Queues, Fairness, and The Go Scheduler

Madhav Jivrajani, VMware

`$ whoami`

- Work @ VMware
- Spend most of my time in the Kubernetes community (API-Machinery, ContribEx, Architecture).
- Currently also an undergrad @ PES University, Bangalore, India

Agenda

- Motivation
- Go’s scheduler model
- Fairness
- Design of the Go scheduler
- Visualizing the scheduler at play
- Looking at the fine print
- Knobs of the runtime scheduler
- Conclusion
- References

Small disclaimer: Everything discussed is in reference to Go 1.17.2

So, why are we here? And why do we care?

Goroutines!

- “Lightweight threads”
- Managed by the Go runtime
- Minimal API (`go`)

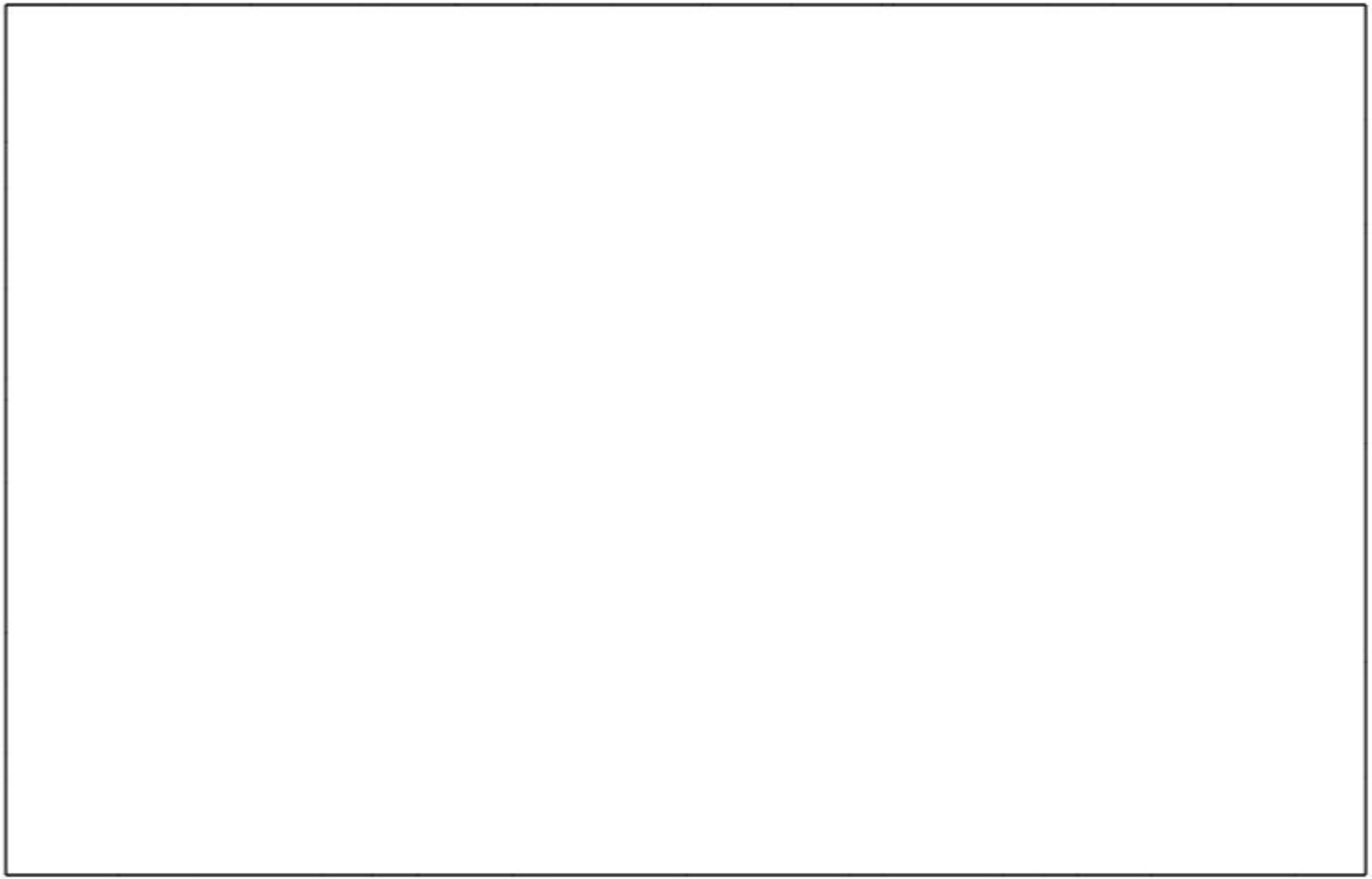
Let’s take a small example

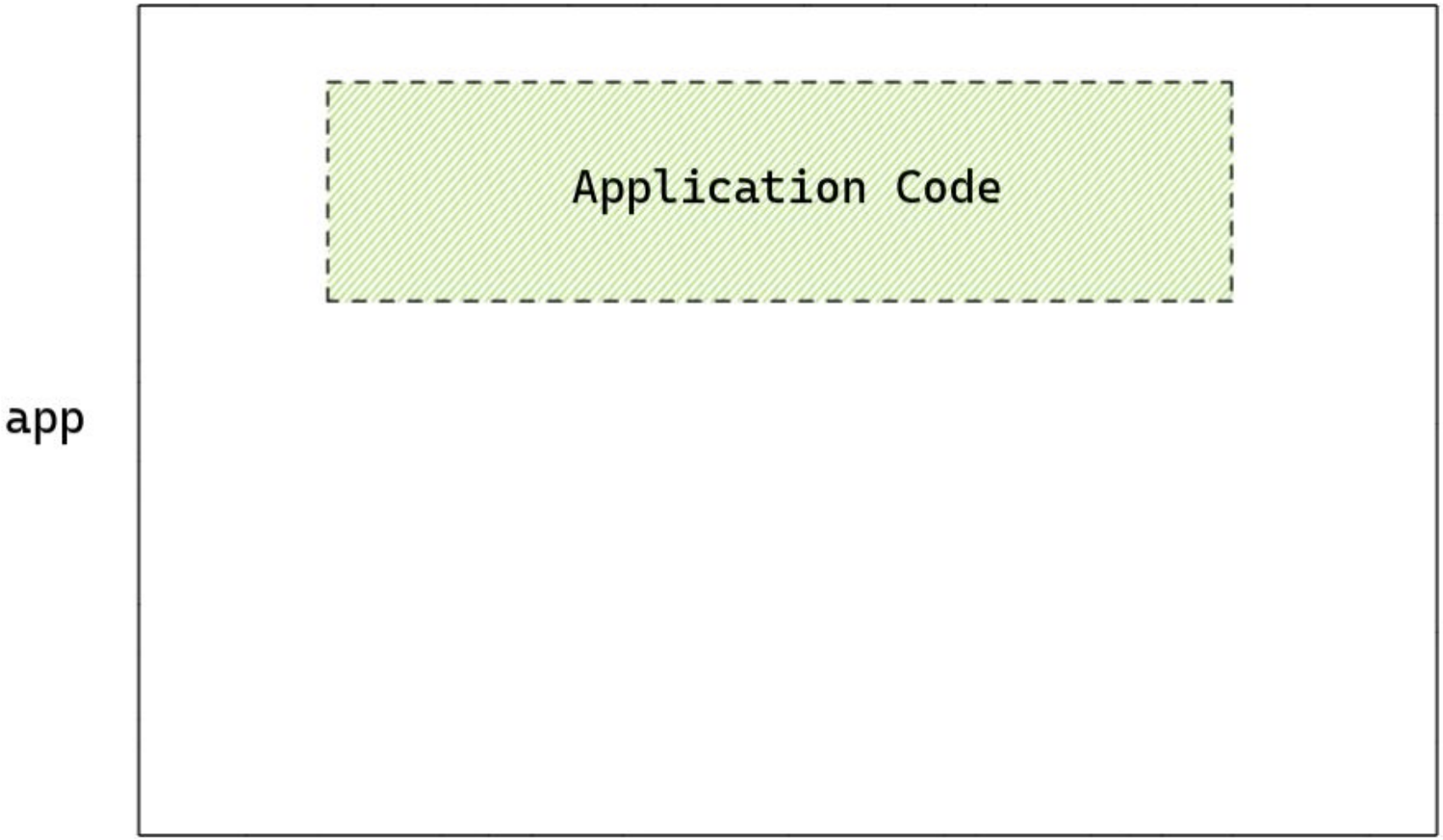
```
// An abridged main.go
func main() {
    go doSomething()
    doAnotherThing()
}
```

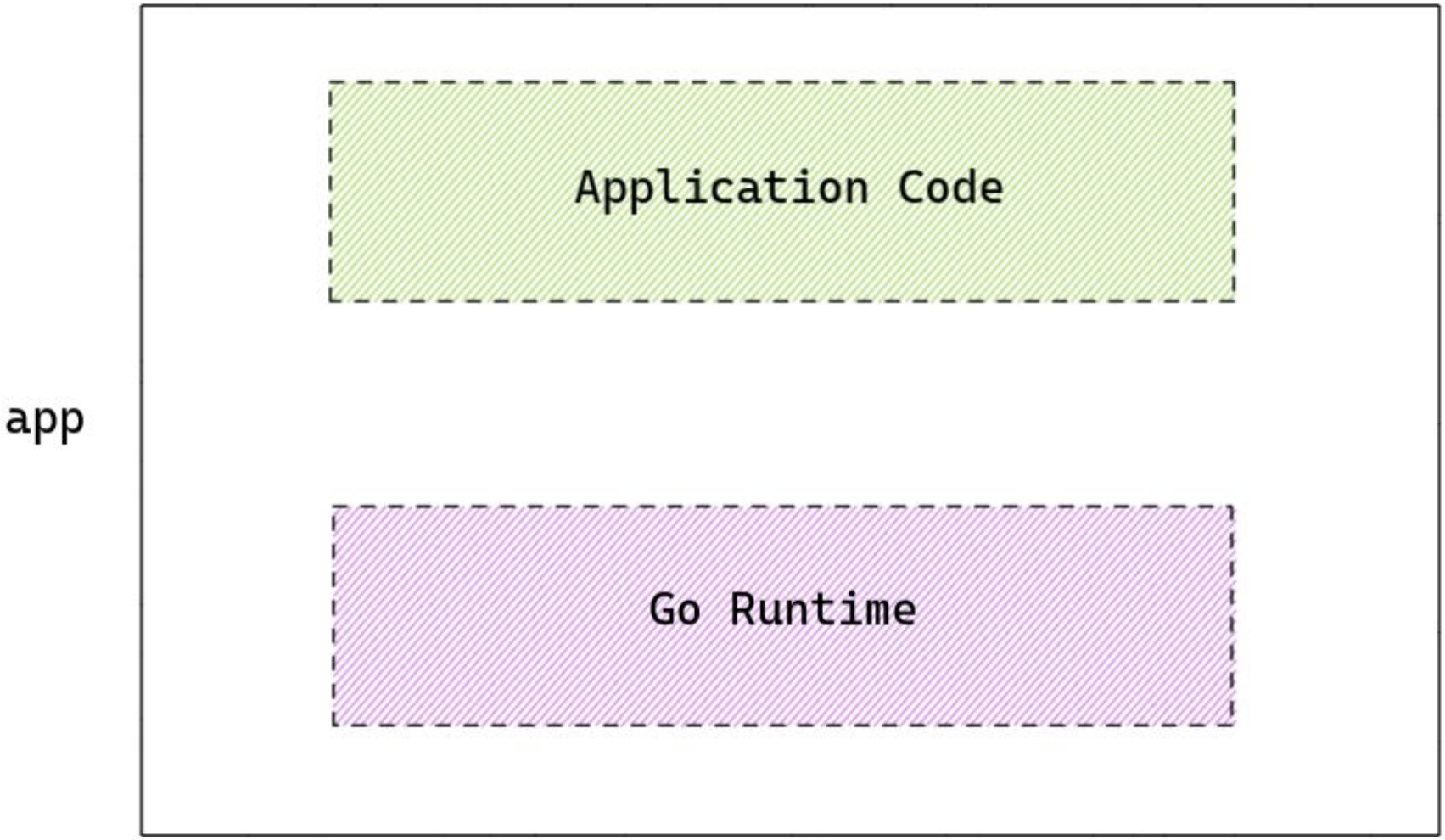
```go
// An abridged main.go
func main() {
    go doSomething()
    doAnotherThing()
}
```

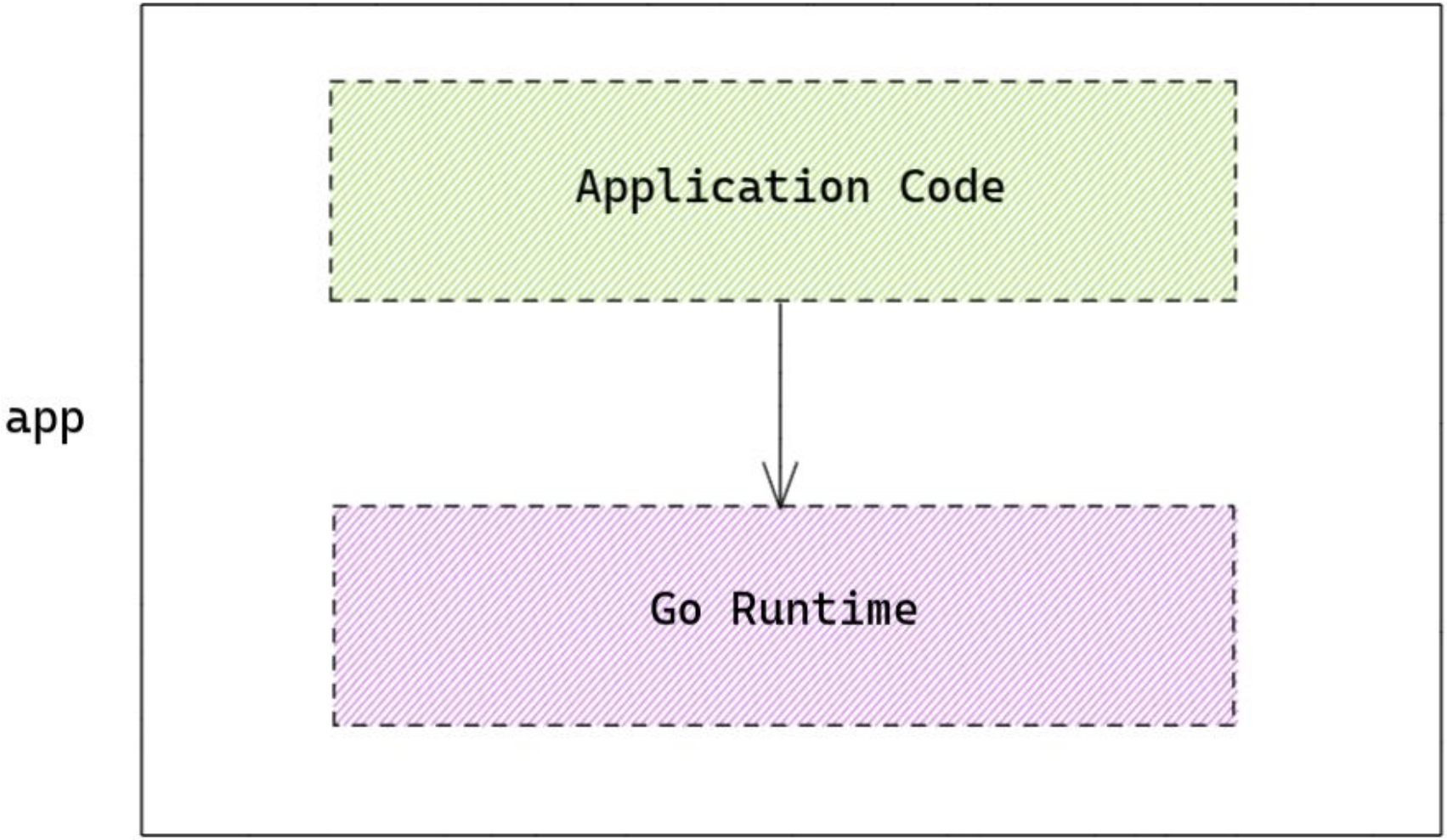
```
go build -o app main.go
```

app




Application Code
app

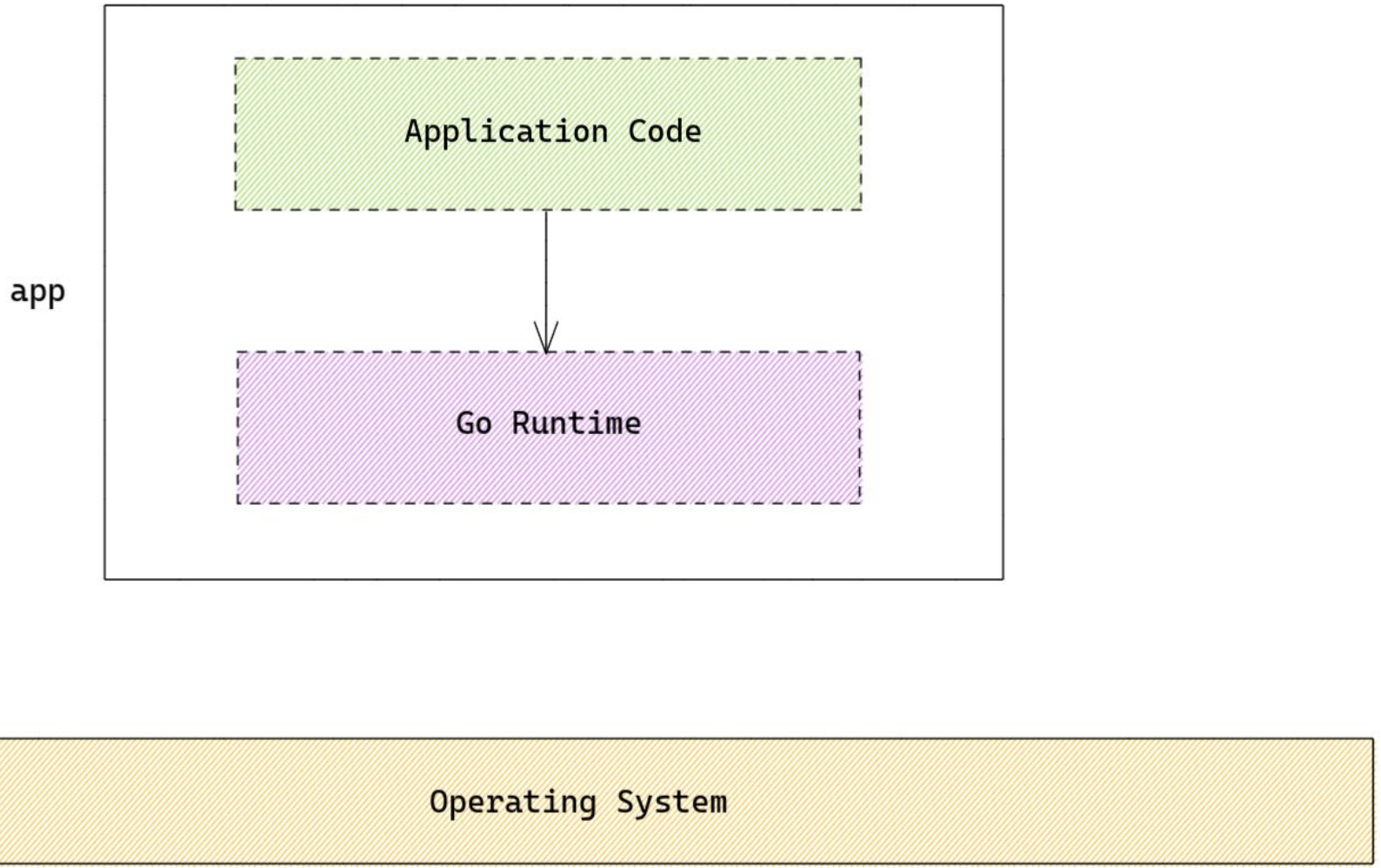

app
Application Code
Go Runtime

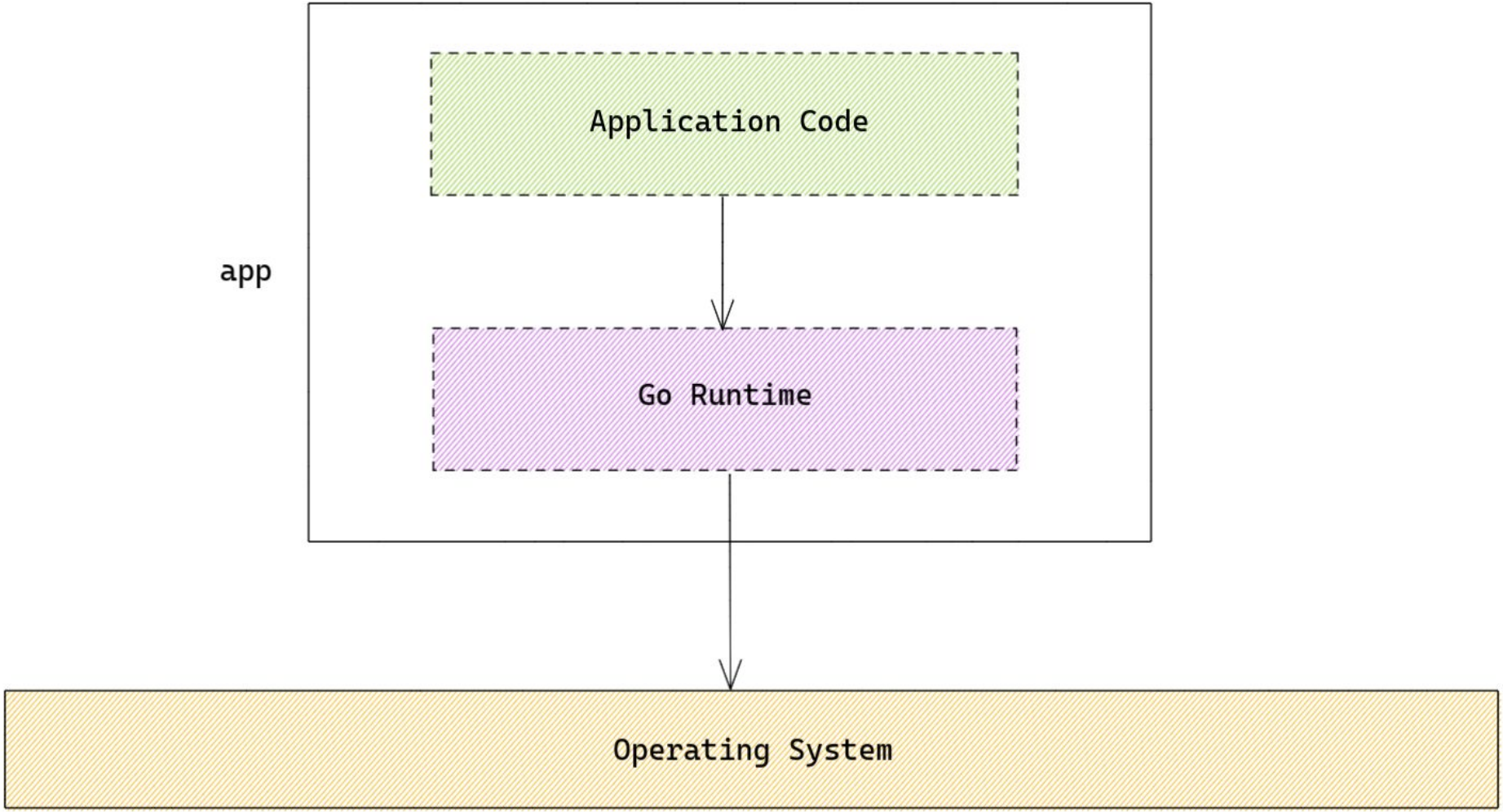

app
Application Code
Go Runtime

One such example of calling into the `runtime`

```
func main() {
    go doSomething()
    doAnotherThing()
}
```

