goroutineの仕組みについて

~Go Conference 2021 Autumn~

自己紹介

- Takuma Shibuya @sivchari
- 22卒 Cyber Agent(予定)
  - バックエンドエンジニア
- linter, OSSが好き

話すこと

- Goのruntimeを支える要素
- goroutineのライフタイム
- goroutineのスケジューリング

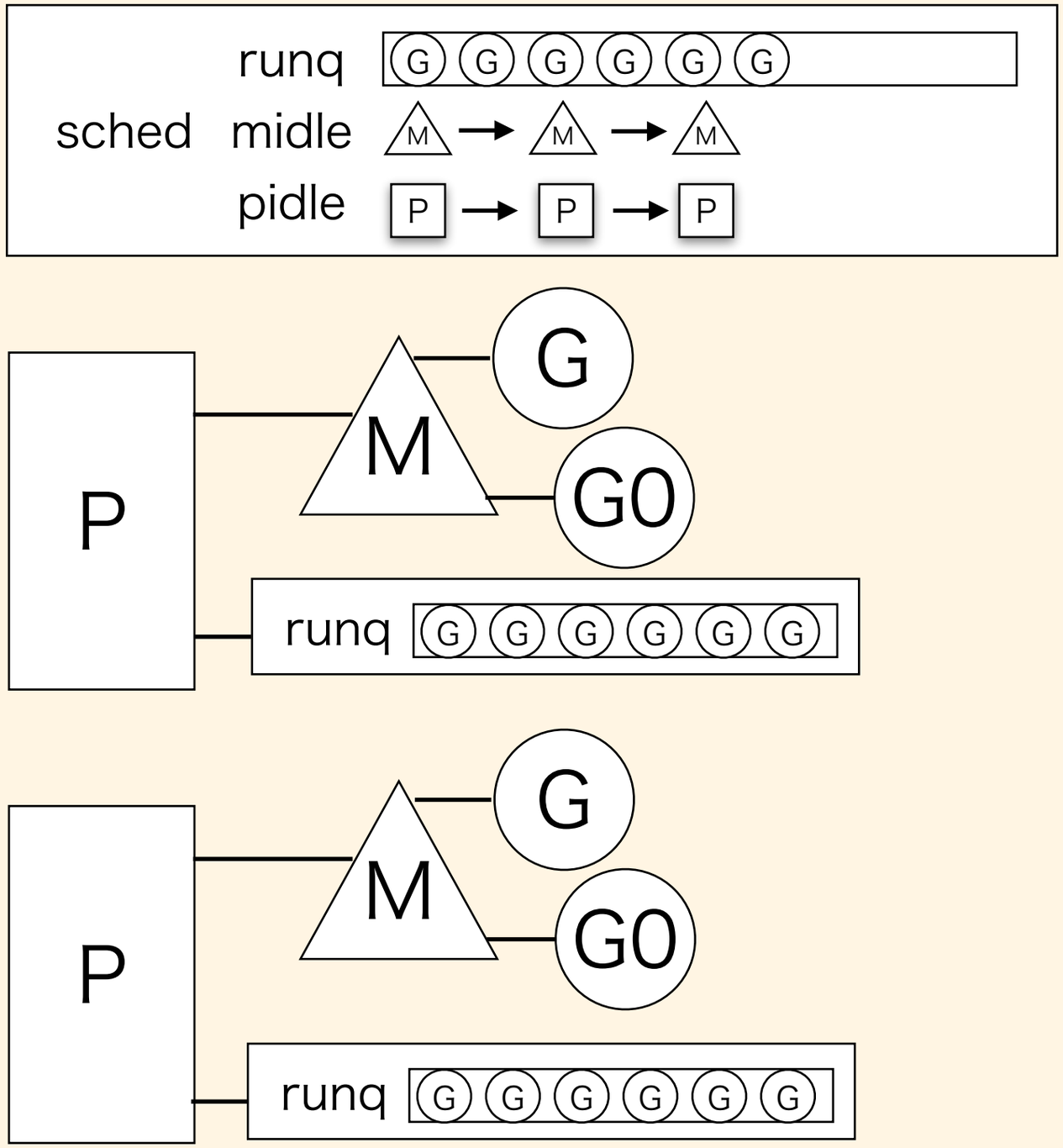
~Let's Go~

~Goのruntimeを支える要素~

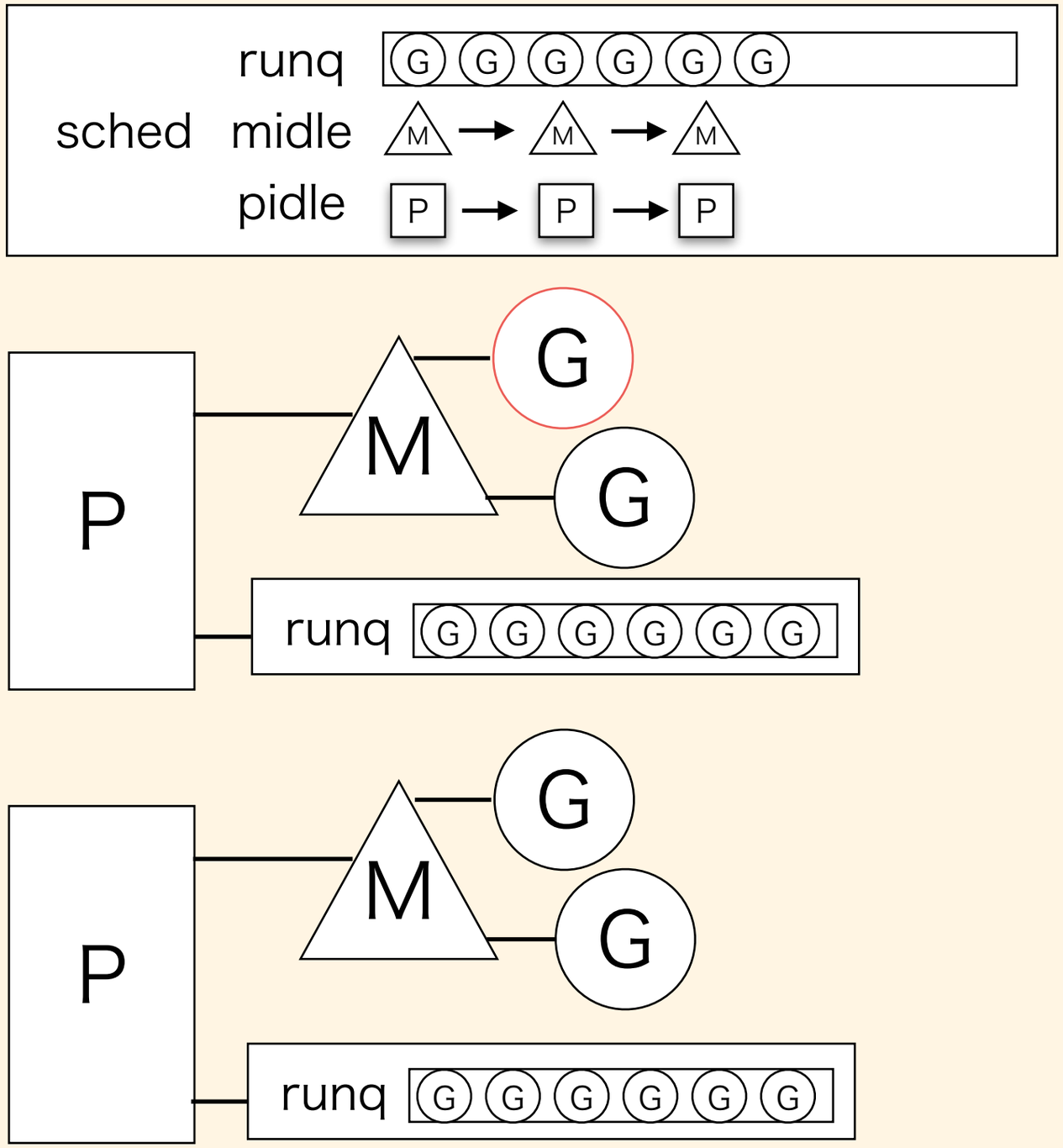
登場人物

- G: goroutine
- M: マシンスレッド
- P: goroutineを実行するリソース
- sched: グローバルでユニーク、全てのPで共有される

全体像


runq
G G G G G G
sched
midle
M M M
pidle
P P P
P
M
G
G0
runq
G G G G G G
P
M
G
G0
runq
G G G G G G

全体像

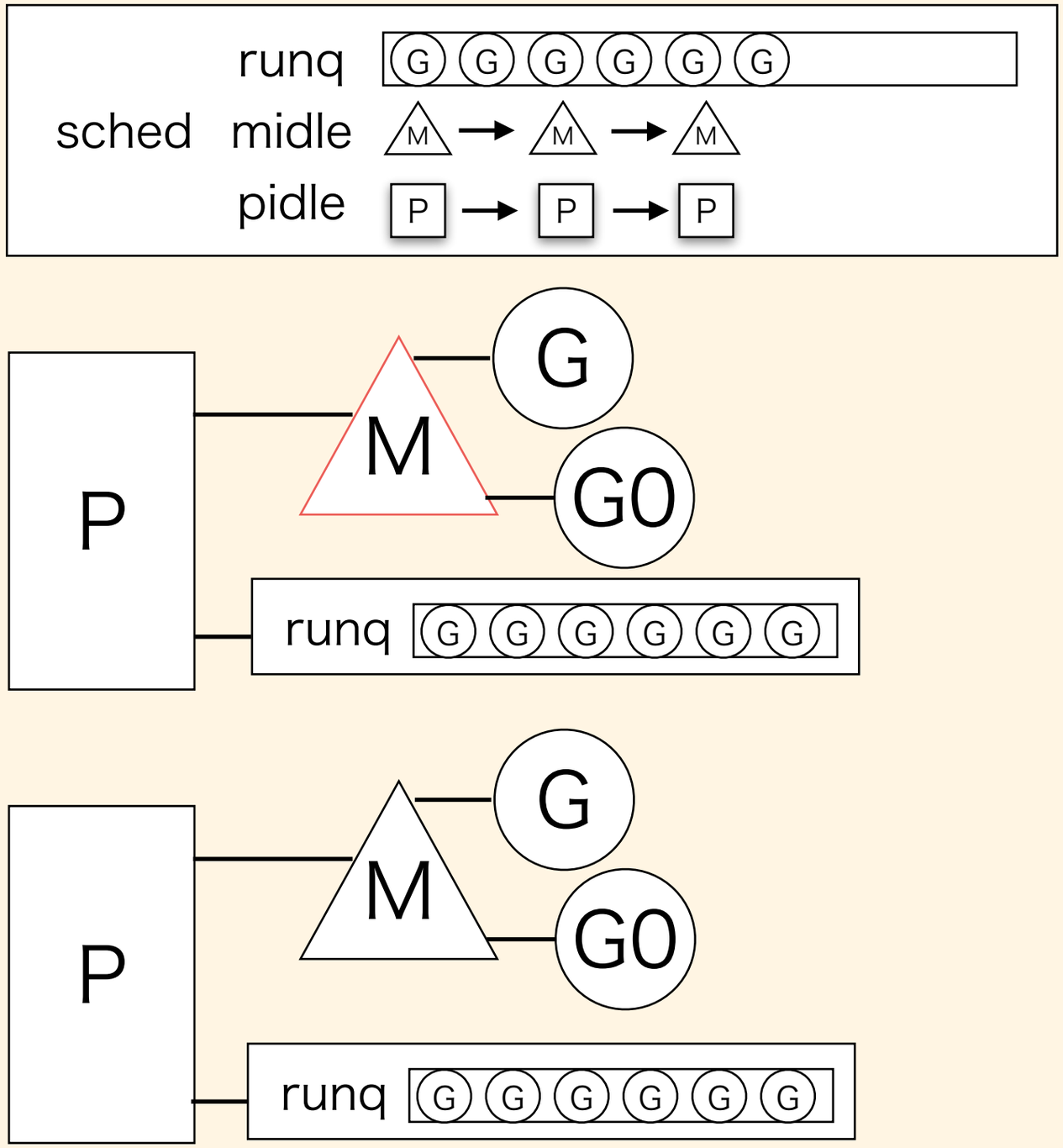

runq
G G G G G G
sched
midle
M M M
pidle
P P P
P
M
G
G
runq
G G G G G G
P
M
G
G
runq
G G G G G G

G

- m: g(G)を実行しているm(M)
- sched: PCなどの値を保持するスタック
- atomicstatus: goroutineの状態(8つ)
- gopc: goroutineを生成したgoのPC
- startpc: goroutineのPC

```
type g struct {
    m           *m // current m; offset known to arm liblink
    sched       gobuf
    atomicstatus uint32
    gopc        uintptr // pc of go statement that created this goroutine
    startpc     uintptr // pc of goroutine function
}
```

