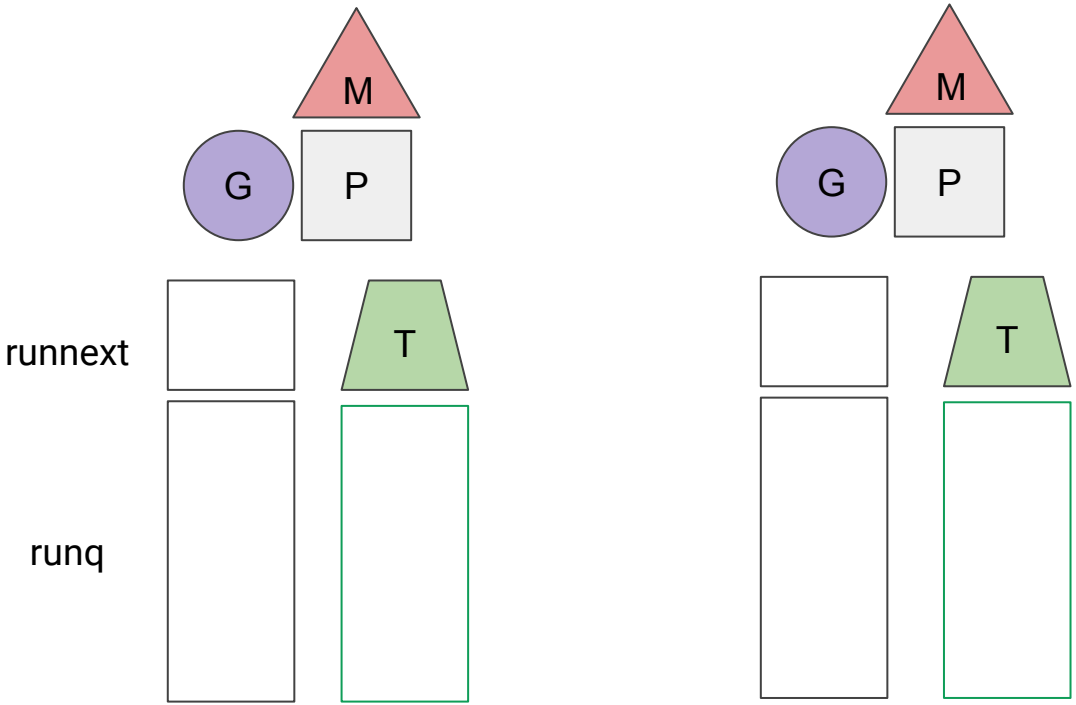
64 个

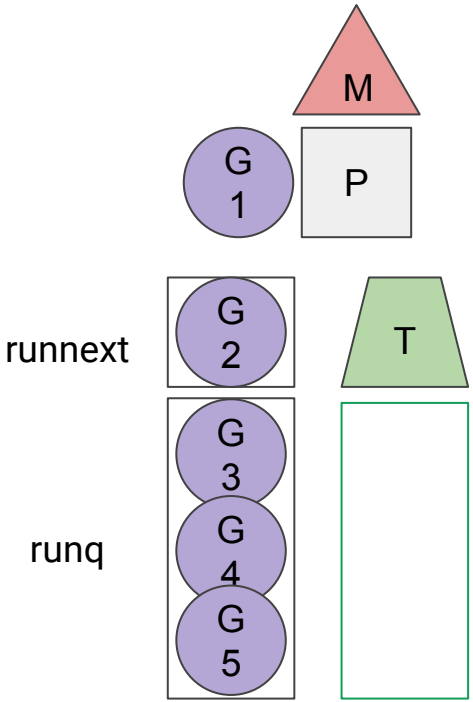


```
func (t *timer) assignBucket()
*timersBucket {
	id := uint8(getg().m.p.ptr().id)
% timersLen
	t.tb = &timers[id].timersBucket
	return t.tb
}
```

Go 1.14?