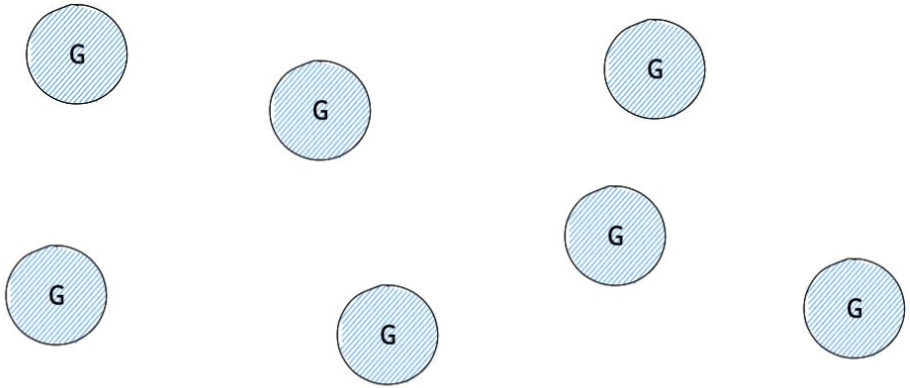
```
func main() {
    runtime.newProc(...)
    doAnotherThing()
}
```

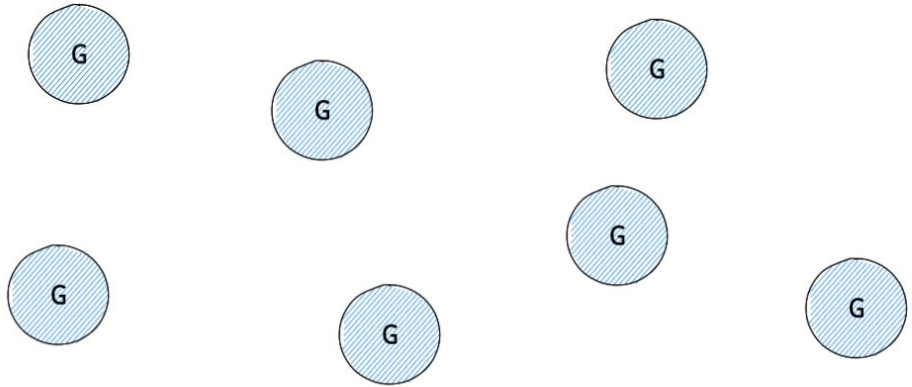

app
Application Code
Go Runtime
Operating System

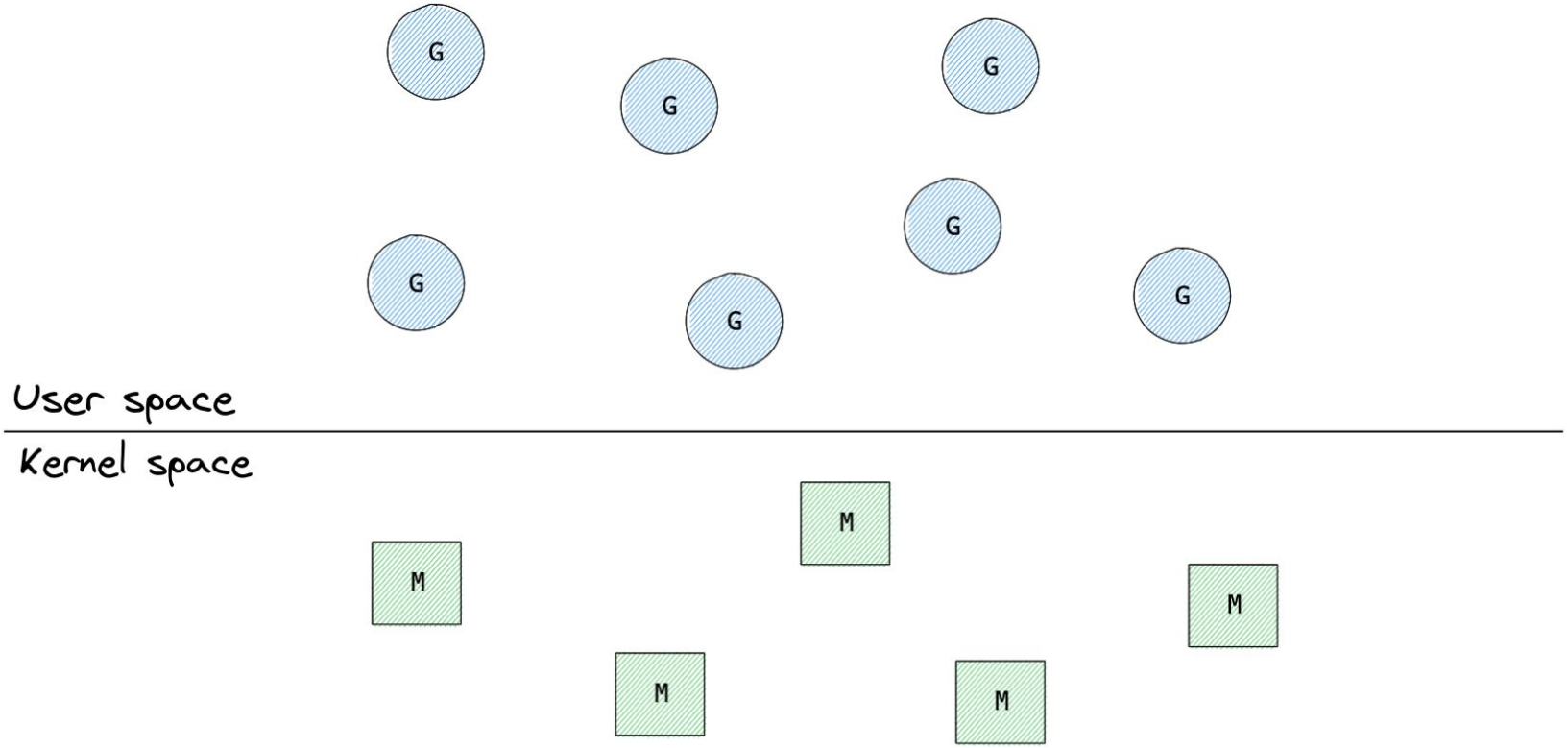

app
Application Code
Go Runtime
Operating System

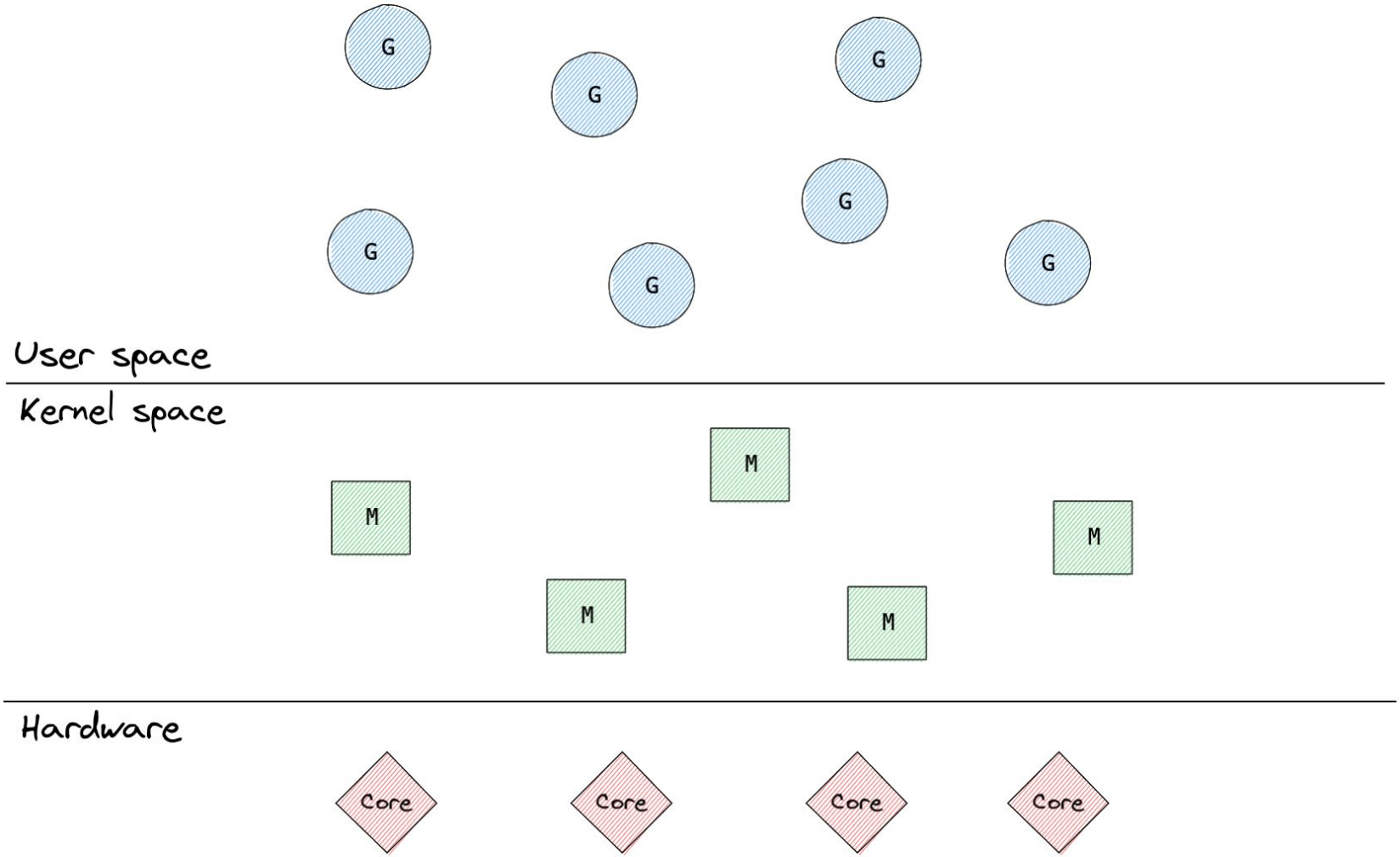
Let’s actually run our code.

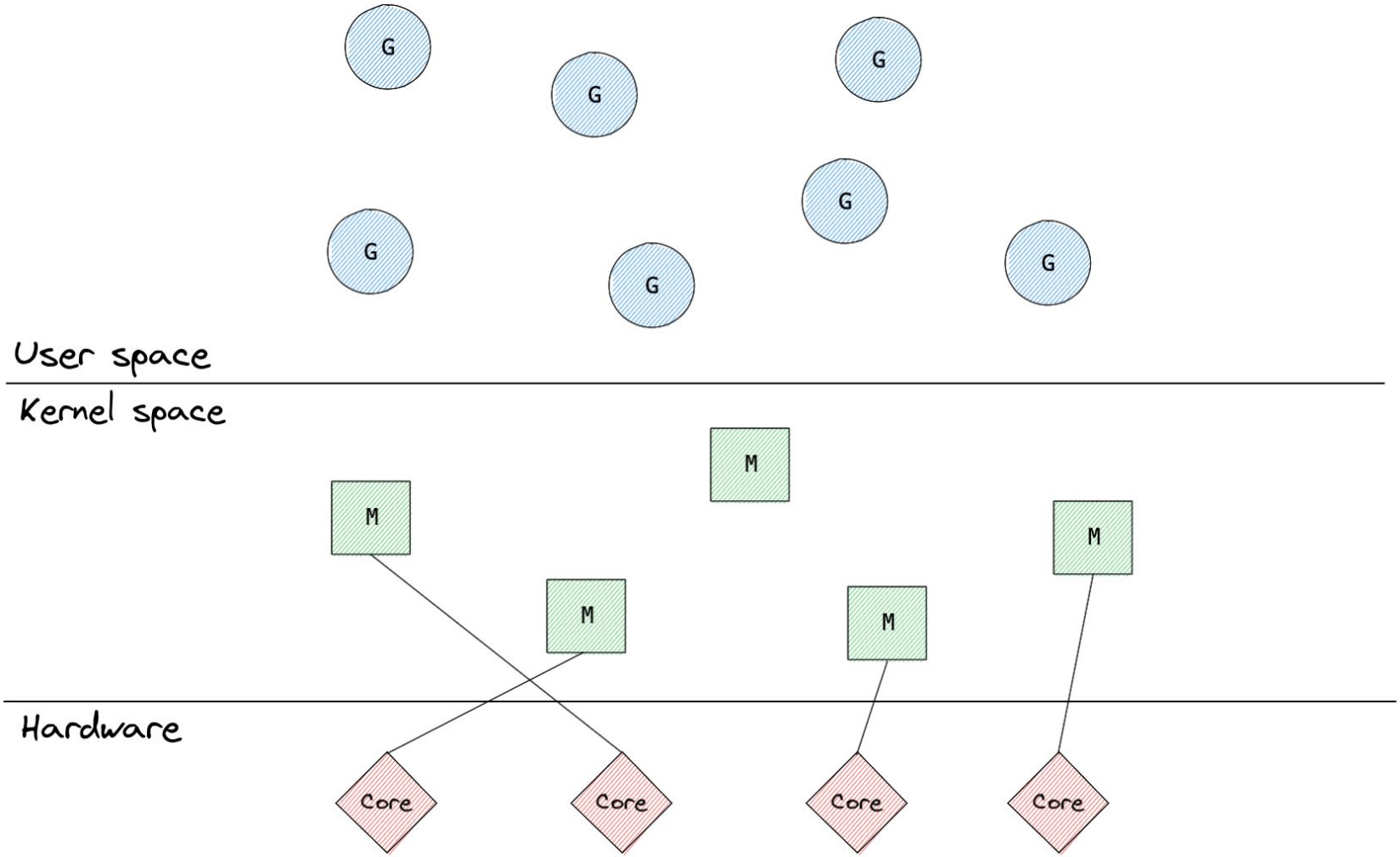
```
./app
```


G
G
G
G
G
G


G
G
G
G
G
G
User space


G
G
G
G
G
G
G
User space
Kernel space
M
M
M
M
M


G
G
G
G
G
G
G
User space
Kernel space
M
M
M
M
M
Hardware
Core
Core
Core
Core

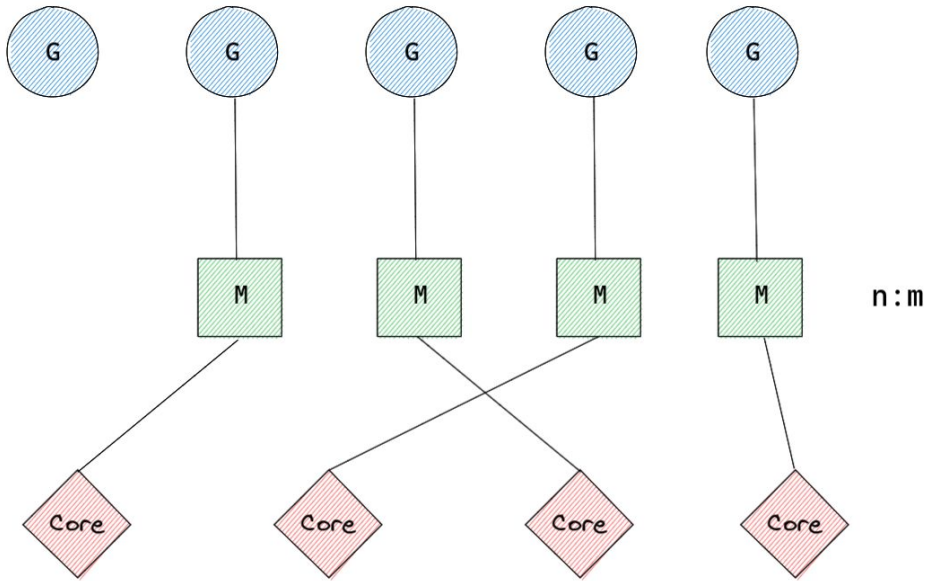

G
G
G
G
G
G
G
User space
Kernel space
M
M
M
M
M
Hardware
Core
Core
Core
Core

How do we get the code “inside” Goroutines to actually run on our hardware?

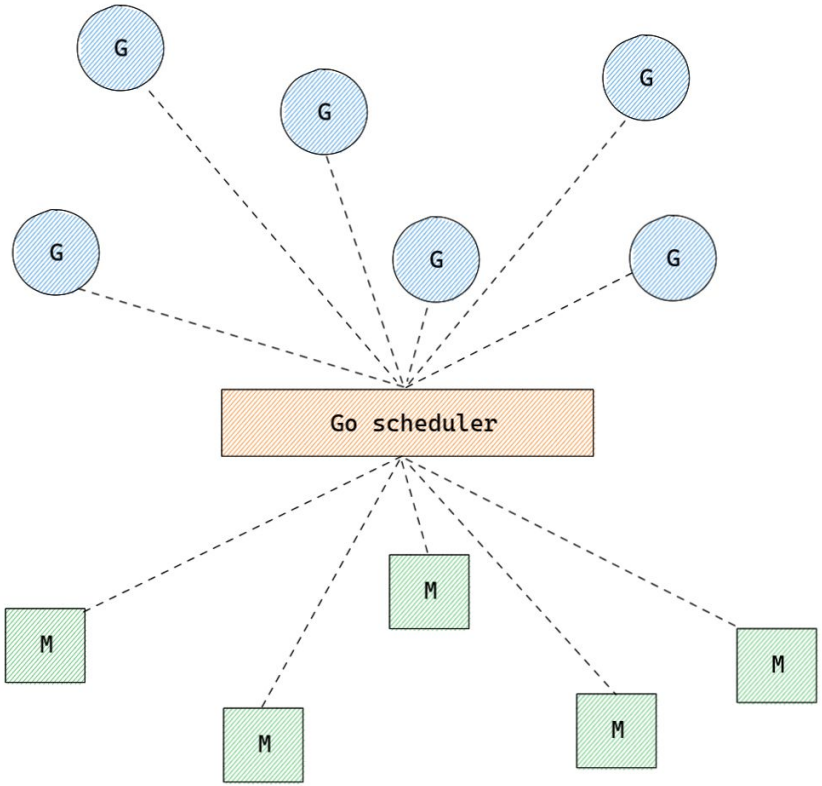
We need some way to map Goroutines to OS threads - user-space scheduling!

n : m scheduling

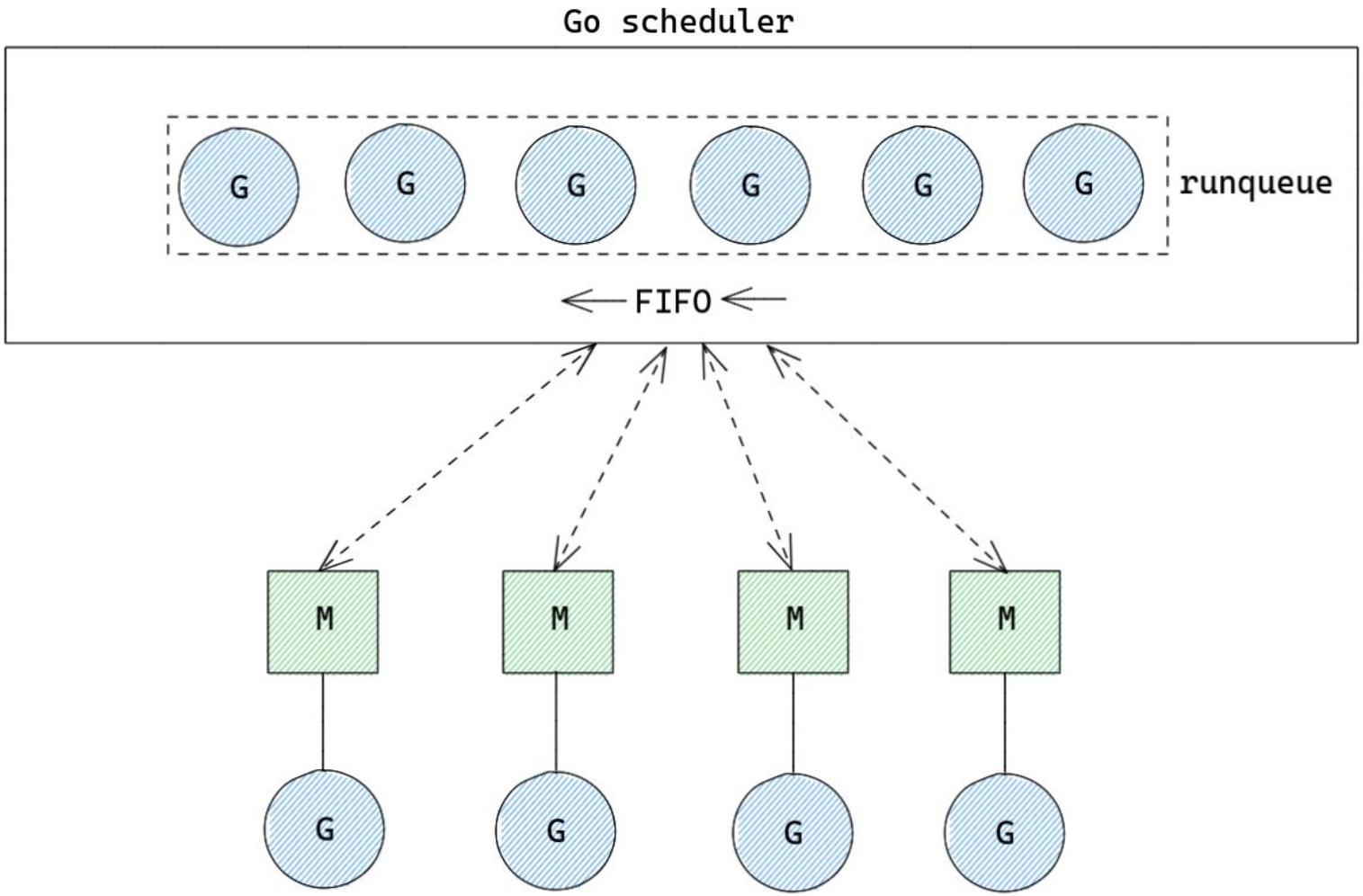
- Increased flexibility
- The number of G's is typically much greater than the number of M's.
- The user-space scheduler multiplexes G's over the available M's.