全体像

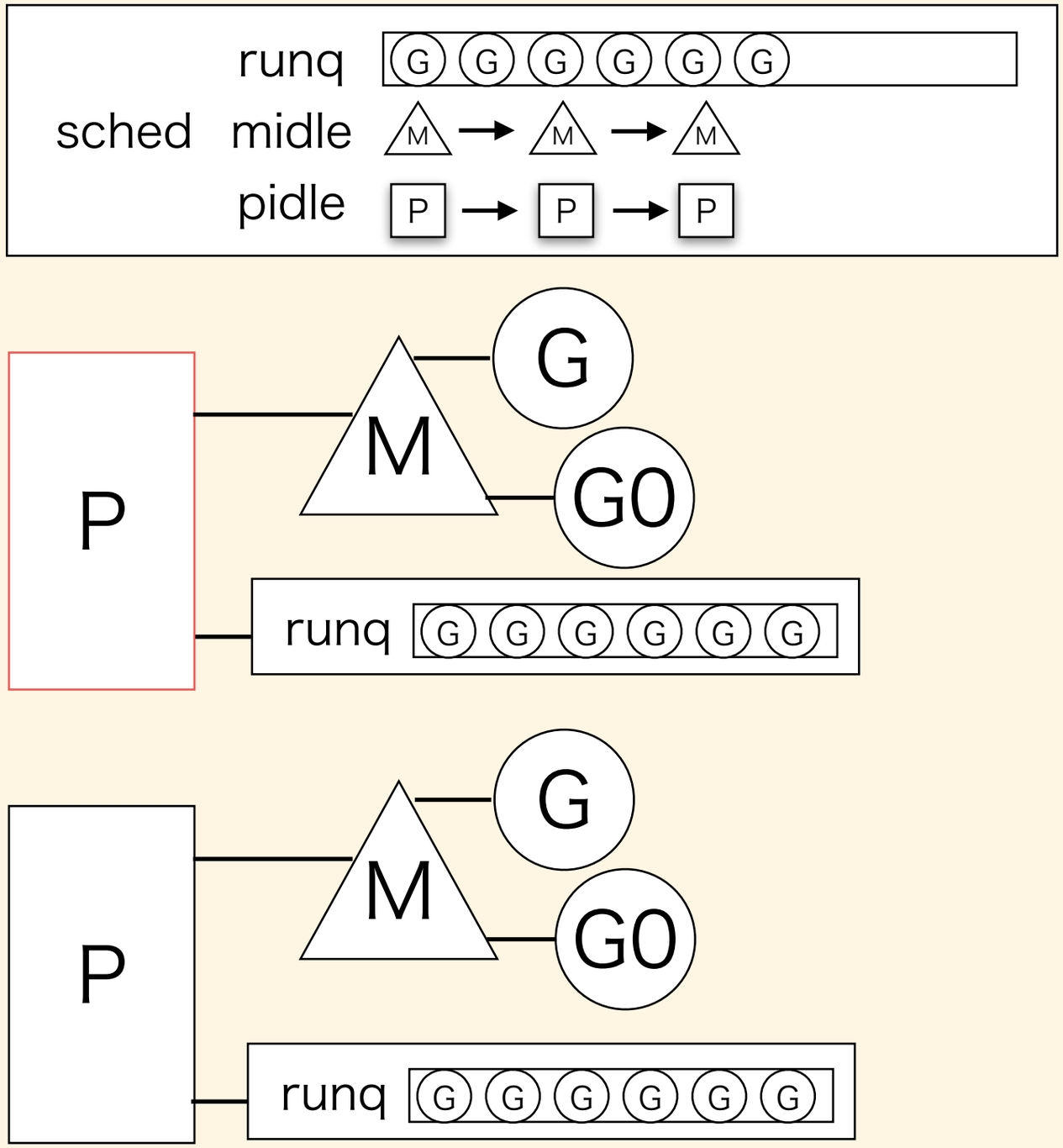

runq
G G G G G G
sched
midle
M M M
pidle
P P P
P
M
G
G0
runq
G G G G G G
P
M
G
G0
runq
G G G G G G

M

- g0: スケジューラーのためのgoroutine
- curg: 現在実行しているgoroutine
- p: Goのコードを実行するために紐づくp(P)

```
type m struct {
    g0            *g     // goroutine with scheduling stack
    curg          *g       // current running goroutine
    p             puintptr // attached p for executing go code (nil if not executing go code)
}
```

全体像

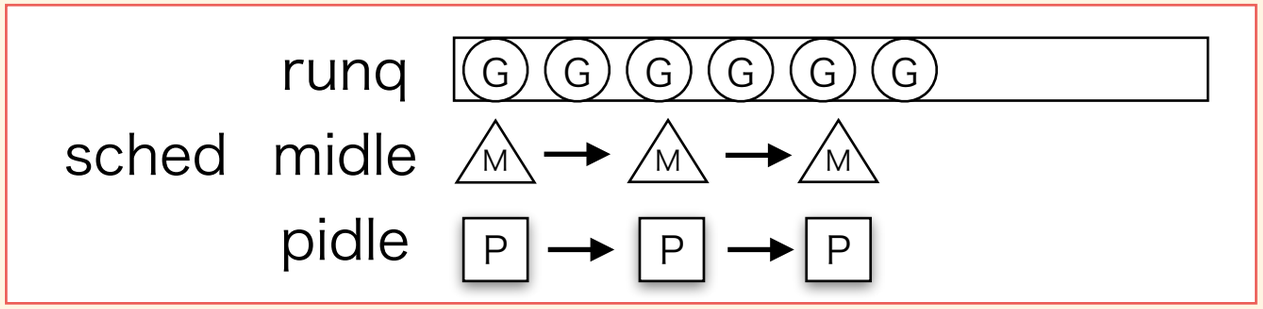

runq
G G G G G G
sched
midle
M M M
pidle
P P P
P
M
G
G0
runq
G G G G G G
P
M
G
G0
runq
G G G G G G

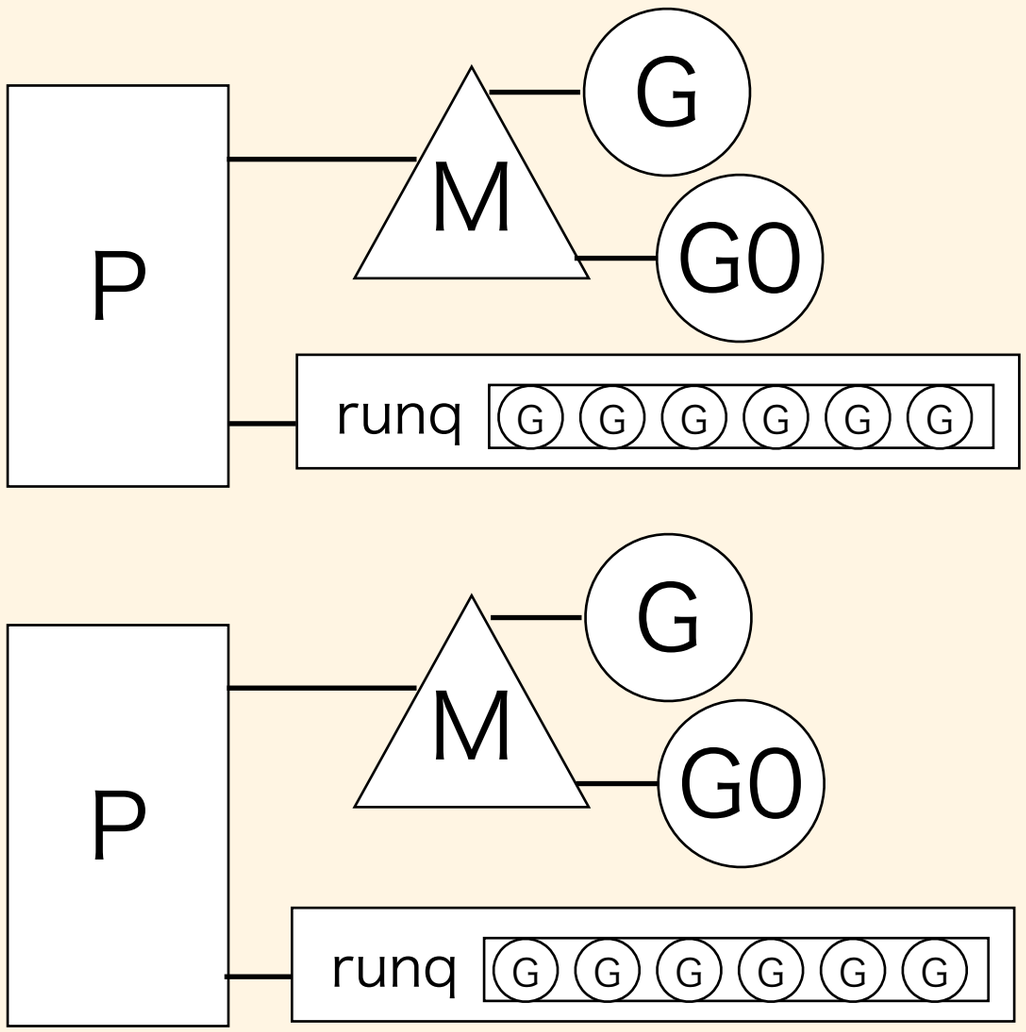
P

- m: p(P)に紐づくm(M)
- runq: Pごとに保持するg(G)のlocal queue
- gfree: 実行していない利用可能なg(G)

```
type p struct {
    m           muintptr   // back-link to associated m (nil if idle)
    // Queue of runnable goroutines. Accessed without lock.
    runqhead uint32
    // Available G's (status == Gdead)
    gfree    *g
}
```

全体像


runq
G G G G G G
sched
midle
M M M
pidle
P P P


P
M
G
G0
runq
G G G G G G
P
M
G
G0
runq
G G G G G G

sched

- midle: idleのm(M)を保持
- pidle: idleのp(P)を保持
- runq*: g(G)を保持するglobal queue

```
type schedt struct {
    midle muintptr // idle m's waiting for work

    pidle puintptr // idle p's

    // Global runnable queue.
    runqhead guintptr
    runqtail guintptr
    runqsize int32
}
```

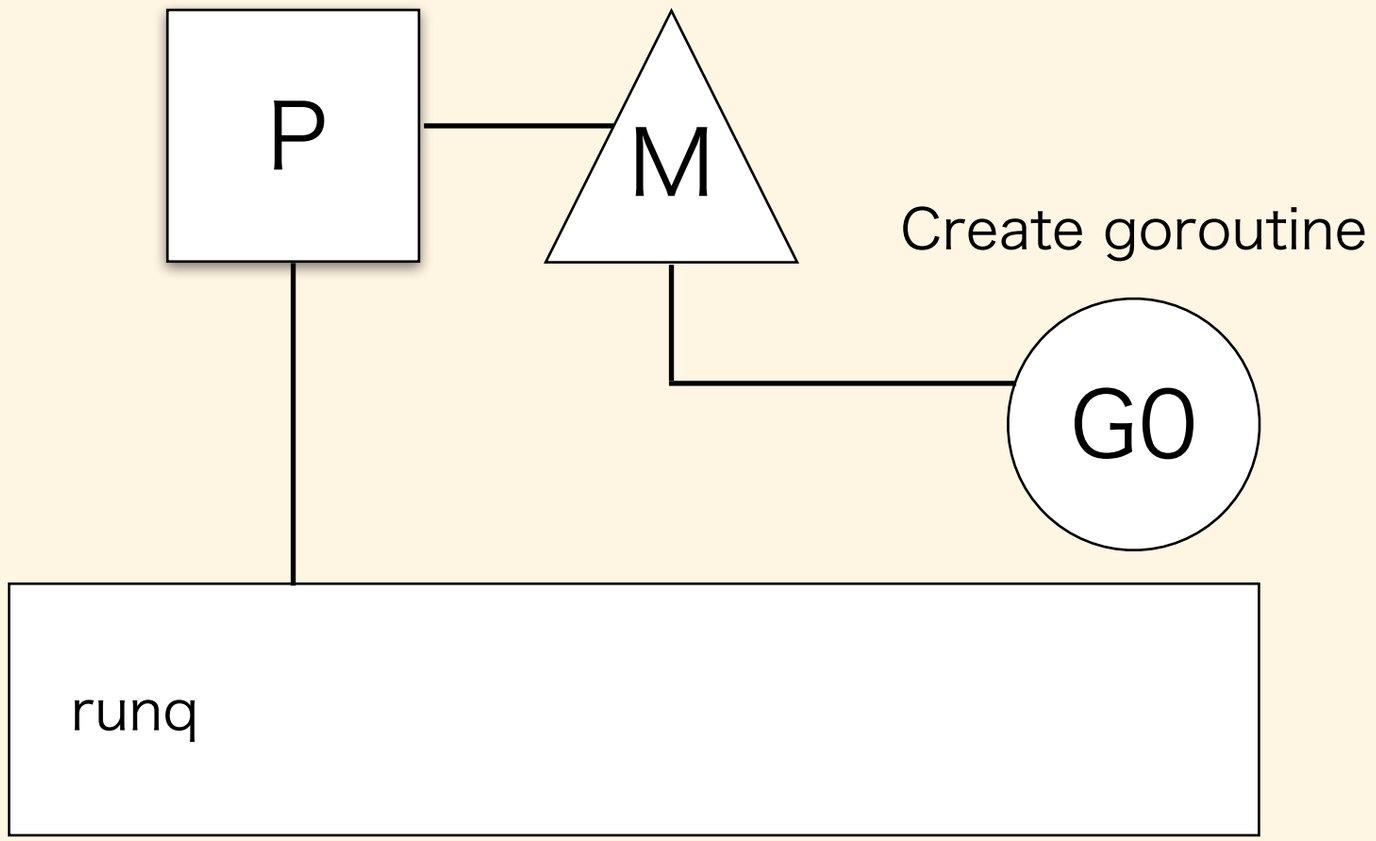
~goroutineのライフタイム~

サンプルコード

```go
func main() {
  var wg sync.WaitGroup
  wg.Add(1)

  go func() {
    defer wg.Done()
    println("Hello GoCon from child goroutine")
  }()

  println("Hello GoCon from main goroutine")

  wg.Wait()
}
```

