# *After go func(): Goroutines Through a Beginner's Eye*

# Vaibhav Gupta

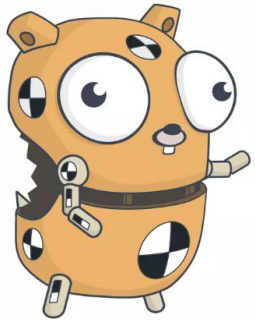
@97vaibhav

- Indian 🇮🇳
- Backend engineer@Qenest Holdings
- A Three-year-old Gopher
- Second Time Speaker@GoConference Tokyo

# *After go func(): Goroutines Through a Beginner's Eye*

# *Outline*

- *Motivation*
- *Goroutines & Go Scheduler Model*
- *Scheduler Internals (Fairness ,Preemption Work Stealing)*
- *Visualization*
- *Beginner Pitfalls & My Learnings*
- *Conclusion*
- *References*
- *Q/A*

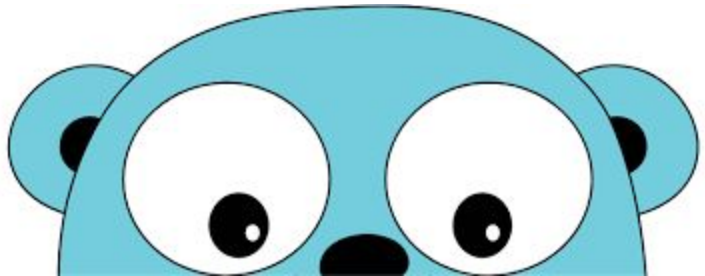


# *Motivation*

- *One keyword, massive power (go func())*
  *1つのキーワードで、ものすごい力 (go func())*

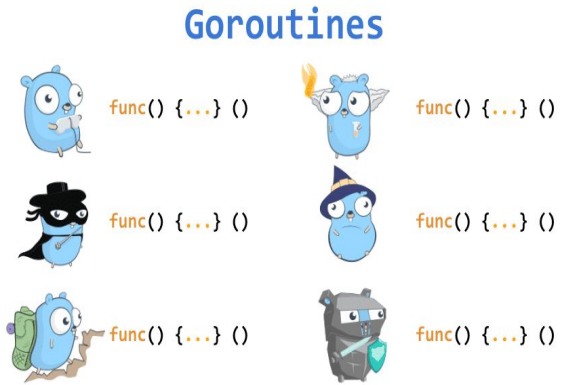
- *From “it works” to “I understand why”*
  *「なぜか動く」から「理解して動かす」へ*

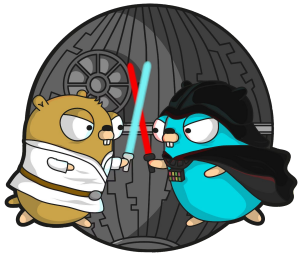
- *Today’s goal: a clear mental model*
  *今日の目標：頭の中にクリアなメンタルモデルを描く*



# Goroutine は何?

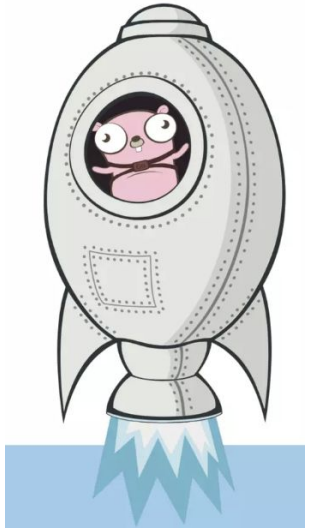
*Goroutines are lightweight threads which are managed by Go runtime*
*ゴルーチンとは、Go ランタイムに管理される軽量スレッド。*





| *Goroutines* | *OS Threads* |
| --- | --- |
| 1. *language-level, managed by Go runtime*<br>2. *very cheap to create 2kb*<br>3. *growable segmented stacks*<br>4. *millions possible; parallelism limited by GOMAXPROCS (P count)* | 1. *kernel-level, managed by the OS scheduler*<br>2. *expensive to create*<br>3. *fixed-size stacks*<br>4. *hundreds–thousands practical; parallelism capped by CPU cores* |

*Goroutines を 見ましょう ...*



```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    wg.Add(11)
    for i := 0; i <= 10; i++ {

        go func(i int) {
            defer wg.Done()
            fmt.Printf("Printing value of i in goroutine - %d\n", i)
        }(i)
    }
    wg.Wait()
    fmt.Println("Hello, Welcome to Goroutines")
}
```

*go run demo1.go*

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    wg.Add(11)
    for i := 0; i <= 10; i++ {

        go func(i int) {
            defer wg.Done()
            fmt.Printf("Printing value of i in goroutine  - %d\n", i)
        }(i)
    }
    wg.Wait()
    fmt.Println("Hello, Welcome to Goroutines")
}
```

*1st Run*

*go run demo1.go*

```
~/golang_projects/github.com/97vaibhav/go-conference-2025/demo1
> go run demo1.go
Printing value of i in goroutine  - 10
Printing value of i in goroutine  - 4
Printing value of i in goroutine  - 7
Printing value of i in goroutine  - 1
Printing value of i in goroutine  - 2
Printing value of i in goroutine  - 3
Printing value of i in goroutine  - 8
Printing value of i in goroutine  - 9
Printing value of i in goroutine  - 5
Printing value of i in goroutine  - 6
Printing value of i in goroutine  - 0
Hello, Welcome to Goroutines
```

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    wg.Add(11)
    for i := 0; i <= 10; i++ {

        go func(i int) {
            defer wg.Done()
            fmt.Printf("Printing value of i in goroutine  - %d\n", i)
        }(i)
    }
    wg.Wait()
    fmt.Println("Hello, Welcome to Goroutines")
}
```

*2nd Run*

*go run demo1.go*

```
~/golang_projects/github.com/97vaibhav/go-conference-2025/demo1
> go run demo1.go
Printing value of i in goroutine  - 1
Printing value of i in goroutine  - 10
Printing value of i in goroutine  - 0
Printing value of i in goroutine  - 3
Printing value of i in goroutine  - 4
Printing value of i in goroutine  - 5
Printing value of i in goroutine  - 6
Printing value of i in goroutine  - 7
Printing value of i in goroutine  - 8
Printing value of i in goroutine  - 9
Printing value of i in goroutine  - 2
Hello, Welcome to Goroutines
```

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    wg.Add(11)
    for i := 0; i <= 10; i++ {

        go func(i int) {
            defer wg.Done()
            fmt.Printf("Printing value of i in goroutine  - %d\n", i)
        }(i)
    }
    wg.Wait()
    fmt.Println("Hello, Welcome to Goroutines")
}
```

*3rd Run*

*go run demo1.go*