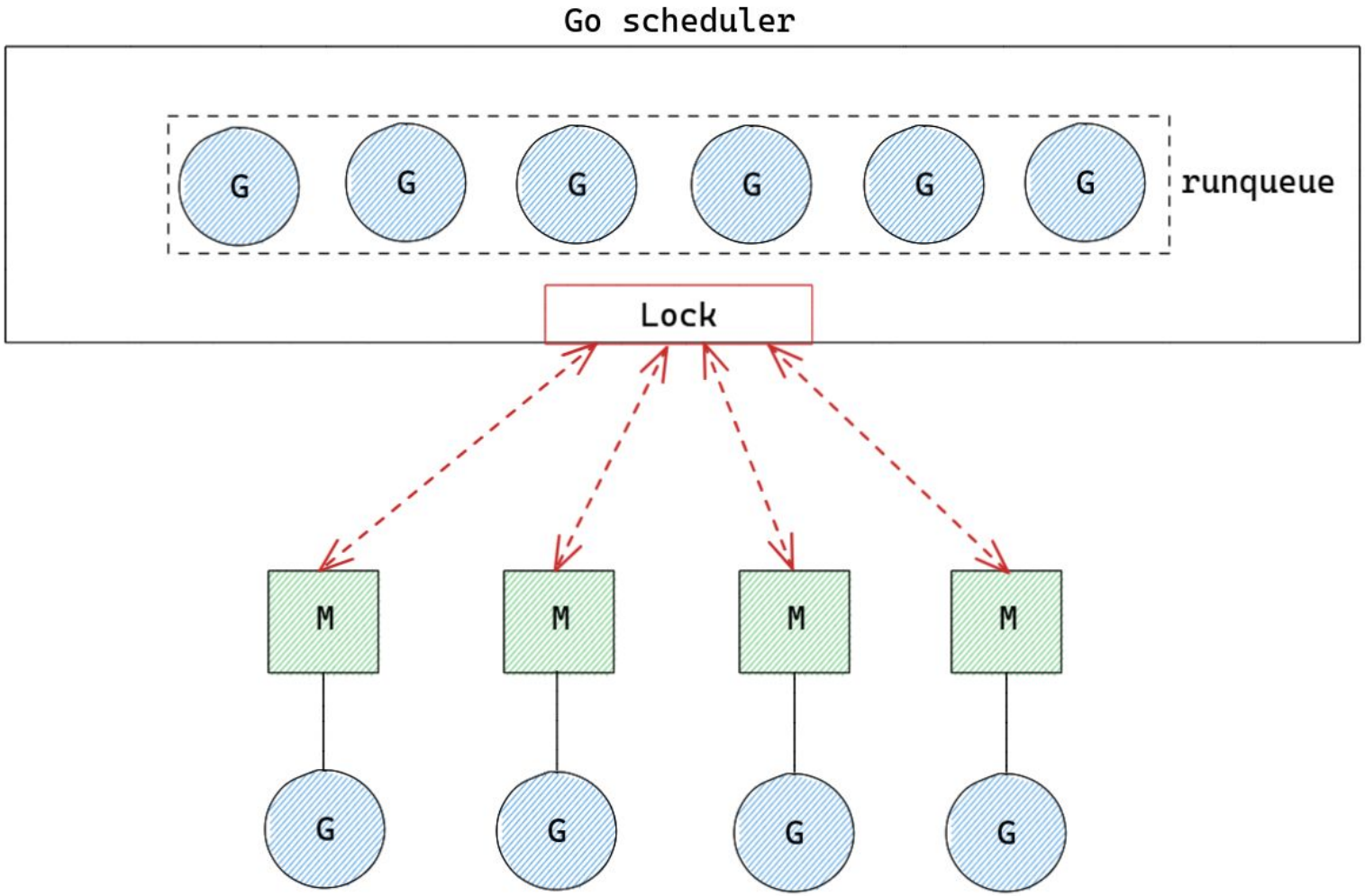
The Go scheduler does *N-M scheduling*.



How do we keep track of Goroutines that are yet to be run?


Go scheduler
G
G
G
G
G
G
runqueue
FIFO
M
M
M
M
G
G
G
G

What are the different considerations we have with the current state?

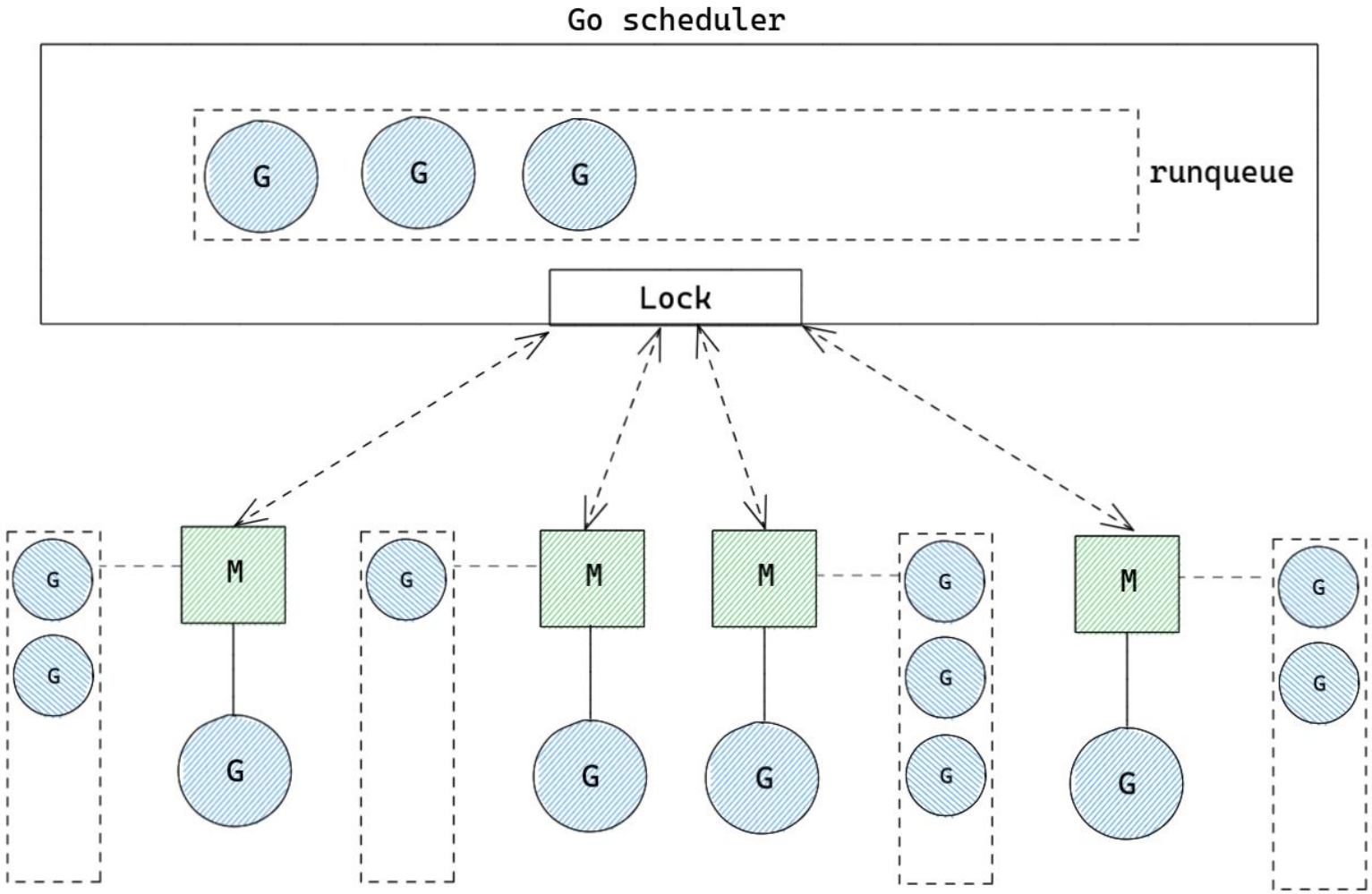

Go scheduler
G
G
G
G
G
G
runqueue
Lock
M
M
M
M
G
G
G
G

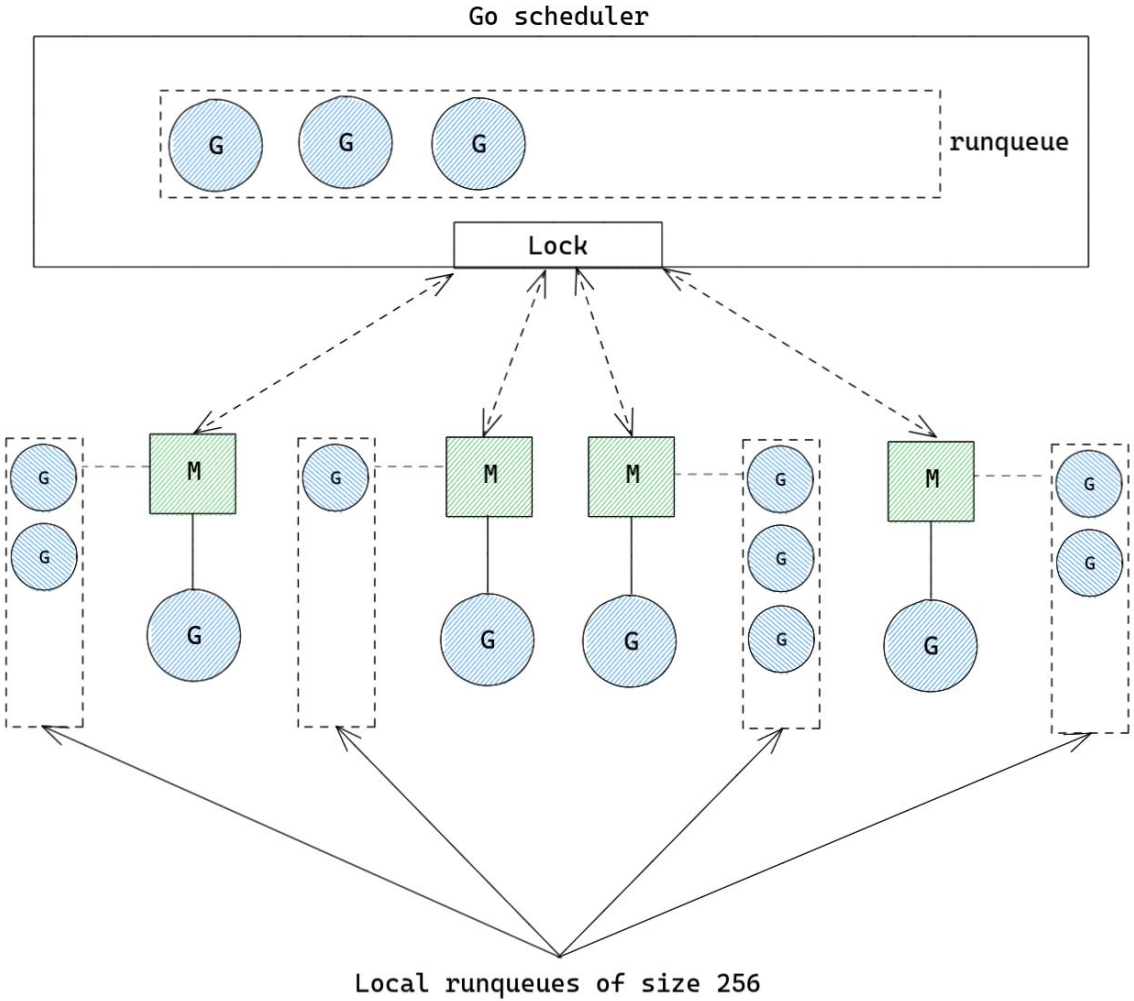
What if the running Goroutines happen to be long-running tight loops?

It is likely that the ones in the `runqueue` will end up starving.

- We could try time-multiplexing.
  - But then with just one global runqueue, each Goroutine would end up getting a short slice of time - leading to poor locality and excessive context switching.

To begin addressing these challenges, the notion of *distributed* `runqueue`s was introduced.


Go scheduler
G
G
G
runqueue
Lock
M
M
M
M
G
G
G
G
G
G
G
G
G
G
G
G
G
G
G


Go scheduler
G
G
G
runqueue
Lock
M
M
M
M
G
G
G
G
G
G
G
G
G
G
G
G
Local runqueues of size 256

Interlude: `GOMAXPROCS`

"The GOMAXPROCS variable limits the number of operating system threads that can execute user-level Go code simultaneously. There is no limit to the number of threads that can be blocked in system calls on behalf of Go code; those do not count against the GOMAXPROCS limit."

https://pkg.go.dev/runtime

Interlude: `GOMAXPROCS`

- This implies that there could be a significantly large number of threads that are blocked and not actively executing Goroutines.
- But we maintain some amount of per-thread state (the local `runqueue` being part of it).
  - We don't really need all this state for threads that are not actually executing code.
- If we were to implement work-stealing in order to re-balance load, the number of threads to check would be unbounded.

How can we tackle this?

Interlude: `GOMAXPROCS`

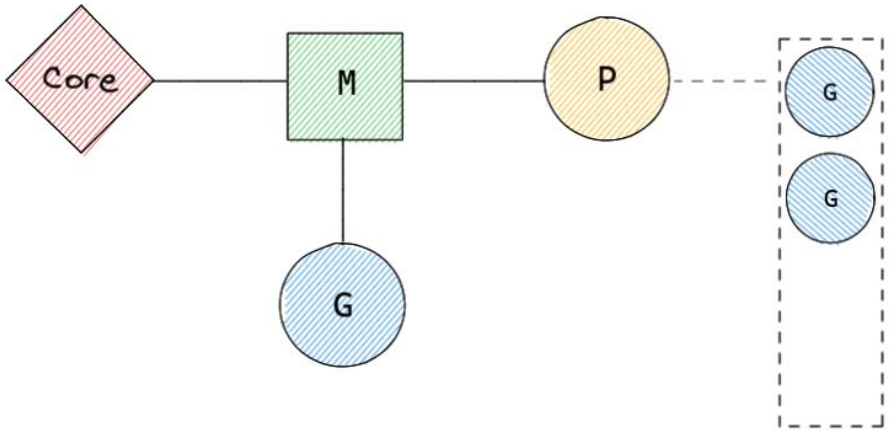
- This implies that there could be a significantly large number of threads that are blocked and not actively executing Goroutines.
- But we maintain some amount of per-thread state (the local `runqueue` being part of it).
  - We don't really need all this state for threads that are not actually executing code.
- If we were to implement work-stealing in order to re-balance load, the number of threads to check would be unbounded.

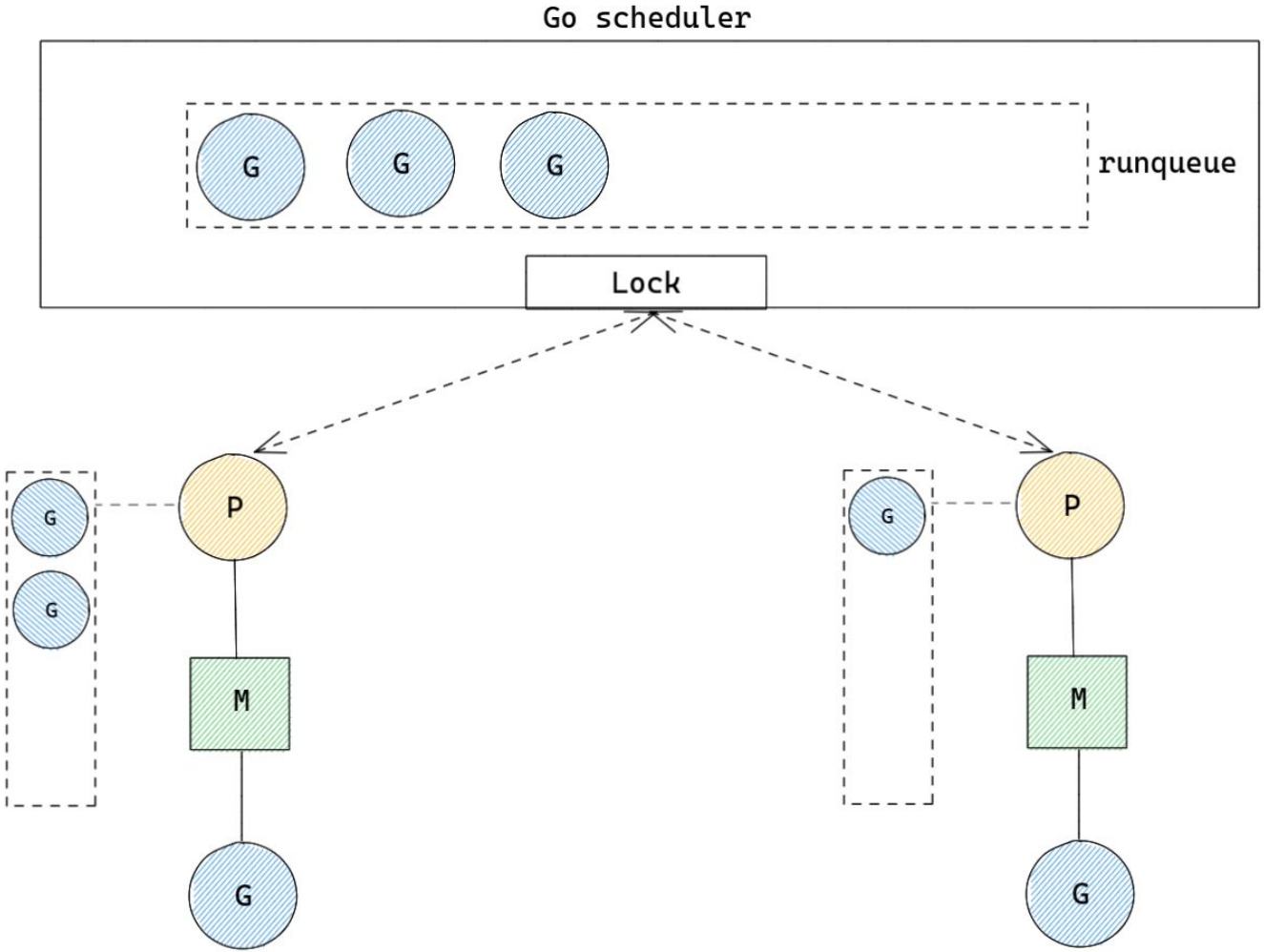
How can we tackle this?