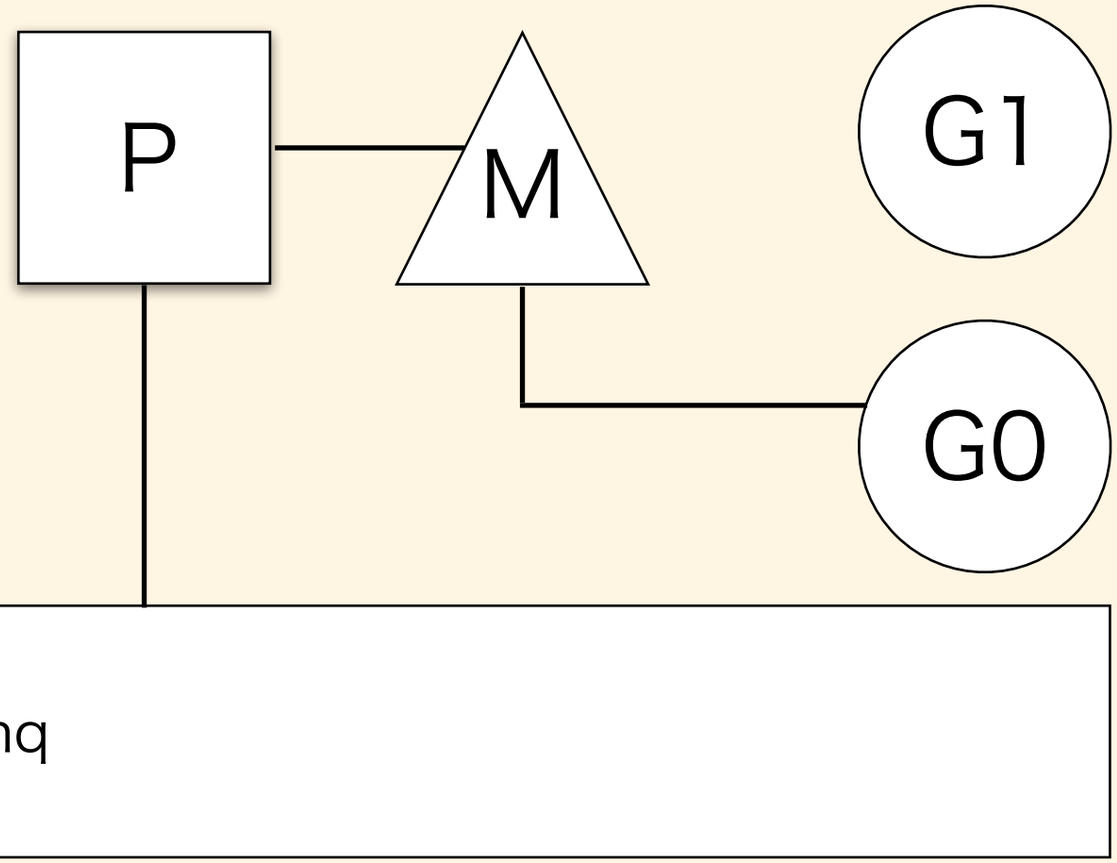
newproc

```go
// Create a new g running fn with siz bytes of arguments.
func newproc(siz int32, fn *funcval) {
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    gp := getg()
    pc := getcallerpc()
    systemstack(func() {
        // the following line
        newg := newproc1(fn, argp, siz, gp, pc)

        _p_ := getg().m.p.ptr()
        runqput(_p_, newg, true)

        if mainStarted {
            wakep()
        }
    })
}
```



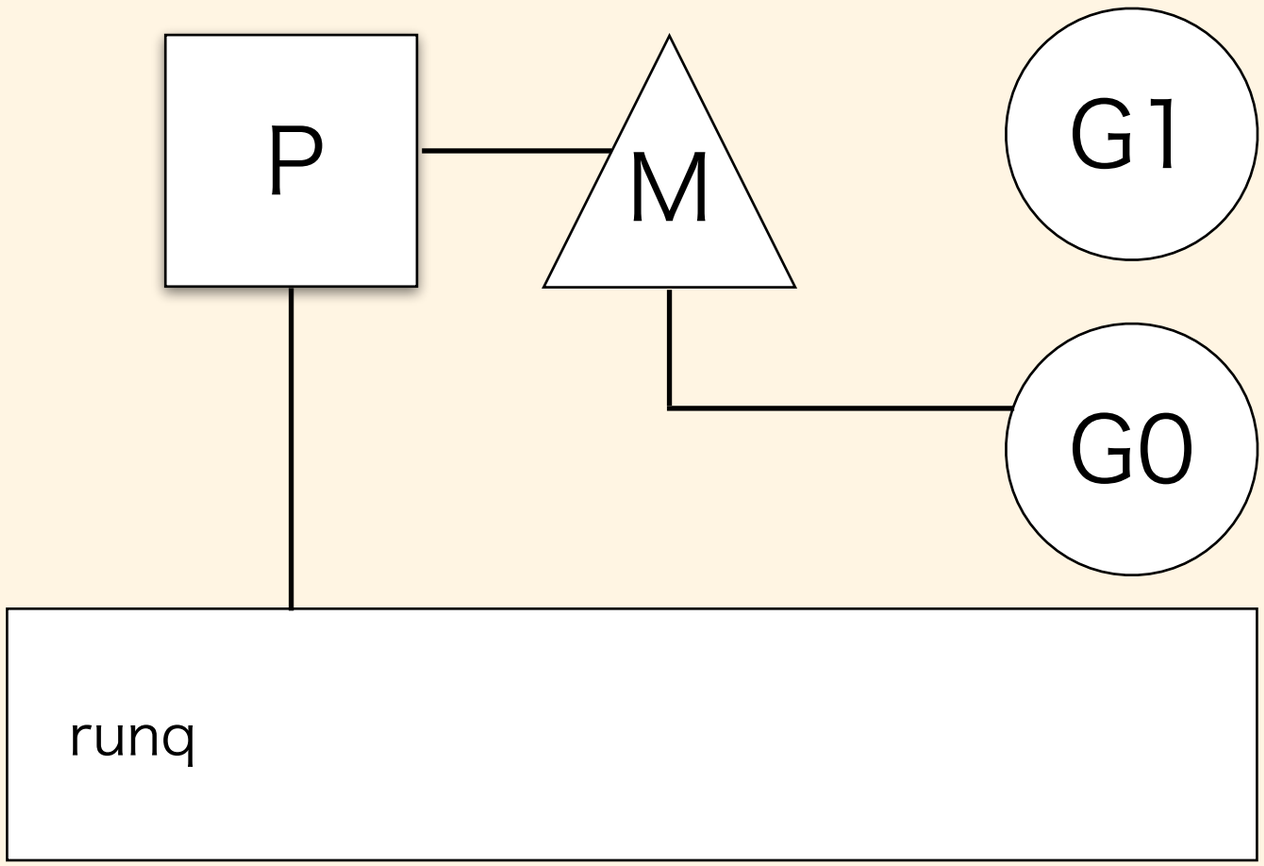
newproc1

```go
// Create a new g in state _Grunnable, starting at fn, with narg bytes
func newproc1(fn *funcval, argp unsafe.Pointer, narg int32, callergp *g, callerpc uintptr) *g {
    _g_ := getg() // 現在のg(G)を取得

    _p_ := _g_.m.p.ptr() // g(G)に紐づくp(P)
    newg := gfget(_p_) // gfreeからg(G)を取得

    newg.sched.pc = abi.FuncPCABI0(goexit) + sys.PCQuantum // pcにgoexitをset
    newg.gopc = callerpc // goroutineを生成したgoのPC
    newg.startpc = fn.fn // goroutineのPC
    return newg
}
```



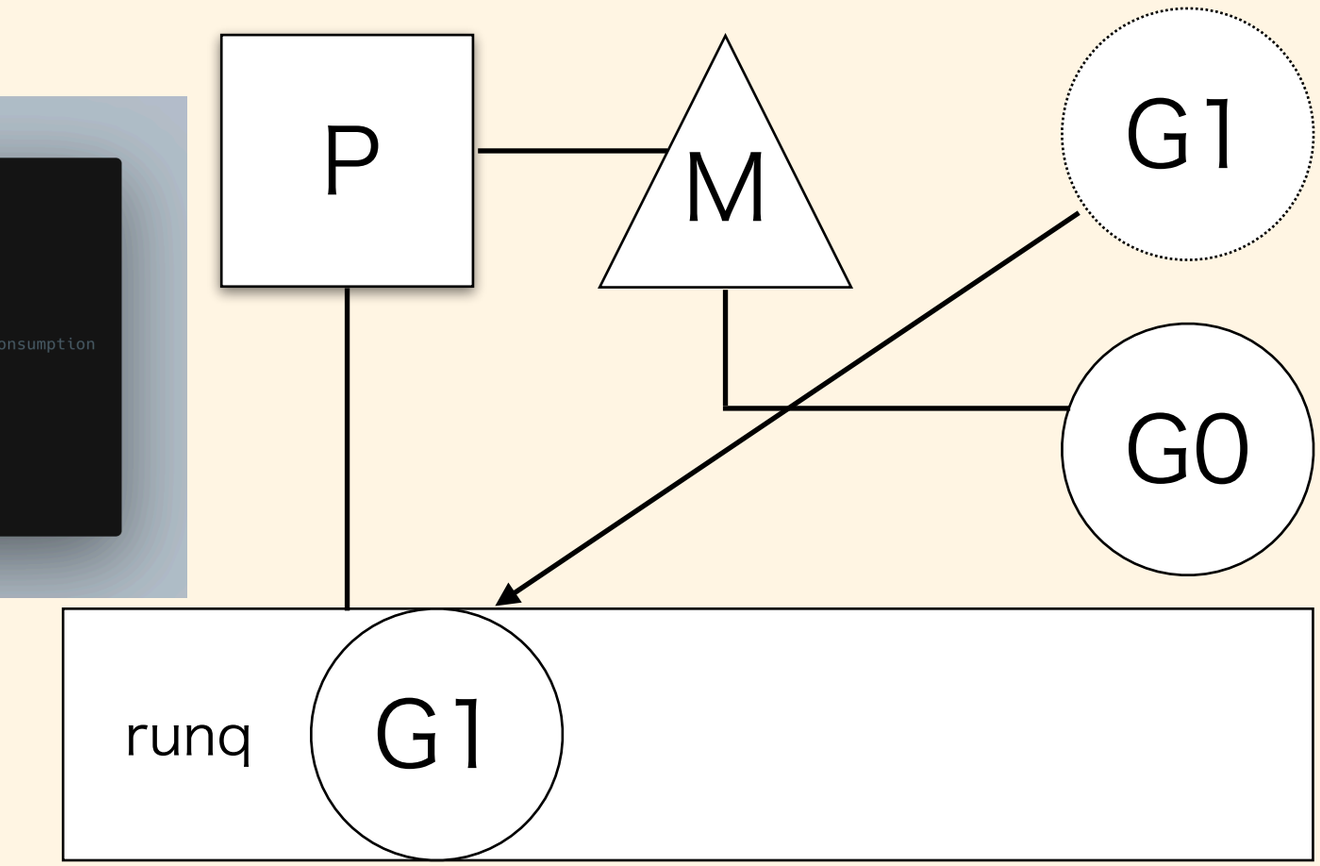
newproc

```go
// Create a new g running fn with siz bytes of arguments.
func newproc(siz int32, fn *funcval) {
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    gp := getg()
    pc := getcallerpc()
    systemstack(func() {
        newg := newproc1(fn, argp, siz, gp, pc)

        _p_ := getg().m.p.ptr()
        // following line
        runqput(_p_, newg, true)

        if mainStarted {
            wakep()
        }
    })
}
```