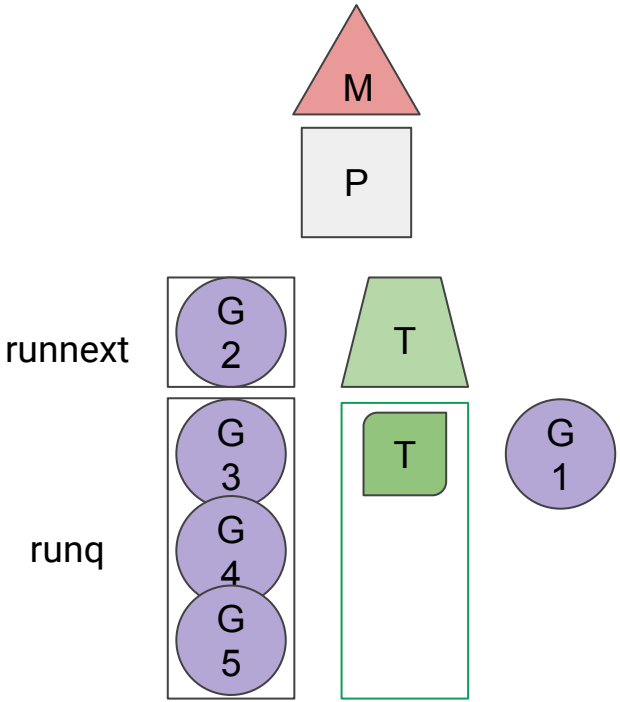
Go 1.14 的优化: 消除多余的 timerproc 和 MP 上下文切换




M
G
1
P
runnext
G
2
T
runq
G
3
G
4
G
5

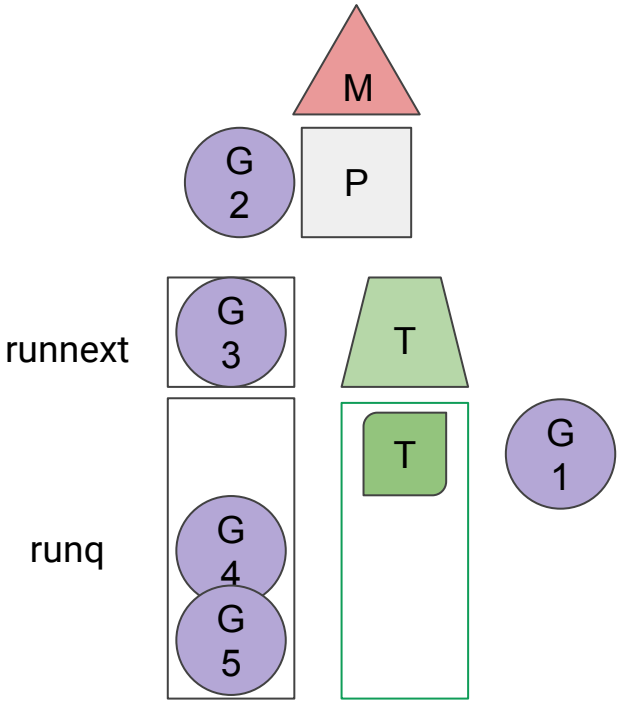
time.Timer: startTimer (addtimer) -> timer.C (sleep)

go 1.14



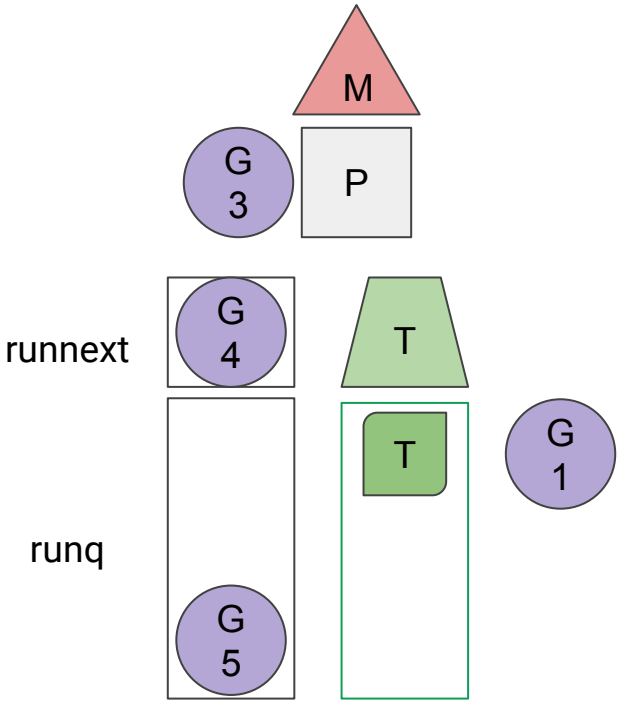
time.Timer: startTimer (addtimer) -> timer.C (sleep) -> schedule