✨Indirection✨

Introducing `p` - processor

- `p` is a heap-allocated data structure that is used to execute Go code.
- A `p` actively executing Go code has an `m` associated with it.
- Much of the state previously maintained per-thread is now part of `p` - such as local `runqueue`.
    - This addresses our previous two concerns!
- No. of `Ps` = `GOMAXPROCS`

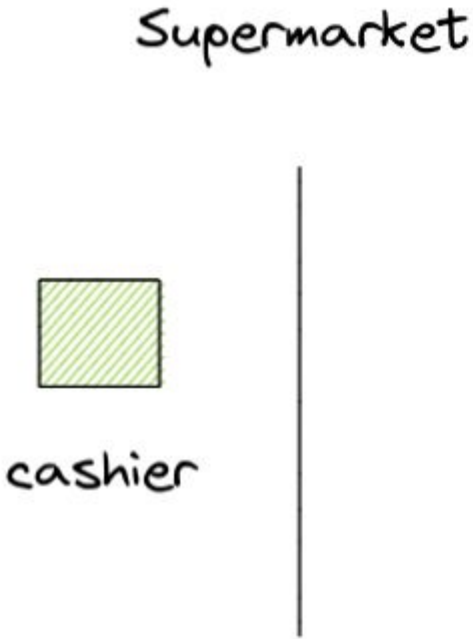


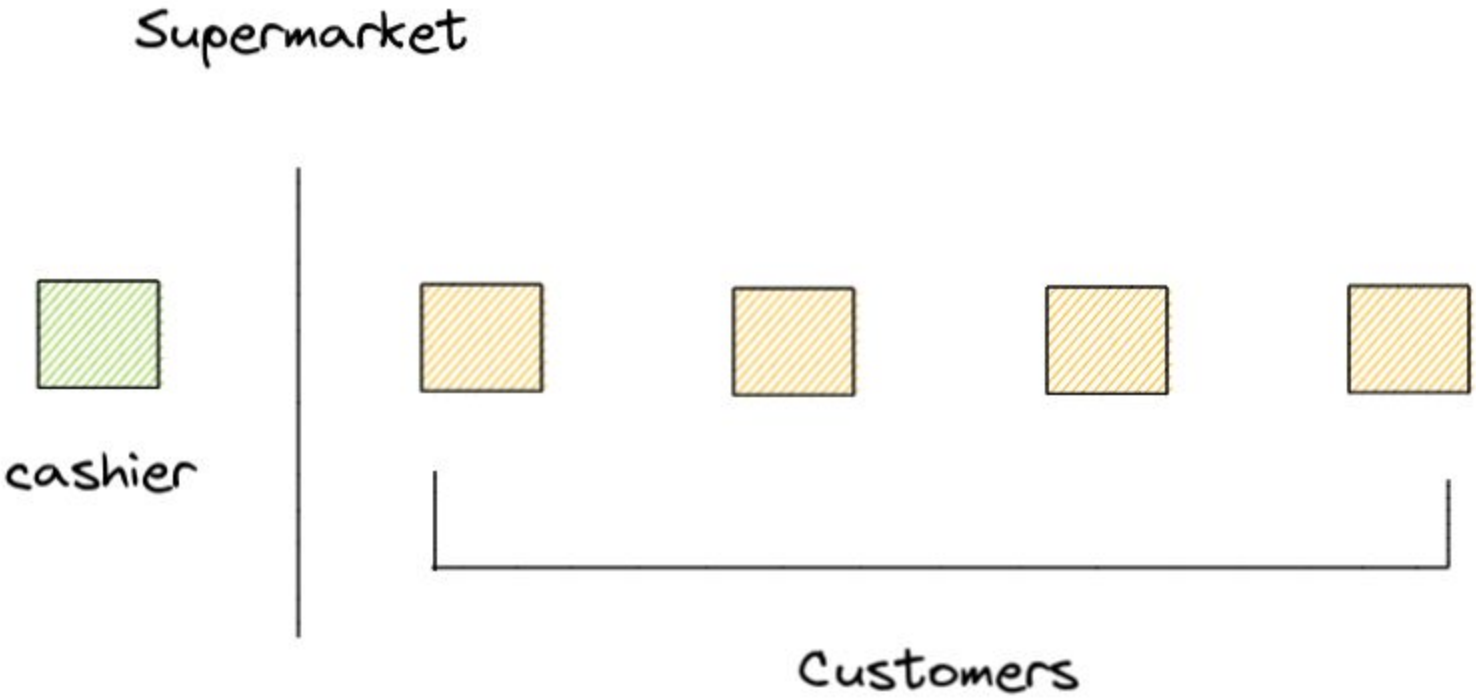

Go scheduler
G
G
G
runqueue
Lock
G
G
P
M
G
G
P
M
G

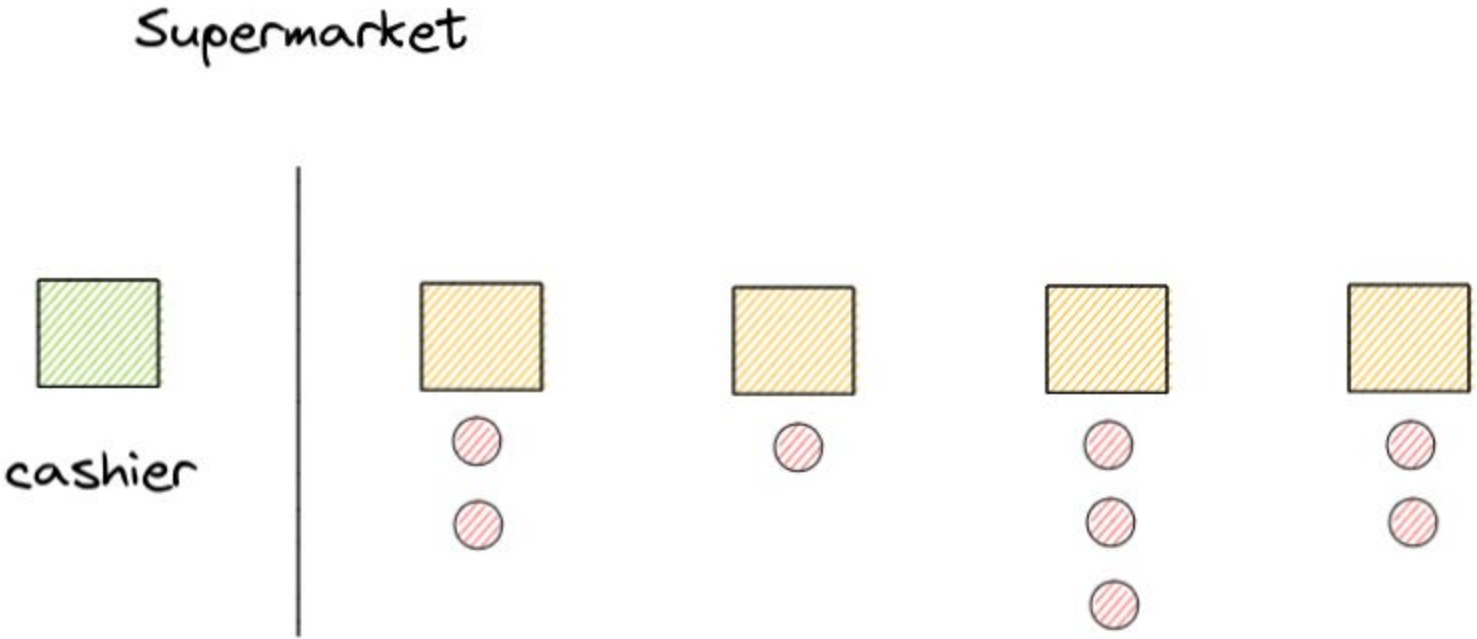
“Go scheduler: Implementing language with lightweight concurrency”
by Dmitry Vyukov

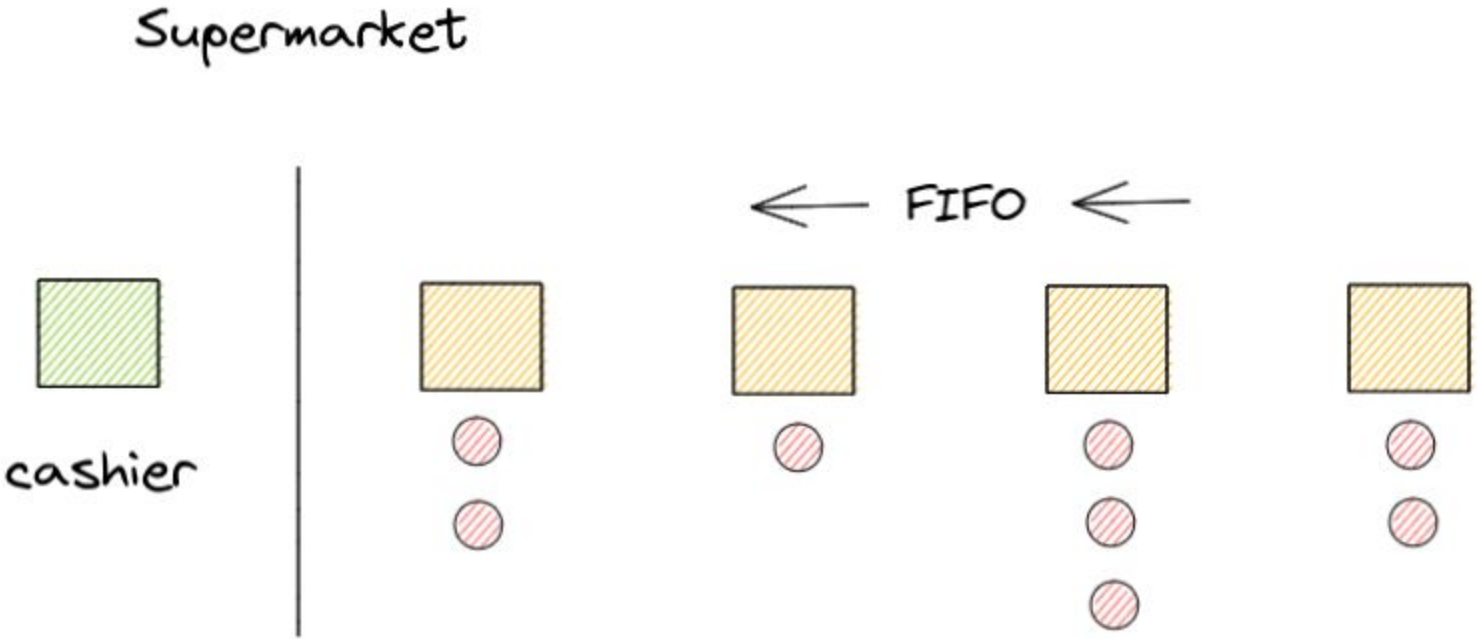
Interlude: Fairness

Let’s consider the following scenario.


Supermarket
cashier

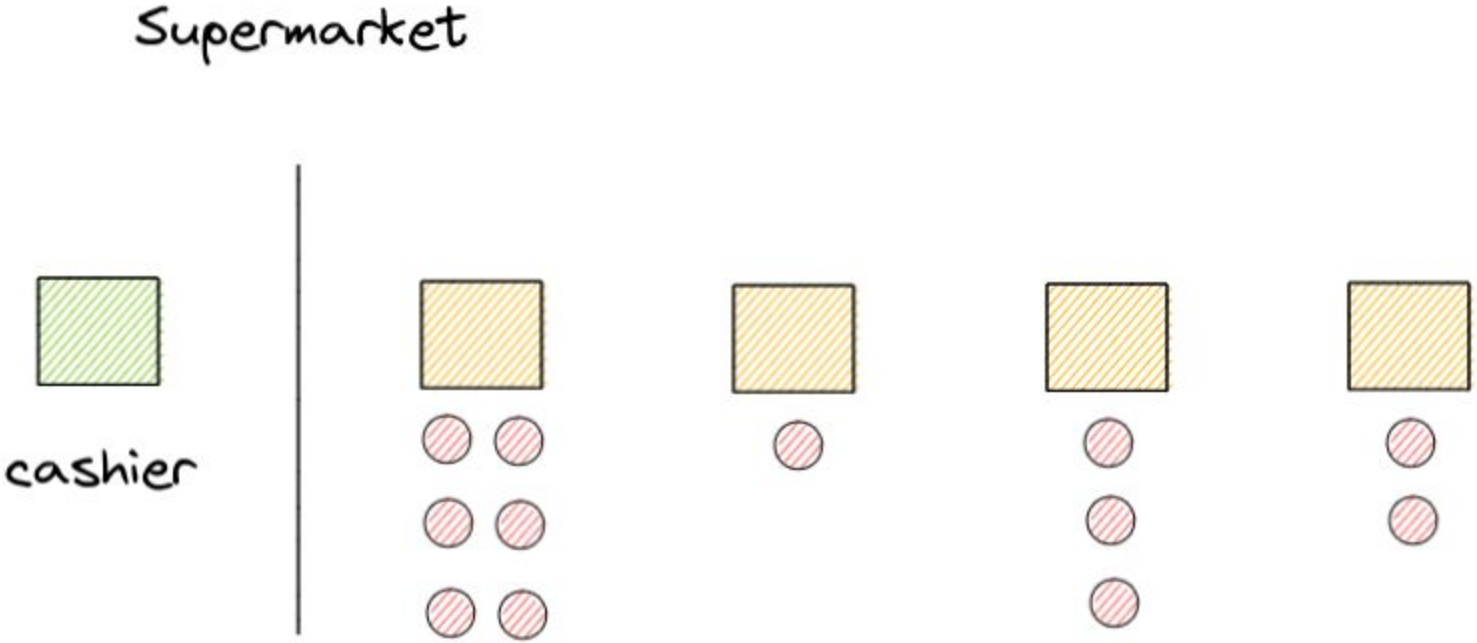

Supermarket
cashier
Customers


Supermarket
cashier


Supermarket
FIFO
cashier

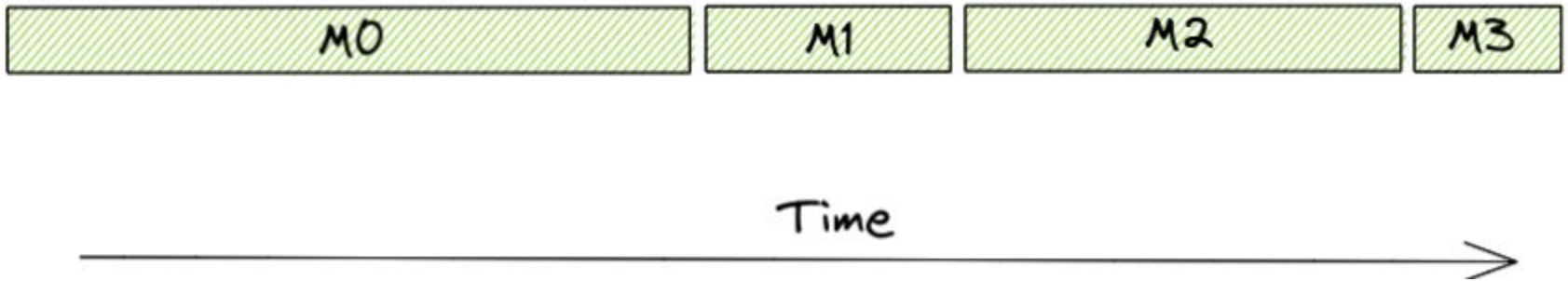
This looks fine.

But, what if...


Supermarket
cashier

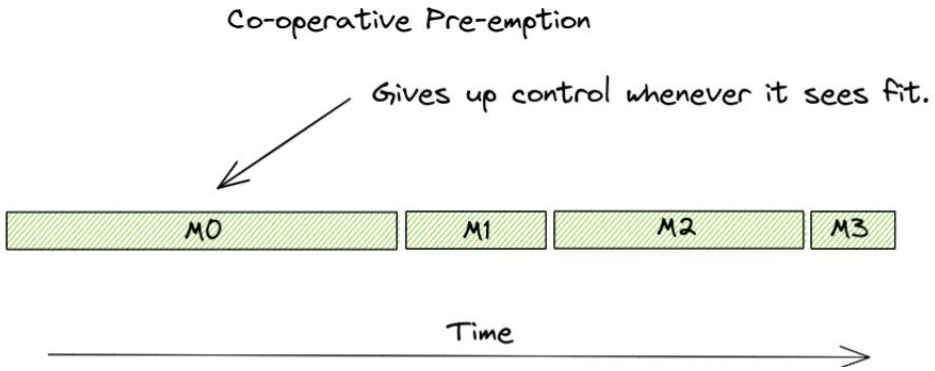
Presence of resource hogs in a FIFO system leads to something known as the *Convoy Effect*.

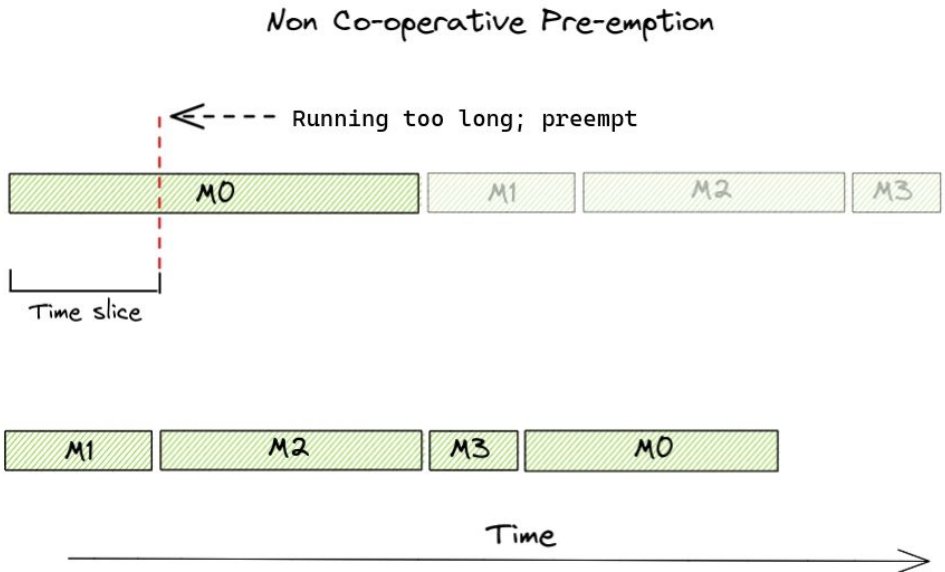
This is a common problem to deal with while considering fairness in scheduling.



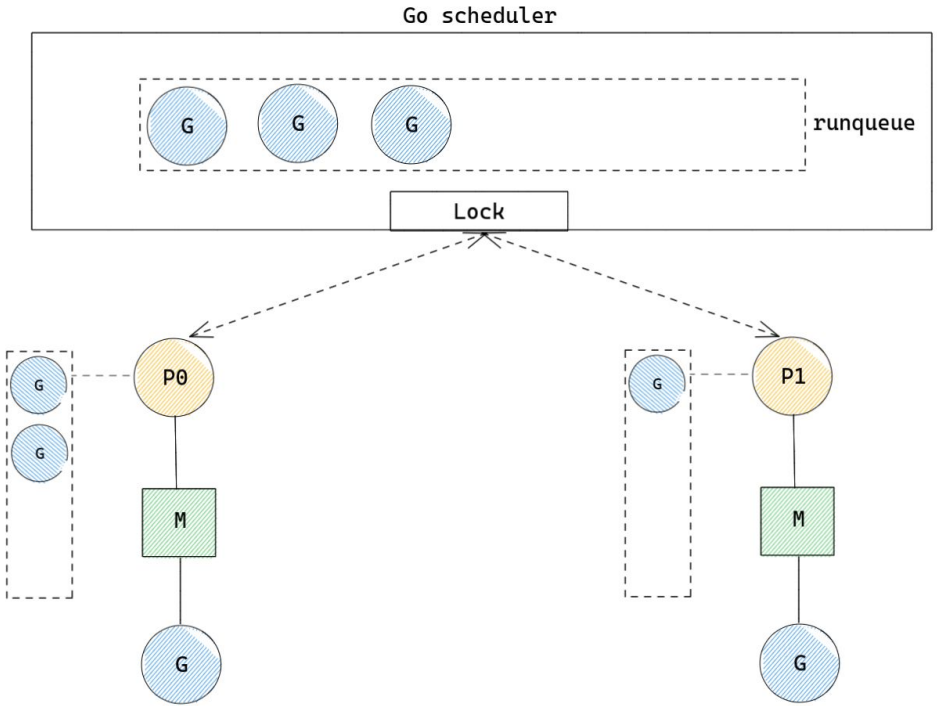
How do we deal with this in scheduling?

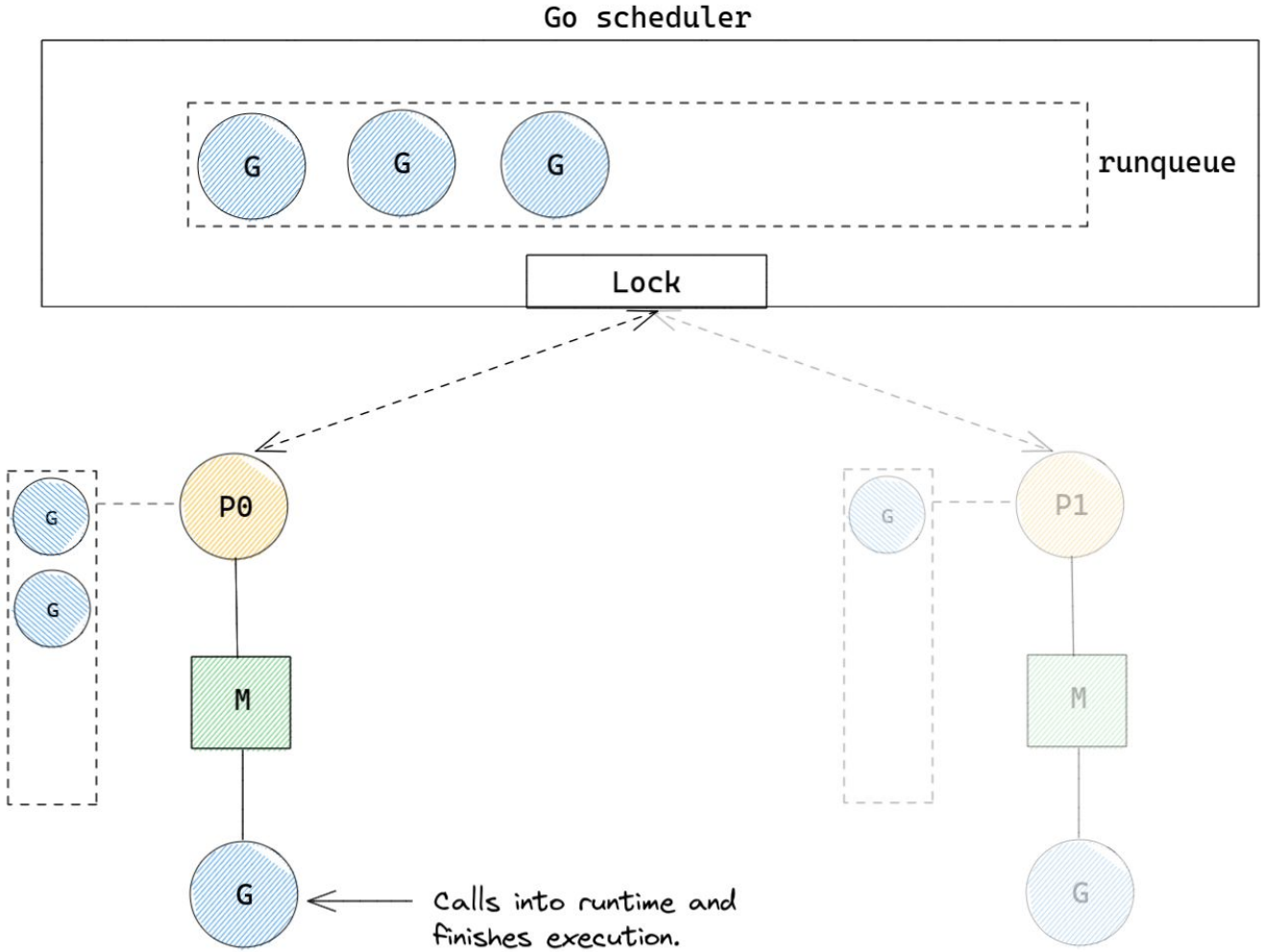
- One way is to just schedule the short running tasks before the long running ones.
  - This would require us knowing what the characteristic of the workload is like.
- Another way - pre-emption!