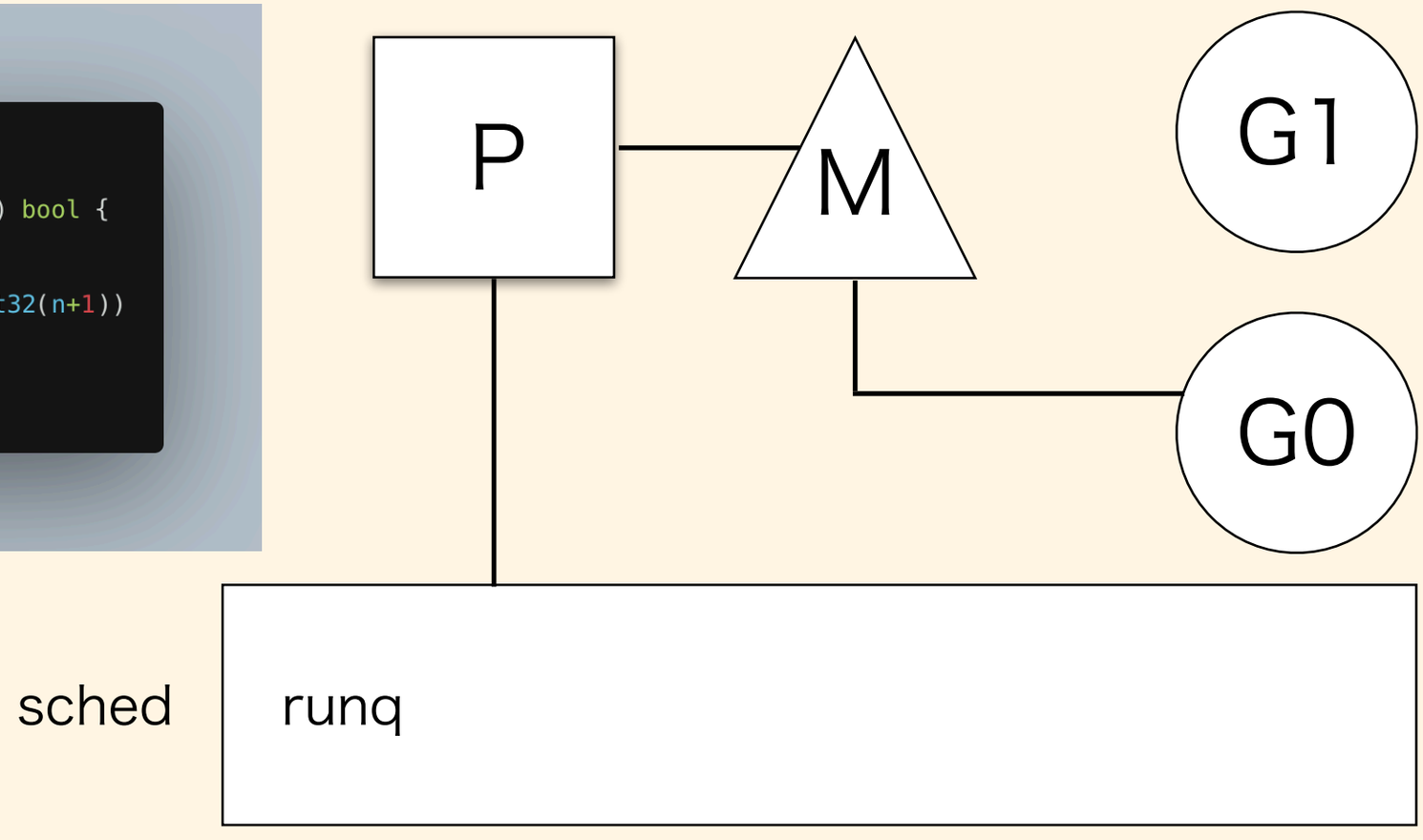
runqput

```go
func runqput(_p_ *p, gp *g, next bool) {
    retry:
    h := atomic.LoadAcq(&_p_.runqhead) // load-acquire, synchronize with consumers
    t := _p_.runqtail
    if t-h < uint32(len(_p_.runq)) { // local runqに追加できるかチェック
        _p_.runq[t%uint32(len(_p_.runq))].set(gp)
        atomic.StoreRel(&_p_.runqtail, t+1) // store-release, makes the item available for consumption
        return
    }
    if runqputslow(_p_, gp, h, t) { // バッチでg(G)をglobal runqに追加する
        return
    }
    // the queue is not full, now the put above must succeed
    goto
}
```



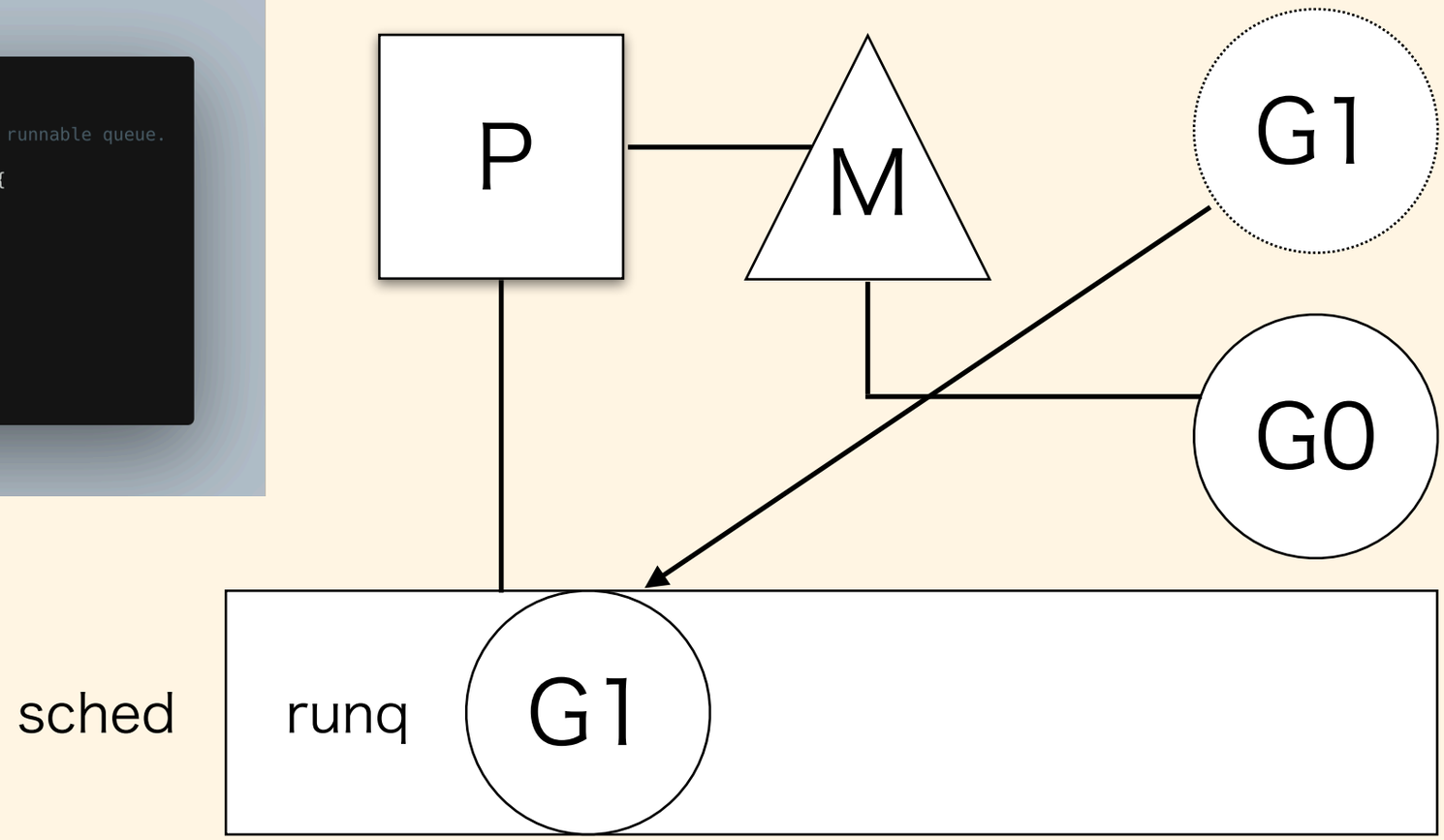
runqputslow

```
func runqputslow(_p_ *p, gp *g, h, t uint32) bool {
    // Now put the batch on global queue.
    lock(&sched.lock)
    globrunqputbatch(batch[0], batch[n], int32(n+1))
    unlock(&sched.lock)
    return true
}
```



globruqputbatch

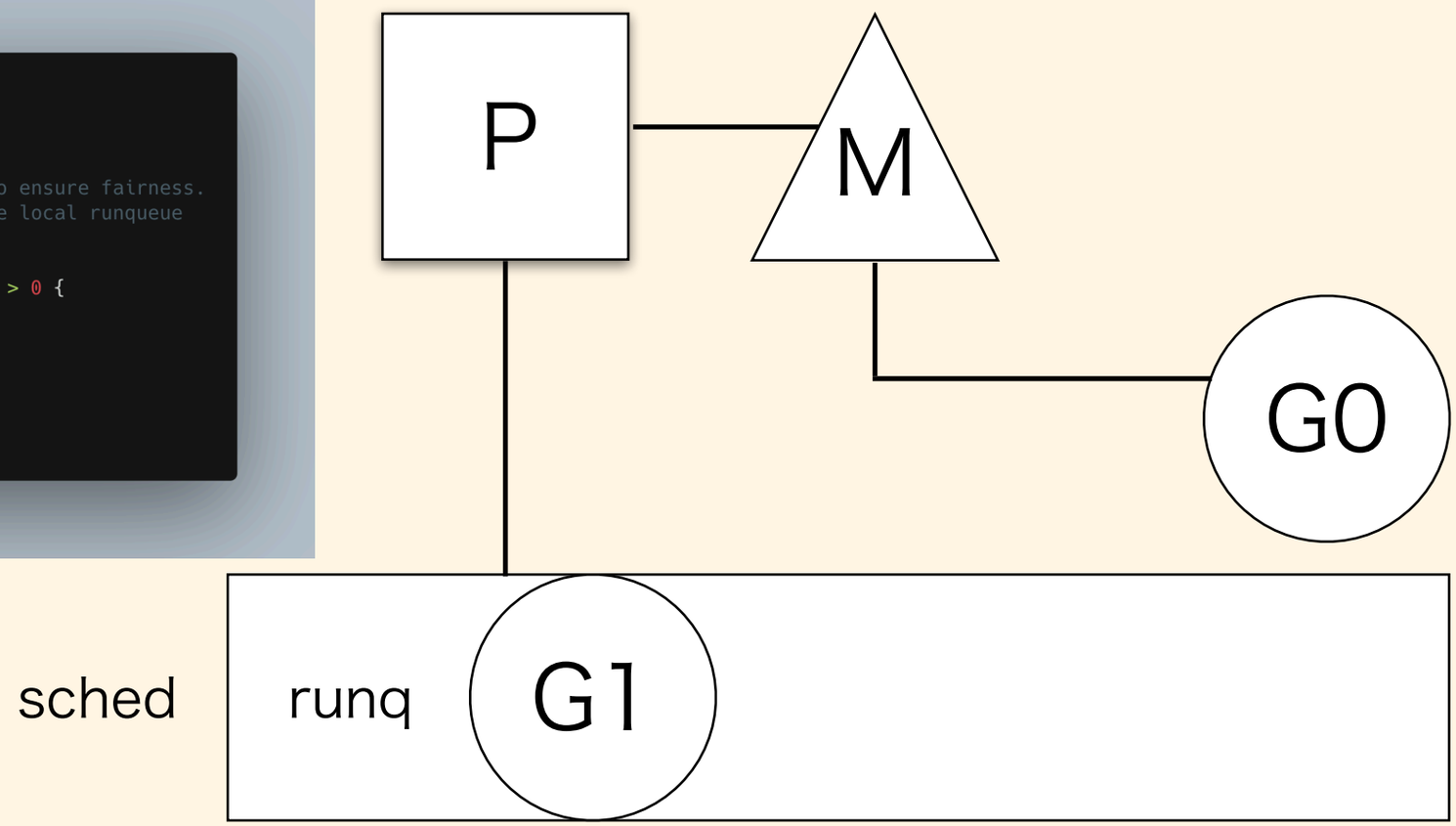
```
// Put a batch of runnable goroutines on the global runnable queue.
// Sched must be locked.
func globrunqputbatch(ghead *g, gtail *g, n int32) {
    gtail.schedlink = 0
    if sched.runqtail != 0 {
        sched.runqtail.ptr().schedlink.set(ghead)
    } else {
        sched.runqhead.set(ghead)
    }
    sched.runqtail.set(gtail)
    sched.runqsize += n
}
```