```
~/golang_projects/github.com/97vaibhav/go-conference-2025/demo1
> go run demo1.go
Printing value of i in goroutine  - 10
Printing value of i in goroutine  - 3
Printing value of i in goroutine  - 0
Printing value of i in goroutine  - 1
Printing value of i in goroutine  - 2
Printing value of i in goroutine  - 4
Printing value of i in goroutine  - 5
Printing value of i in goroutine  - 8
Printing value of i in goroutine  - 7
Printing value of i in goroutine  - 6
Printing value of i in goroutine  - 9
Hello, Welcome to Goroutines
```

- *How did 11 goroutines run concurrently? Magic?*
  *11個のgoroutineがどうやって並行して実行されたのでしょうか？魔法でしょうか？*

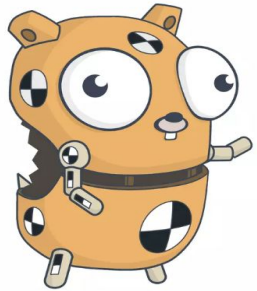
- *In What Order 11 goroutines ran?*
  *11 個の goroutine はどのような順序で実行されましたか?*

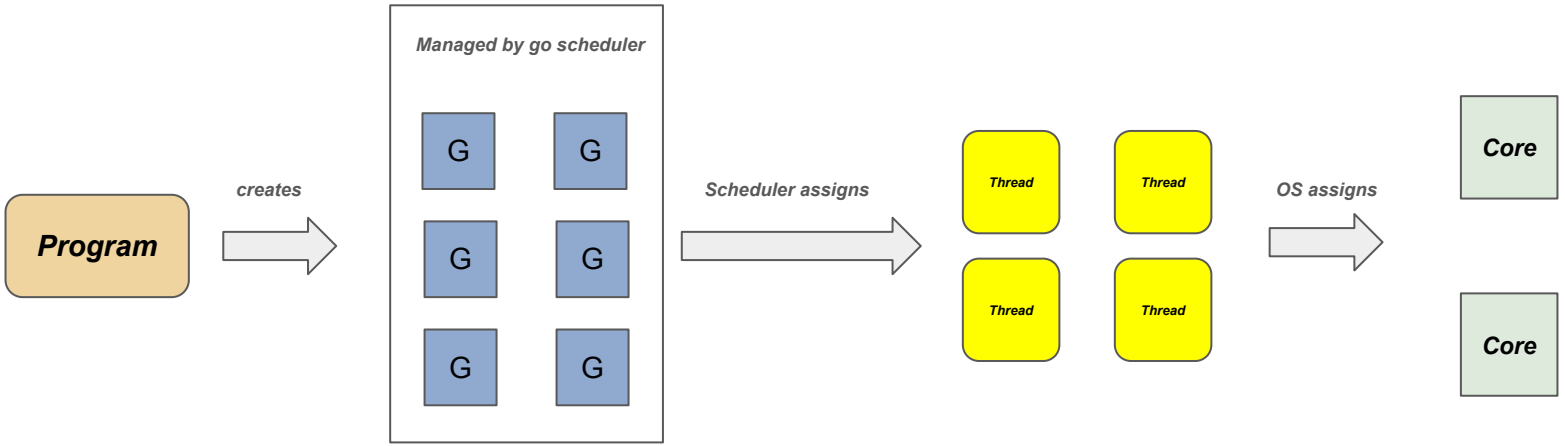
GO CONFERENCE 2025

“If I Tell you in what order goroutines run, it won’t happen” - Dr Gopher Strange.

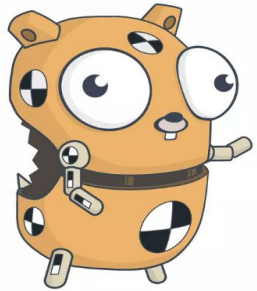
# *Go Scheduler Model*

- *We need some way to map goroutines onto os threads-* ***User Space Scheduling***
  *ゴルーチンをOSスレッドにマッピングする方法が必要です -* ***ユーザー空間スケジューリング***




Program
creates
Managed by go scheduler
G
G
G
G
G
G
Scheduler assigns
Thread
Thread
Thread
Thread
OS assigns
Core
Core

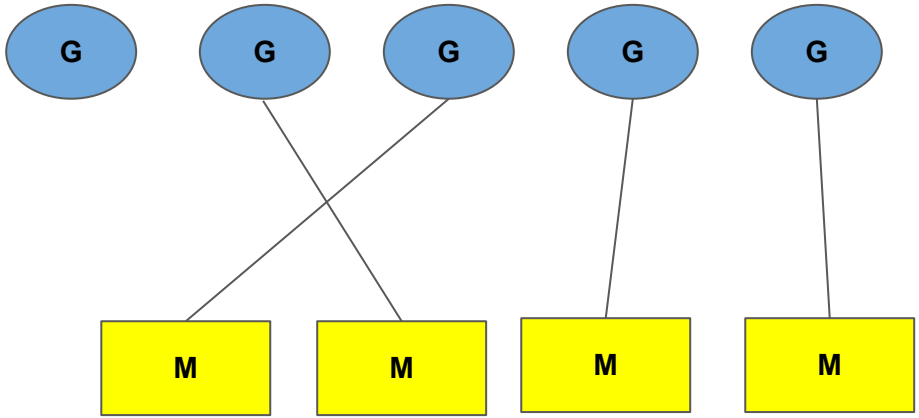
- *We need some way to map goroutines onto os threads-* ***User Space Scheduling***
  *ゴルーチンをOSスレッドにマッピングする方法が必要です -* ***ユーザー空間スケジューリング***



# M:N Scheduling

- *The no of G can be greater than number of M*
  *Goroutine(G) の数は Thread(M) より多くてよい*