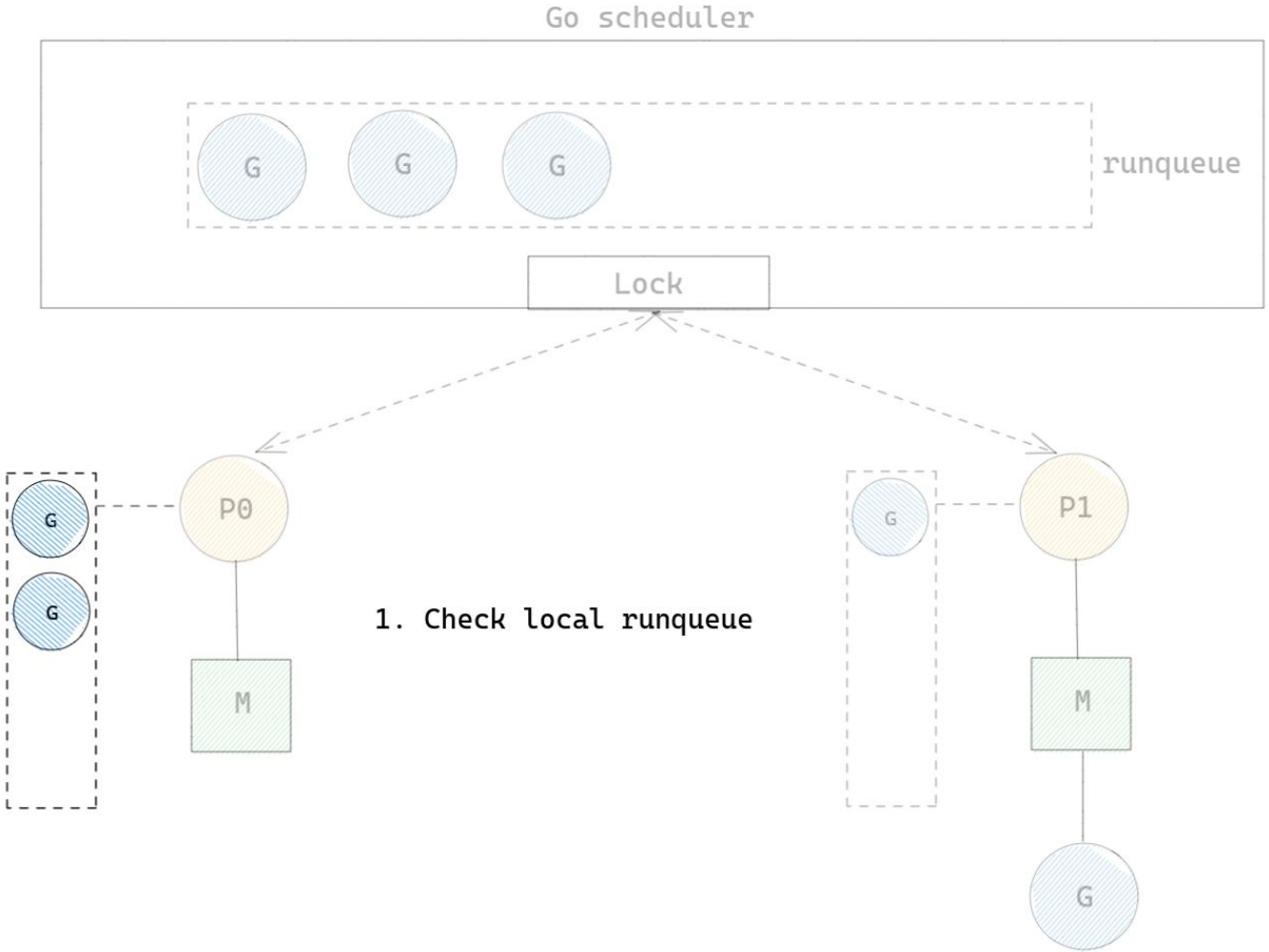


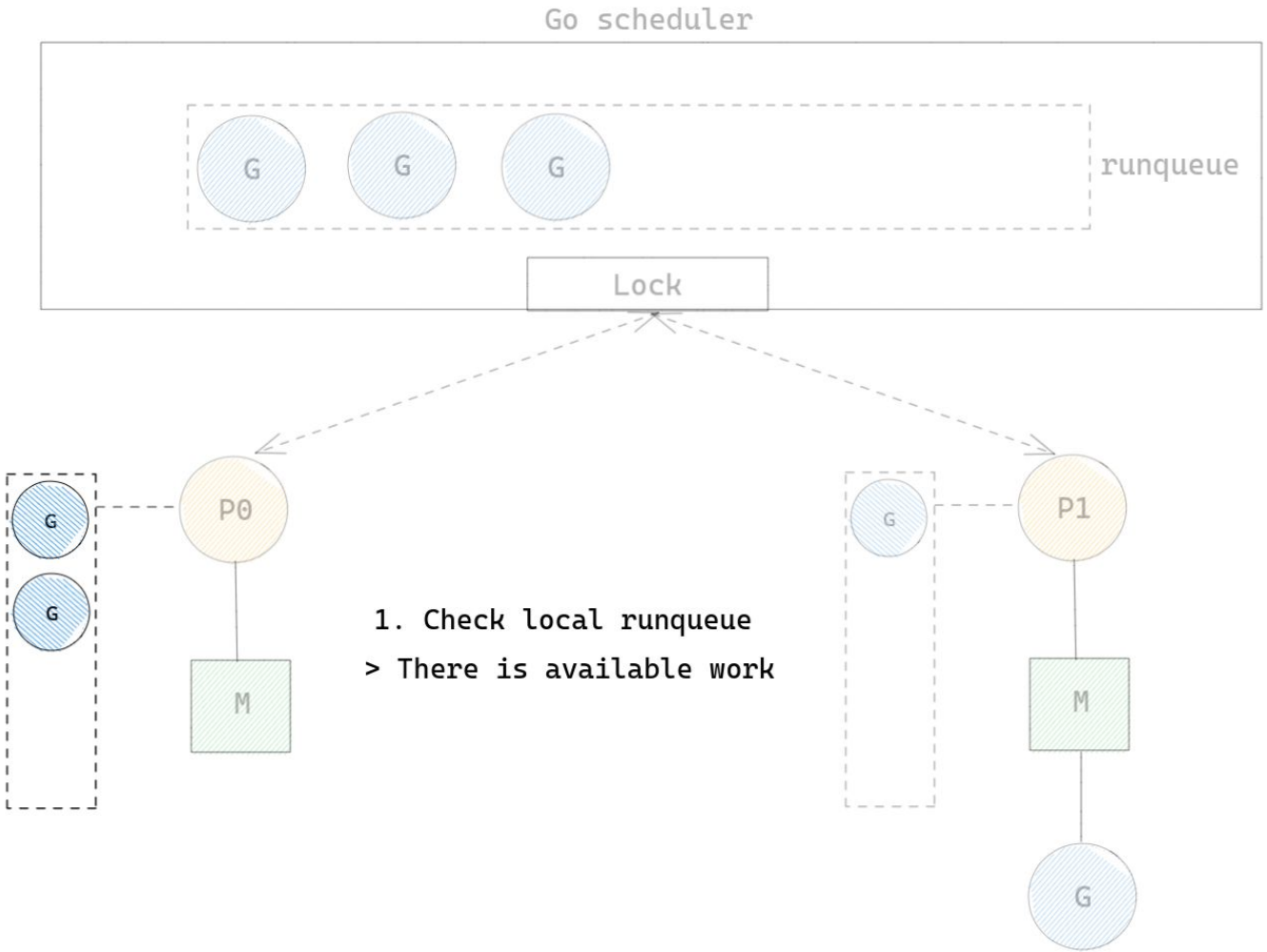


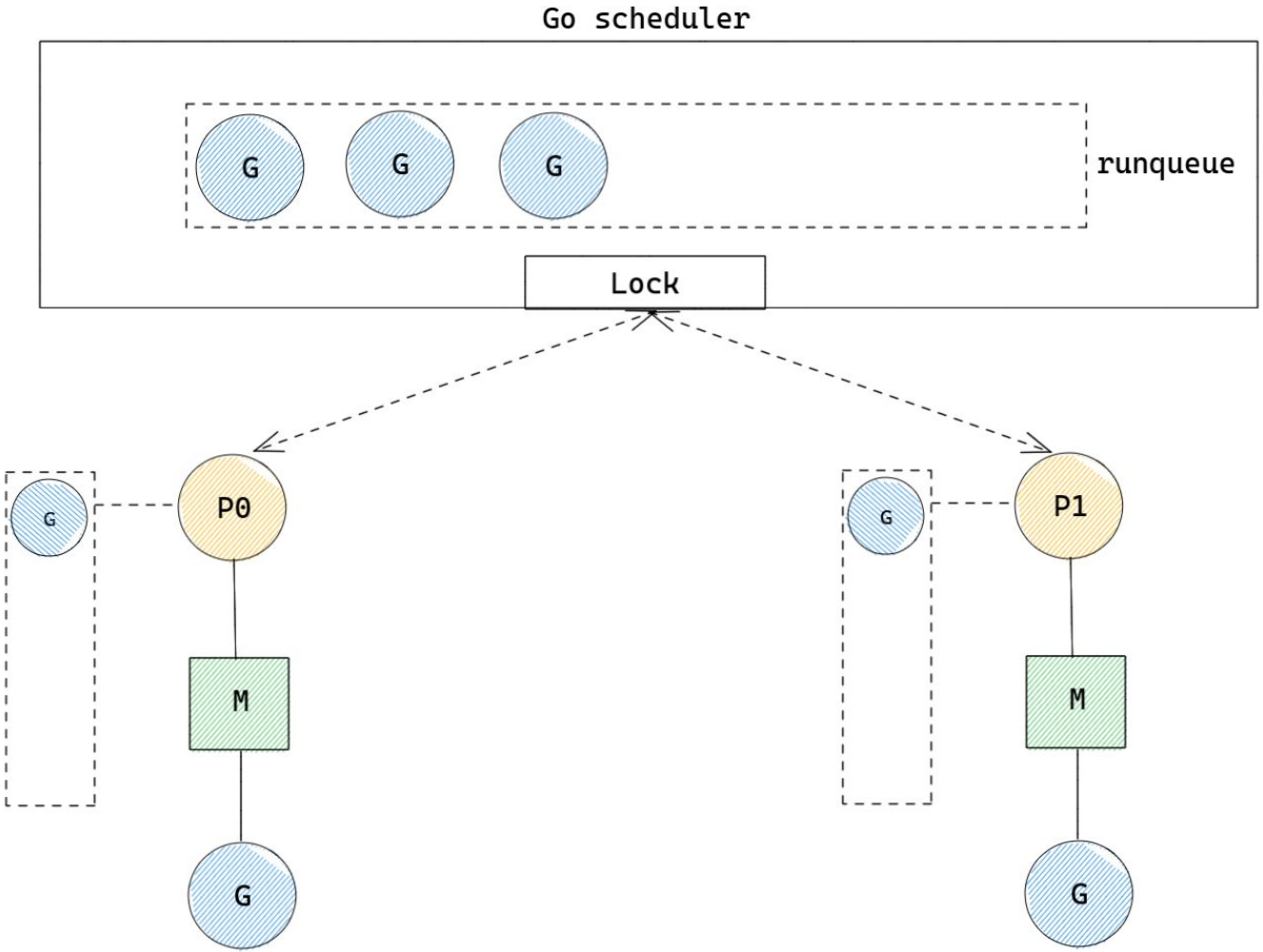
Alright, now that we have enough context, the first thing to ask ourselves is “how do we choose which Goroutine to run?”



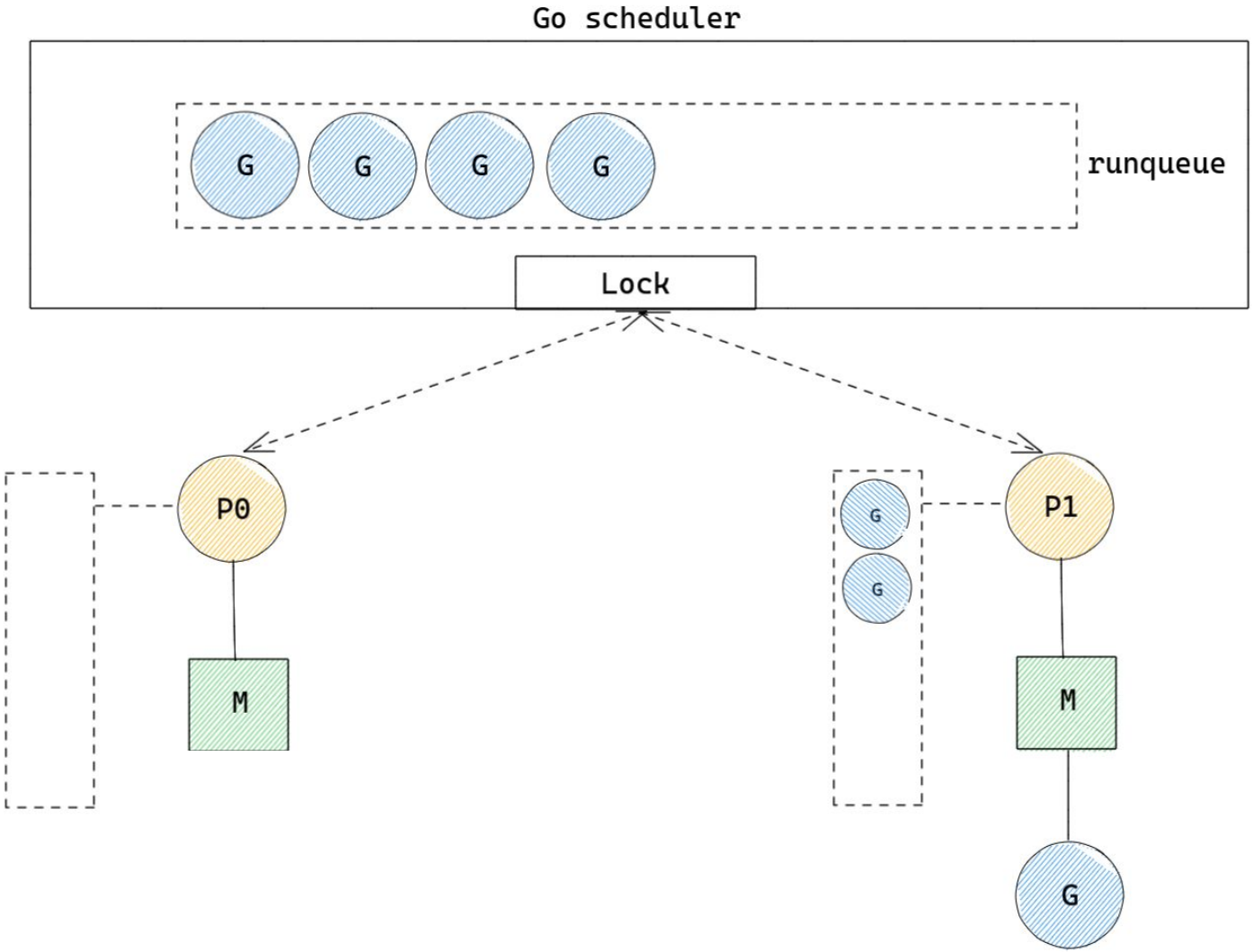

Go scheduler
G
G
G
runqueue
Lock
G
G
P0
M
G
Calls into runtime and
finishes execution.
G
P1
M
G

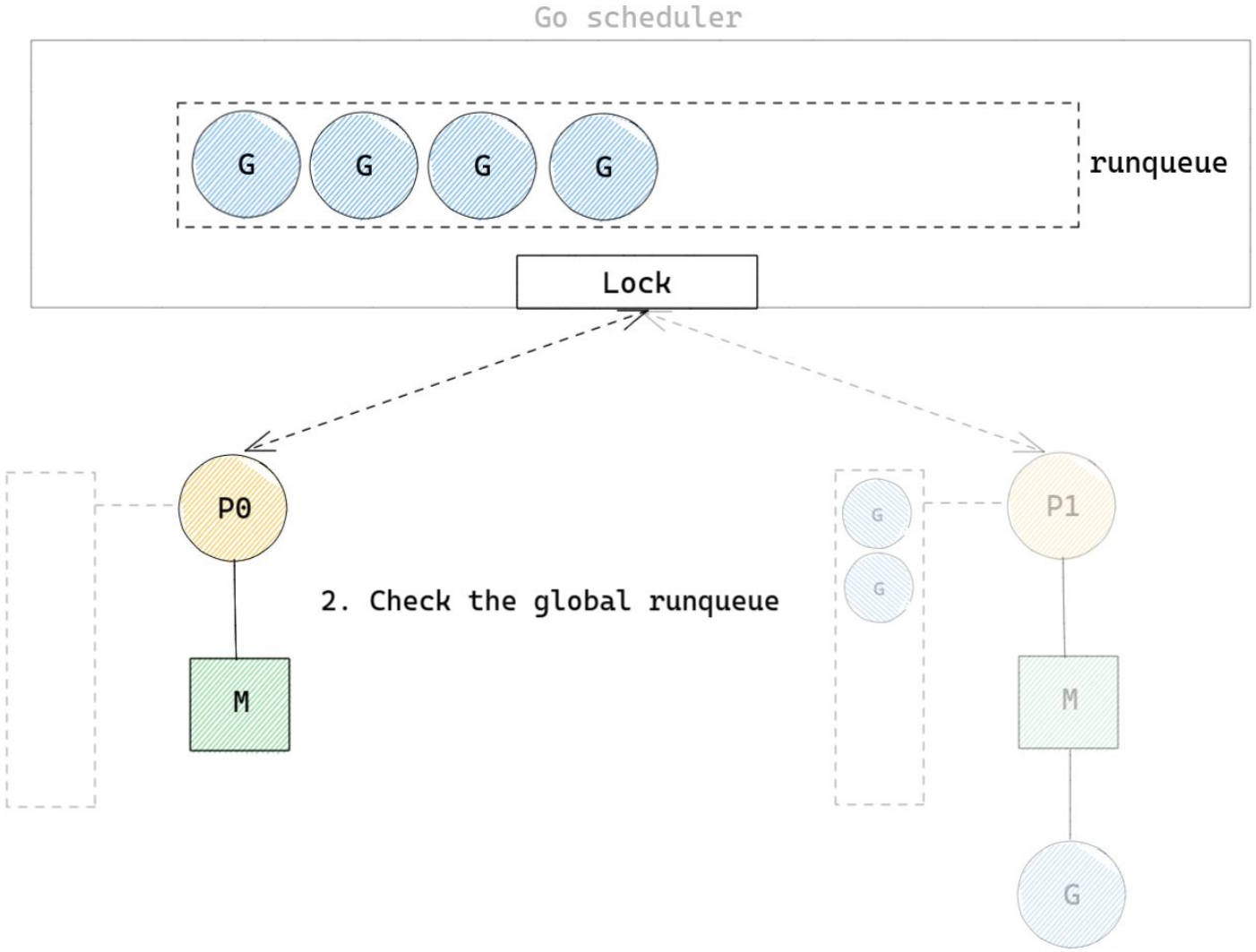

Go scheduler
G
G
G
runqueue
Lock
G
G
P0
M
1. Check local runqueue
G
P1
M
G

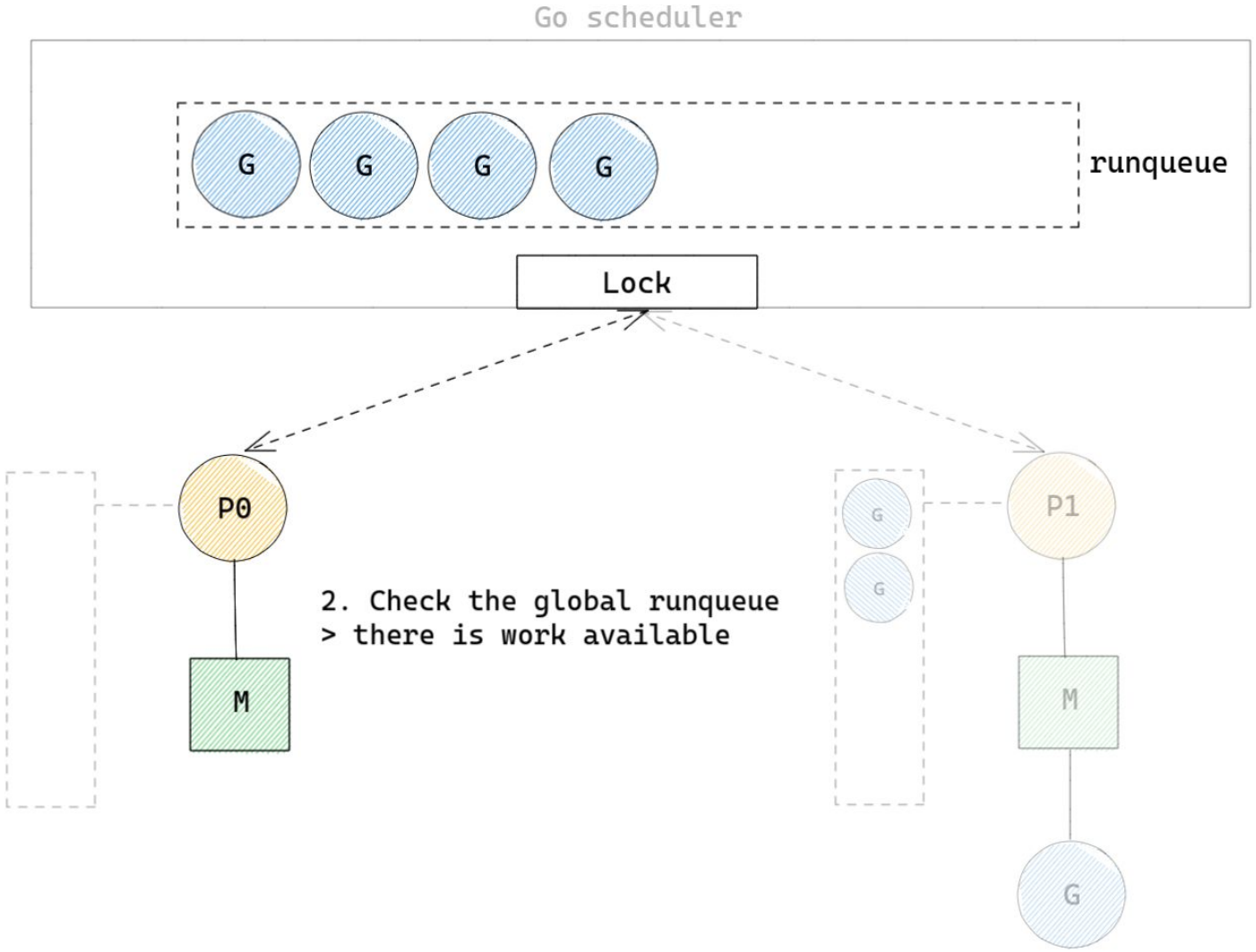

Go scheduler
G
G
G
runqueue
Lock
G
G
P0
M
1. Check local runqueue
> There is available work
G
P1
M
G