~goroutineのスケジューリング~

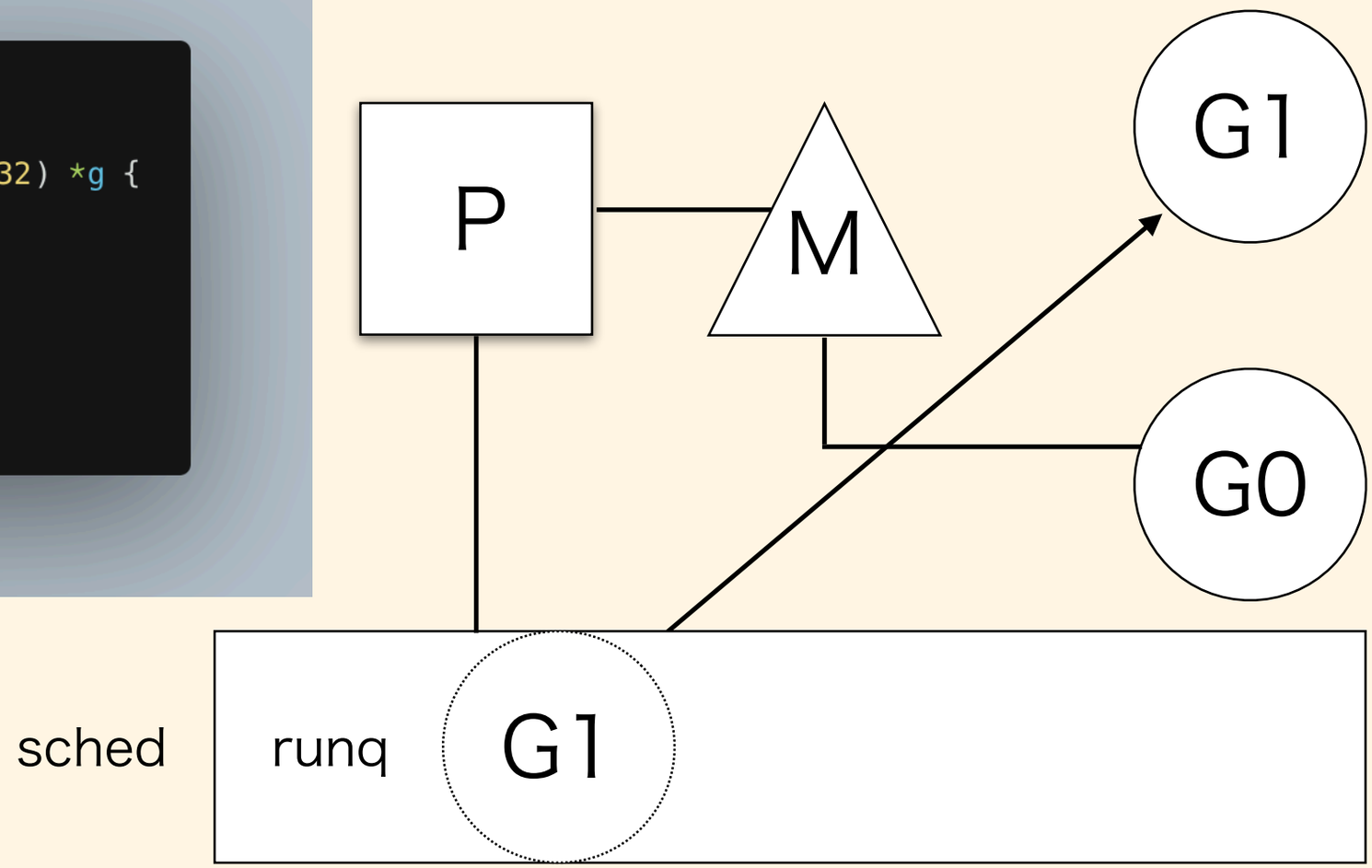
schedule

```go
func schedule() {
    if gp == nil {
        // Check the global runnable queue once in a while to ensure fairness.
        // Otherwise two goroutines can completely occupy the local runqueue
        // by constantly respawning each other.
        // 61回に1回global runqが空でなければ取得
        if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
            lock(&sched.lock)
            gp = globrunqget(_g_.m.p.ptr(), 1)
            unlock(&sched.lock)
        }
    }
}
```



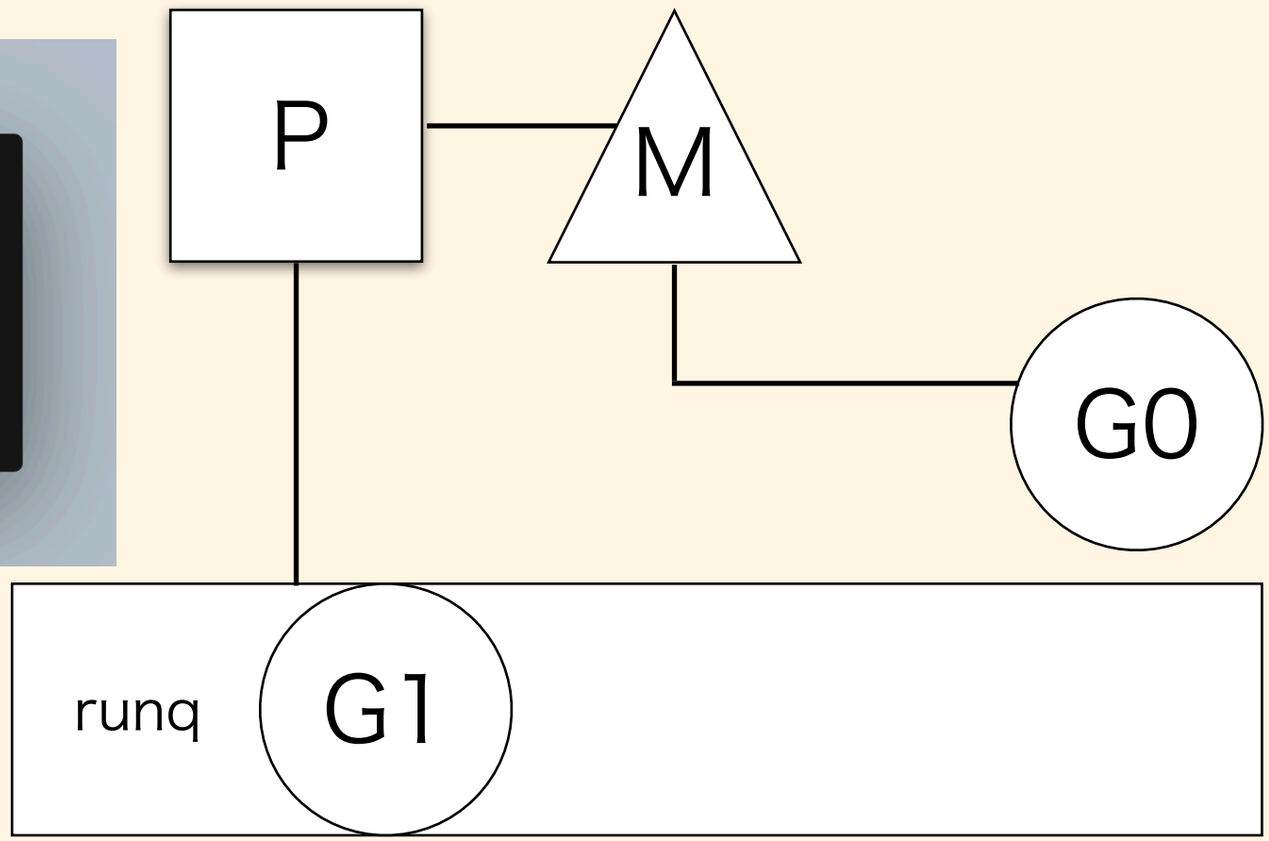
globrunqget

```
func globrunqget(_p_ *p, max int32) *g {
    if sched.runqsize == 0 {
        return nil
    }
    gp := sched.runq.pop()
    return gp
}
```



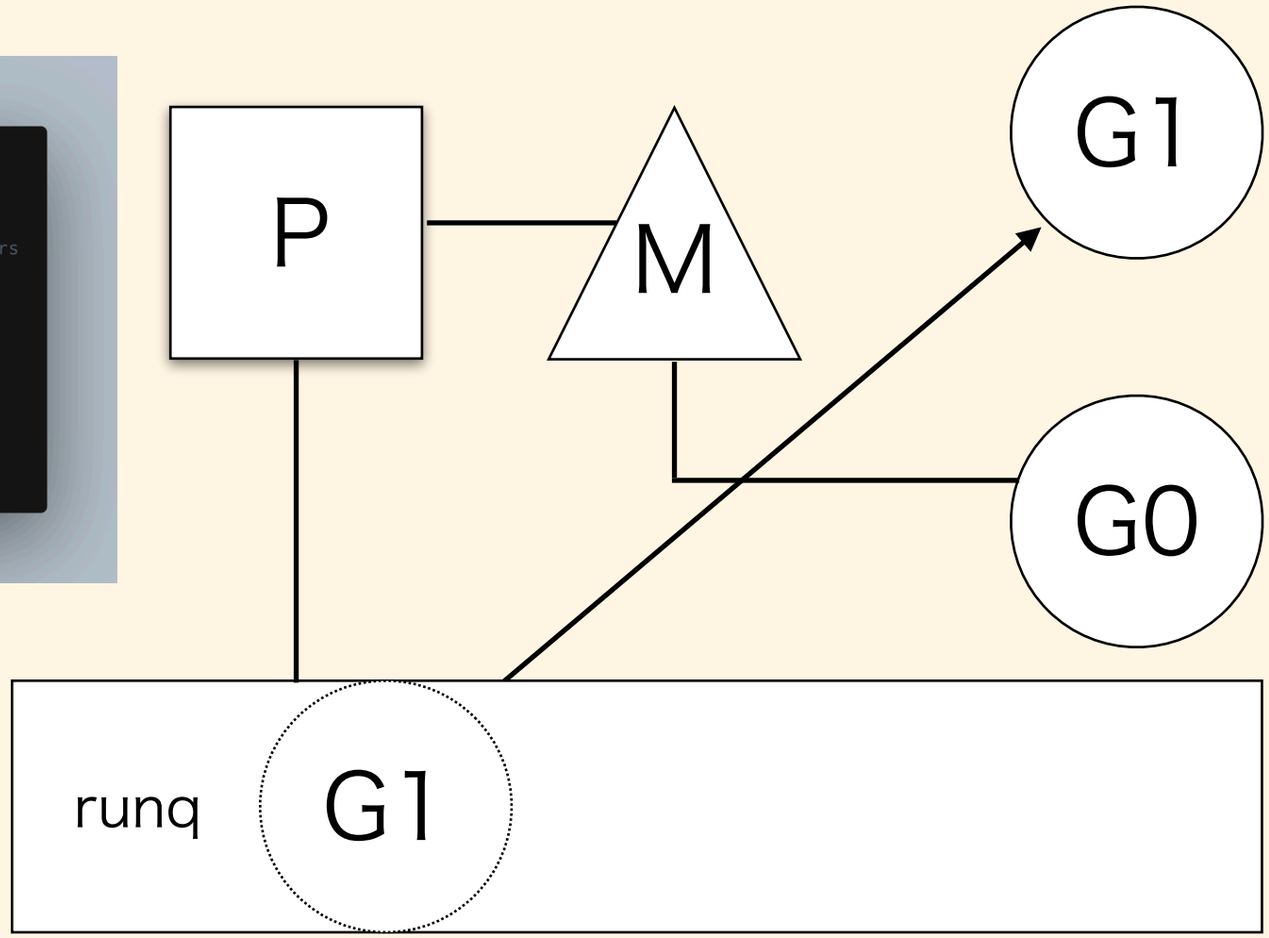
schedule

```
func schedule() {
    if gp == nil {
        gp, inheritTime = runqget(_g_.m.p.ptr())
        // We can see gp != nil here even if the M is spinning,
        // if checkTimers added a local goroutine via goready.
    }
}
```