go 1.14



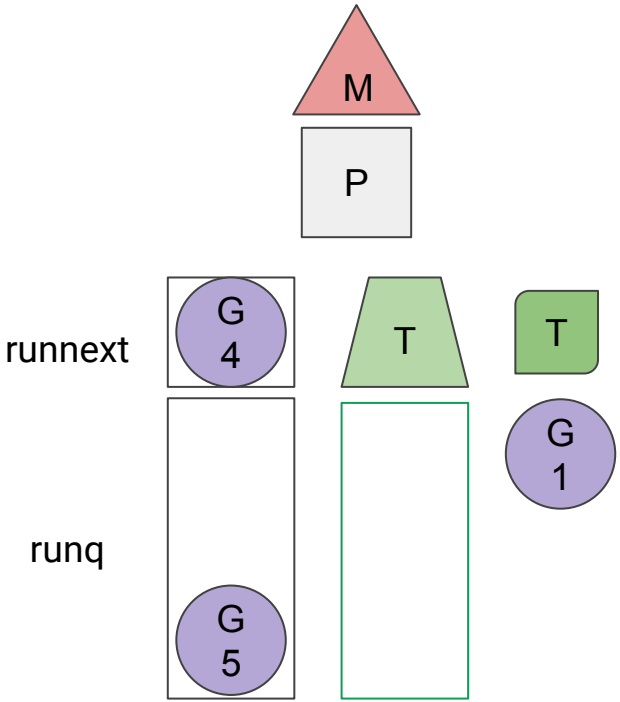
time.Timer: startTimer (addtimer) -> timer.C (sleep) -> schedule*2

go 1.14



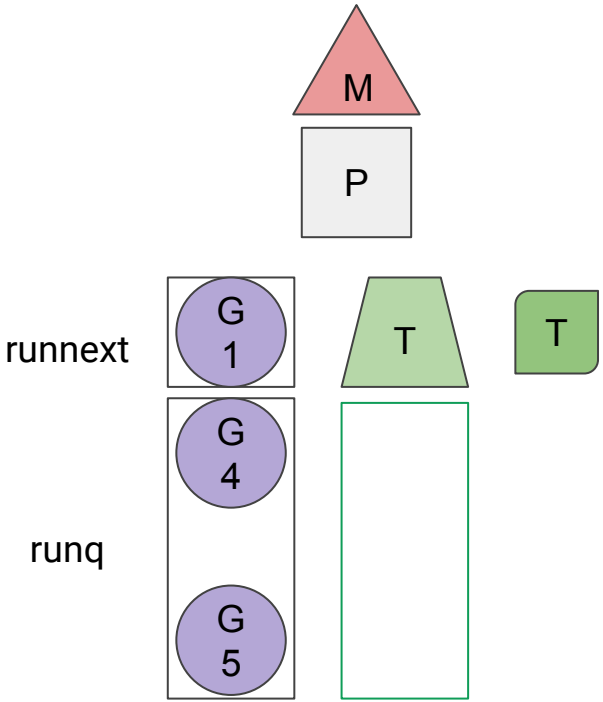
time.Timer: startTimer (addtimer) -> timer.C (sleep) -> schedule*2 -> schedule (checkTimers)

go 1.14



time.Timer: startTimer (addtimer) -> timer.C (sleep) -> schedule*2 -> schedule (checkTimers)