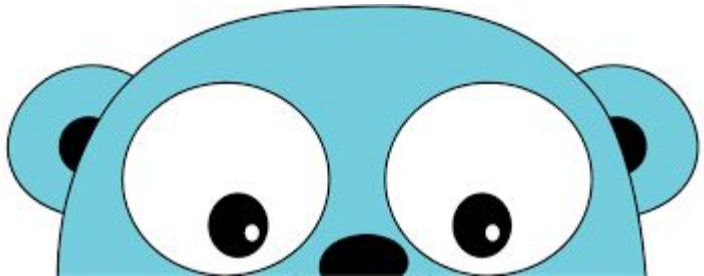
- *Go scheduler multiplexes G onto available M*
  *Go スケジューラが賢くMにGを割り当ててくれる*



# *Go scheduler Internals*

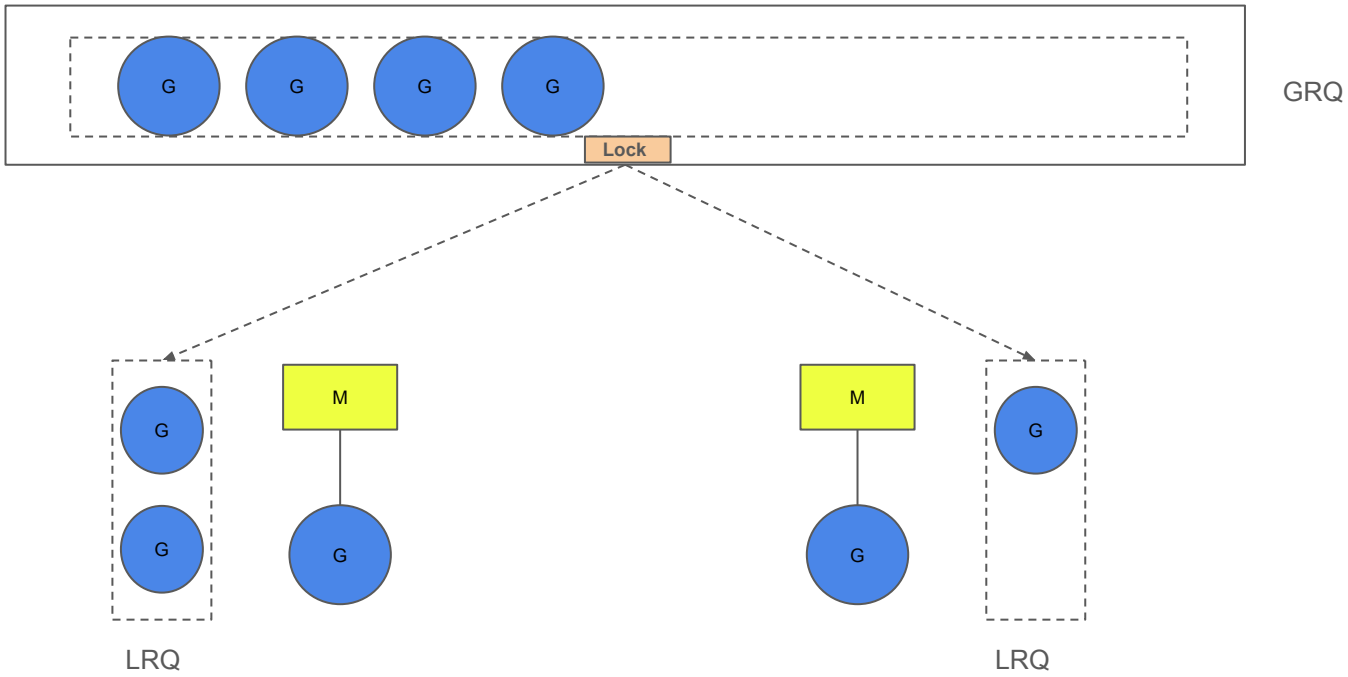
*How do we keep track of goroutine that are yet to be run or are running ?*

*まだ実行されていないゴルーチンを、どうやって管理しますか？*



Goのスケジューラには、2つの実行キューがあります

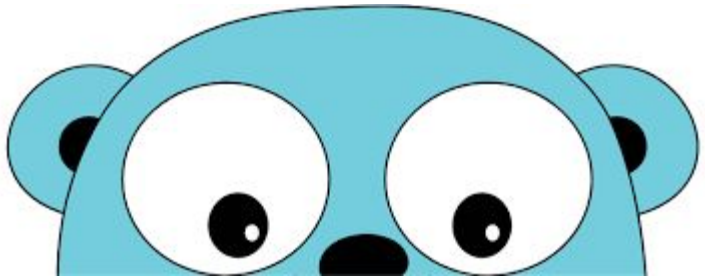
- Global Run Queue (GRQ)
- Local Run Queue (LRQ)



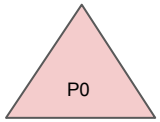

GRQ = Global Run Queue
LRQ = Local Run Queue

*But Wait there's a Problem !!*

*Long running tasks or System Calls ??* 🥲



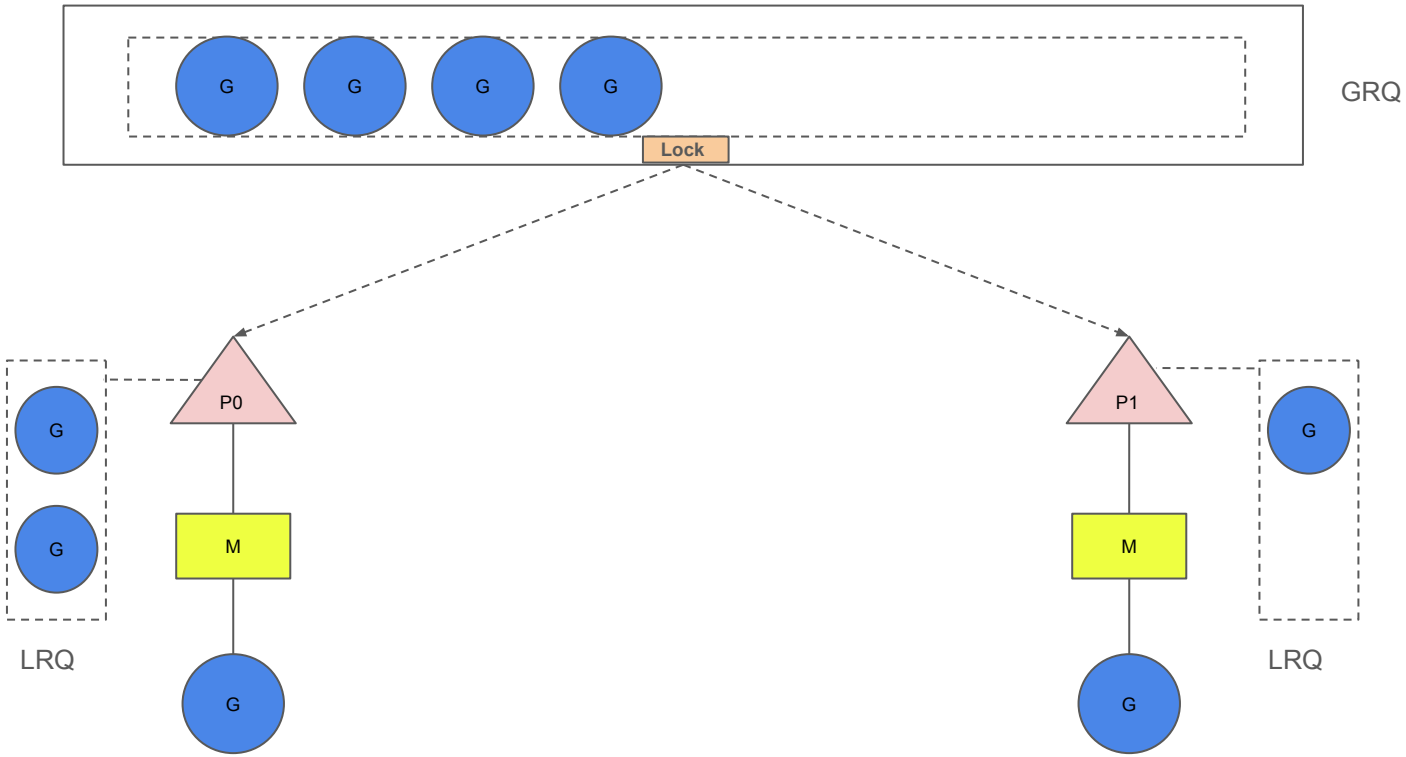
# *Processor*




Processor


System Thread


Goroutine

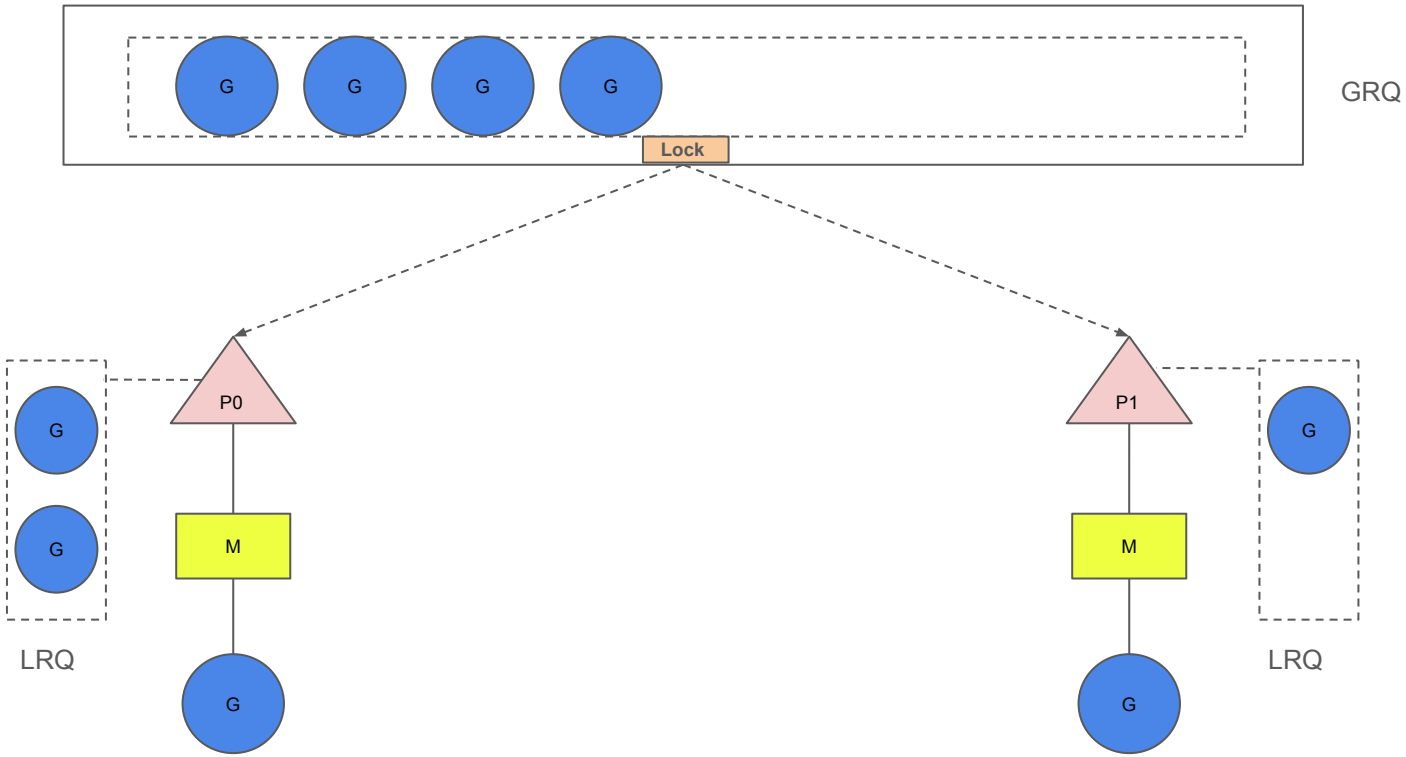



GRQ = Global Run Queue
LRQ = Local Run Queue

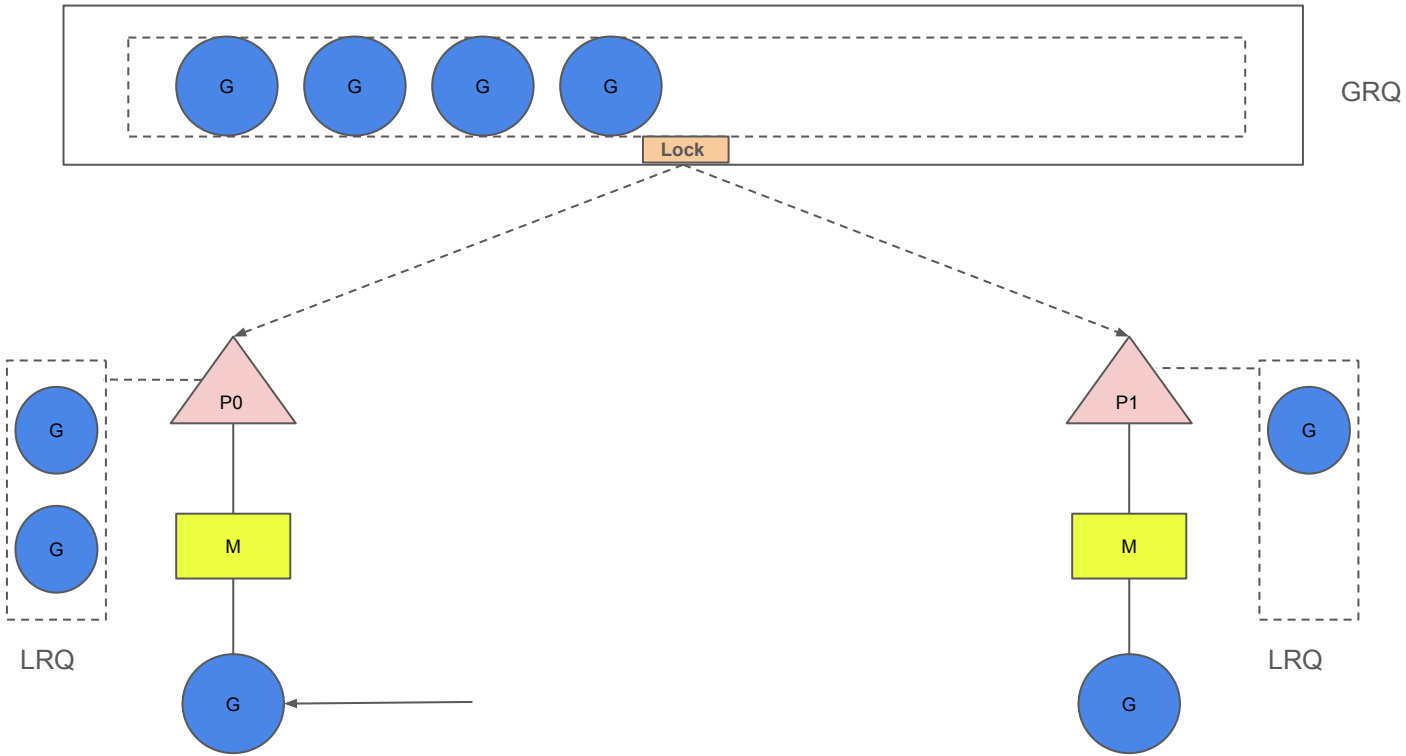
✨ プロセッサの数は最大数GOMAXPROCSです

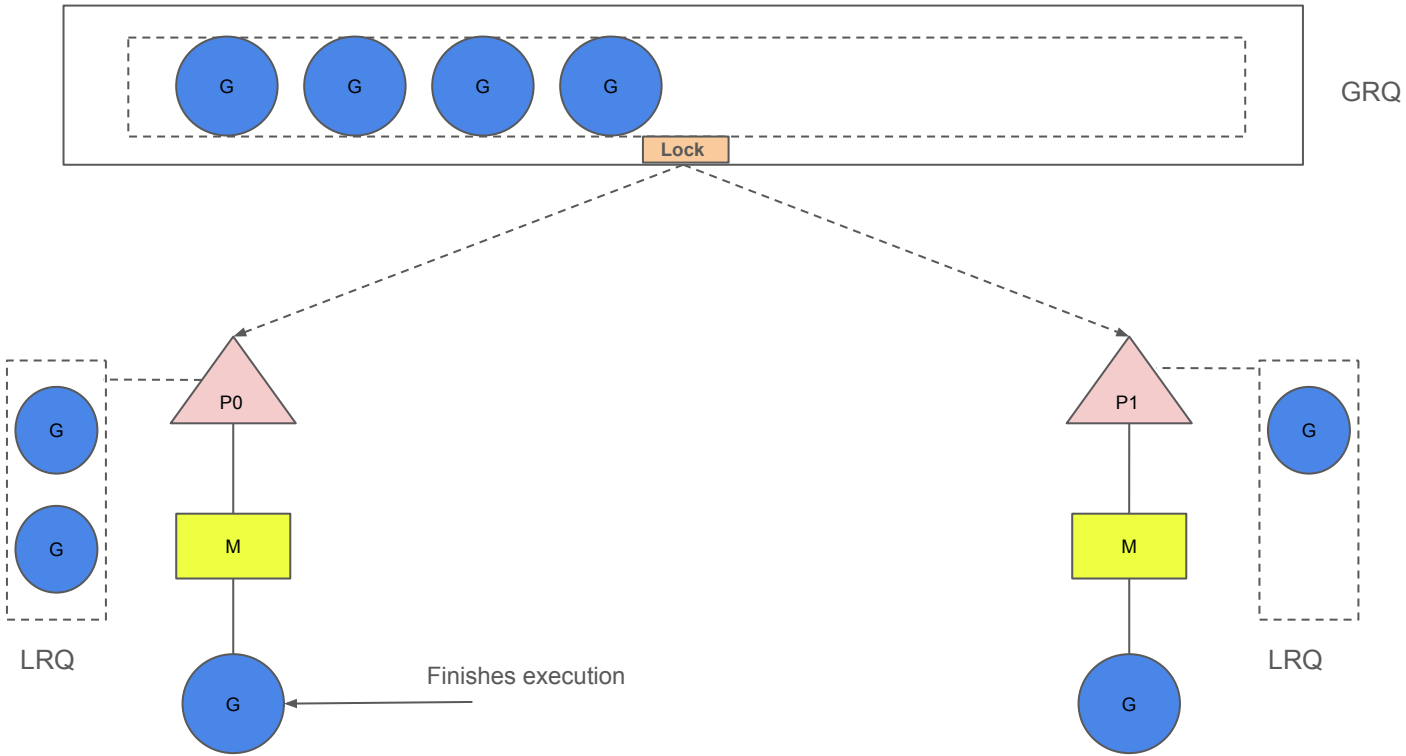
✨ *Now we have enough explanation*

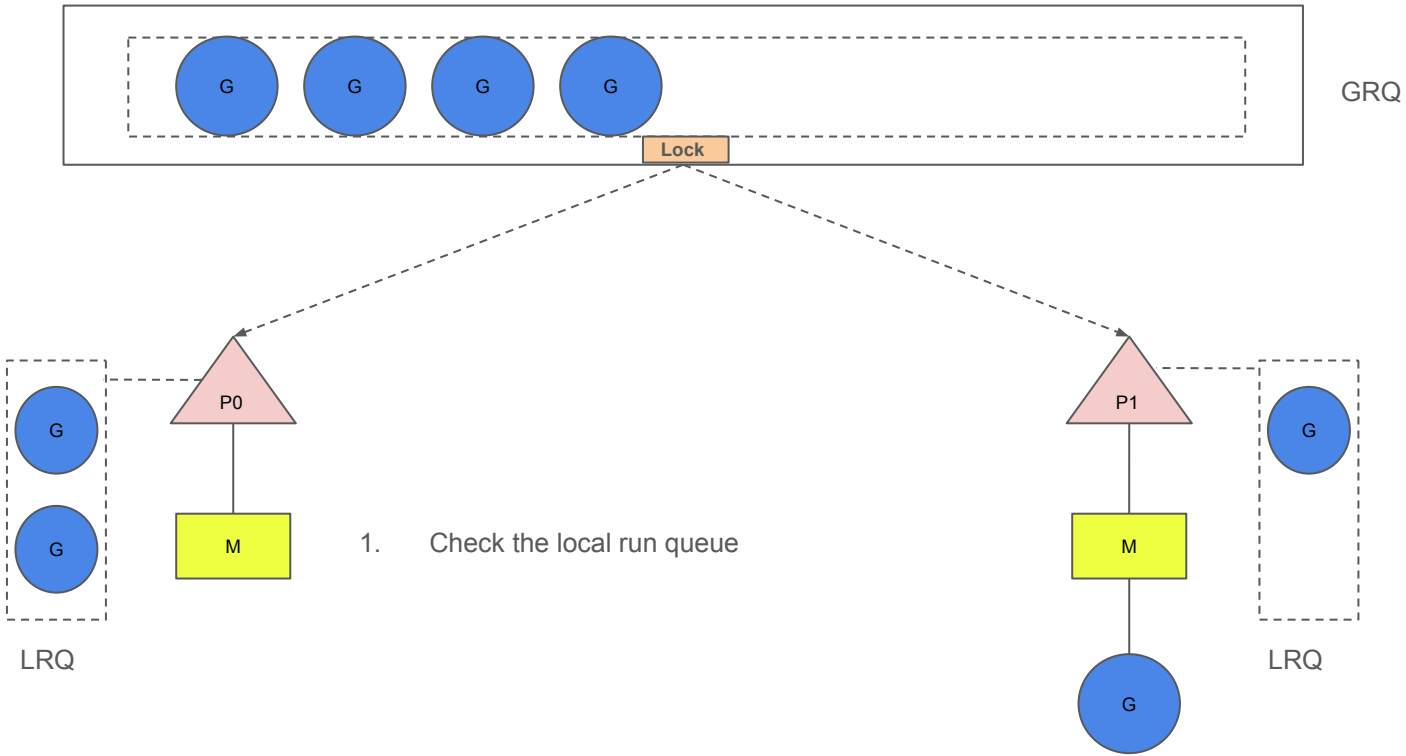
*“How do we choose which go routine to run“*

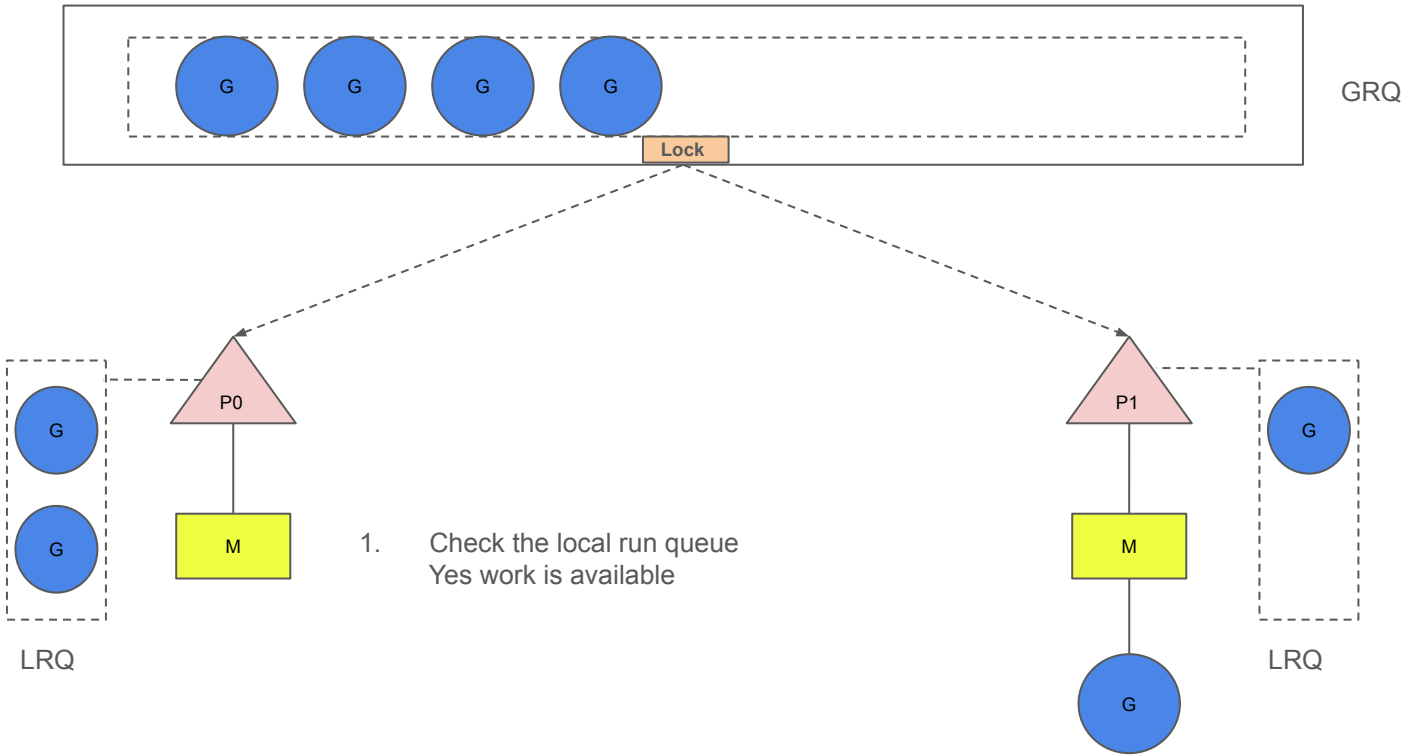


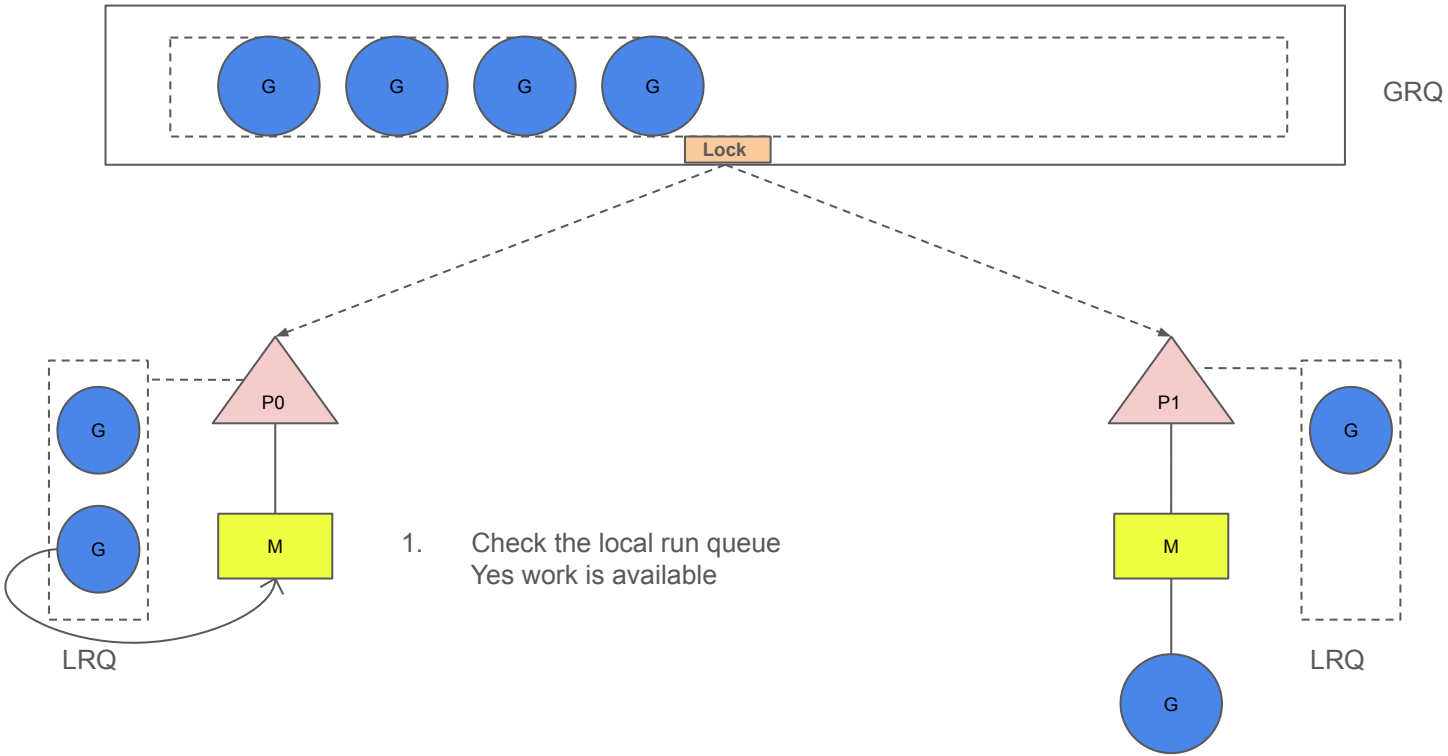

GRQ = Global Run Queue
LRQ = Local Run Queue

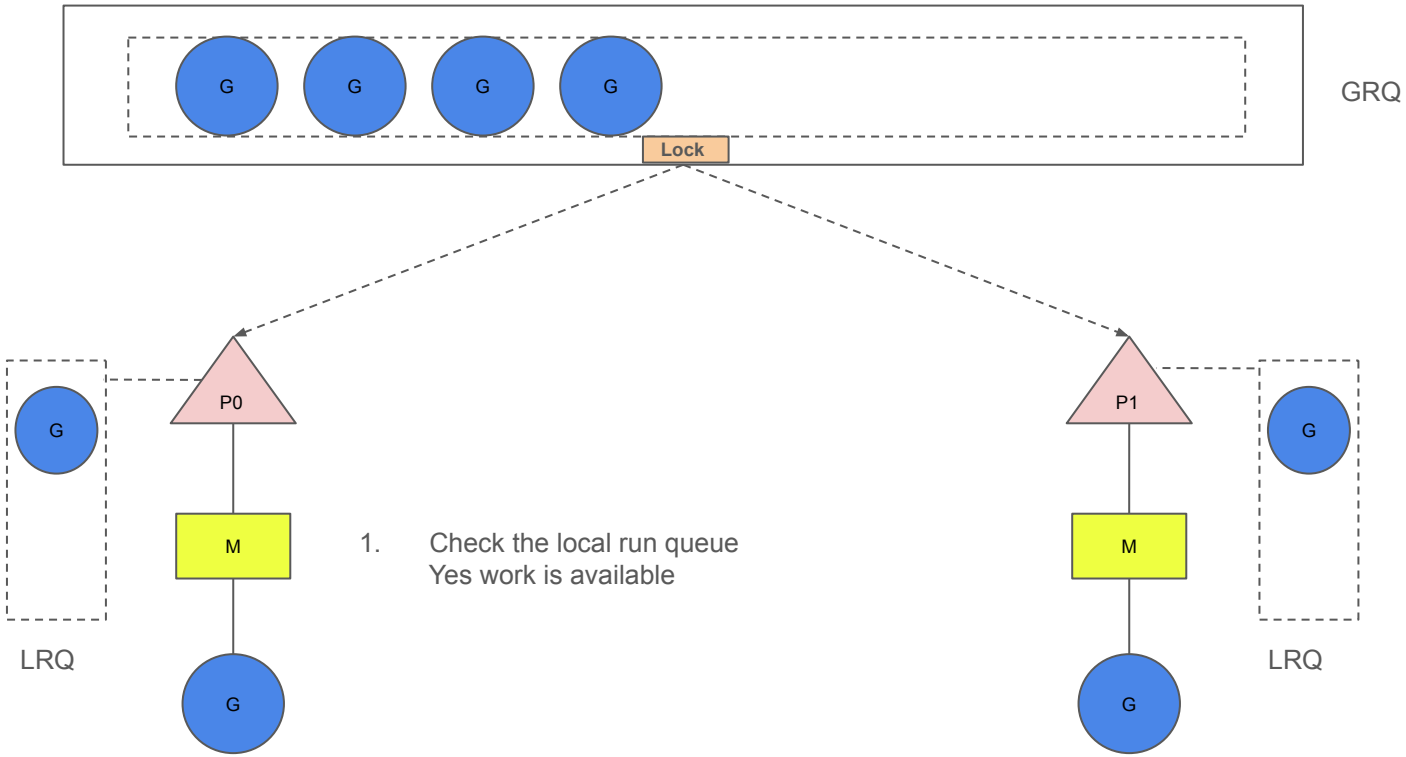

G
G
G
G
GRQ
Lock
P0
P1
G
G
M
M
LRQ
LRQ
G
G


G
G
G
G
GRQ
Lock
P0
P1
G
G
M
M
G
G
LRQ
LRQ
Finishes execution
G
G

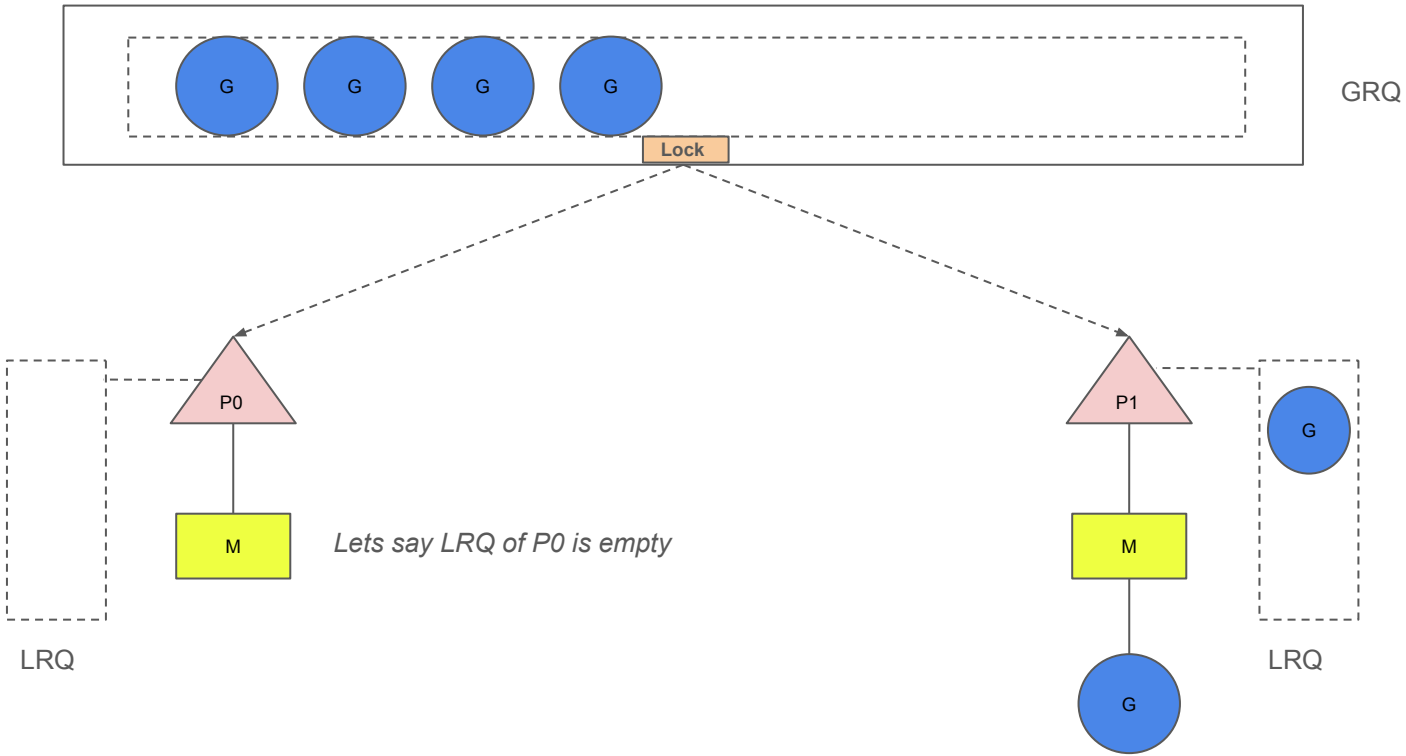

G
G
G
G
GRQ
Lock
P0
P1
G
G
M
1. Check the local run queue
M
G
G
LRQ
LRQ

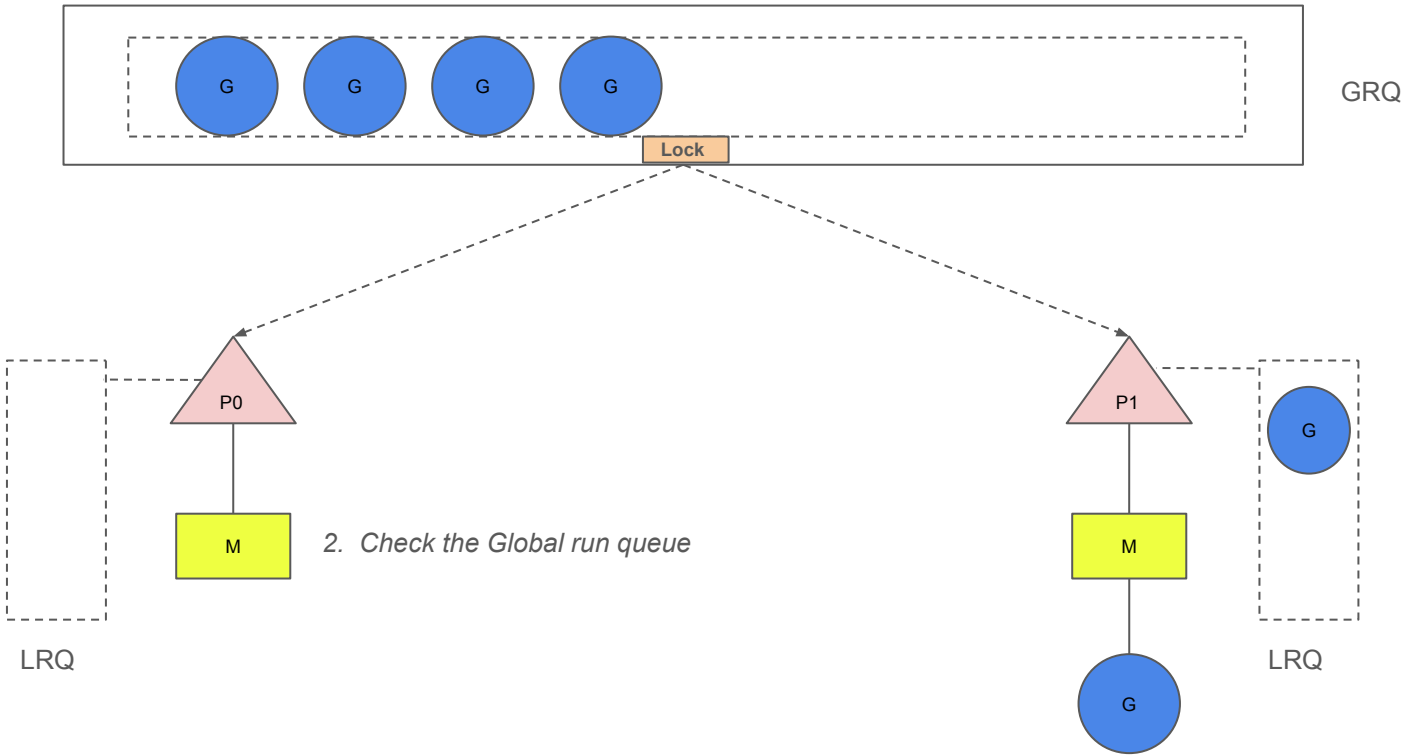

G
G
G
G
GRQ
Lock
P0
P1
G
G
M
M
G
G
LRQ
LRQ
1. Check the local run queue