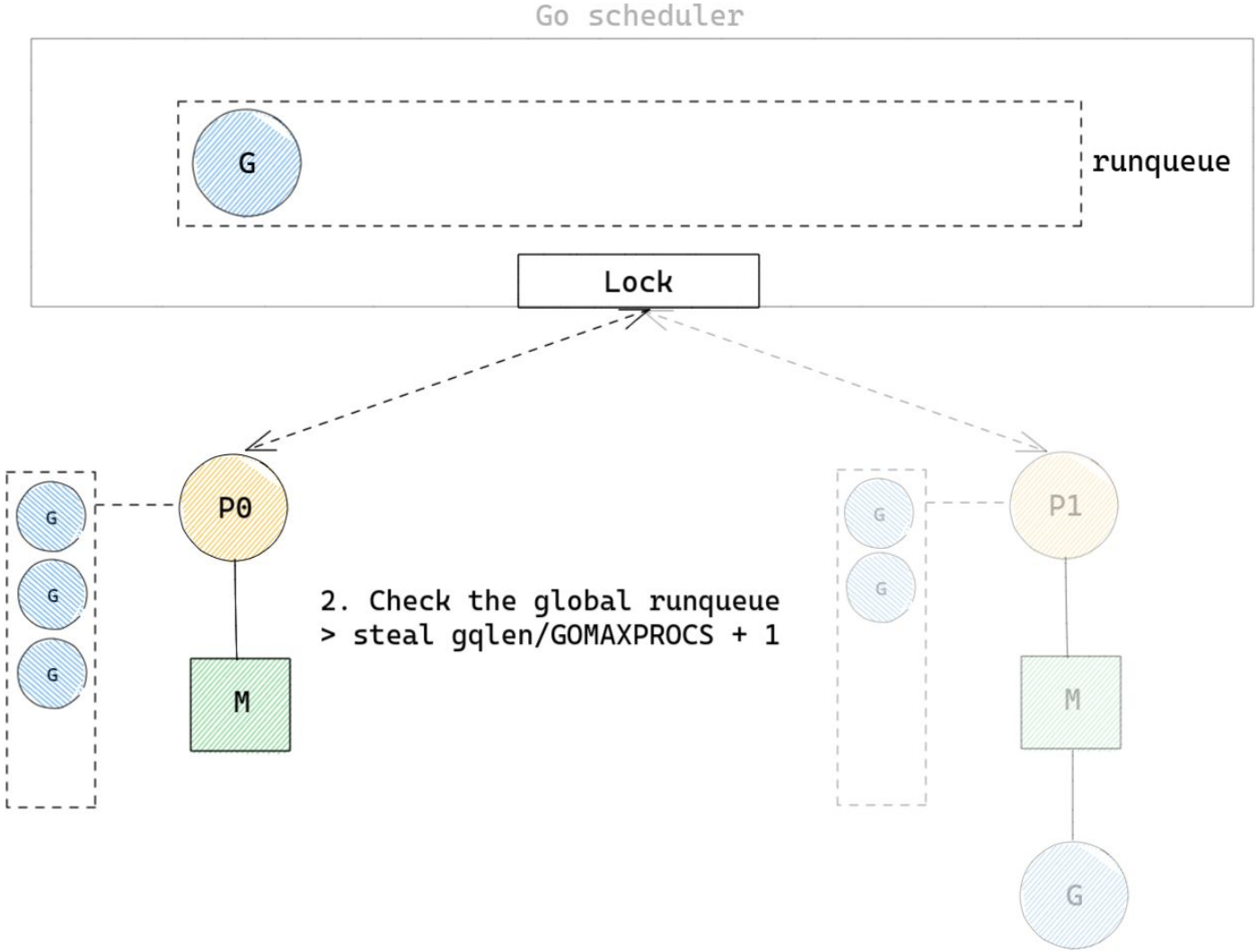

Go scheduler
G
G
G
runqueue
Lock
G
P0
M
G
G
P1
M
G

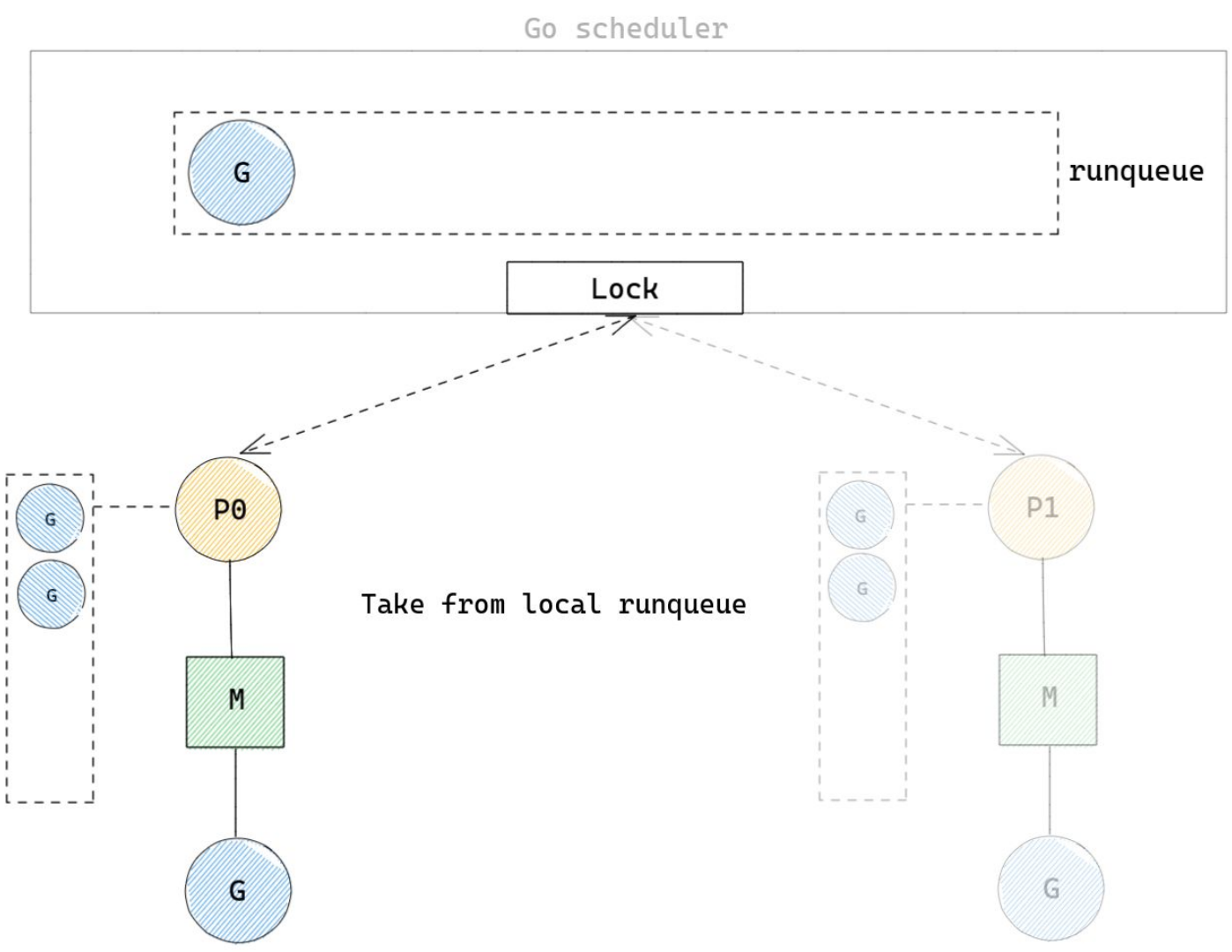
What if the situation is something like this?


Go scheduler
G
G
G
G
runqueue
Lock
P0
M
G
G
P1
M
G

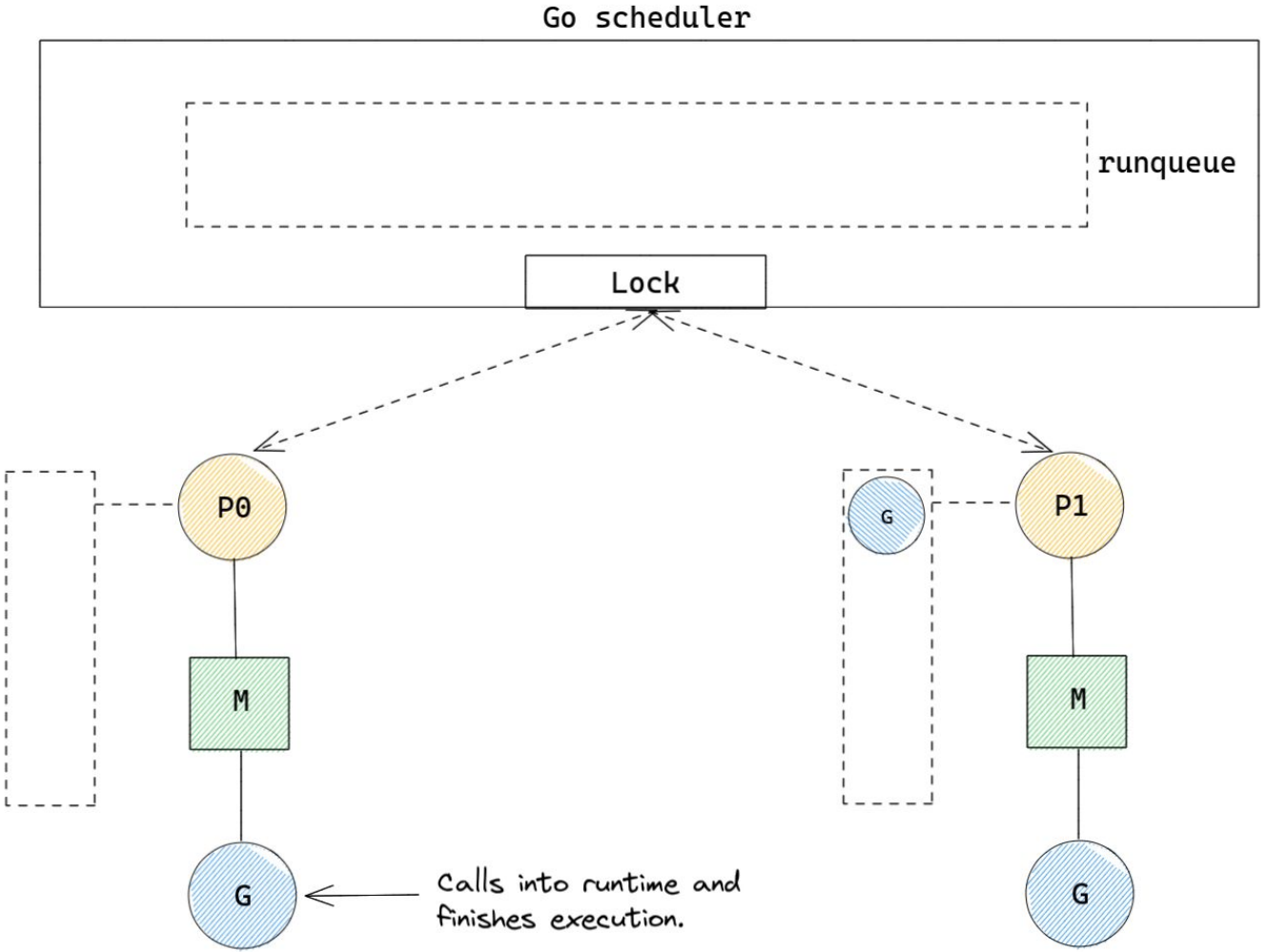

Go scheduler
G
G
G
G
runqueue
Lock
P0
M
2. Check the global runqueue
G
G
P1
M
G

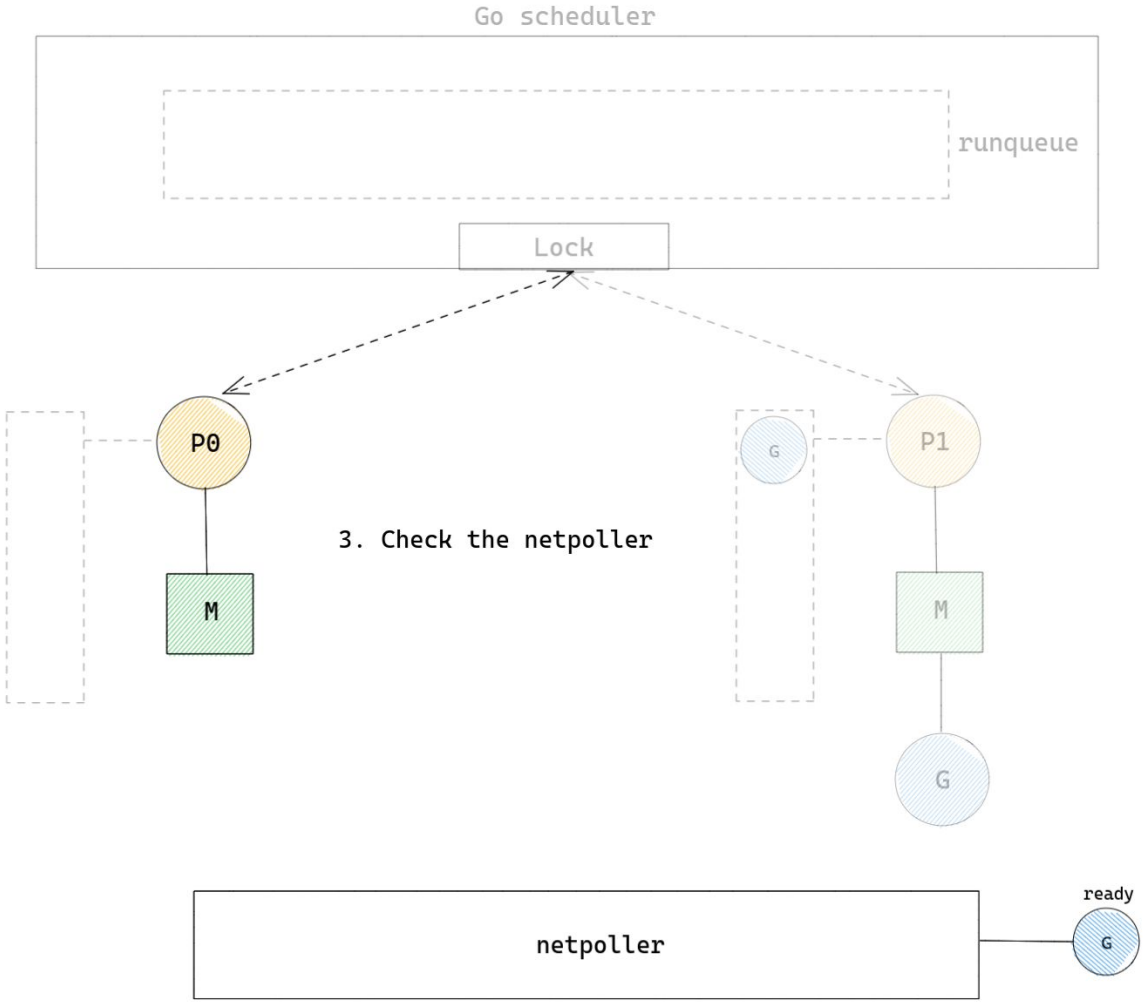

Go scheduler
G
G
G
G
runqueue
Lock
P0
M
2. Check the global runqueue
> there is work available
G
G
P1
M
G

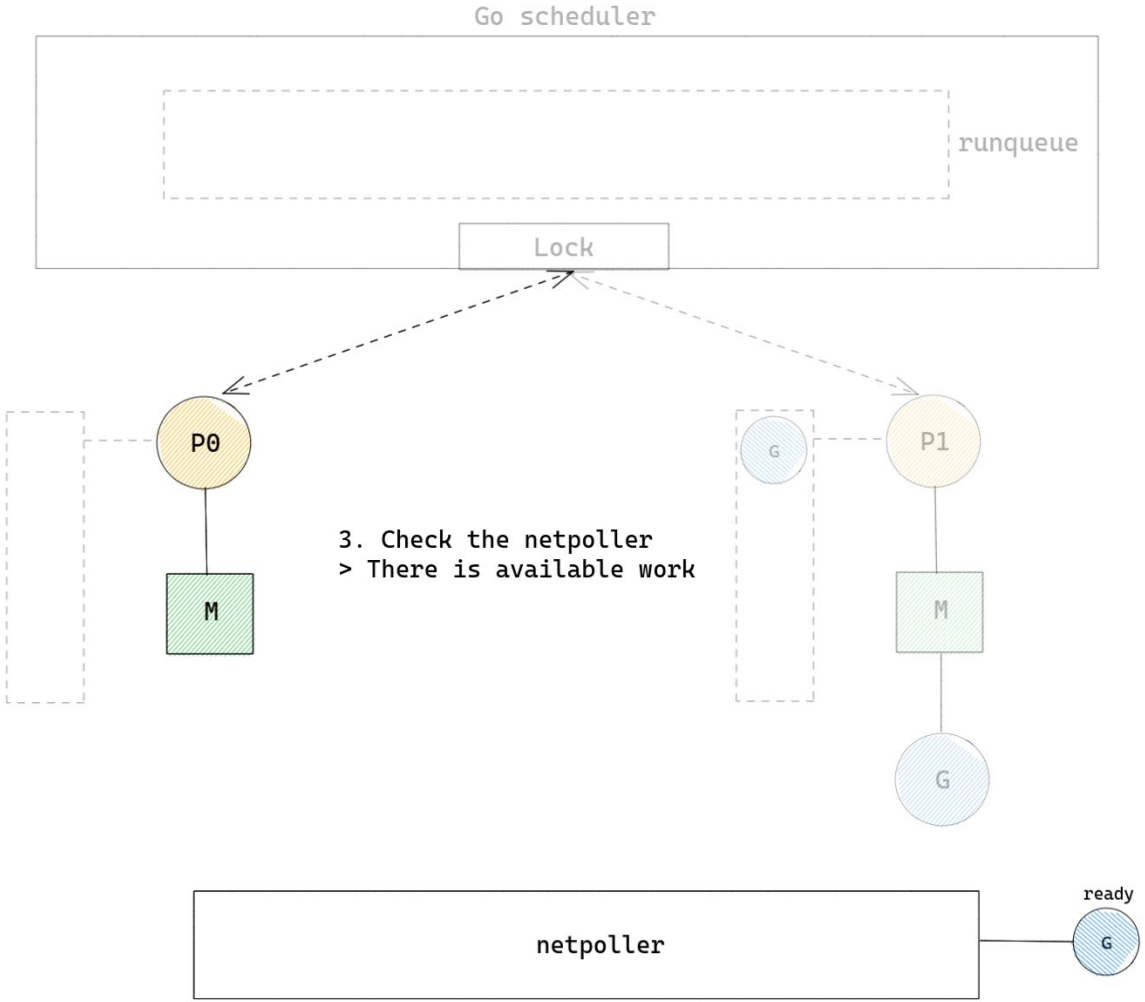

Go scheduler
G
runqueue
Lock
P0
G
G
G
M
2. Check the global runqueue
> steal gqlen/GOMAXPROCS + 1
G
G
P1
M
G