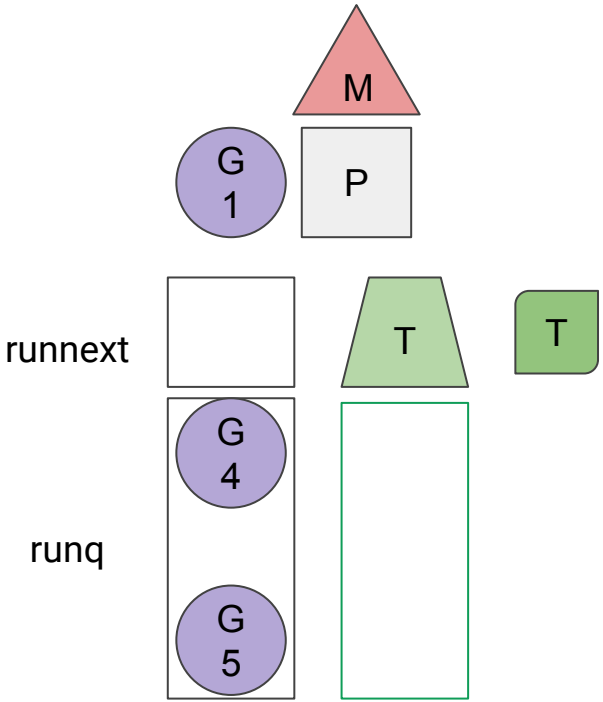
go 1.14



chansend -> goready(g1) -> ready(g1, next)

time.Timer: startTimer (addtimer) -> timer.C (sleep) -> schedule*2 -> schedule (checkTimers -> findrunnable)

go 1.14



读源码时间

```go
type Timer struct {
    C <-chan Time
    r timer
}
func NewTimer(d Duration) *Timer {
    c := make(chan Time, 1)
    t := &Timer{
        C: c,
        r: runtimeTimer{
            when: when(d),
            f:    sendTime,
            arg:  c,
        },
    }
    startTimer(&t.r)
    return t
}
```

```go
type timer struct {
    pp       uintptr
    when     int64
    period   int64
    f        func(interface{}, uintptr)
    arg      interface{}
    seq      uintptr
    nextwhen int64
    status   uint32
}
```

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

```go
type p struct {
    (...)
    timersLock mutex
    timers []*timer
    (...)
}
```

```go
func schedule() {
    _g_ := getg()

    (...)

top:
    pp := _g_.m.p.ptr()
    (...)

    checkTimers(pp, 0)

    (...)
    execute(...)
}
```

```go
func checkTimers(pp *p, now int64) {
    lock(&pp.timersLock)
    (...)
    for len(pp.timers) > 0 {
        if tw := runtimer(pp, rnow); tw != 0 {
            break
        }
    }
    (...)
    unlock(&pp.timersLock)
}
```