Yes work is available

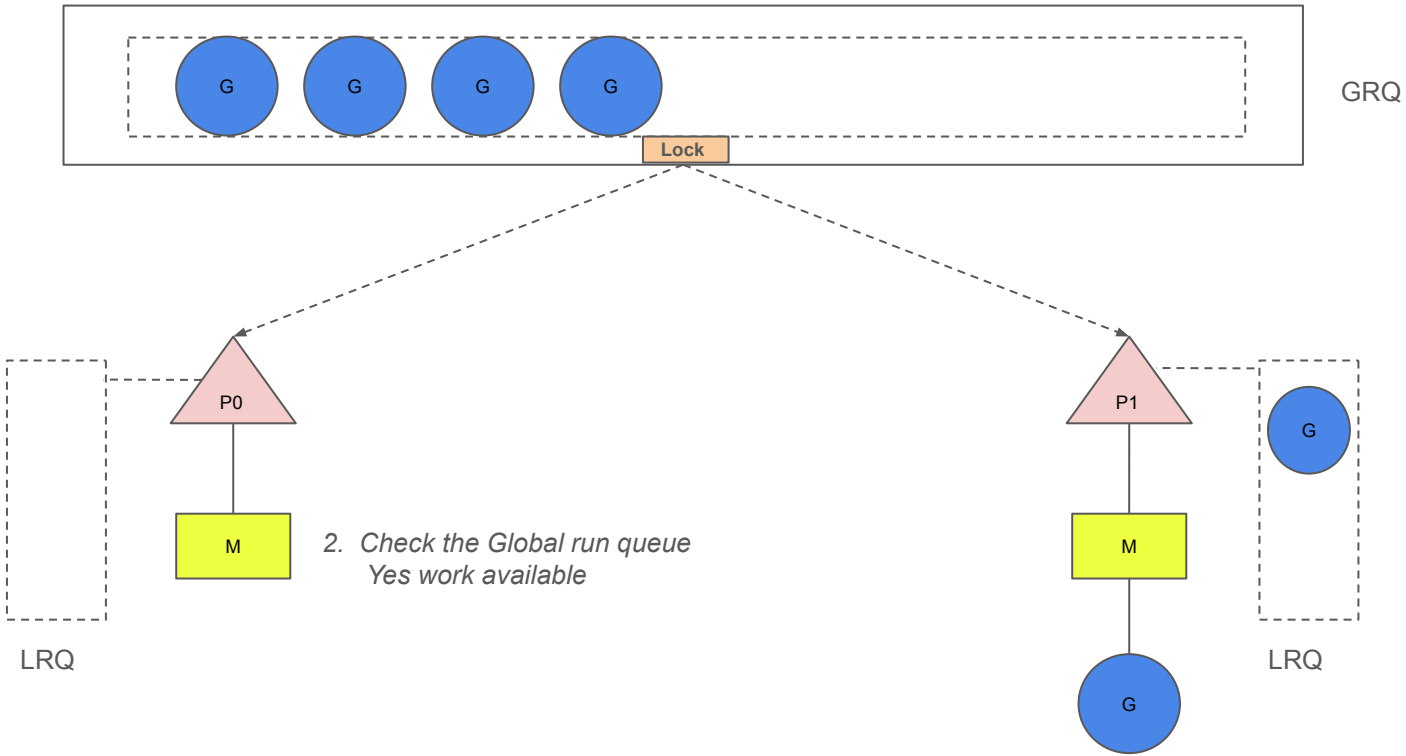

G
G
G
G
GRQ
Lock
P0
G
G
M
LRQ
1. Check the local run queue
Yes work is available
P1
M
G
G
LRQ

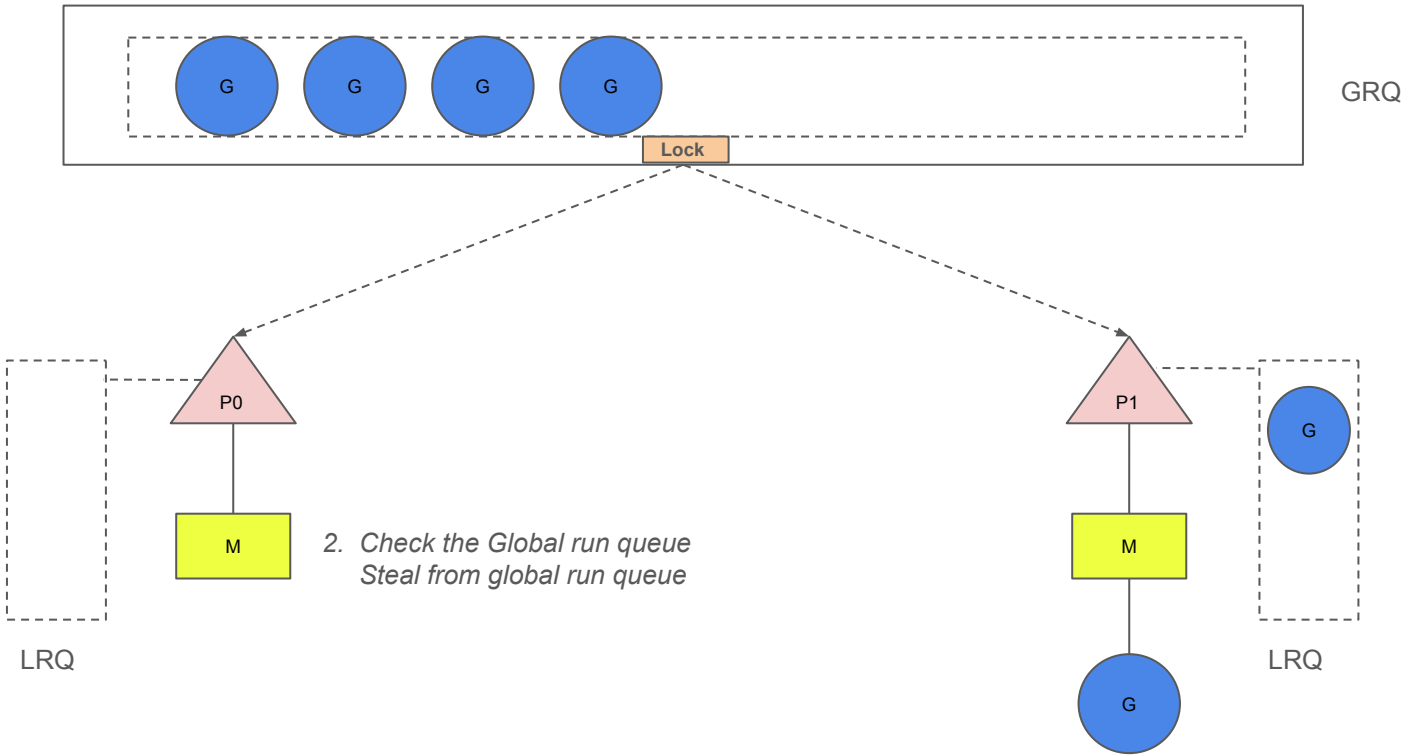

G
G
G
G
GRQ
Lock
P0
P1
G
G
M
M
1. Check the local run queue
Yes work is available
LRQ
LRQ
G
G

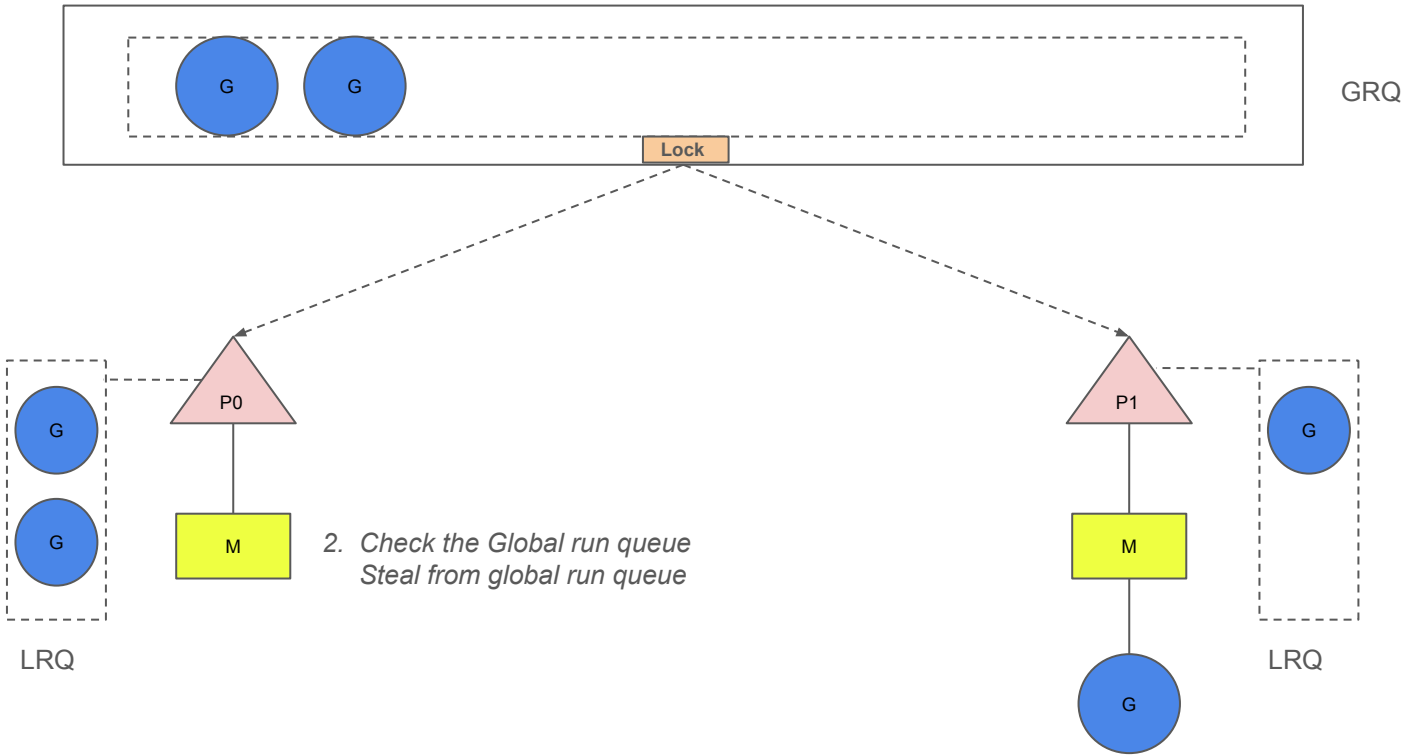
*Local Run Queue は空ですか? 😅*

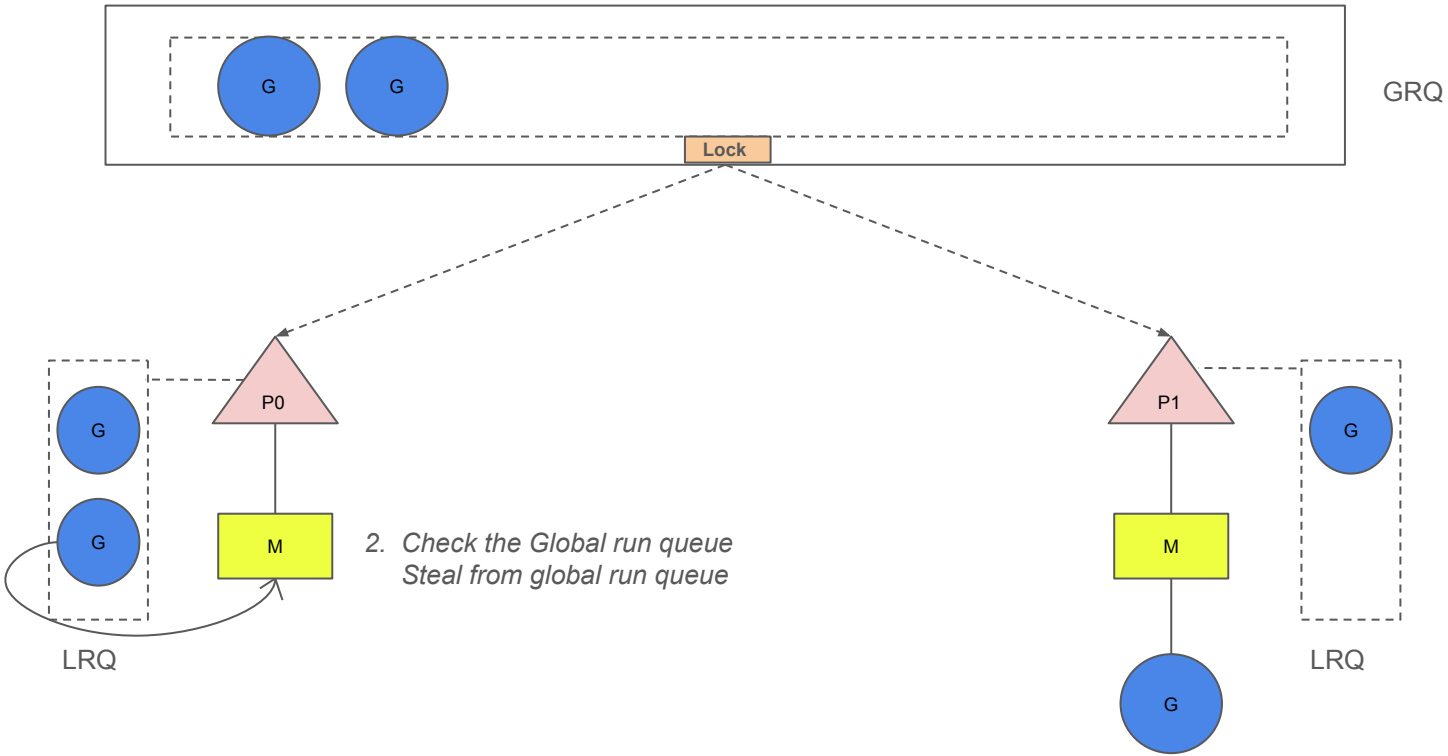

G
G
G
G
GRQ
Lock
P0
P1
G
M
Lets say LRQ of P0 is empty
M
LRQ
G
LRQ

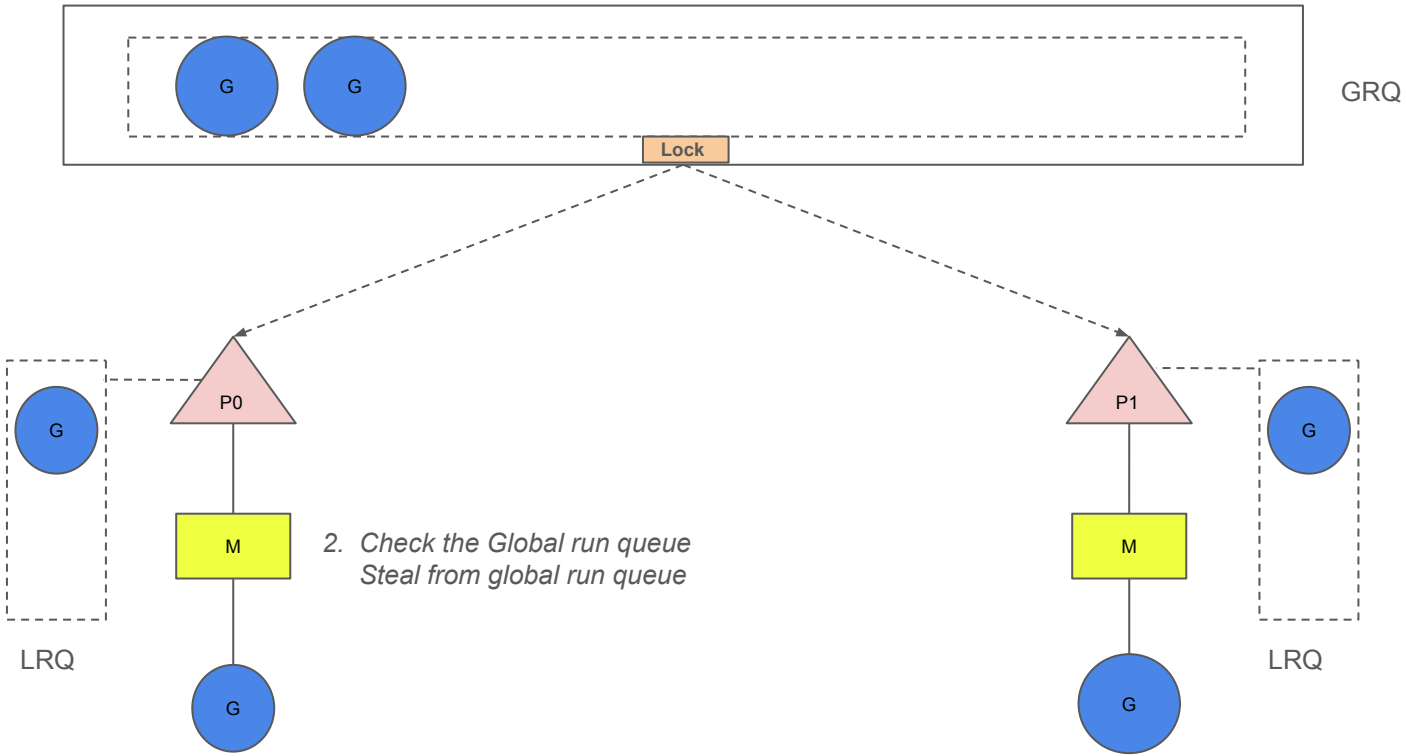

G
G
G
G
GRQ
Lock
P0
P1
G
M
2. Check the Global run queue
M
LRQ
G
LRQ


G
G
G
G
GRQ
Lock
P0
M
LRQ
2. Check the Global run queue
Yes work available
P1
G
M
LRQ
G

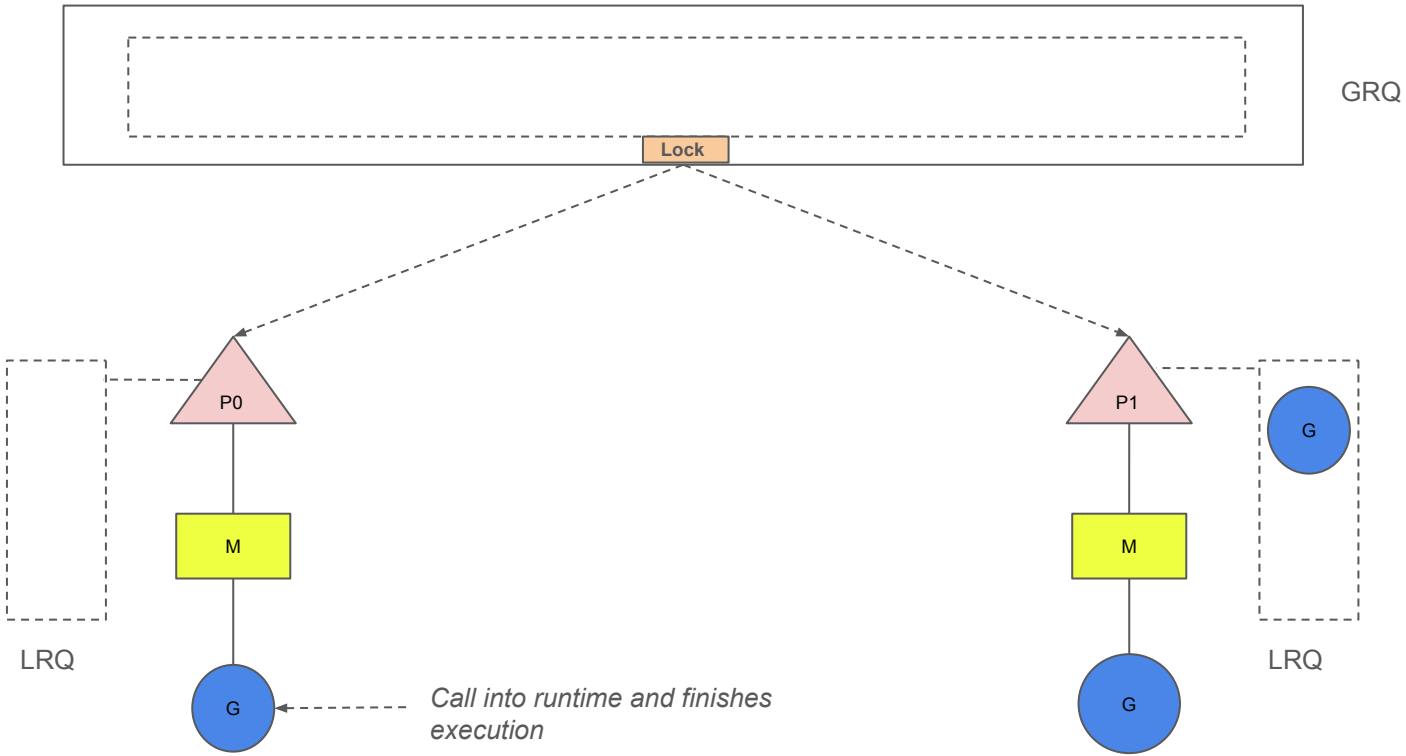

G
G
G
G
GRQ
Lock
P0
M
LRQ
2. Check the Global run queue
Steal from global run queue
P1
M
G
G
LRQ

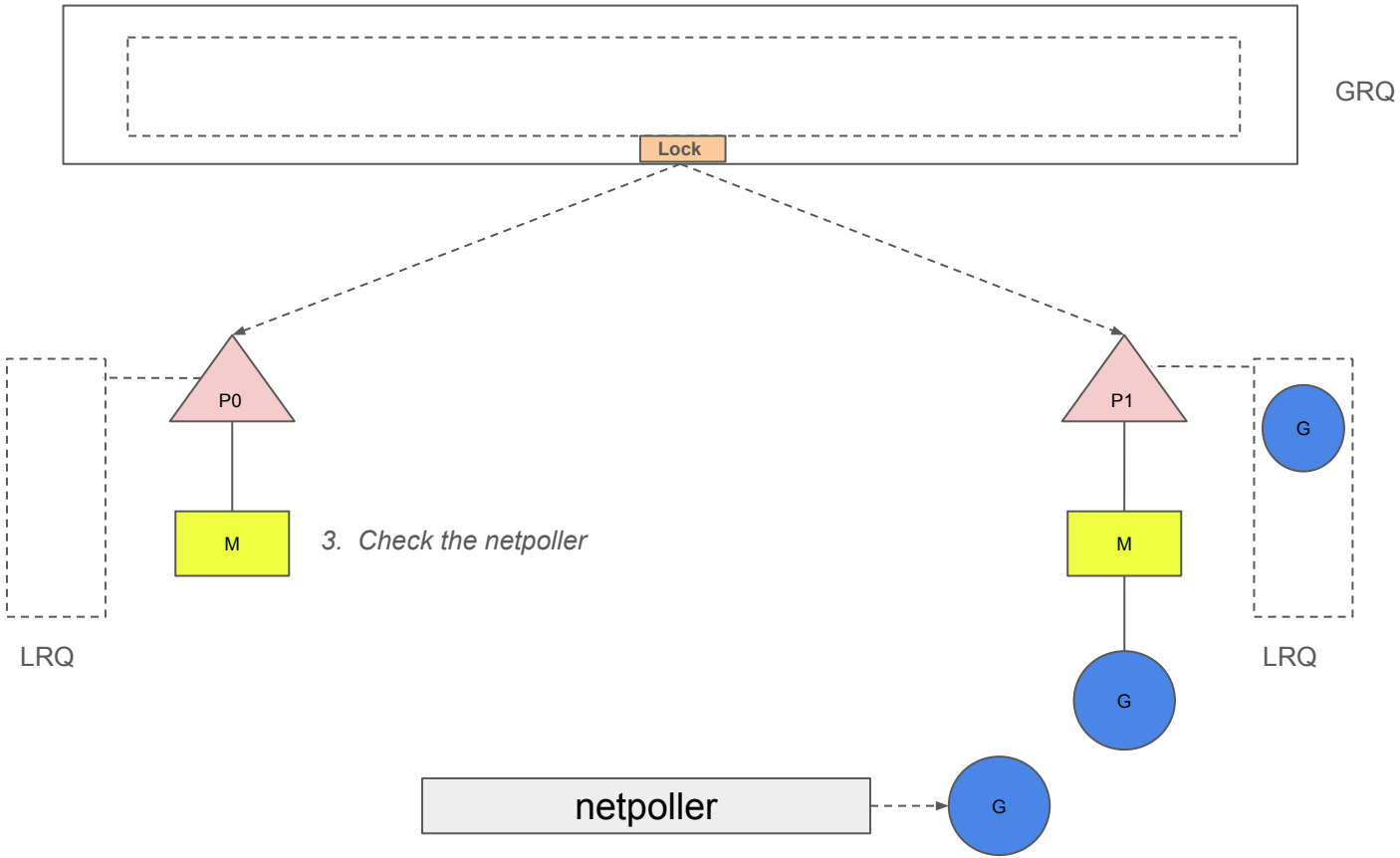

G
G
GRQ
Lock
P0
P1
G
G
M
M
G
G
LRQ
LRQ
2. Check the Global run queue
Steal from global run queue

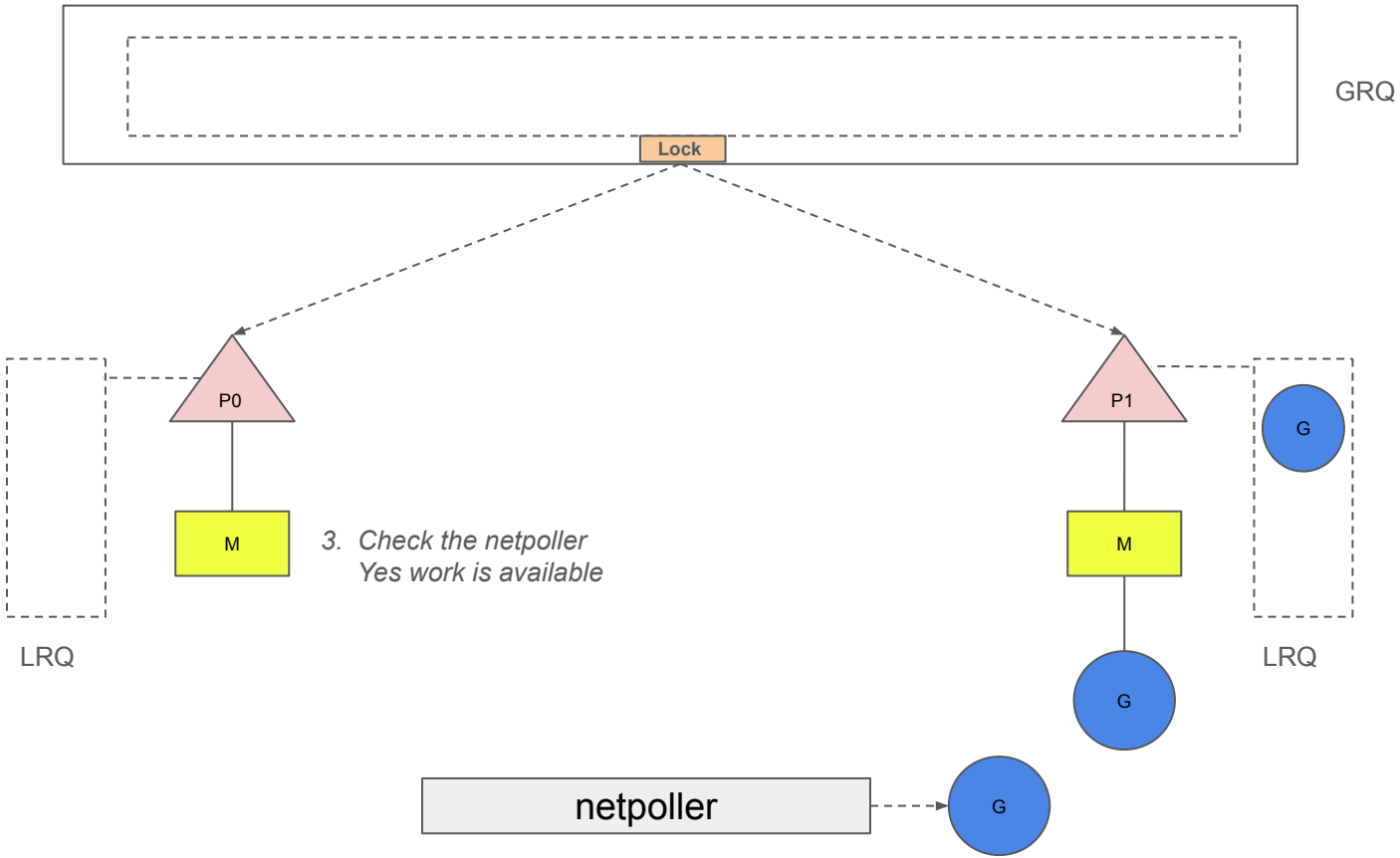

G
G
GRQ
Lock
P0
P1
G
G
M
M
G
G
2. Check the Global run queue
Steal from global run queue
LRQ
LRQ

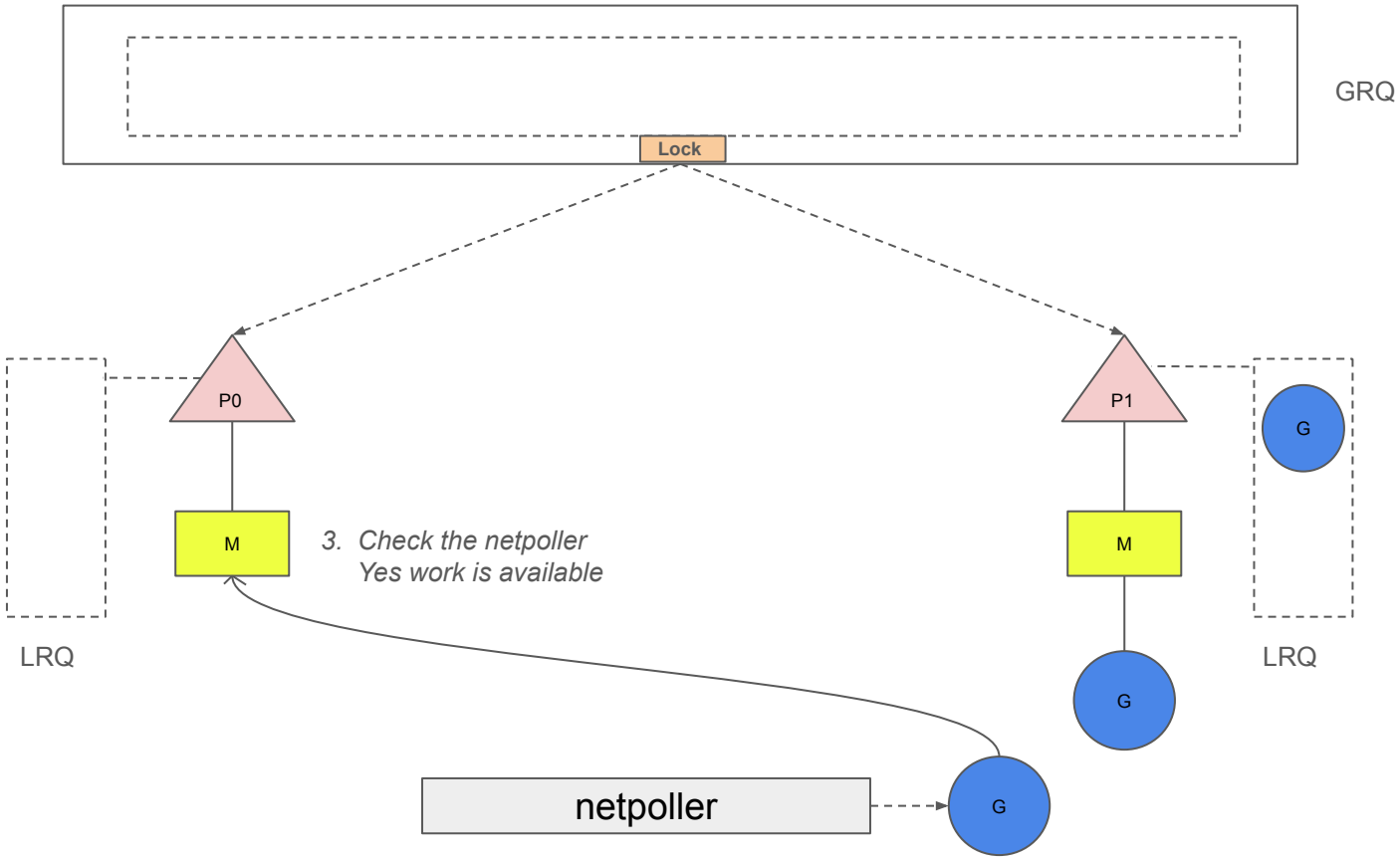

G
G
GRQ
Lock
P0
P1
G
G
M
M
2. Check the Global run queue
Steal from global run queue
LRQ
LRQ
G
G

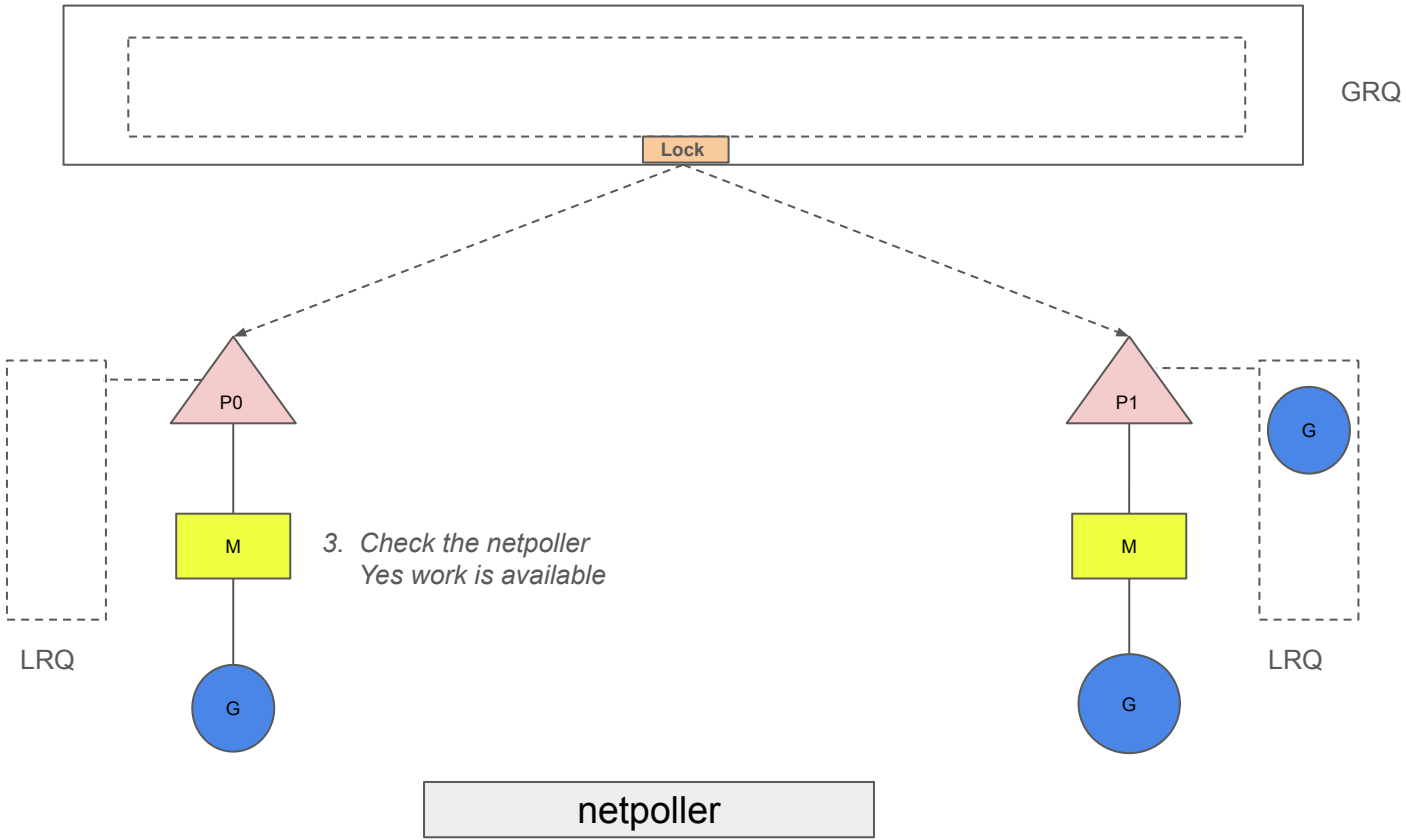
*Global Run Queue 実行キューも空ですか？🙃*


GRQ
Lock
P0
P1
M
M
G
G
G
LRQ
LRQ
Call into runtime and finishes execution


GRQ
Lock
P0
P1
G
M
3. Check the netpoller
M
LRQ
LRQ
G
netpoller
G

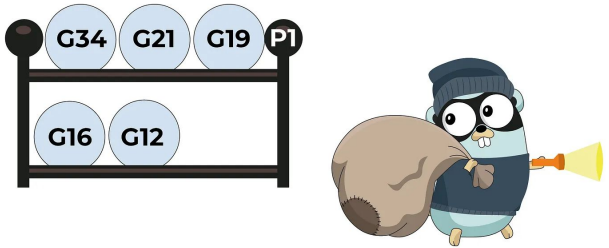

GRQ