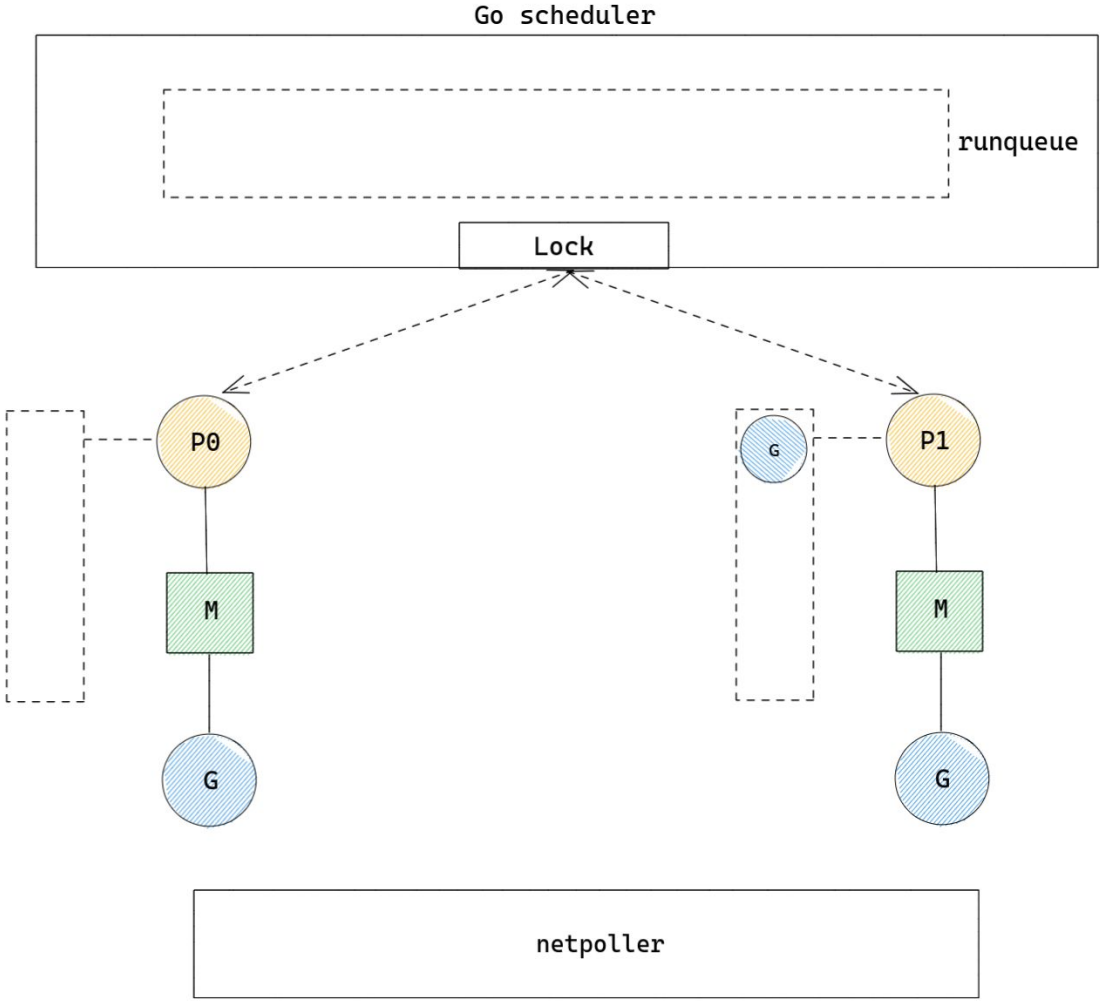

Go scheduler
G
runqueue
Lock
P0
G
G
M
G
Take from local runqueue
P1
G
G
M
G

Global `runqueue` empty too?

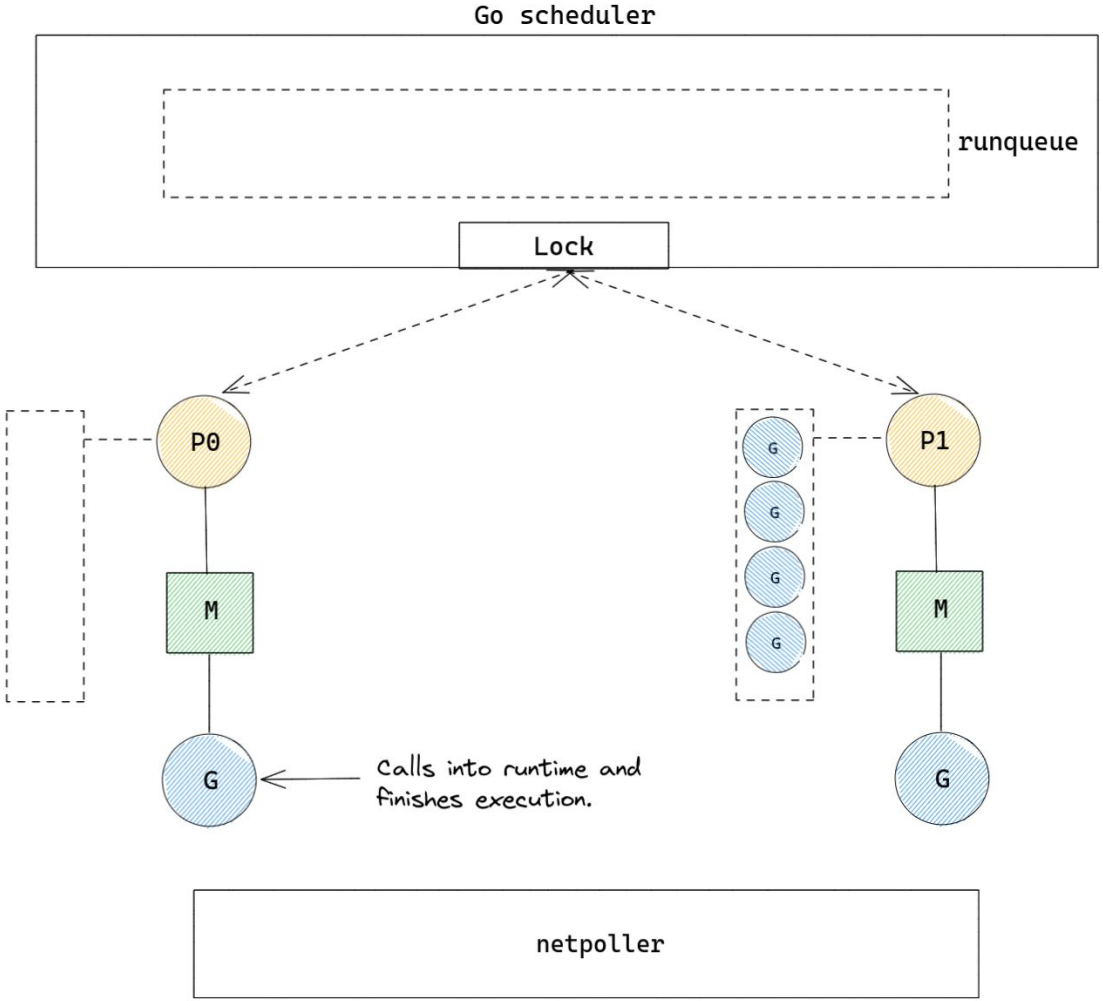

Go scheduler
runqueue
Lock
P0
P1
G
M
M
G
G
Calls into runtime and
finishes execution.

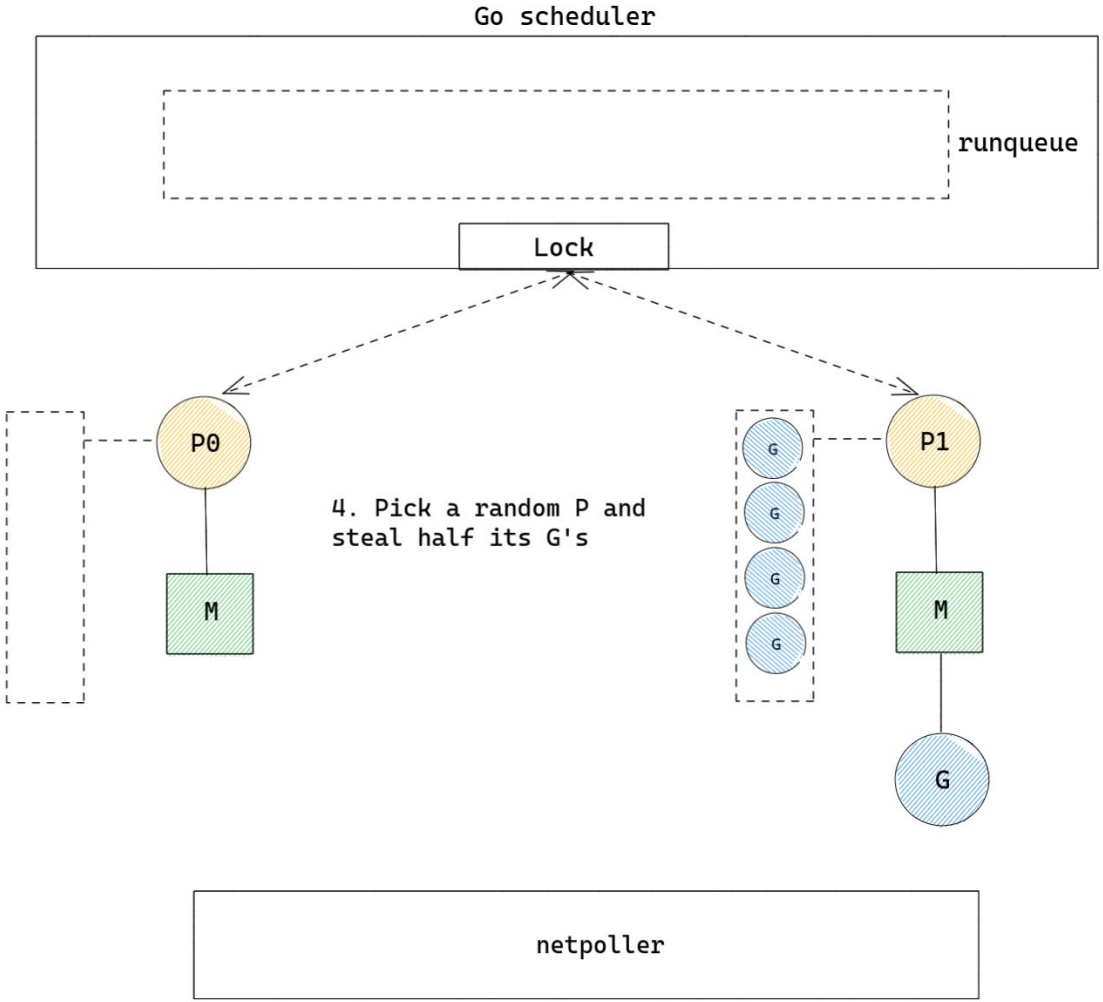

Go scheduler
runqueue
Lock
P0
M
3. Check the netpoller
G
P1
M
G
netpoller
ready
G

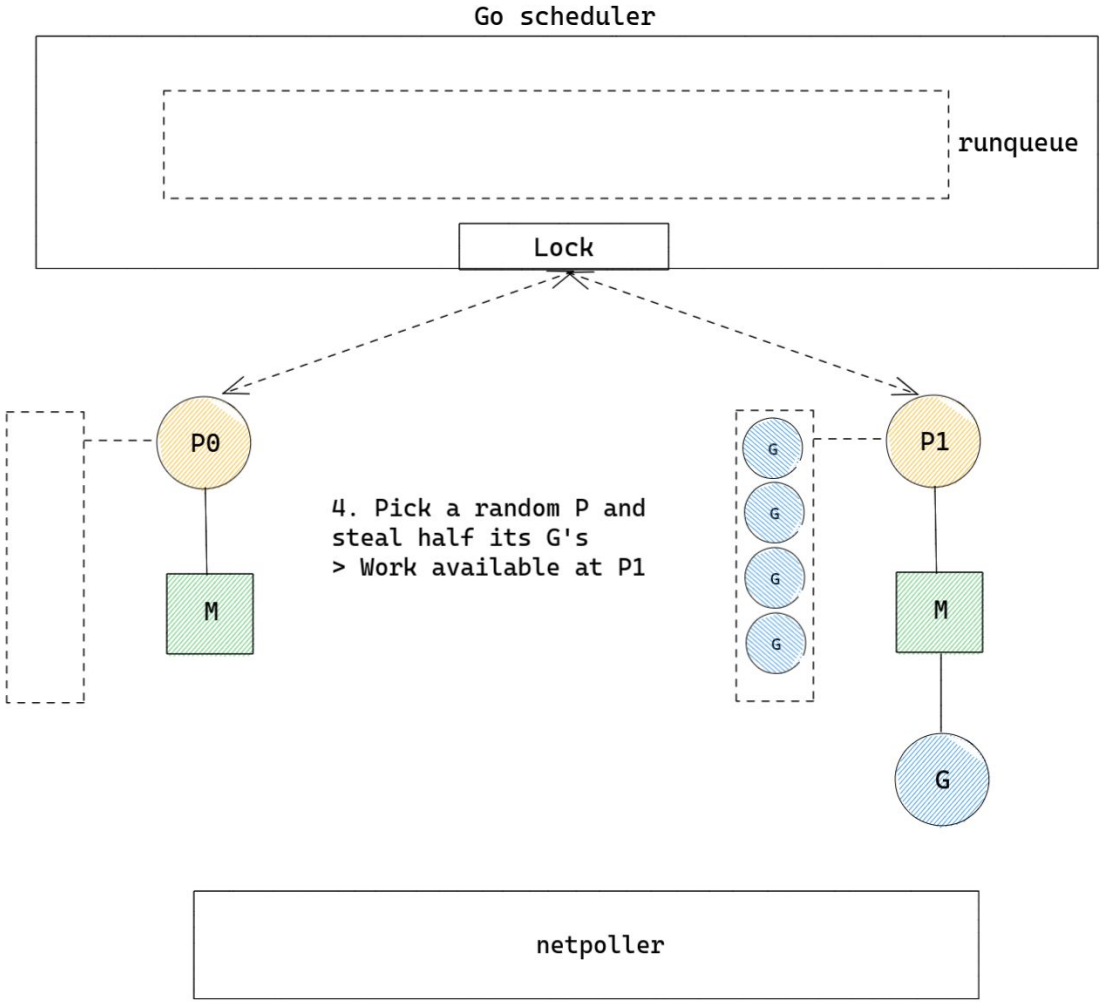

Go scheduler
runqueue
Lock
P0
M
3. Check the netpoller
> There is available work
G
P1
M
G
netpoller
ready
G

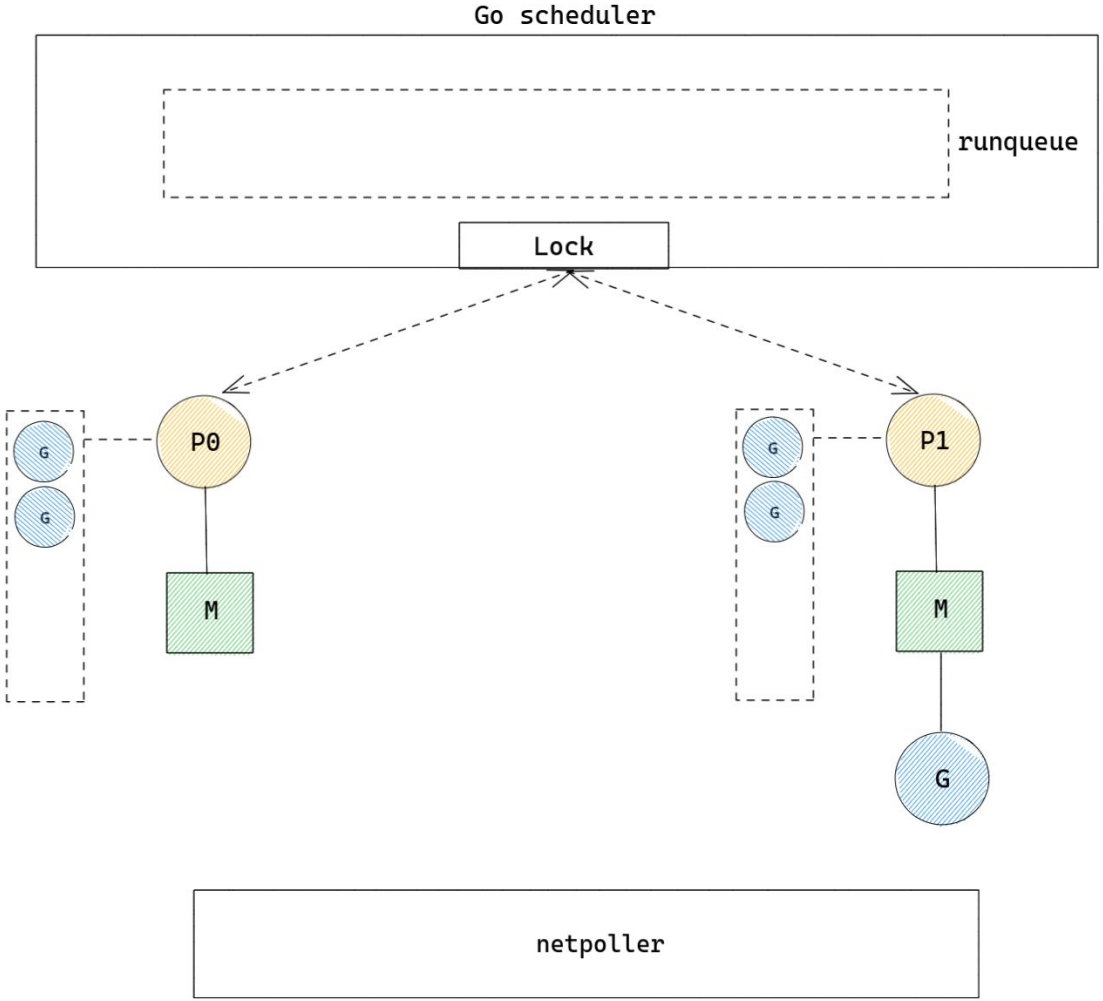

Go scheduler
runqueue
Lock
P0
M
G
G
P1
M
G
netpoller

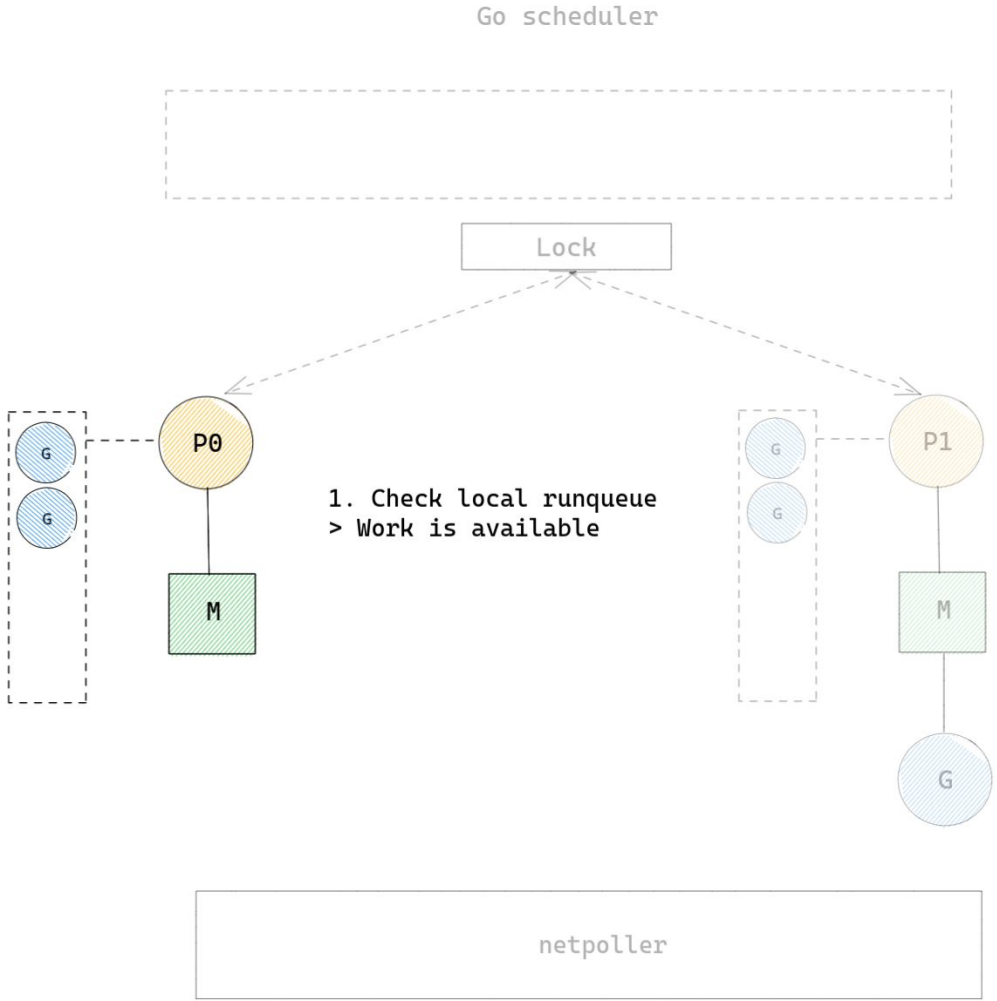
What if `netpoller` doesn't have any ready Goroutines and let's assume the situation looked something like this.