运行 Timer

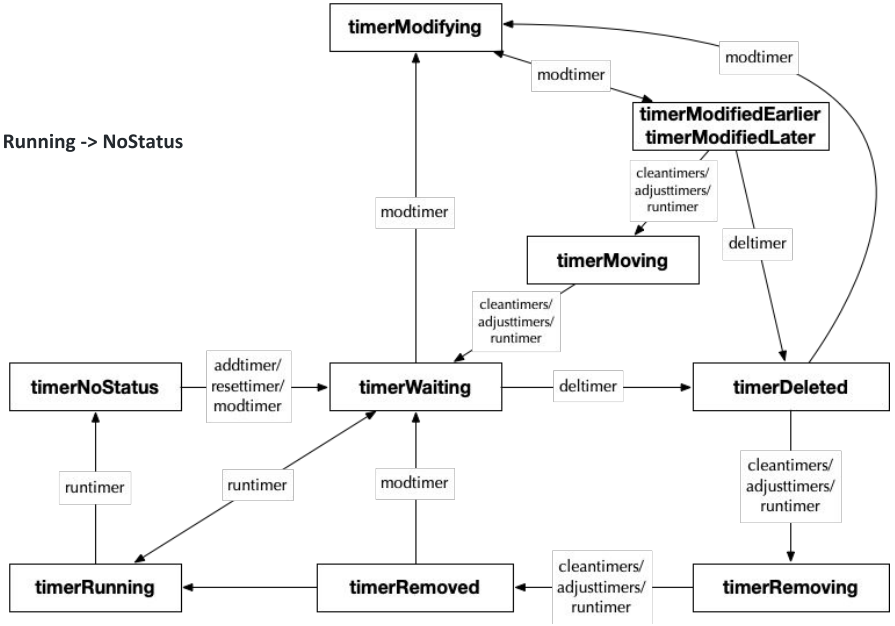
go 1.14

```
func runtimer(pp *p, now int64) int64 {
    for {
        t := pp.timers[0]
        (...)
        switch s := atomic.Load(&t.status); s {
        case timerWaiting:
            if t.when > now {
                return t.when
            }
            (...)
            runOneTimer(pp, t, now)
            return 0
        (...)
        }
    }
}
```

```
func runOneTimer(pp *p, t *timer, now int64) {
    (...)
    f := t.f
    arg := t.arg
    seq := t.seq
    dodeltimer0(pp)
    atomic.Cas(&t.status, timerRunning, timerNoStatus)
    (...)
    unlock(&pp.timersLock)
    f(arg, seq) // 触发 sendTime 信号 通知用户 goroutine
    lock(&pp.timersLock)
    (...)
}
```

更多细节处理：Timer 状态机

1. 一个 Timer 的标准生命周期为：
   - **NoStatus -> Waiting -> Running -> NoStatus**
2. 当人为的对 Timer 进行删除时：
   - **NoStatus -> Waiting -> Deleted -> Removing -> Removed**
3. 当人为的对 Timer 进行修改时：
   - **NoStatus -> Waiting -> Modifying -> ModifiedEarlier/ModifiedLater -> Moving -> Waiting -> Running -> NoStatus**
4. 当人为的对 Timer 进行重置时：
   - **NoStatus -> Waiting -> Deleted -> Removing -> Removed -> Waiting -> Running -> NoStatus**



实验

性能测试


```
name                      old time/op  new time/op  delta
AfterFunc-12              1.57ms ± 1%  0.07ms ± 1%  -95.42%  (p=0.000 n=10+8)
After-12                  1.63ms ± 3%  0.11ms ± 1%  -93.54%  (p=0.000 n=9+10)
Stop-12                   78.3µs ± 3%  73.6µs ± 3%   -6.01%  (p=0.000 n=9+10)
SimultaneousAfterFunc-12   138µs ± 1%   111µs ± 1%  -19.57%  (p=0.000 n=10+9)
StartStop-12              28.7µs ± 1%  31.5µs ± 5%   +9.64%  (p=0.000 n=10+7)
Reset-12                  6.78µs ± 1%  4.24µs ± 7%  -37.45%  (p=0.000 n=9+10)
Sleep-12                   183µs ± 1%   125µs ± 1%  -31.67%  (p=0.000 n=10+9)
Ticker-12                 5.40ms ± 2%  0.03ms ± 1%  -99.43%  (p=0.000 n=10+10)
Sub-12                     114ns ± 1%   113ns ± 3%     ~     (p=0.069 n=9+10)
Now-12                    37.2ns ± 1%  36.8ns ± 3%     ~     (p=0.287 n=8+8)
NowUnixNano-12            38.1ns ± 2%  37.4ns ± 3%   -1.87%  (p=0.020 n=10+9)
Format-12                  252ns ± 2%   195ns ± 3%  -22.61%  (p=0.000 n=9+10)
FormatNow-12               234ns ± 1%   177ns ± 2%  -24.34%  (p=0.000 n=10+10)
MarshalJSON-12             320ns ± 2%   250ns ± 0%  -21.94%  (p=0.000 n=8+8)
MarshalText-12             320ns ± 2%   245ns ± 2%  -23.30%  (p=0.000 n=9+10)
Parse-12                   206ns ± 2%   208ns ± 4%     ~     (p=0.084 n=10+10)
ParseDuration-12          89.1ns ± 1%  86.6ns ± 3%   -2.78%  (p=0.000 n=10+10)
Hour-12                   4.43ns ± 2%  4.46ns ± 1%     ~     (p=0.324 n=10+8)
Second-12                 4.47ns ± 1%  4.40ns ± 3%     ~     (p=0.145 n=9+10)
Year-12                   14.6ns ± 1%  14.7ns ± 2%     ~     (p=0.112 n=9+9)

Day-12                    20.1ns ± 3%  20.2ns ± 1%     ~     (p=0.404 n=10+9)
```