Lock
P0
P1
G
M
3. Check the netpoller
Yes work is available
M
LRQ
LRQ
G
netpoller
G

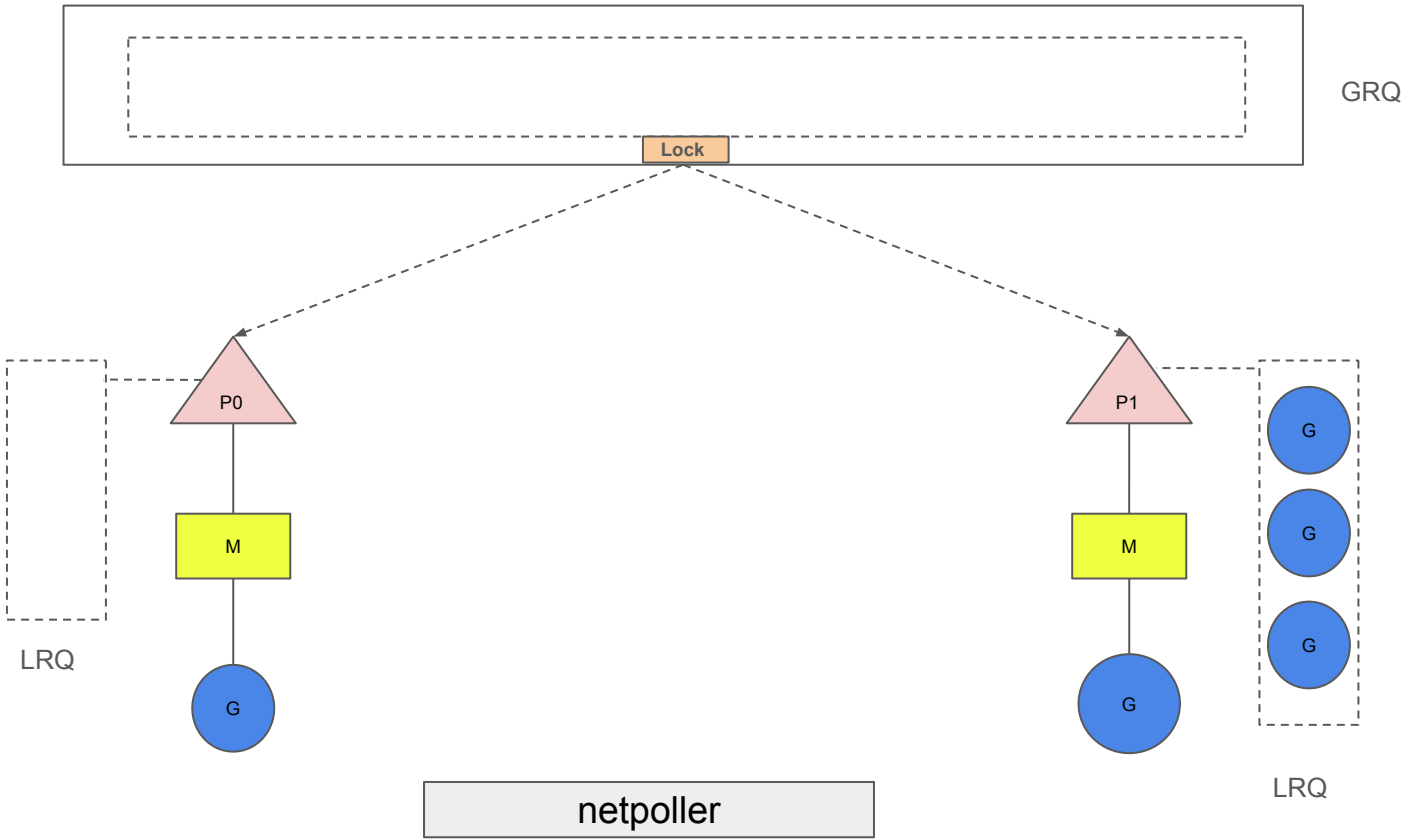

GRQ
Lock
P0
P1
G
M
3. Check the netpoller
Yes work is available
M
LRQ
LRQ
G
netpoller
G

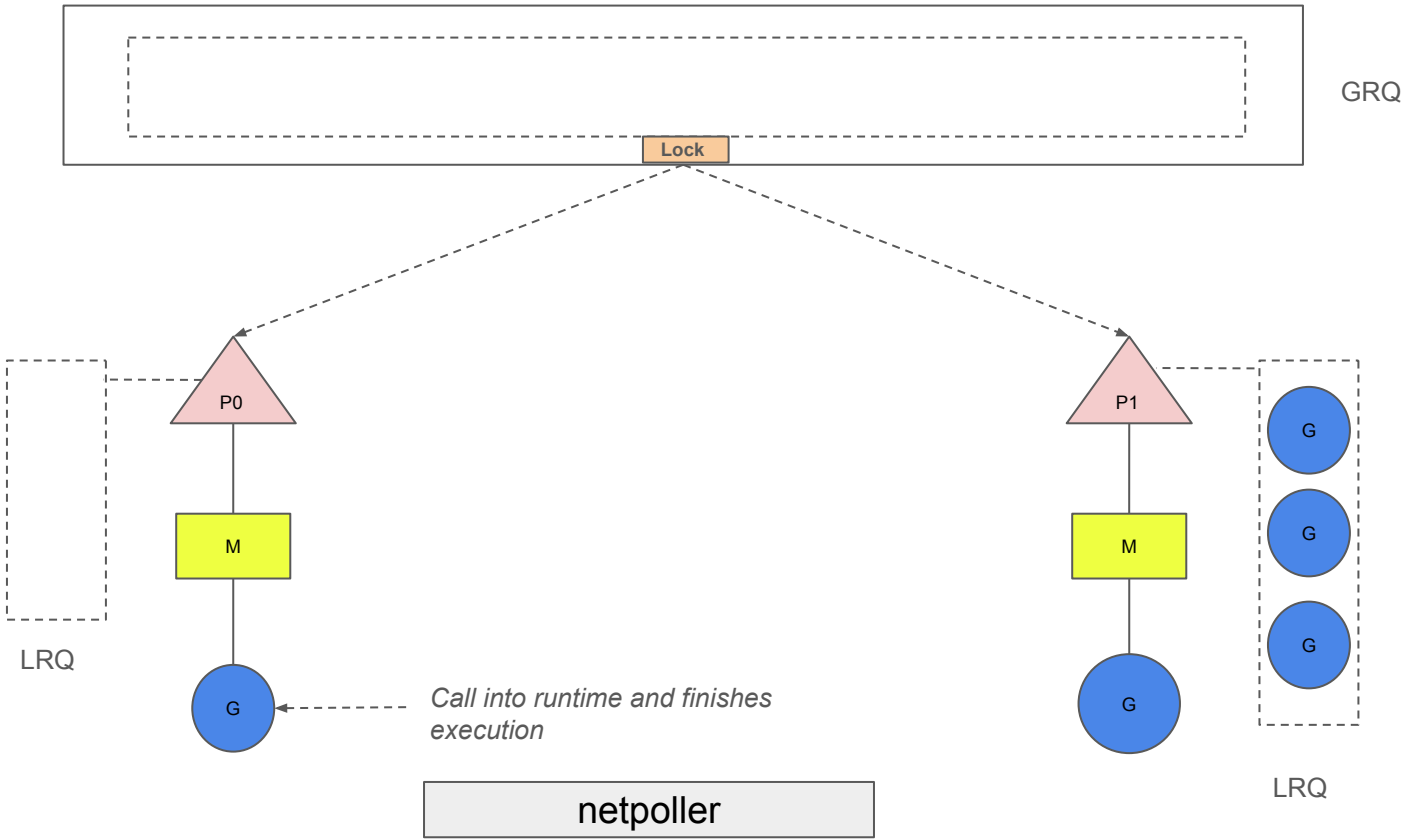

GRQ
Lock
P0
P1
G
M
3. Check the netpoller
Yes work is available
M
LRQ
G
LRQ
G
netpoller

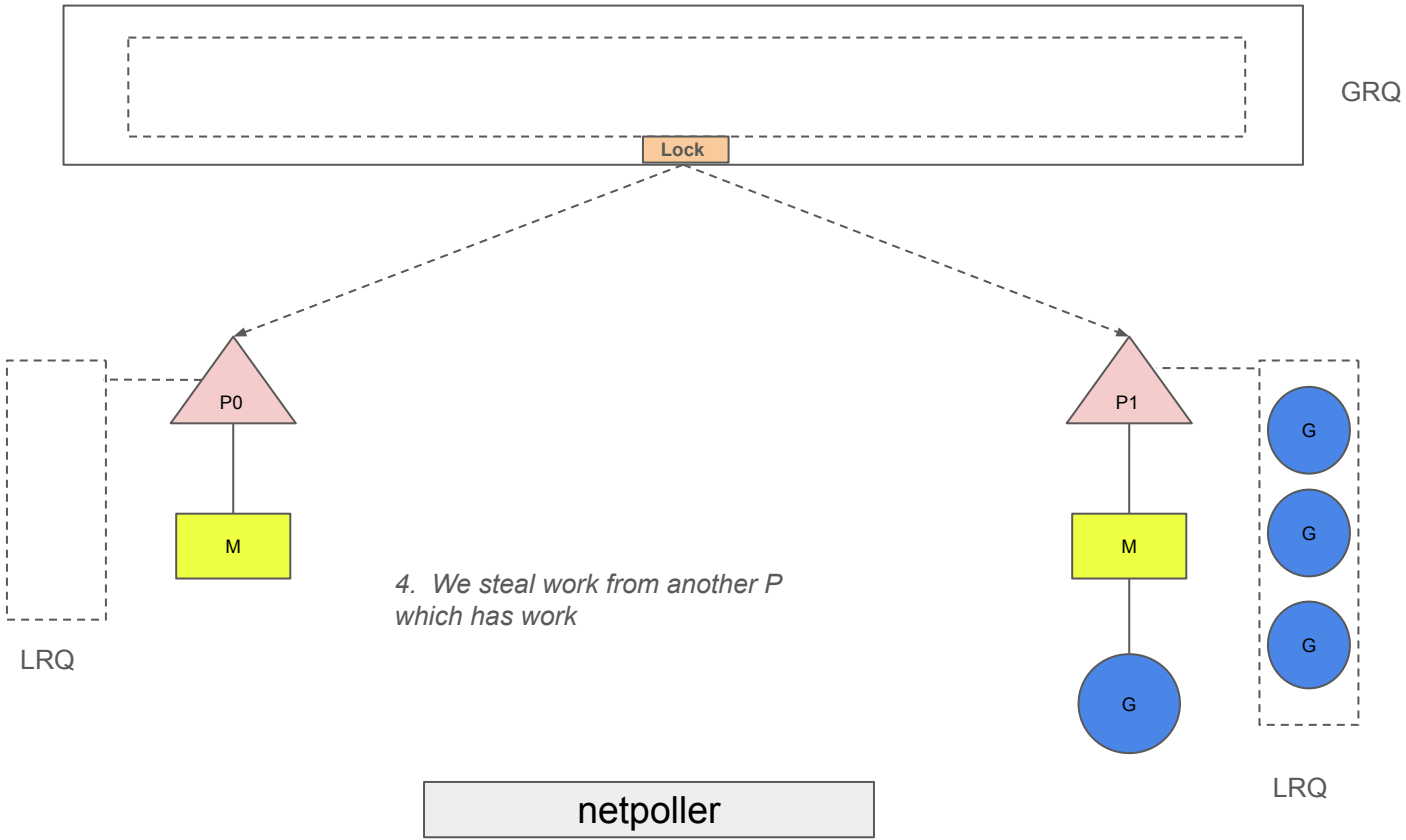
*Netpoller も空の場合はどうなるでしょうか？ 🤨*

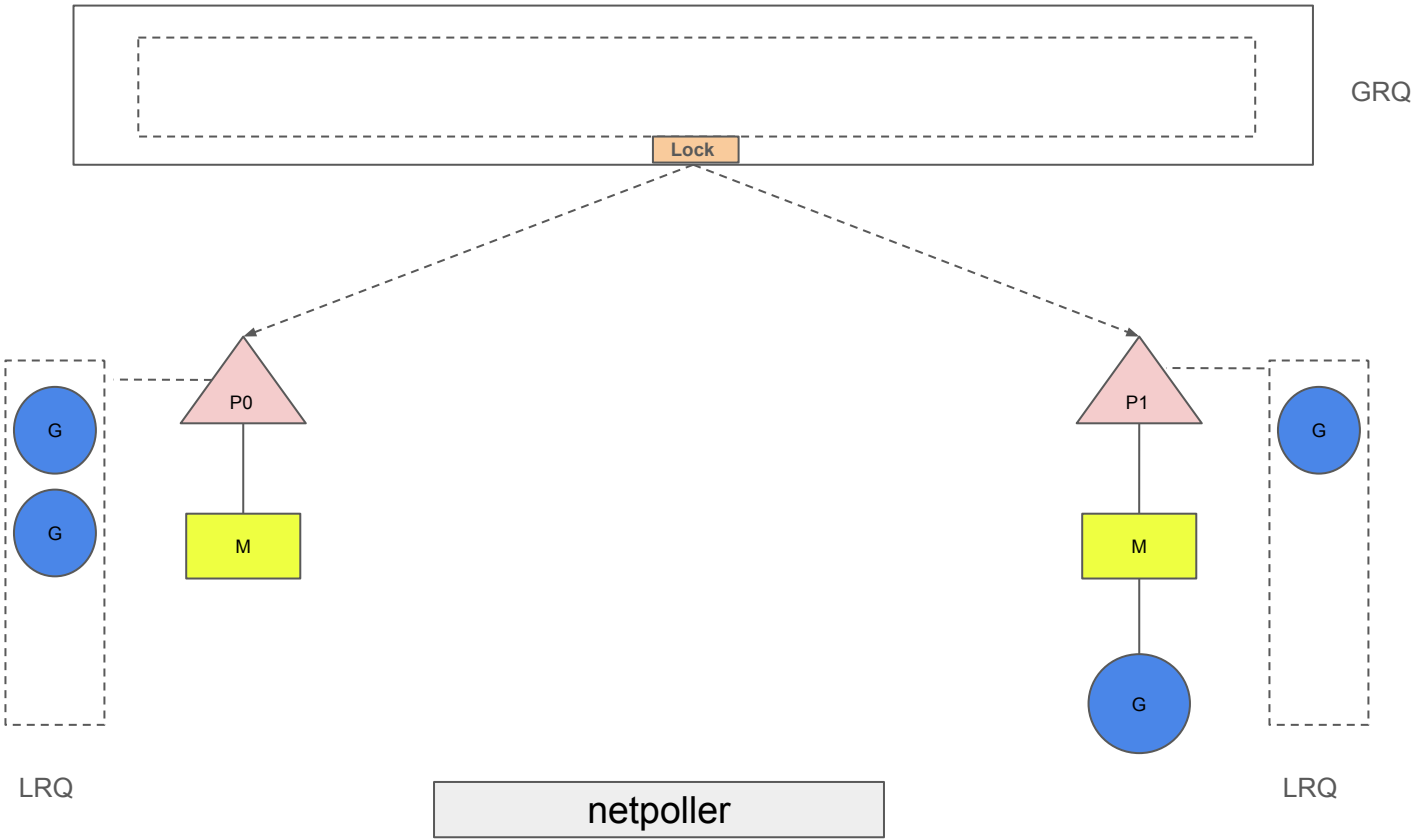
# *Work Stealing*

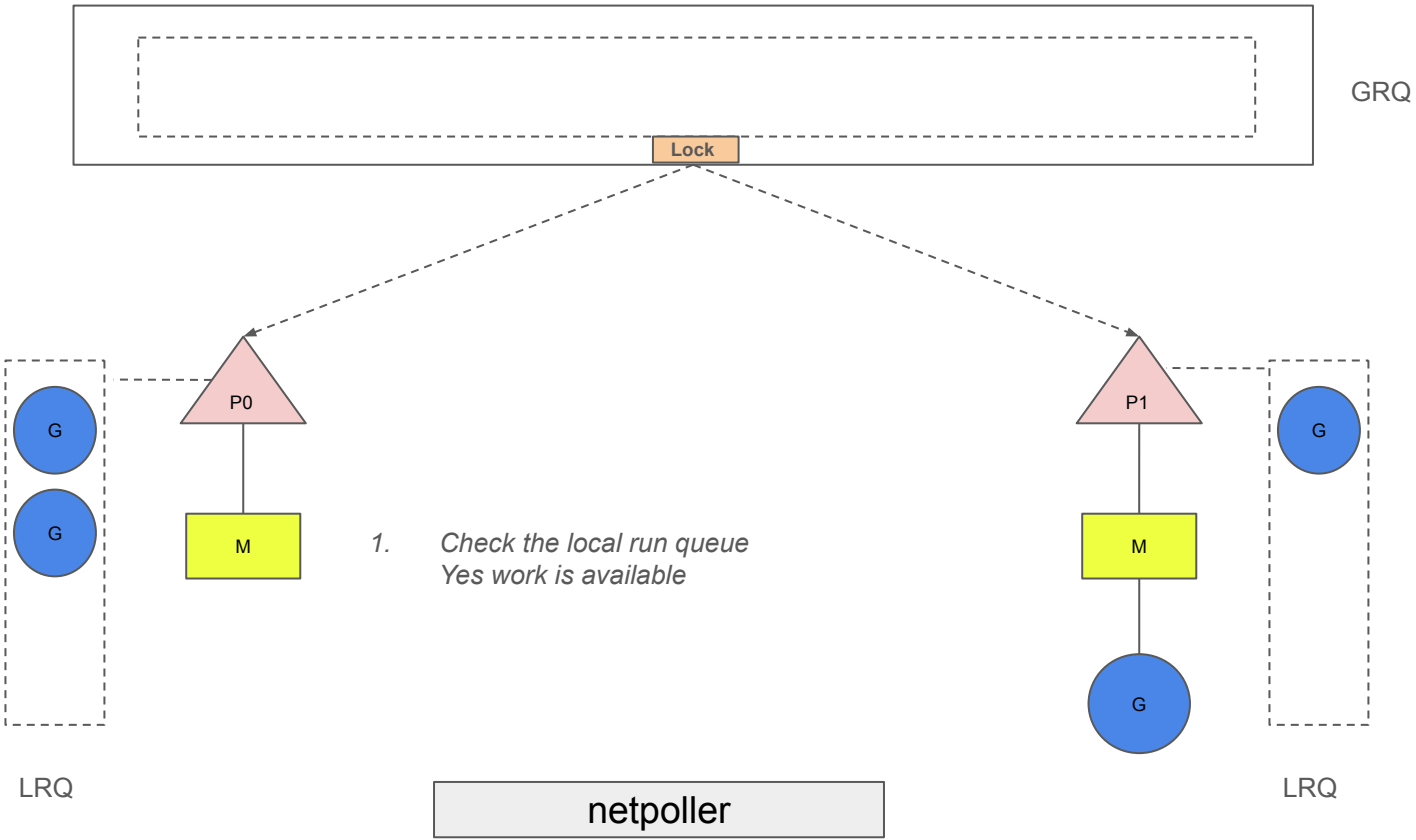


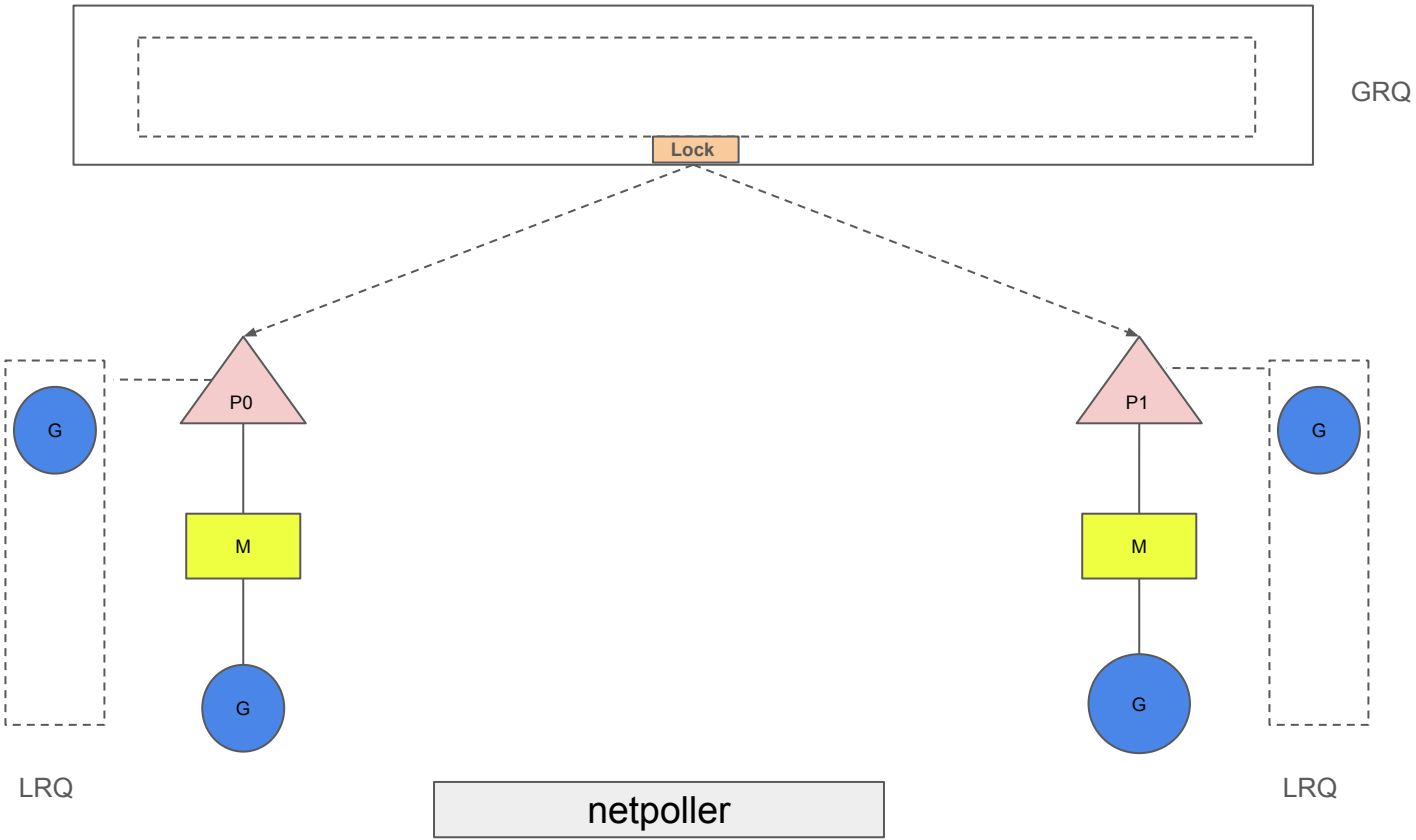

GRQ
Lock
P0
P1
M
M
G
G
G
G
G
LRQ
LRQ
netpoller


GRQ
Lock
P0
P1
M
M
G
G
G
G
G
LRQ
LRQ
Call into runtime and finishes execution
netpoller


GRQ
Lock
P0
M
LRQ
4. We steal work from another P
which has work
P1
M
G
G
G
G
LRQ
netpoller


GRQ
Lock
P0
P1
G
G
M
M
G
G
LRQ
LRQ
netpoller

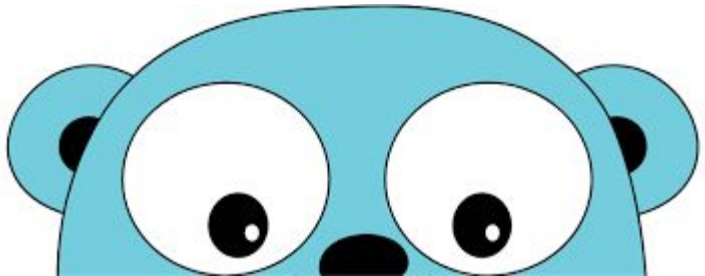

GRQ
Lock
P0
P1
G
G
M
M
G
G
1. Check the local run queue
Yes work is available
LRQ
LRQ
netpoller

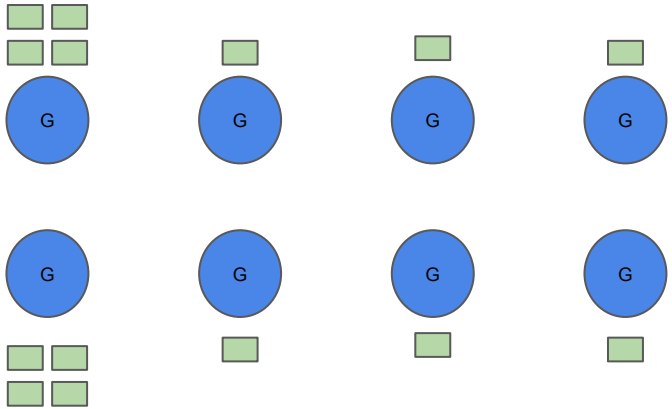

GRQ
Lock
P0
P1
G
M
G
G
M
G
LRQ
netpoller
LRQ

*Reference to the order of work stealing execution - runtime/proc.go*

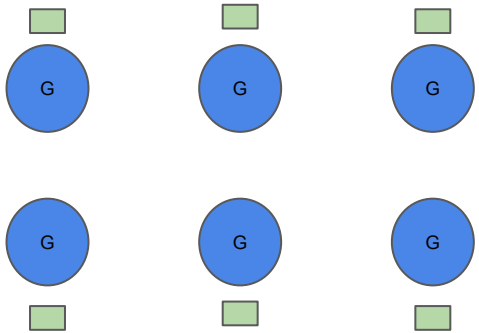
*What about long running task*

# *Preemption*



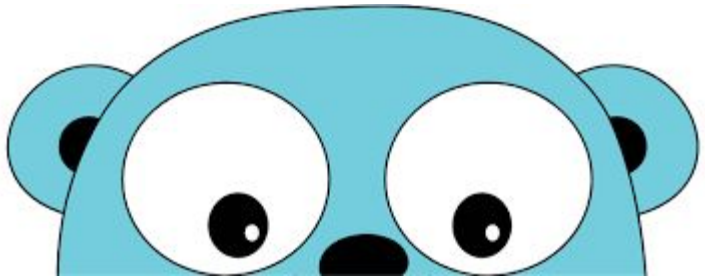

G
G
G
G
G
G
G
G


G
G

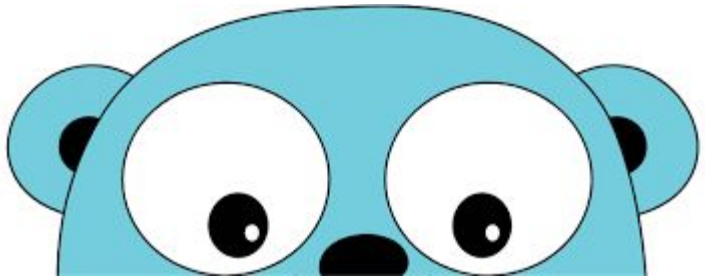

G
G
G
G
G
G

## *Upto Go 1.10*

- *Go has used cooperative preemption with safe-points only at function calls.*