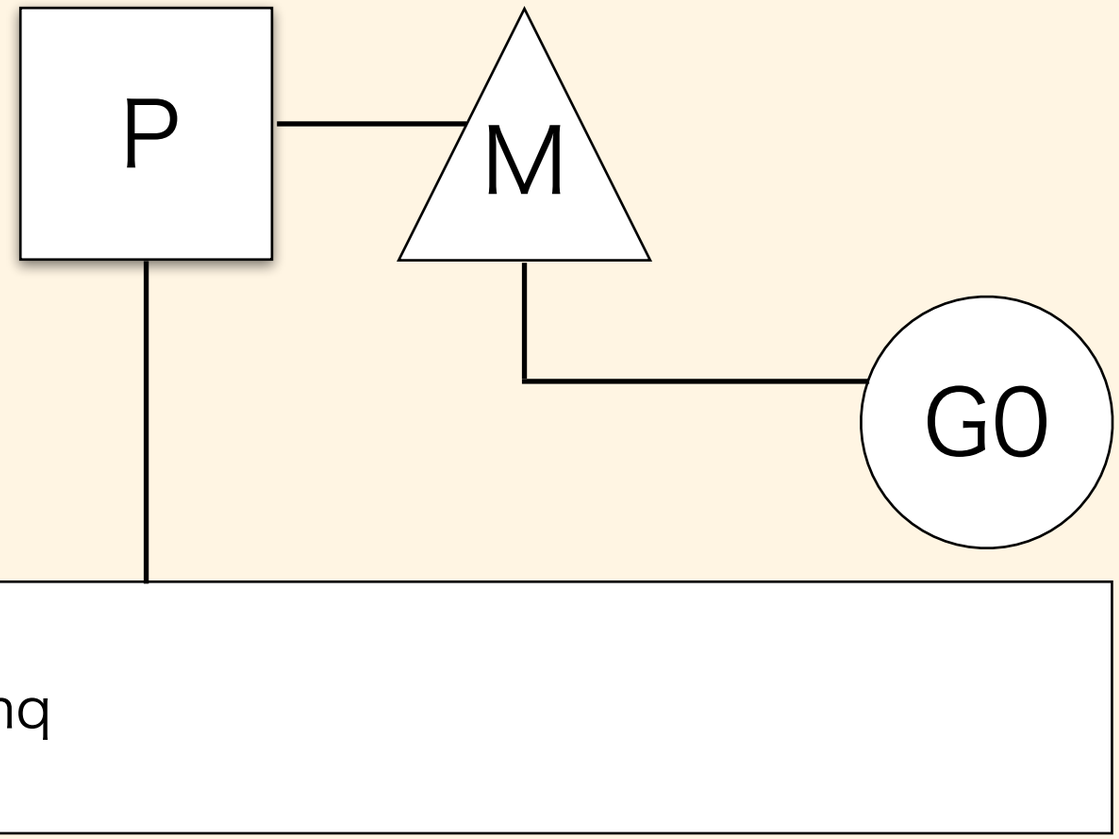
runqget

```go
func runqget(_p_ *p) (gp *g, inheritTime bool) {
    for {
        h := atomic.LoadAcq(&_p_.runqhead) // load-acquire, synchronize with other consumers
        t := _p_.runqtail
        if t == h {
            return nil, false // local runqがempty
        }
        gp := _p_.runq[h%uint32(len(_p_.runq))].ptr()
        if atomic.CasRel(&_p_.runqhead, h, h+1) { // cas-release, commits consume
            return gp, false
        }
    }
}
```



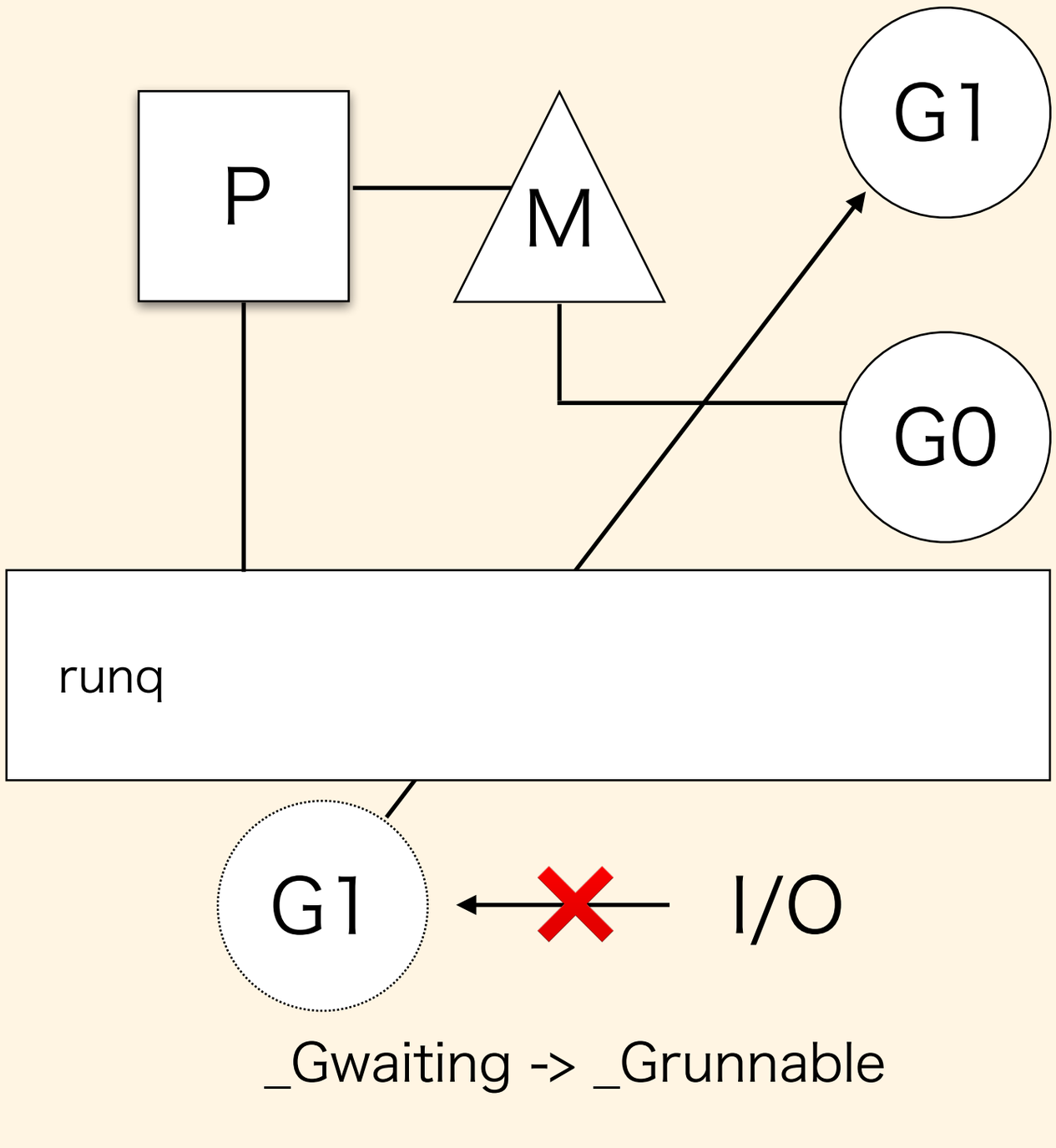
schedule

```
func schedule() {
    if gp == nil {
        gp, inheritTime = findrunnable() // blocks until work is available
    }
}
```



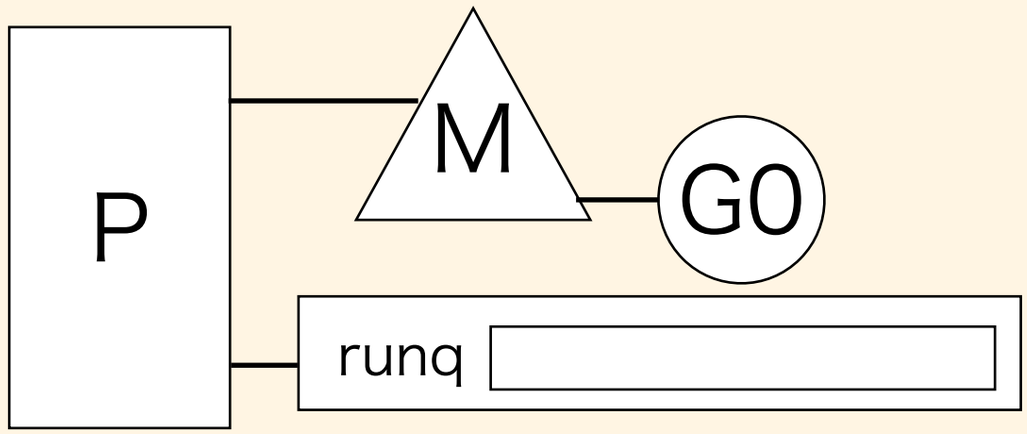
findrunnable

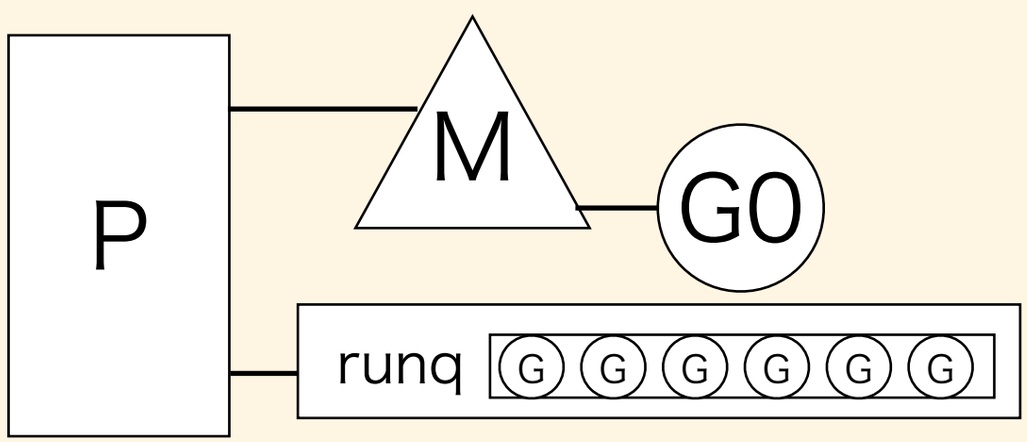
```go
func findrunnable() (gp *g, inheritTime bool) {
    // Poll network.
    // This netpoll is only an optimization before we resort to stealing.
    // We can safely skip it if there are no waiters or a thread is blocked
    // in netpoll already. If there is any kind of logical race with that
    // blocked thread (e.g. it has already returned from netpoll, but does
    // not set lastpoll yet), this thread will do blocking netpoll below
    // anyway.
    // I/O待ちが終了し、別プログラムの実行ができる(switch from _Gwaiting to _Grunnable)
    if netpollinited() && atomic.Load(&netpollWaiters) > 0 && atomic.Load64(&sched.lastpoll) != 0 {
        if list := netpoll(0); !list.empty() { // non-blocking
            gp := list.pop()
            injectglist(&list) // goroutineのlistを更新
            casgstatus(gp, _Gwaiting, _Grunnable)
            if trace.enabled {
                traceGoUnpark(gp, 0)
            }
            return gp, false
        }
    }
}
```



findrunnable

```go
func findrunnable() (gp *g, inheritTime bool) {
    // Spinning Ms: steal work from other Ps.
    //
    // Limit the number of spinning Ms to half the number of busy Ps.
    // This is necessary to prevent excessive CPU consumption when
    // GOMAXPROCS>>1 but the program parallelism is low.
    procs := uint32(gomaxprocs)
    if _g_.m.spinning || 2*atomic.Load(&sched.nmspinning) < procs-atomic.Load(&sched.npidle) {
        if !_g_.m.spinning {
            _g_.m.spinning = true
            atomic.Xadd(&sched.nmspinning, 1)
        }

        // the following line
        gp, inheritTime, tnow, w, newWork := stealWork(now)
    }
}
```