示例代码

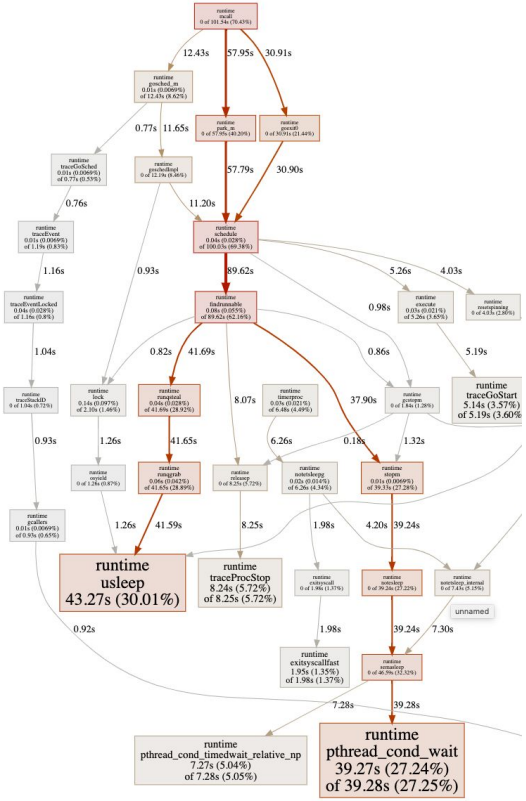
github.com/changkun/sched

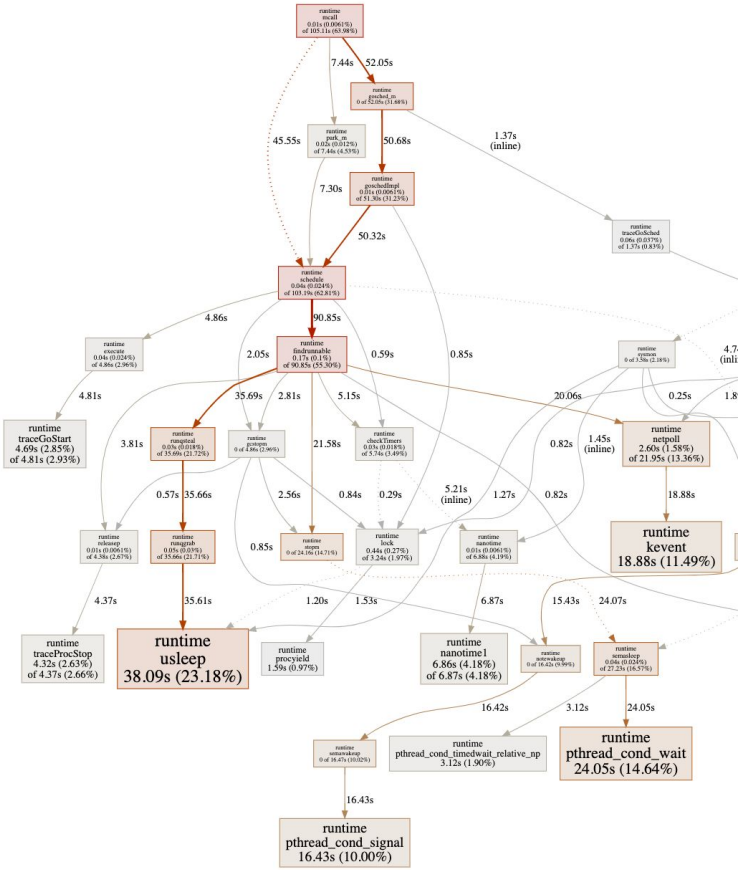
→

simsched: pure timer multiplexing

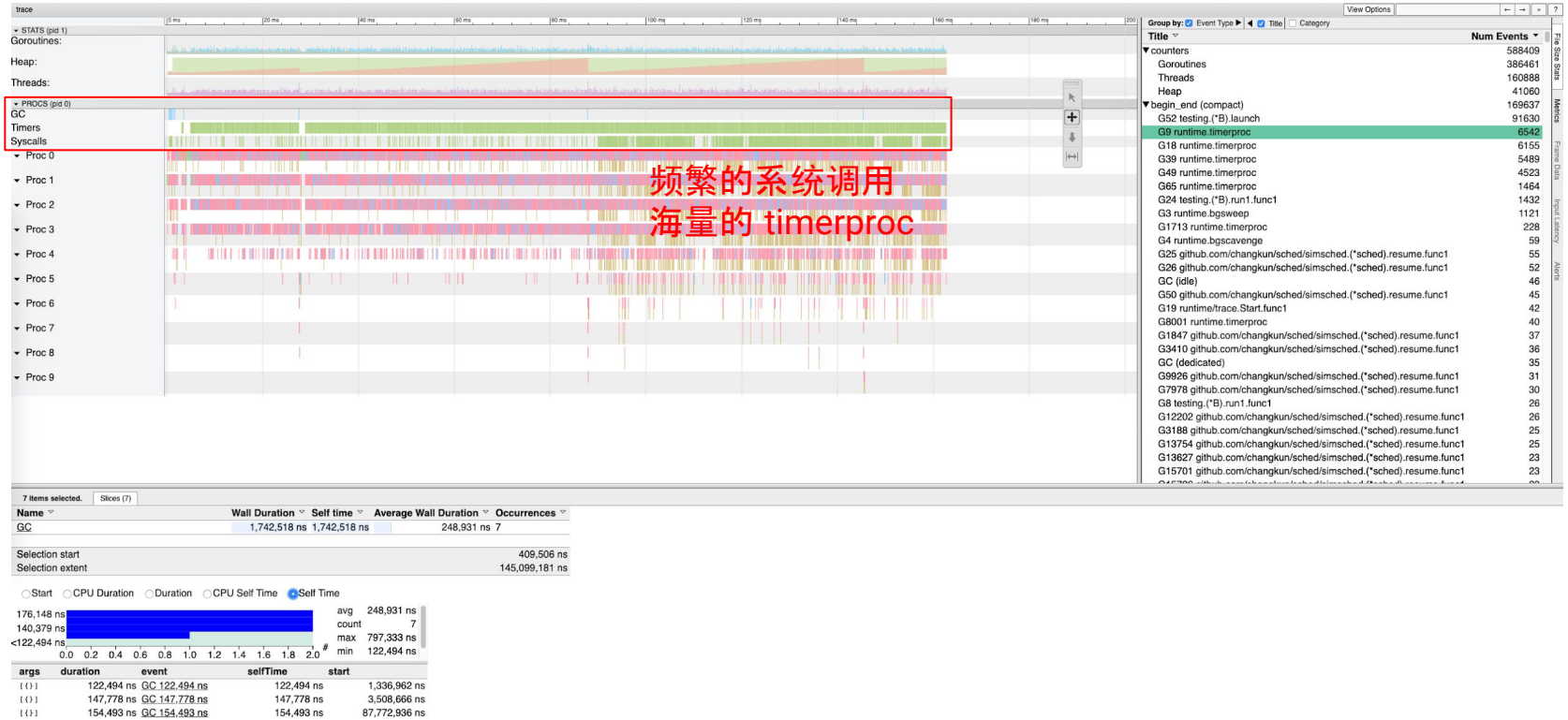
```go
func BenchmarkSubmit(b *testing.B) {
    for size := 10; size < 100000; size *= 10 {
        ss := size
        b.Run(fmt.Sprintf("#tasks-%d", ss), func(b *testing.B) {
            ts := newTasks(ss)
            b.ResetTimer()
            for i := 0; i < b.N; i++ {
                b.StopTimer()
                for j := 0; j < ss-1; j++ {
                    Submit(ts[j])
                }
                b.StartTimer()

                Submit(ts[ss-1])
                Wait()
            }
        })
    }
}
```

pprof (1.13 → 1.14)