- *From execution point of view you can give goroutine processor time (execution time) only on specific events (safe-points) which are function calls.*



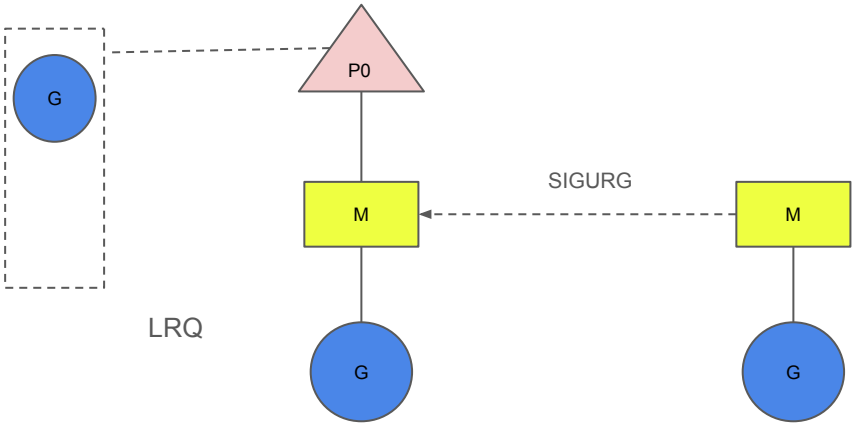
*What's Now ...*



## *Non-Cooperative Preemption*

- *Go introduced non-cooperative preemption because of the problems mentioned above.*
  *Go は、先ほどの問題を解決するために「非協調型プリエンプション」を導入しました。*

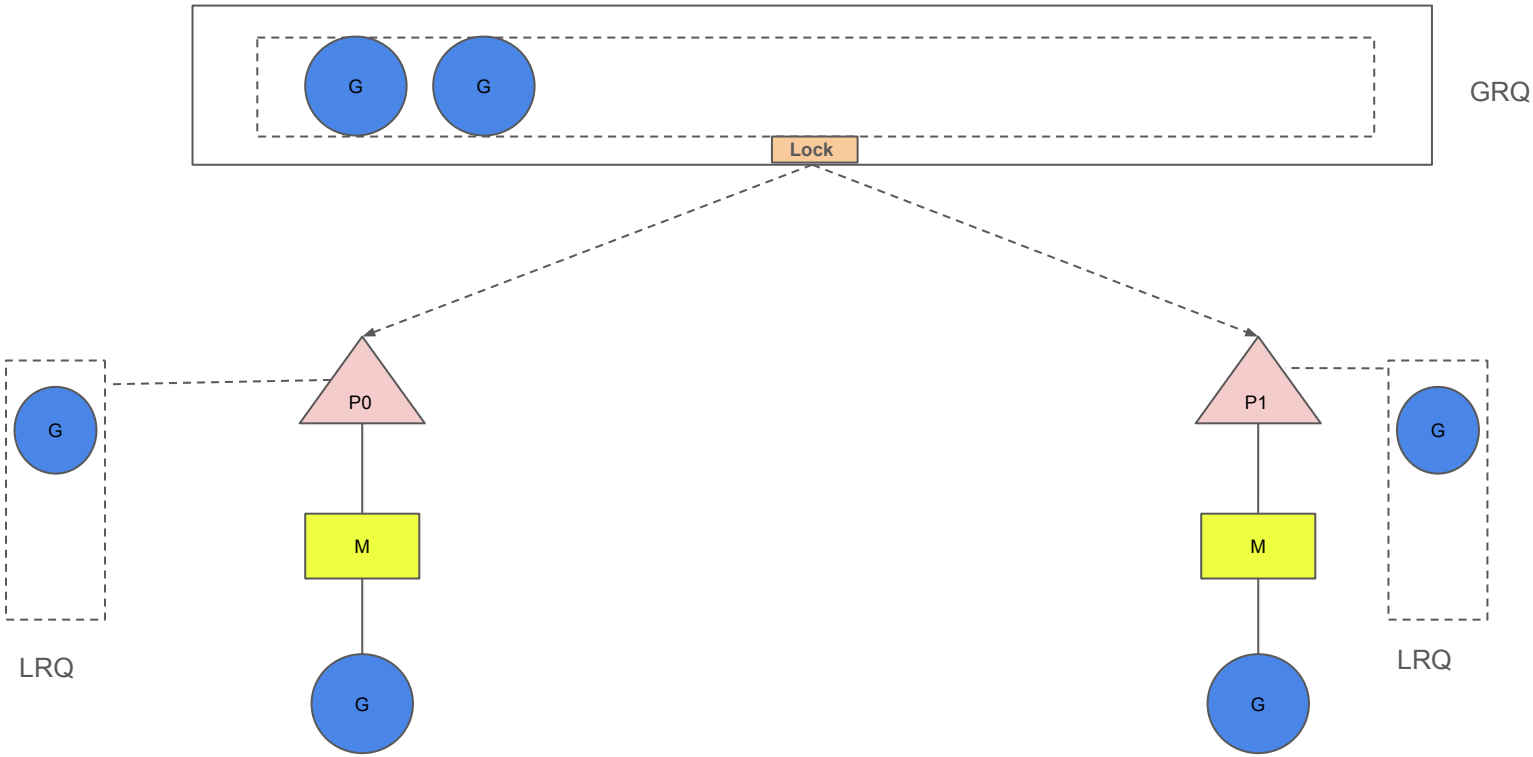
- *In non-cooperative preemption, the Go runtime can forcibly pause a running goroutine even if it doesn't explicitly yield control. This preemptive behavior ensures that no single goroutine can monopolize the CPU for an extended period.*
  *非協調型では、Goroutine が自ら制御を譲らなくても、Go ランタイムが強制的に一時停止させることができます。この仕組みによって、1つの Goroutine が延々と CPU を独占することが防がれます。*



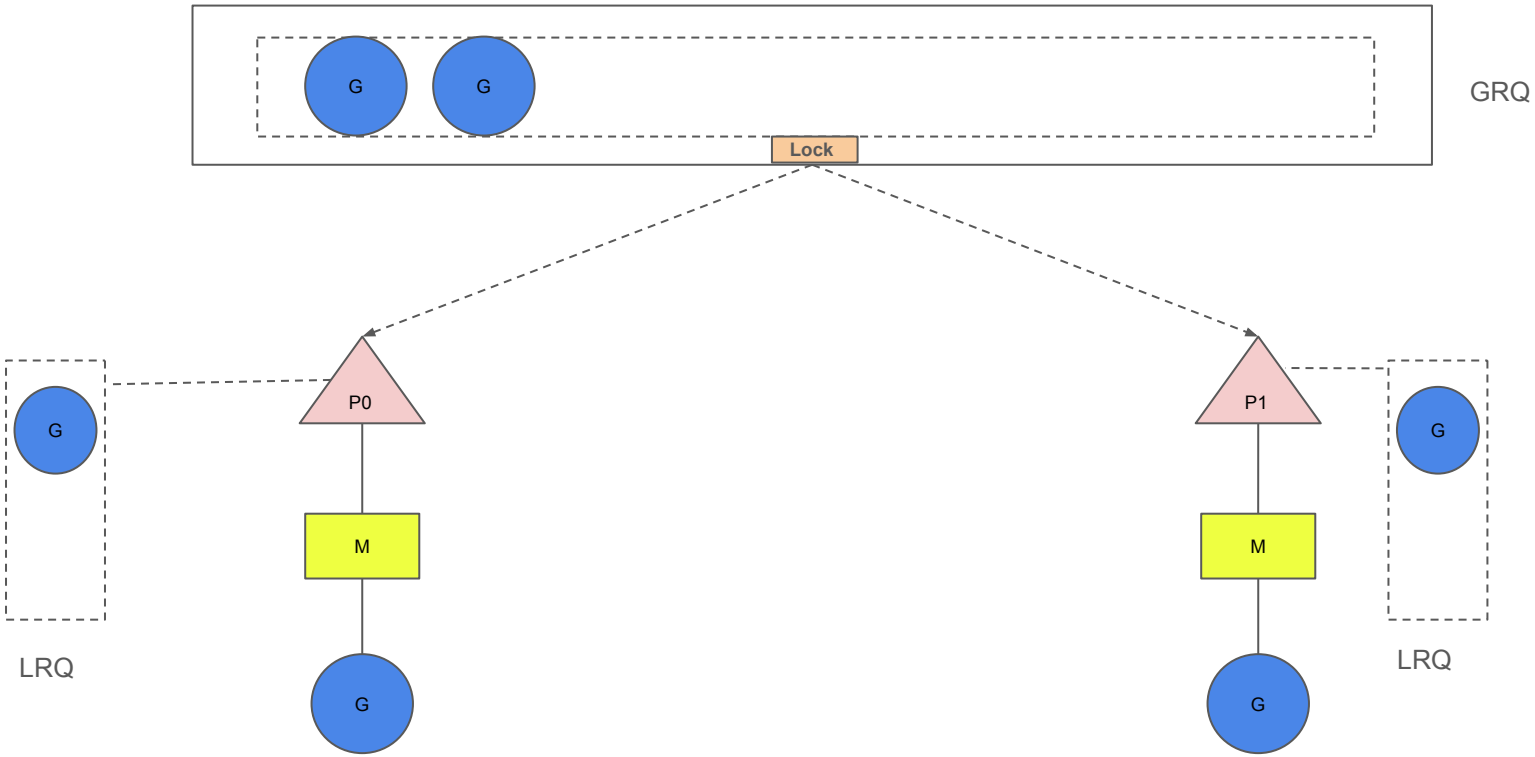

## Sysmon Daemon


*Been running for 10 ms*

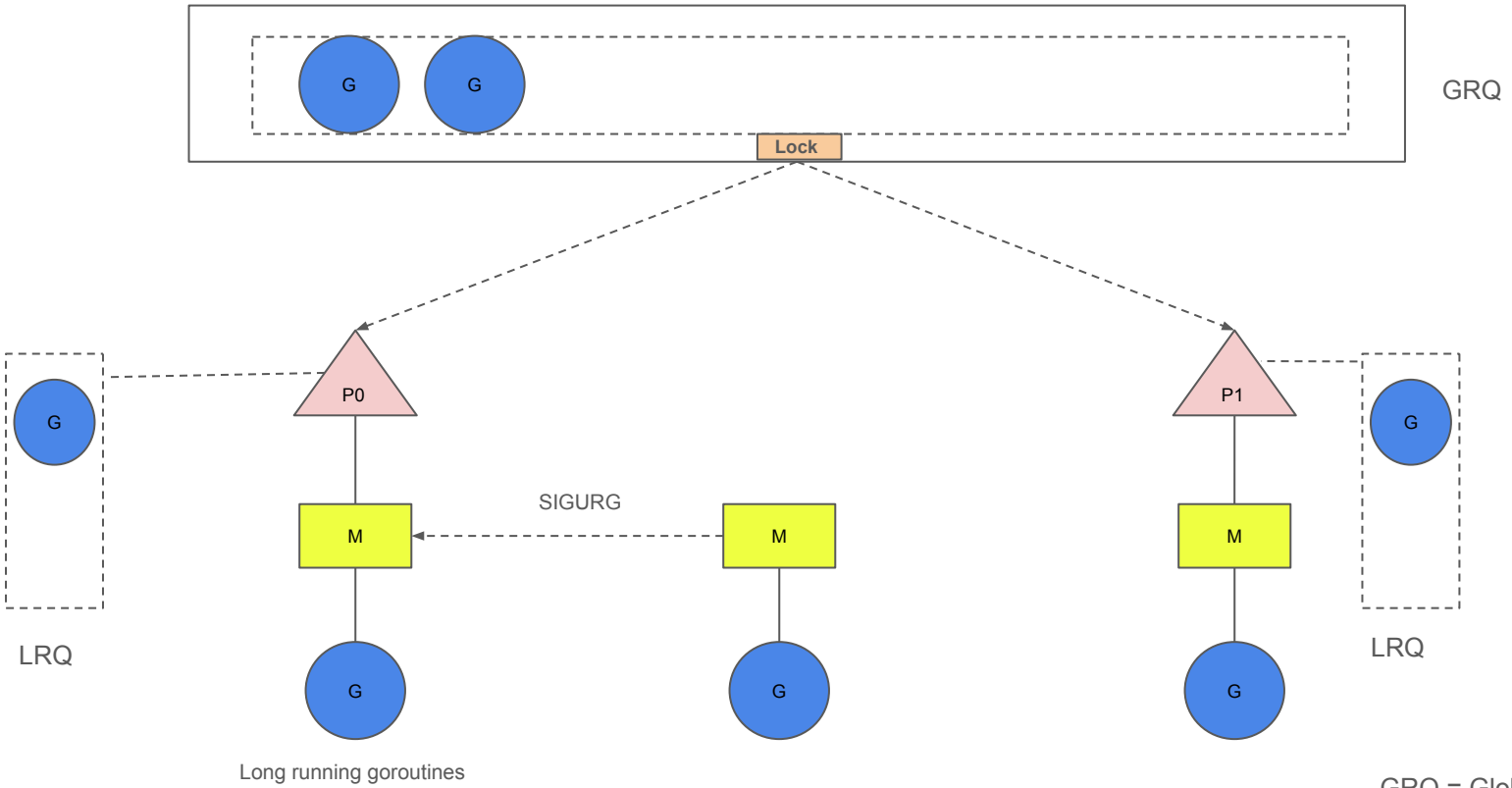
*Where does the preempted go routine ends up going 🤔?*



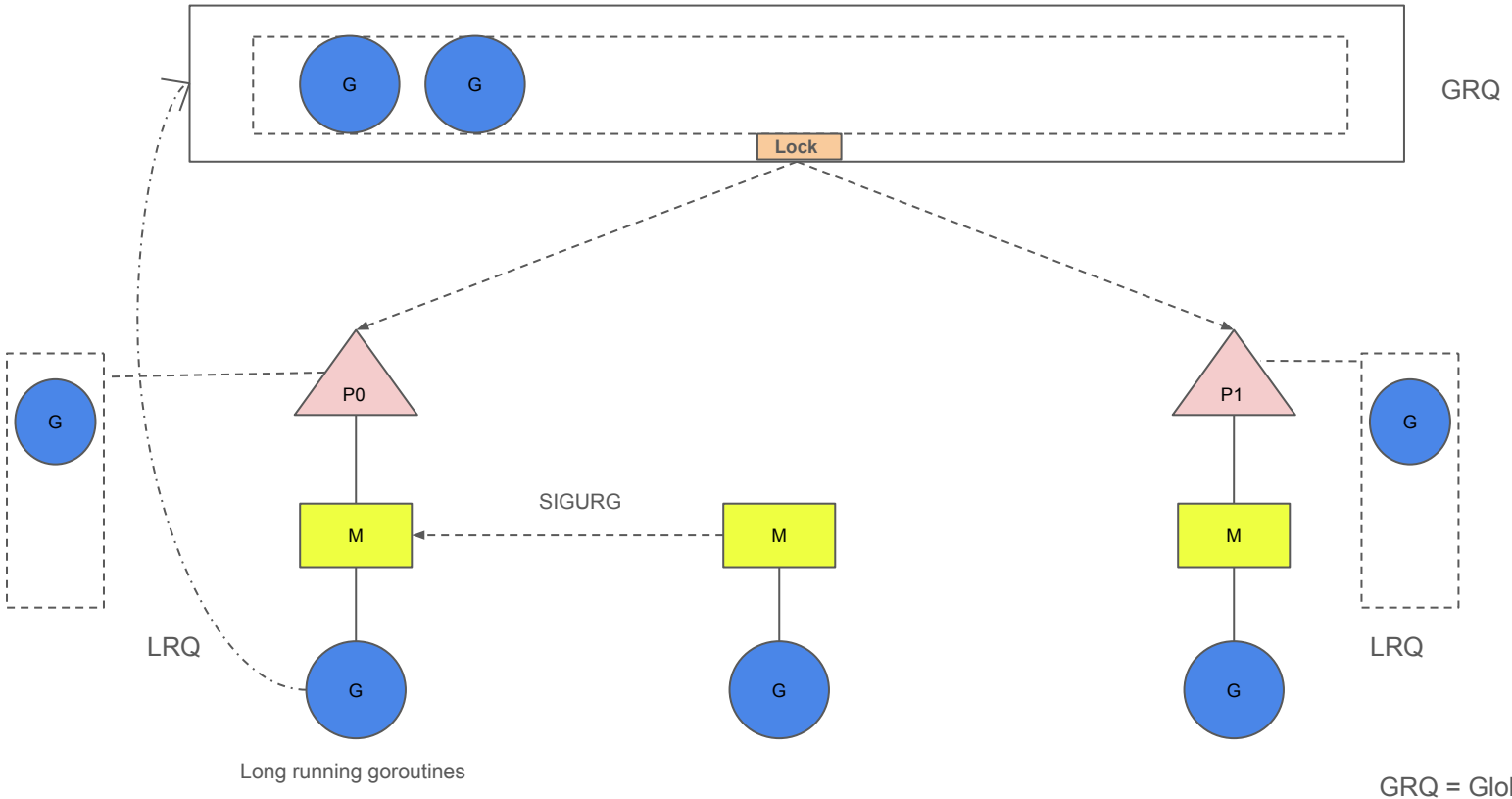

GRQ = Global Run Queue
LRQ = Local Run Queue



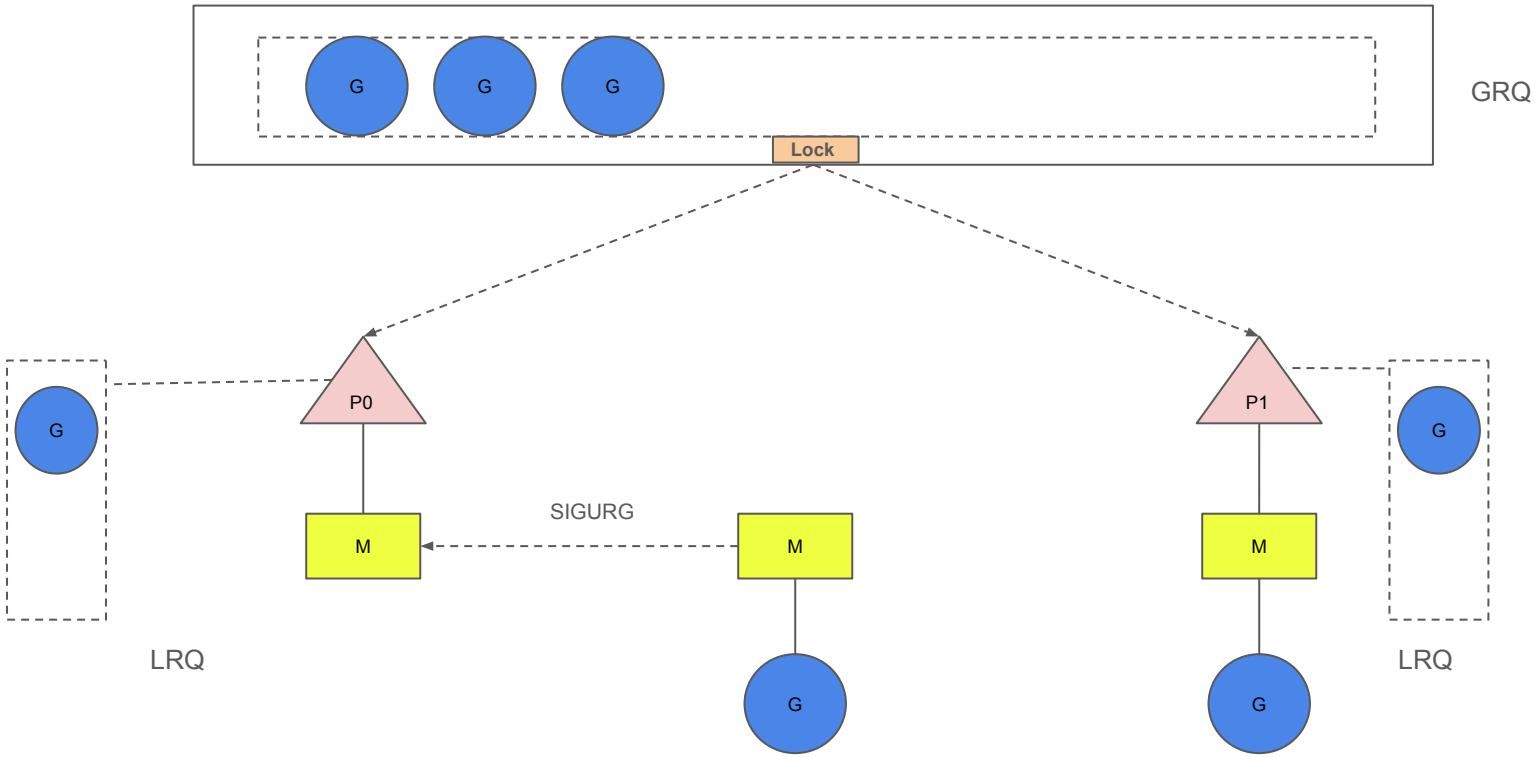

GRQ = Global Run Queue
LRQ = Local Run Queue



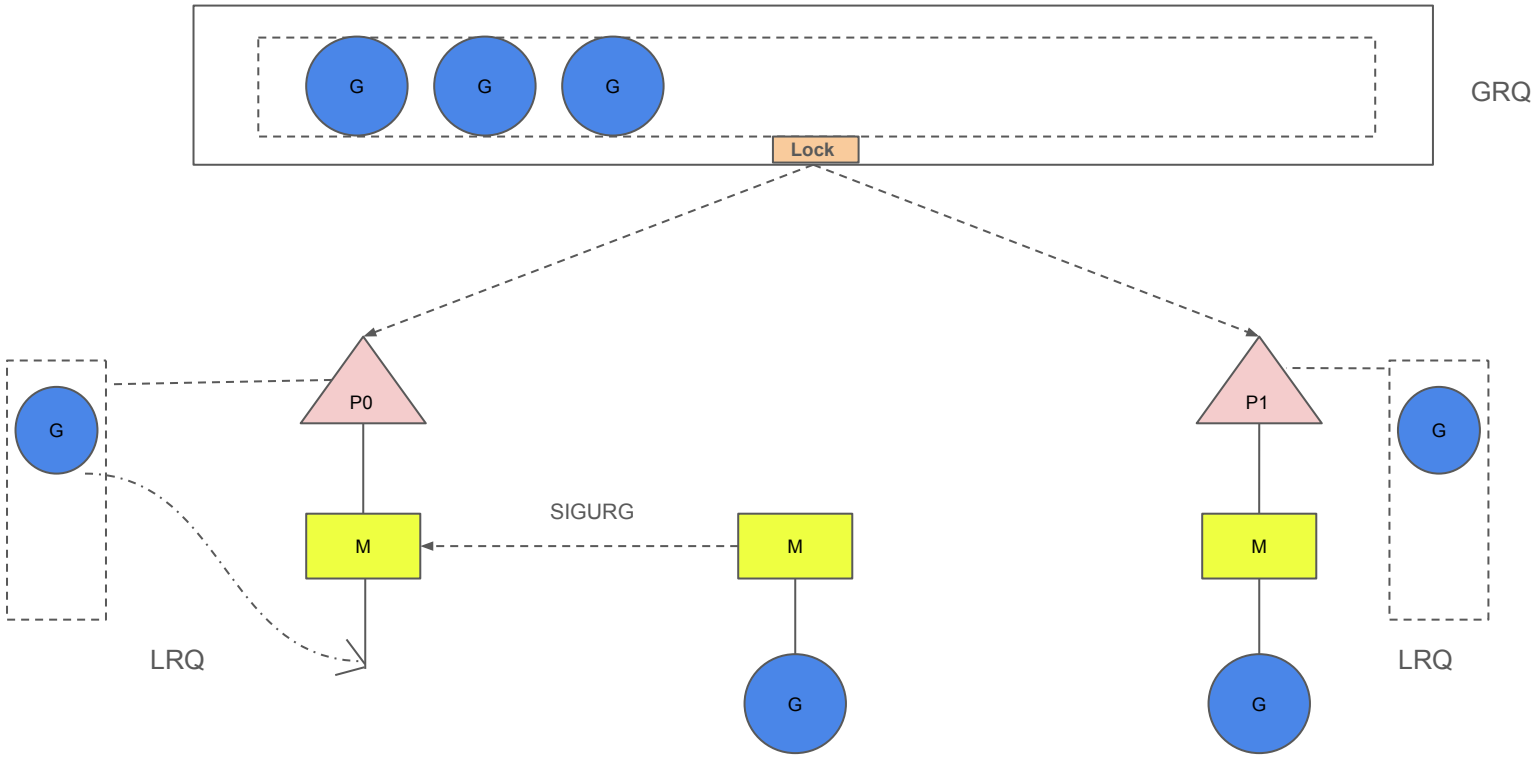

GRQ = Global Run Queue
LRQ = Local Run Queue



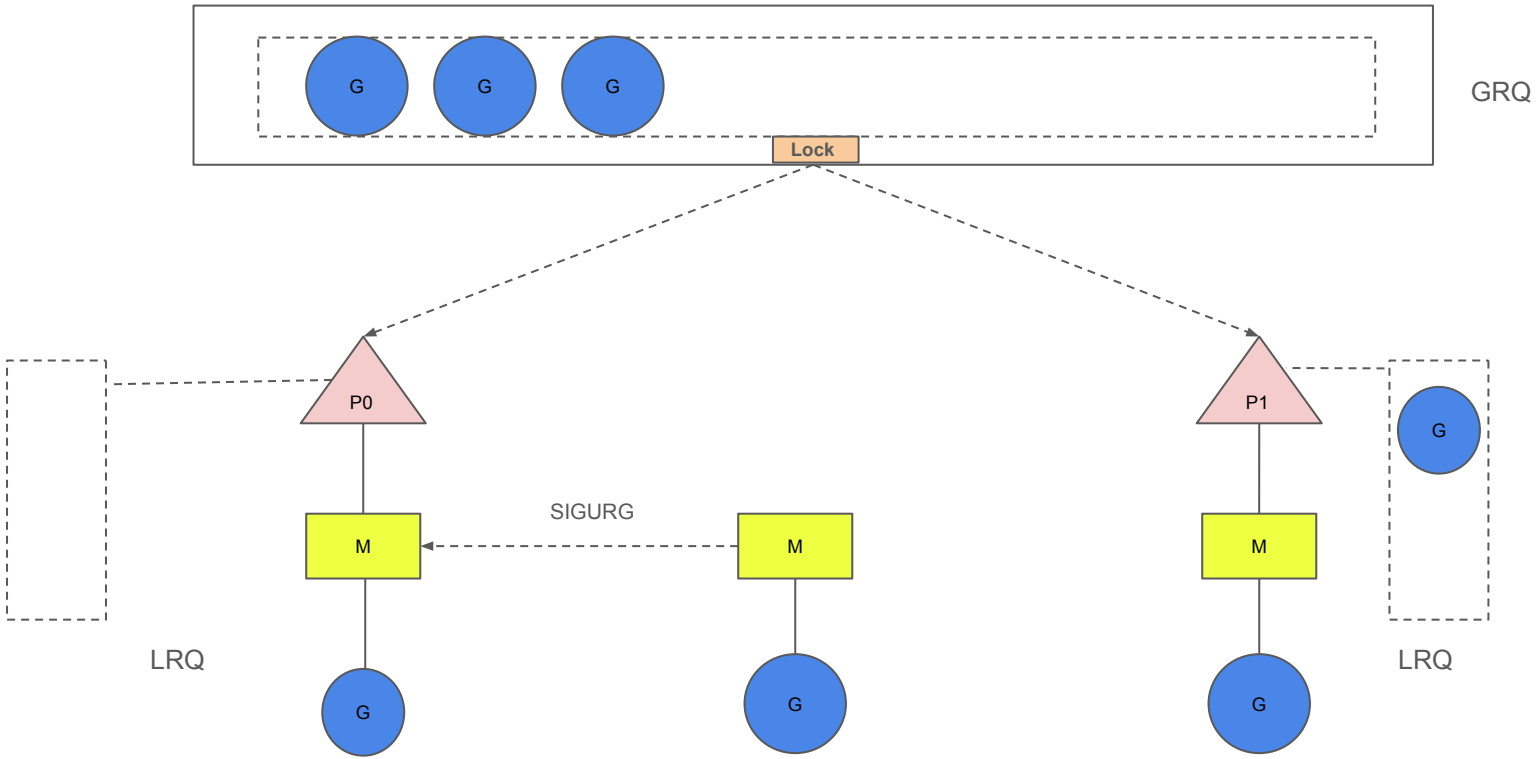

GRQ = Global Run Queue
LRQ = Local Run Queue




GRQ = Global Run Queue
LRQ = Local Run Queue




GRQ = Global Run Queue
LRQ = Local Run Queue




GRQ = Global Run Queue
LRQ = Local Run Queue

# Visualization

GO CONFERENCE 2025

# Visualize scheduling, preemption, and runnable queues with runtime/trace