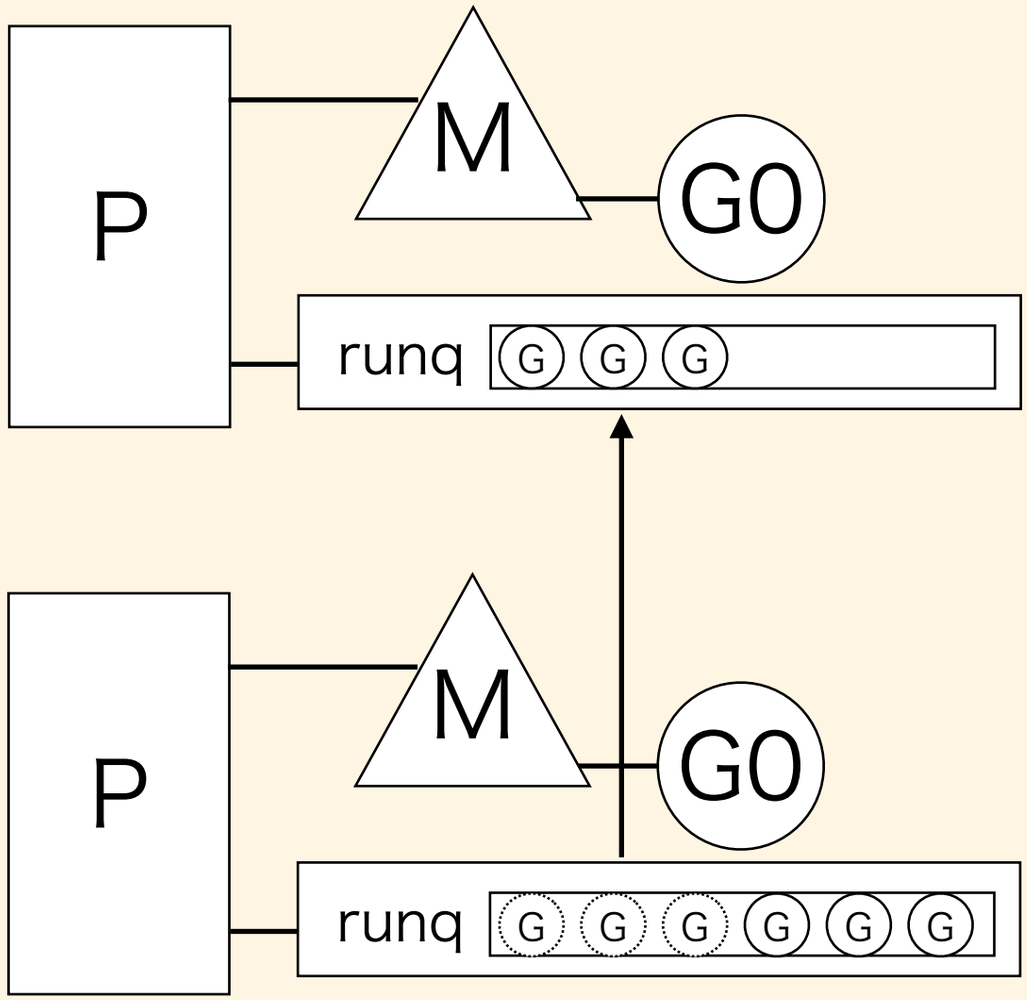




stealWork/runqsteal

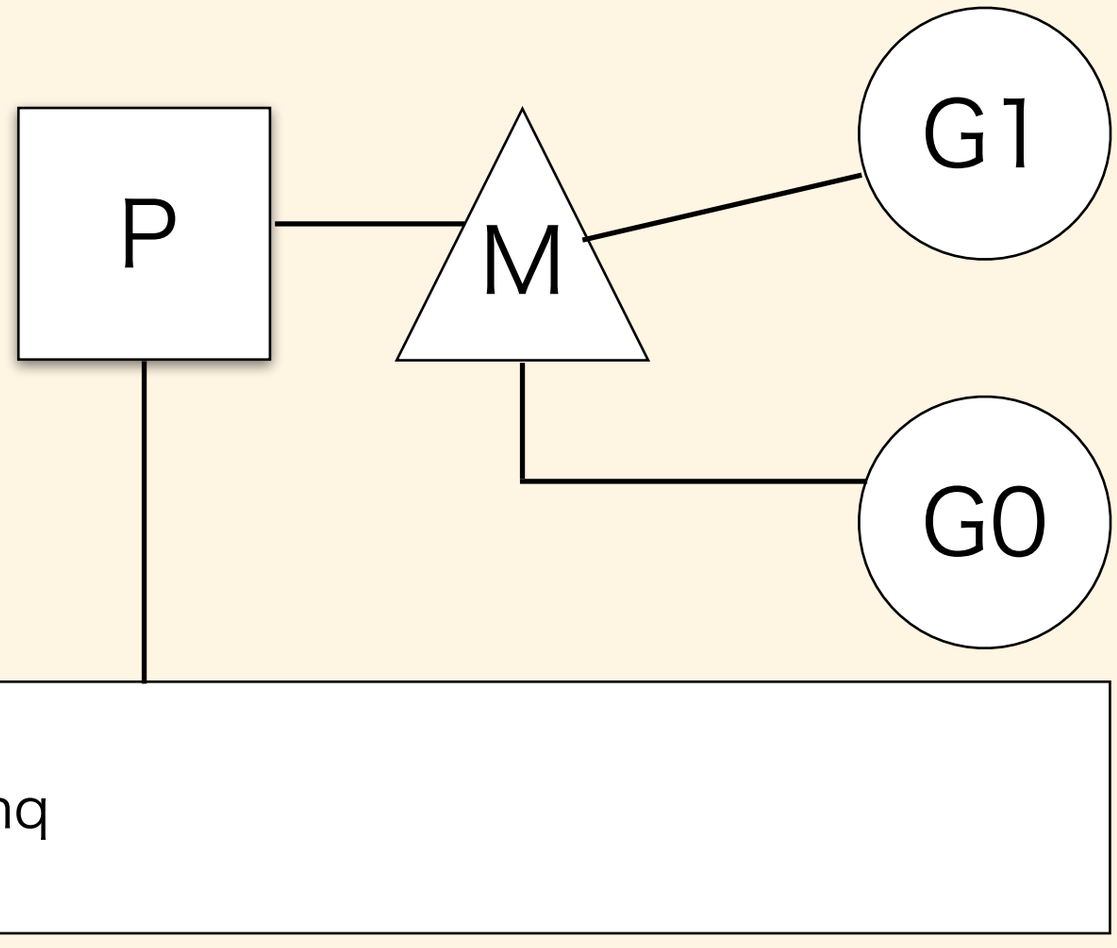
```go
func stealWork(now int64) (gp *g, inheritTime bool, rnow, pollUntil int64, newWork bool) {
    pp := getg().m.p.ptr()
    const stealTries = 4
    for i := 0; i < stealTries; i++ {
        for enum := stealOrder.start(fastrand()); !enum.done(); enum.next() {
            // Don't bother to attempt to steal if p2 is idle.
            if !idlepMask.read(enum.position()) {
                if gp := runqsteal(pp, p2, stealTimersOrRunNextG); gp != nil {
                    return gp, false, now, pollUntil, ranTimer
                }
            }
        }
    }
    // No goroutines found to steal. Regardless, running a timer may have
    // made some goroutine ready that we missed. Indicate the next timer to
    // wait for.
    return nil, false, now, pollUntil, ranTimer
}
```

```go
func runqsteal(_p_, p2 *p, stealRunNextG bool) *g {
    t := _p_.runqtail
    n := runqgrab(p2, &_p_.runq, t, stealRunNextG)
    if n == 0 {
        return nil
    }
    n--
    gp := _p_.runq[(t+n)%uint32(len(_p_.runq))].ptr()
    if n == 0 {
        return gp
    }
    h := atomic.LoadAcq(&_p_.runqhead) // load-acquire, synchronize with consumers
    if t-h+n >= uint32(len(_p_.runq)) {
        throw("runqsteal: runq overflow")
    }
    atomic.StoreRel(&_p_.runqtail, t+n) // store-release, makes the item available for consumption
    return gp
}
```



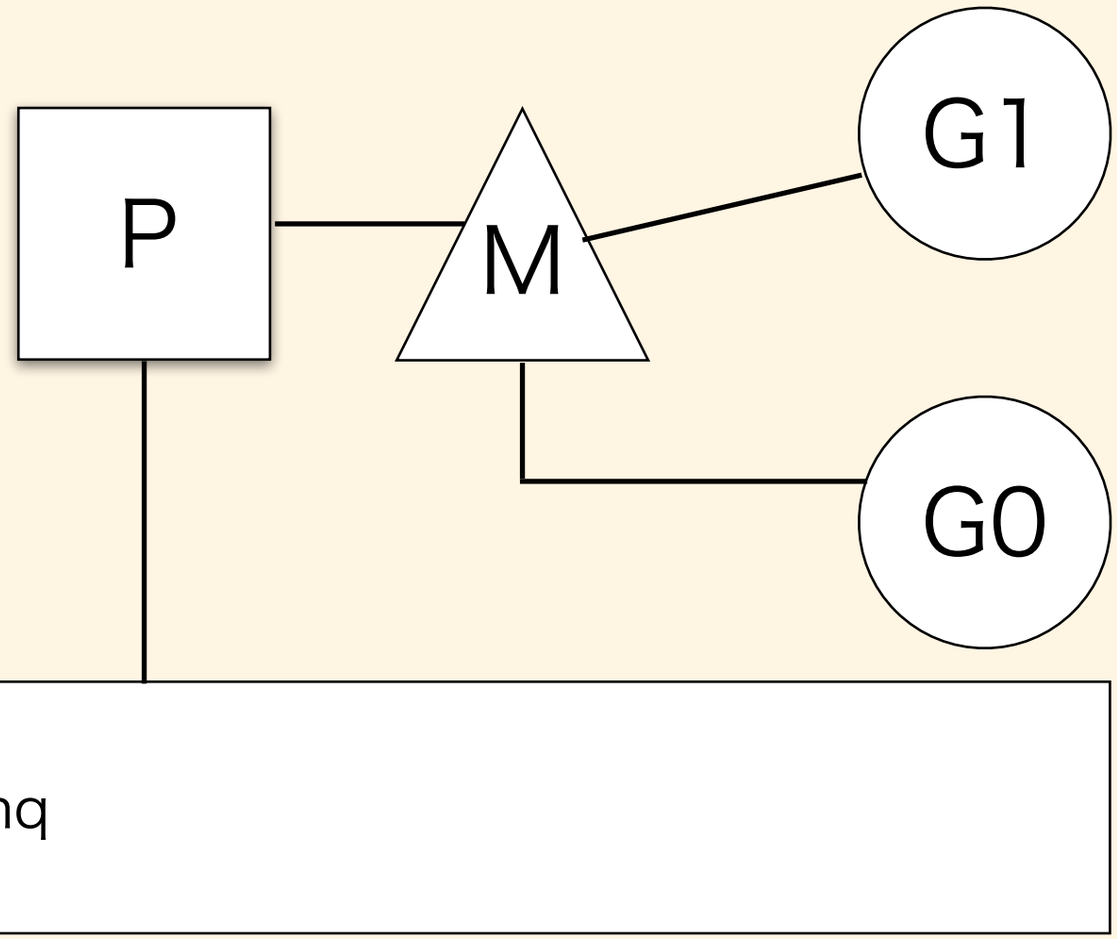
execute

```go
func execute(gp *g, inheritTime bool) {
    _g_ := getg()

    // Assign gp.m before entering _Grunning so running Gs have an
    // M.
    _g_.m.curg = gp
    gp.m = _g_.m
    casgstatus(gp, _Grunnable, _Grunning)
    // GoアセンブリにschedにあるSP、PCなどを渡してg(G)を実行する
    gogo(&gp.sched)
}
```