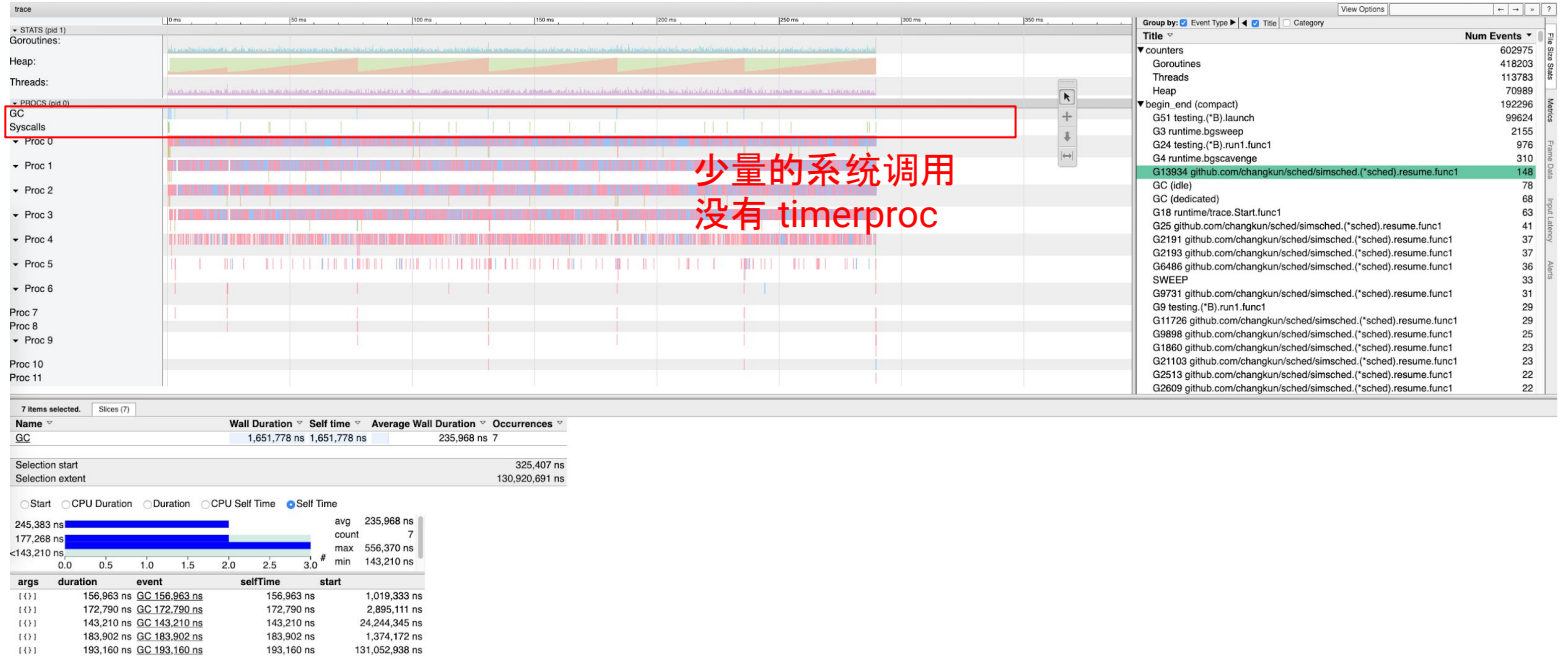




trace (1.13)



trace (1.14)



进一步阅读的参考文献

- runtime: improve timers scalability on multi-CPU systems: issue, commit
- runtime: make timers faster: issue
- Go 夜读第 56 期：channel & select 源码分析
- Go 语言原本：`time.*` 的计时器 Timer
- task scheduler: `github.com/changkun/sched`