```go
func cpuBound(id int, iters int, wg *sync.WaitGroup) {
    defer wg.Done()
    sum := 0
    for i := 0; i < iters; i++ {
        sum += i ^ (i >> 3)
        if i%500000 == 0 {
            // Hint to runtime: allow preemption points
            runtime.Gosched()
        }
    }
    _ = sum
}
```

- *Tight arithmetic to consume CPU*
- *The inner loop does simple integer ops to keep the compiler from optimizing away work*
- *runtime.Gosched() adds explicit yield points*

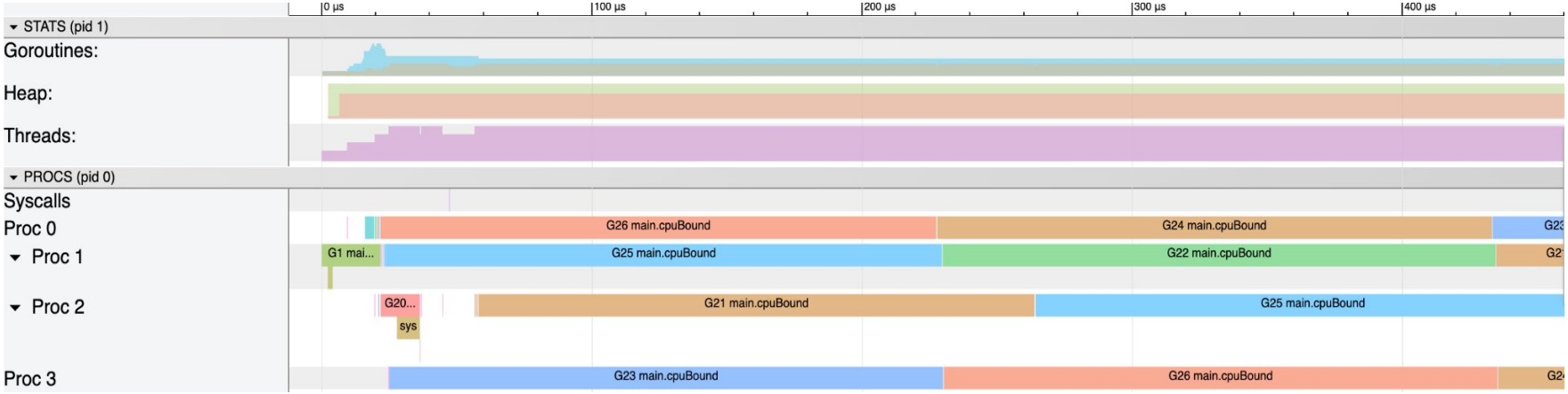
```go
func ioBlocked(id int, wg *sync.WaitGroup) {
    defer wg.Done()
    // Simulate "blocking" on sleep to show goroutine parking/unparking
    time.Sleep(300 * time.Millisecond)
}
```

- *ioBlocked Function Just sleeps for 300 ms to simulate a blocking operation*