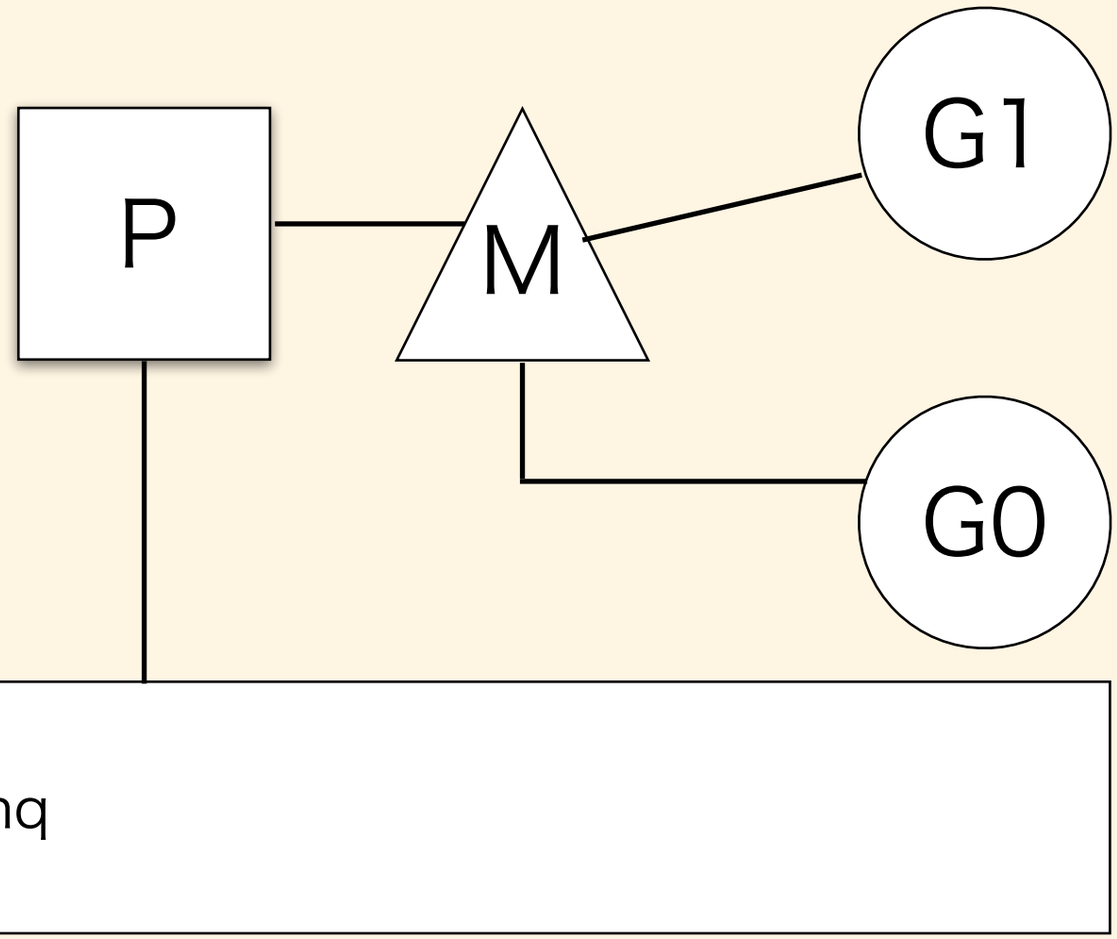
goexit

```go
// goexit is the return stub at the top of every goroutine call stack.
// Each goroutine stack is constructed as if goexit called the
// goroutine's entry point function, so that when the entry point
// function returns, it will return to goexit, which will call goexit1
// to perform the actual exit.
//
// This function must never be called directly. Call goexit1 instead.
// gentraceback assumes that goexit terminates the stack. A direct
// call on the stack will cause gentraceback to stop walking the stack
// prematurely and if there is leftover state it may panic.
func goexit(neverCallThisFunction)
```



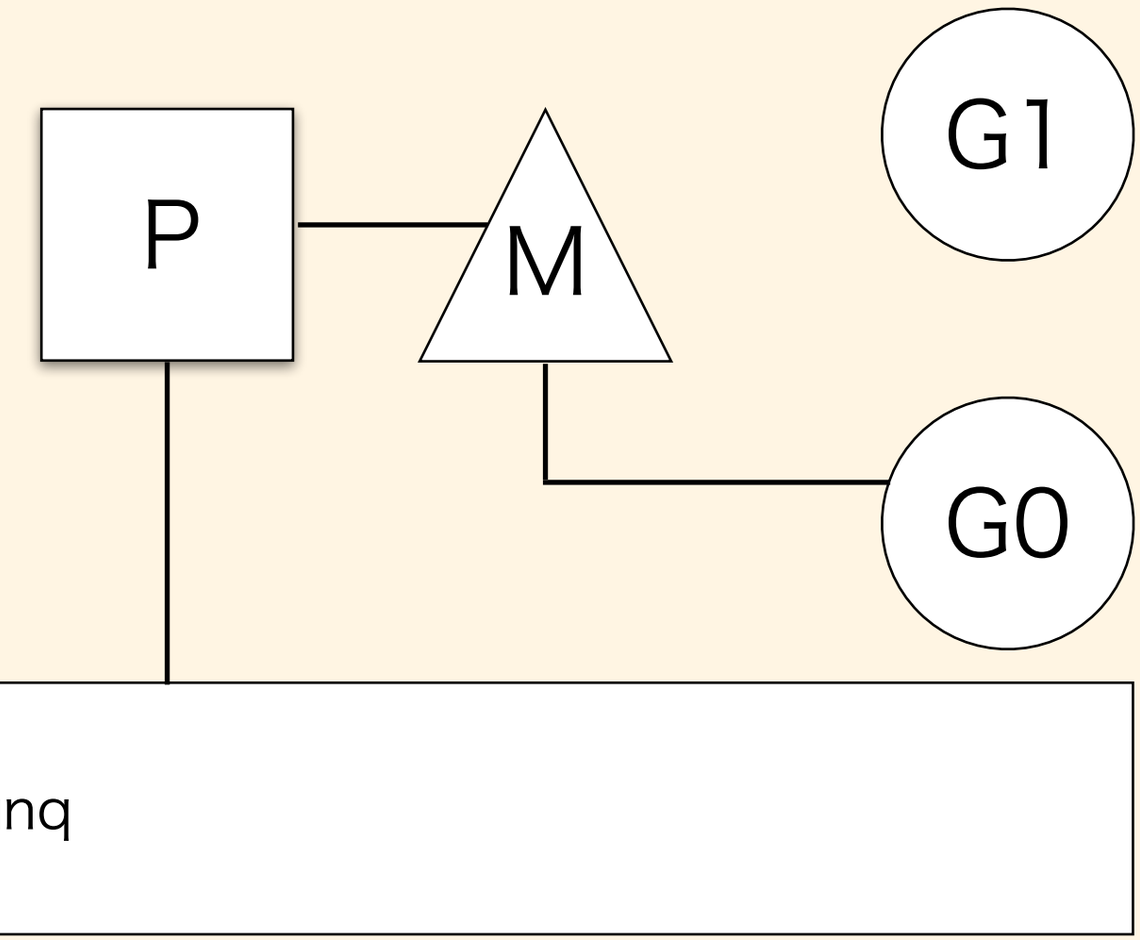
goexit1

```go
// Finishes execution of the current goroutine.
func goexit1() {
    if raceenabled {
        racegoend()
    }
    if trace.enabled {
        traceGoEnd()
    }
    mcall(goexit0)
}
```



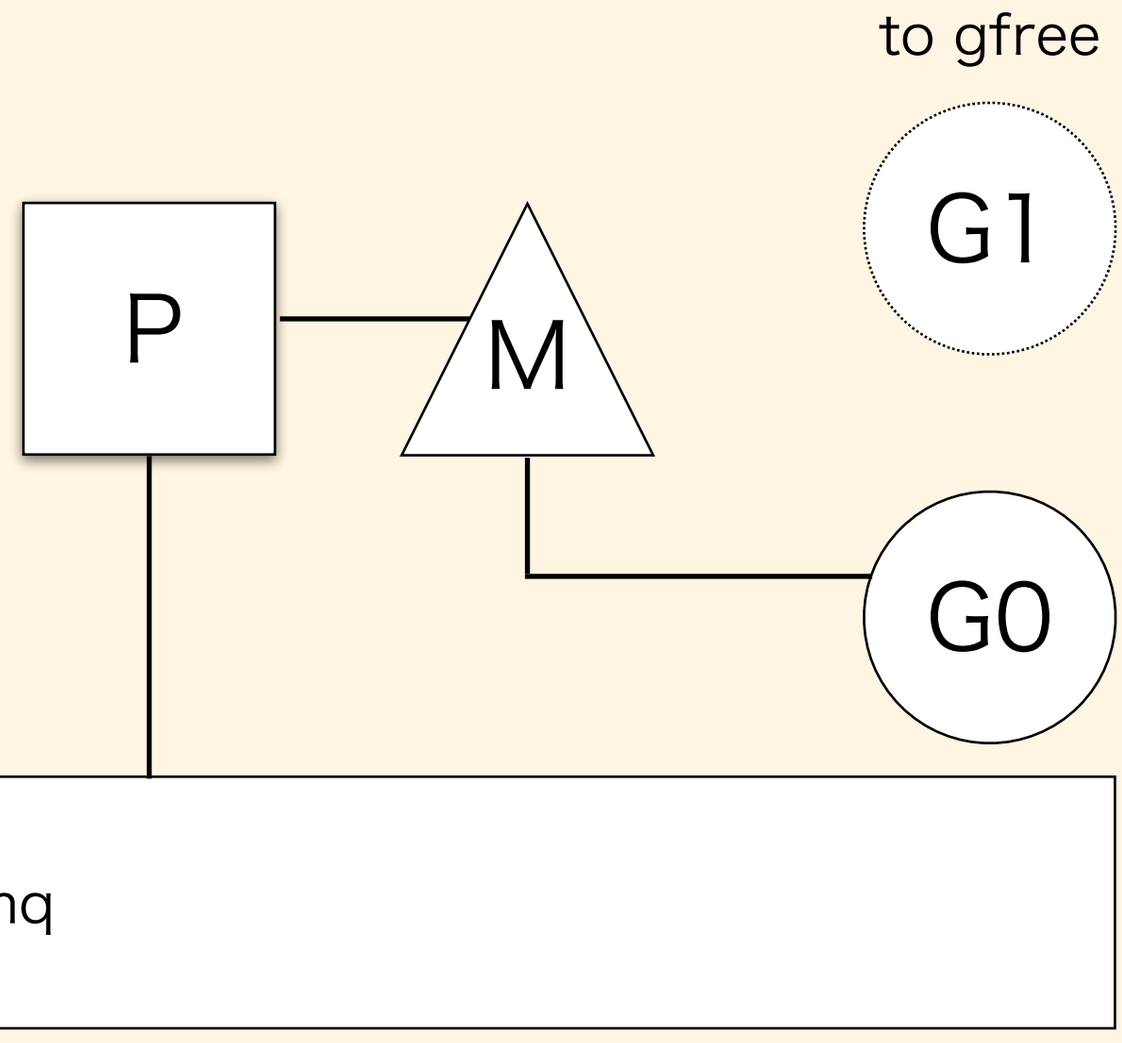
mcall

```go
// mcall switches from the g to the g0 stack and invokes fn(g),
// where g is the goroutine that made the call.
// mcall saves g's current PC/SP in g->sched so that it can be restored later.
// It is up to fn to arrange for that later execution, typically by recording
// g in a data structure, causing something to call ready(g) later.
// 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.
//
// mcall can only be called from g stacks (not g0, not gsignal).
//
// This must NOT be go:noescape: if fn is a stack-allocated closure,
// fn puts g on a run queue, and g executes before fn returns, the
// closure will be invalidated while it is still executing.
func mcall(fn func(*g))
```



goexit0

```go
// goexit continuation on g0.
func goexit0(gp *g) {
    _g_ := getg()

    // g(G)を初期化
    casgstatus(gp, _Grunning, _Gdead)
    gp.m = nil
    locked := gp.lockedm != 0
    gp.lockedm = 0
    _g_.m.lockedg = 0
    gp.preemptStop = false
    gp.paniconfault = false
    gp._defer = nil // should be true already but just in case.
    gp._panic = nil // non-nil for Goexit during panic. points at stack-allocated data.
    gp.writebuf = nil
    gp.waitreason = 0
    gp.param = nil
    gp.labels = nil
    gp.timer = nil

    // localのgfreeにg(G)を戻す。gree listに空きがないならglobalへ
    gfput(_g_.m.p.ptr(), gp)

    // g0を用いてスケジューリングし直す
    schedule()
}
```



サンプルコード

```go
func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    re := regexp.MustCompile(`^(\S.+)\.(\S.+)$`)
    go func() {
        var skip int
        for {
            pc, _, _, ok := runtime.Caller(skip)
            if !ok {
                break
            }
            fn := runtime.FuncForPC(pc)
            _fn := re.FindStringSubmatch(fn.Name())
            fmt.Printf("%d: %s.%s\n", skip, _fn[1], _fn[2])
            skip++
        }

        wg.Done()
    }()
    wg.Wait()
}
```

```
0: main.main.func1
1: runtime.goexit
```

https://play.golang.org/p/wWPrkCxnH3f

まとめ

- Goは独自のスケジューラーでgoroutineを制御している
- 各Pがrunqを持つことでプログラムの効率が高まる
- 複雑な実装のおかげで簡単に並行処理を実装できる

~Tank you for watching~