```go
func main() {
    // Encourage multiple Ps to see scheduling across threads
    runtime.GOMAXPROCS(4)

    f, err := os.Create("trace.out")
    if err != nil {
        panic(err)
    }
    defer f.Close()

    if err := trace.Start(f); err != nil {
        panic(err)
    }
    defer trace.Stop()

    var wg sync.WaitGroup
    rand.Seed(time.Now().UnixNano())

    // Mix cpu-bound and "IO"-blocked goroutines
    cpuG := 6
    ioG := 4

    wg.Add(cpuG + ioG)
    for i := 0; i < cpuG; i++ {
        it := 3_000_000 + rand.Intn(1_000_000)
        go cpuBound(i, it, &wg)
    }
    for i := 0; i < ioG; i++ {
        go ioBlocked(i, &wg)
    }

    wg.Wait()
    fmt.Println("done; trace written to trace.out")
}
```

- *The program mixes CPU-bound goroutines and blocking sleepers*
- *Then records a runtime execution trace so the scheduler's behavior, park/unpark, and preemption are visible in the trace UI.*

# Demonstration




- *The 4 Ps thanks to GOMAXPROCS(4) colored slices are goroutines running on a P's M, and blank gaps mean the P had nothing runnable or we're between events.*
- *The sleepers park on time.Sleep see them disappear from Ps then a timer wakes them and they become Runnable and run again, that's unpark.*
- *Thee cpu Bound goroutines yield at Gosched, so we see short slices and frequent context switches*
  **removing Gosched shows longer runs until the runtime preempts (10 ms).*
- *Notice the same GID switching P lanes that Work Stealing .*
- *Preemption occurs either cooperatively (runtime.Gosched()) or asynchronously Removing Gosched shows the runtime-driven preemption more clearly as longer uninterrupted slices that get cut by async preemption .*

GO CONFERENCE 2025

*Beginner Pitfalls & My Learnings*

# *Blocking isn't just I/O*

- *Network/disk/syscalls block a G*
- *Long CPU loops starve others*
- *Big buffers hide backpressure*
- *Actions: timeouts, contexts, small critical sections*

```
// Bad: long work under lock
mu.Lock()
data = heavyCompute(load()) // blocks others
mu.Unlock()

// Better: shrink the critical section
x := load()
y := heavyCompute(x) // outside lock
mu.Lock()
data = y
mu.Unlock()
```

# GOMAXPROCS: Measure, Don't Guess

- Controls parallel goroutine execution
- CPU-bound: ≈ NumCPU
- I/O-bound: too high → context switching
- Start at default; tune from traces

```
// Inspect and set intentionally
n := runtime.GOMAXPROCS(0)
_ = n // current setting
runtime.GOMAXPROCS(runtime.NumCPU())
```

# Preemption: Give the Scheduler Air

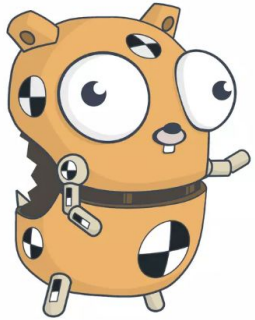
- Tight CPU loops can hog a P
- Insert calls/checks; use contexts
- Break big tasks into steps

```
func crunch(ctx context.Context) {
    for i := 0; i < bigN; i++ {
        if i%1_000 == 0 {
            select {
            case <-ctx.Done():
                return
            default:
            }
        }
        step(i) // function call = preemption point
    }
}
```

# *My Quick Fix Checklist*

- *Where can this block? (I/O, locks, channels)*
- *Is concurrency bounded ?*
- *Are tasks too chunky?*
- *Are locks too coarse?*
- *Do traces show runnable goroutines waiting?*

# Conclusion

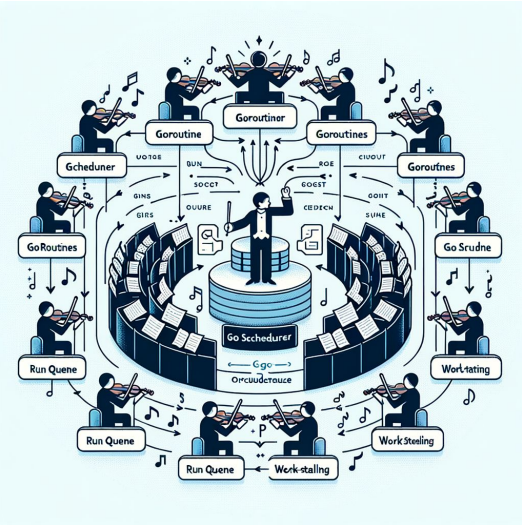


## *Why Understanding the Scheduler Matters*

- *Predictability under load*
- *Fewer “mystery slowdowns”*
- *Better decisions: pooling, buffering, timeouts*
- *Faster debugging with data, not guesses*

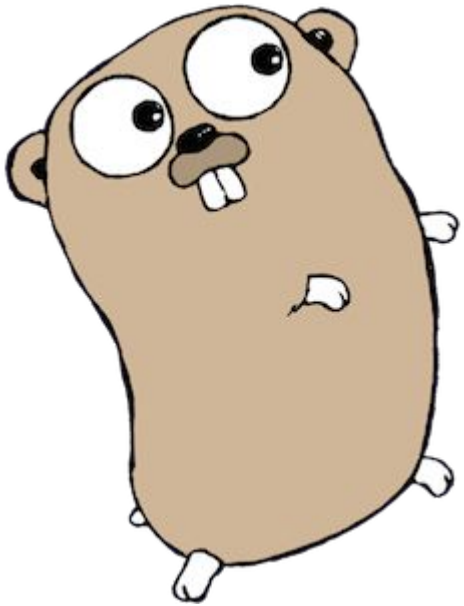
## *After go func(), the Scheduler Conducts*

- *Make it easy: bounded, balanced, measurable*
- *Learn the patterns; trust the traces*
- *Takeaway: Measure, don’t guess*



# References :


- https://community.sap.com/t5/additional-blog-posts-by-sap/mastering-concurrency-unveiling-the-magic-of-go-s-scheduler/ba-p/13577437
- https://go.dev/src/runtime/proc.go
- https://github.com/golang/proposal/blob/master/design/24543-non-cooperative-preemption.md
- https://www.cs.columbia.edu/~aho/cs6998/reports/12-12-11_DeshpandeSponslerWeiss_GO.pdf
- https://medium.com/@hatronix/inside-the-go-scheduler-a-step-by-step-look-at-goroutine-management-1a8cbe9d5dbd
- https://medium.com/a-journey-with-go/go-work-stealing-in-go-scheduler-d439231be64d
- https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit?tab=t.0#heading=h.mmq8lm48qfcw
- https://www.youtube.com/watch?v=S-MaTH8WpOM&ab_channel=Hypermode



聞いてくれて
ありがとうございます

Session Code Repo

Session Slides

# Q/A