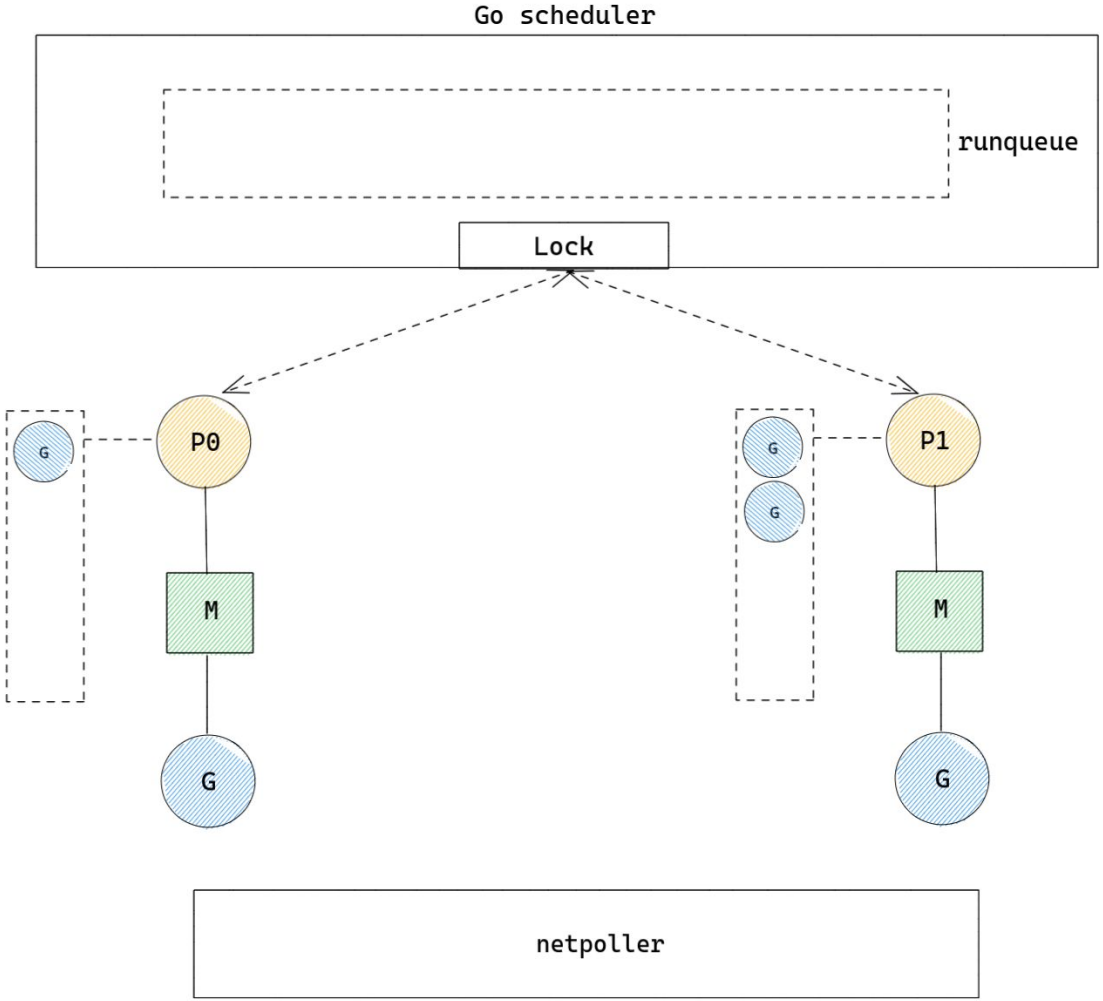

Go scheduler
runqueue
Lock
P0
M
G
Calls into runtime and finishes execution.
G
G
G
G
P1
M
G
netpoller


Go scheduler
runqueue
Lock
P0
M
4. Pick a random P and
steal half its G's
G
G
G
G
P1
M
G
netpoller


Go scheduler
runqueue
Lock
P0
M
4. Pick a random P and
steal half its G's
> Work available at P1
G
G
G
G
P1
M
G
netpoller


Go scheduler
runqueue
Lock
P0
P1
G
G
G
G
M
M
G
netpoller


Go scheduler
Lock
P0
G
G
M
1. Check local runqueue
> Work is available
P1
G
G
M
G
netpoller


Go scheduler
runqueue
Lock
P0
P1
G
G
G
G
M
M
G
G
netpoller

What about the convoy effect? Will that be taken care of?

We spoke about pre-emption earlier, let’s see how Go did it then and now.

Purely co-operative pre-emption

```
for {
    doSomething()
    ...
}
```

convoy

Compiler baked-in pre-emption
(function call pre-emption)

convoy

```
for {
    // doSomething() inlined
    ...
}
```

Non co-operative pre-emption
(compiler baked-in pre-emption
still happens)

Non Co-operative Pre-emption

- Each Goroutine is given a time-slice of 10ms after which, pre-emption is attempted.
    - 10ms is a soft limit.

```
// forcePreemptNS is the time slice given to a G before it is
// preempted.
const forcePreemptNS = 10 * 1000 * 1000 // 10ms
```

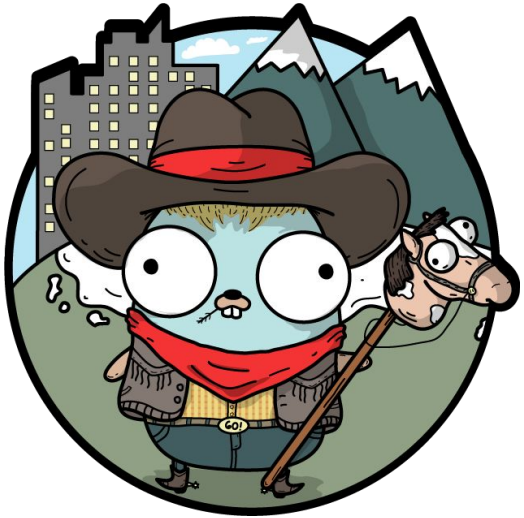
- Pre-emption occurs by sending a userspace signal to the thread running the Goroutine that needs to be pre-empted.
    - Similar to interruption based pre-emption in the kernel.
- The `SIGURG` signal is sent to the thread whose Goroutine needs to be pre-empted.

"Pardon the Interruption: Loop Preemption in Go 1.14" by Austin Clements

Non Co-operative Pre-emption

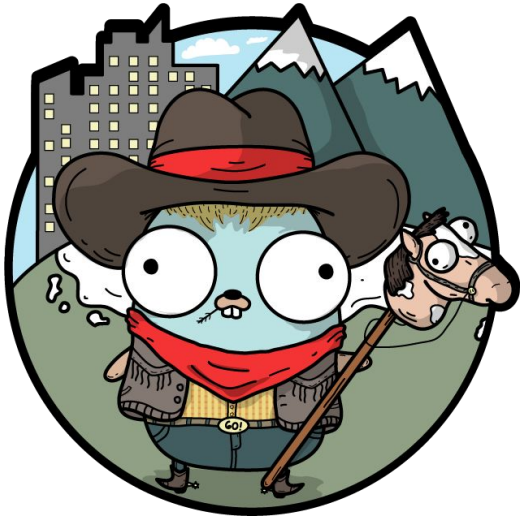
- Who sends this signal?

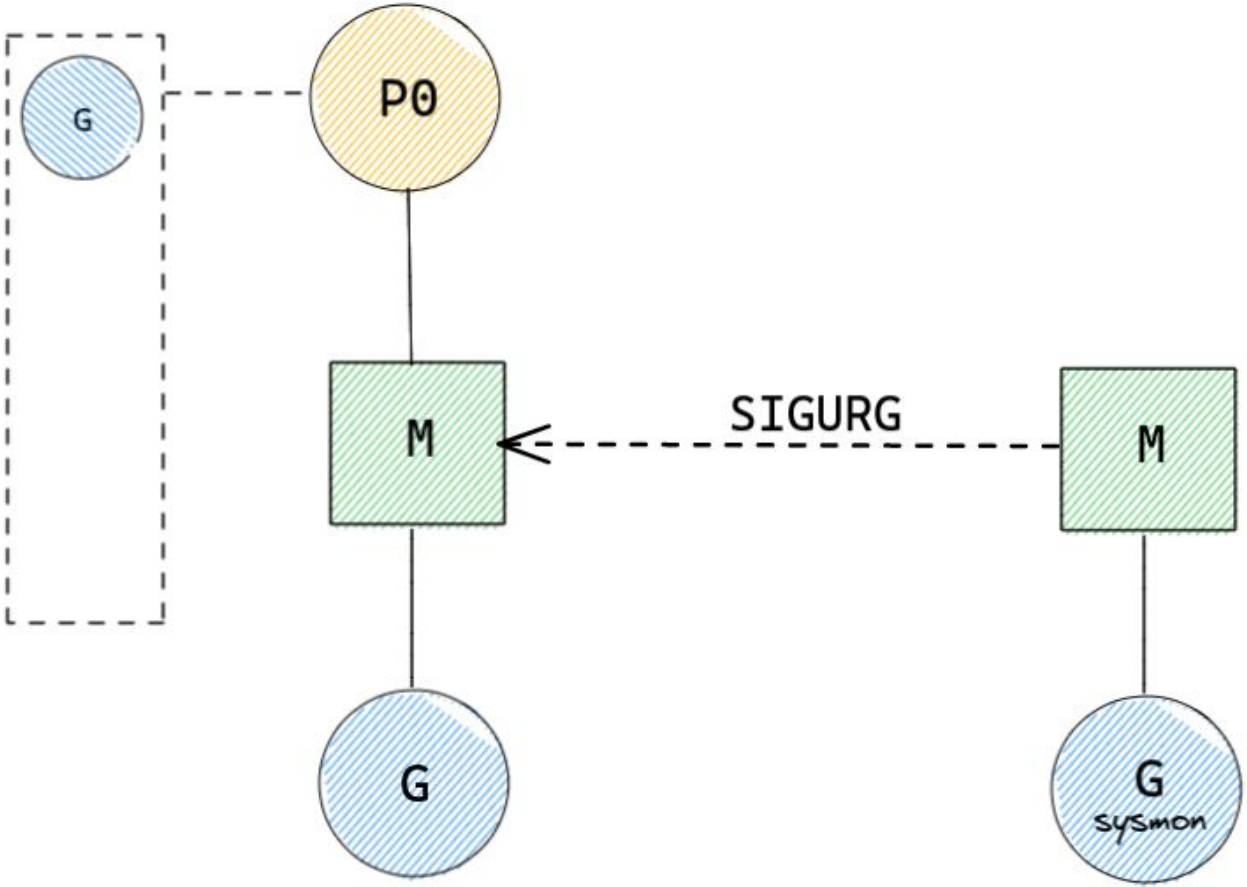
`sysmon`



Non Co-operative Pre-emption

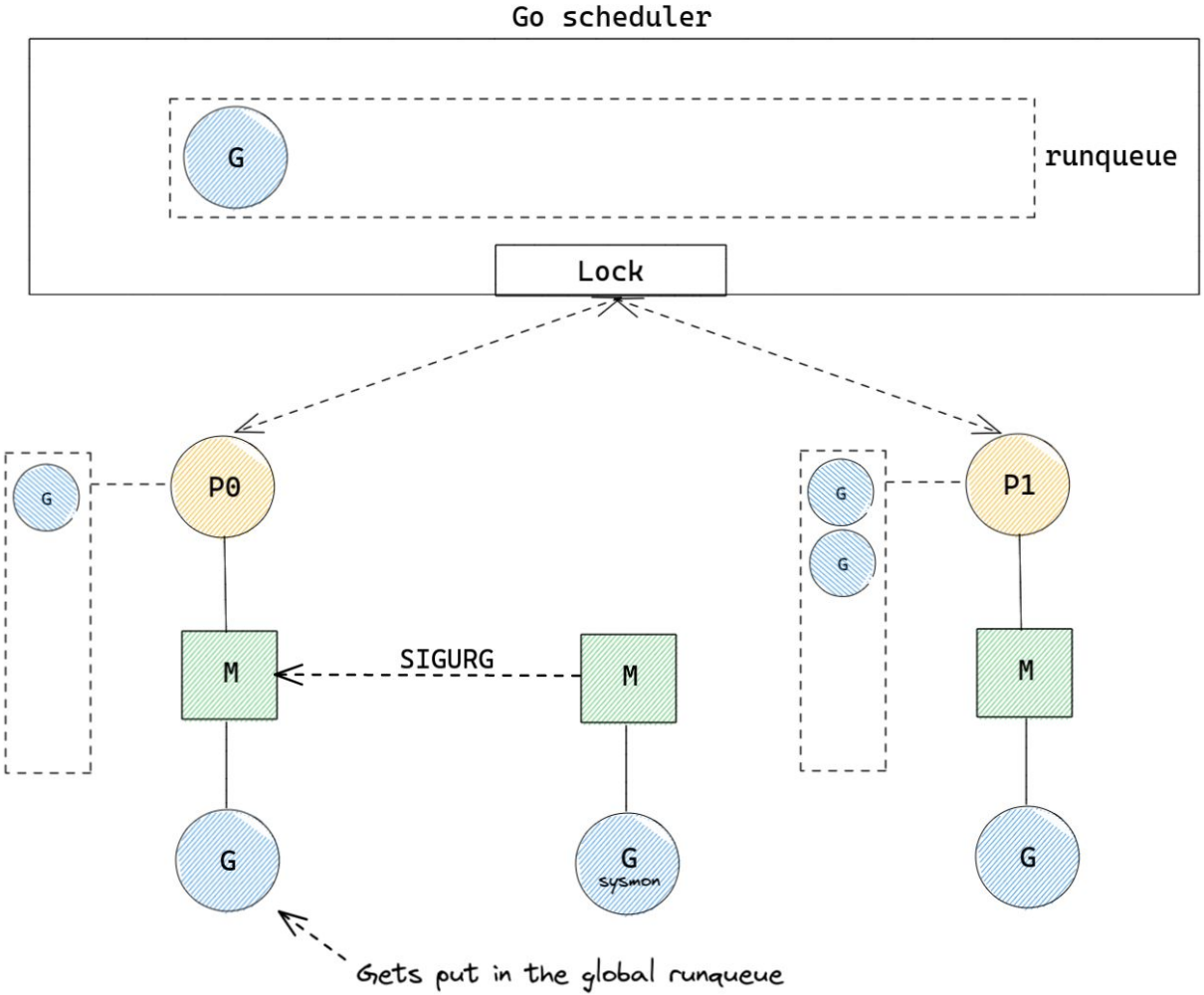
- `sysmon`
  - Daemon running without a `p`
  - Issues pre-emption *requests* for long-running Goroutines

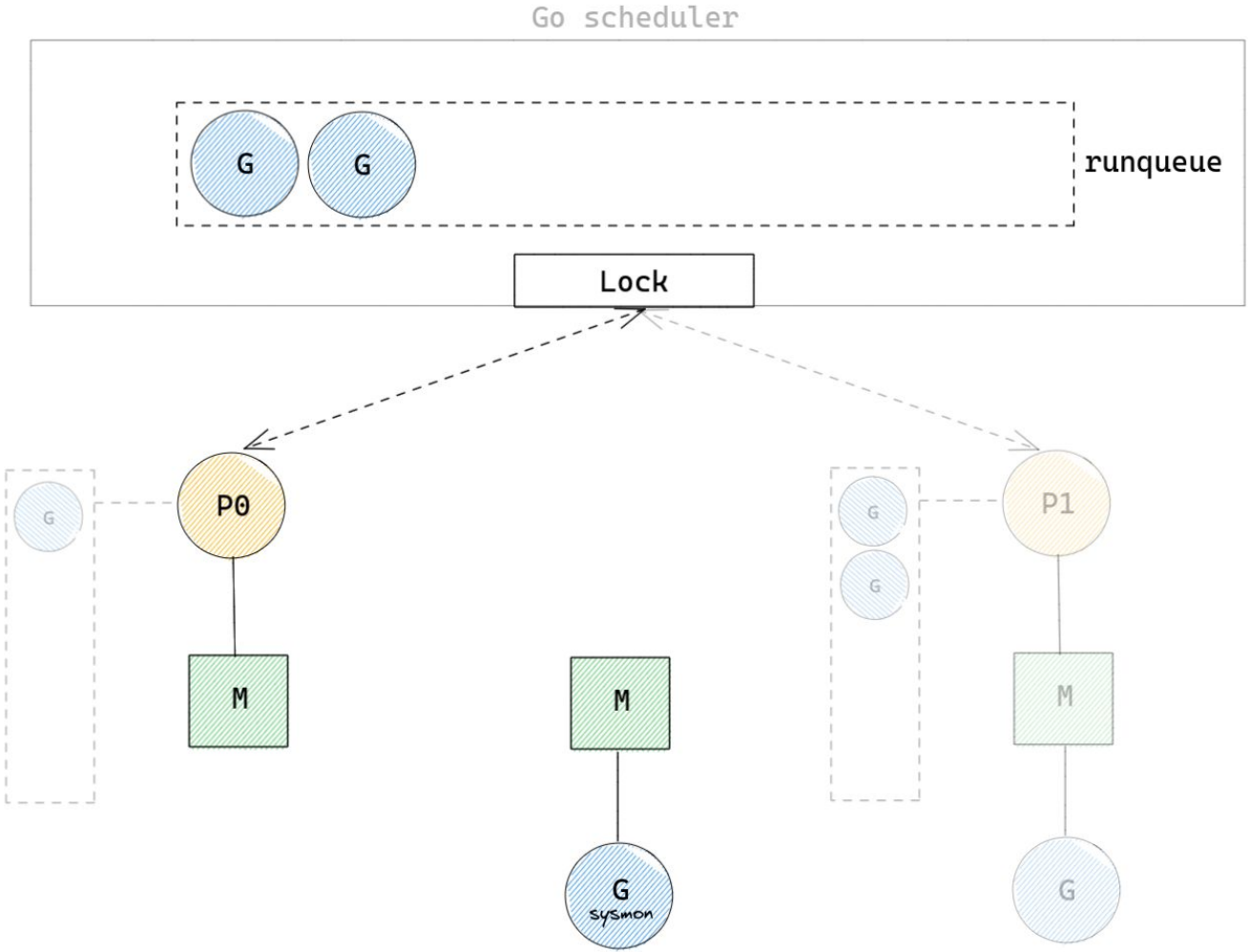


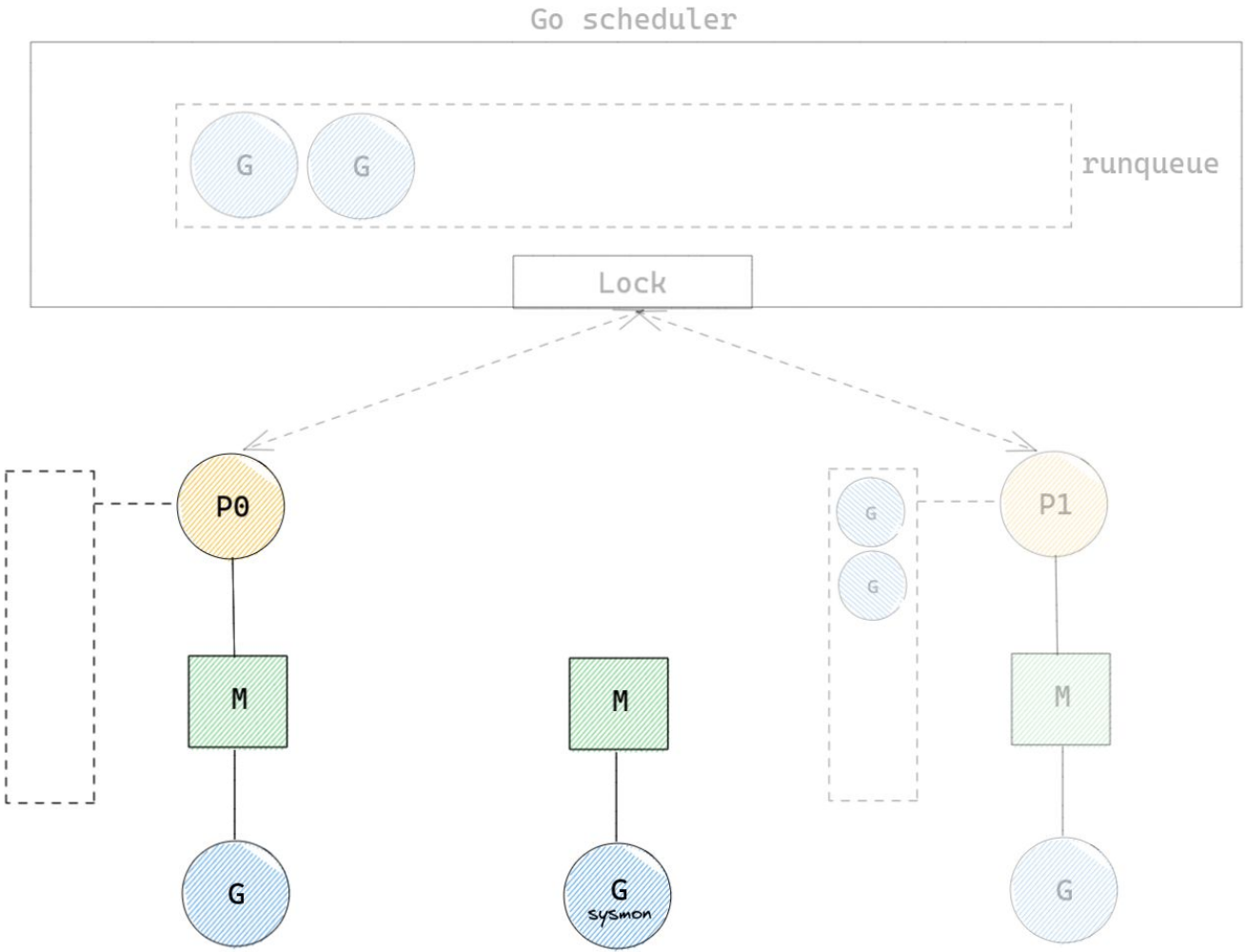


Been running for 10ms

Where does the pre-empted Goroutine go?


Go scheduler
G
runqueue
Lock
G
P0
G
P1
G
M
SIGURG
M
M
G
G
sysmon
G
Gets put in the global runqueue


Go scheduler
G
G
runqueue
Lock
P0
P1
G
G
G
M
M
M
G
sysmon
G


Go scheduler
G
G
runqueue
Lock
P0
M
G
M
G
sysmon
G
G
P1
M
G

Let’s try and actually visualize what we’ve discussed so far!

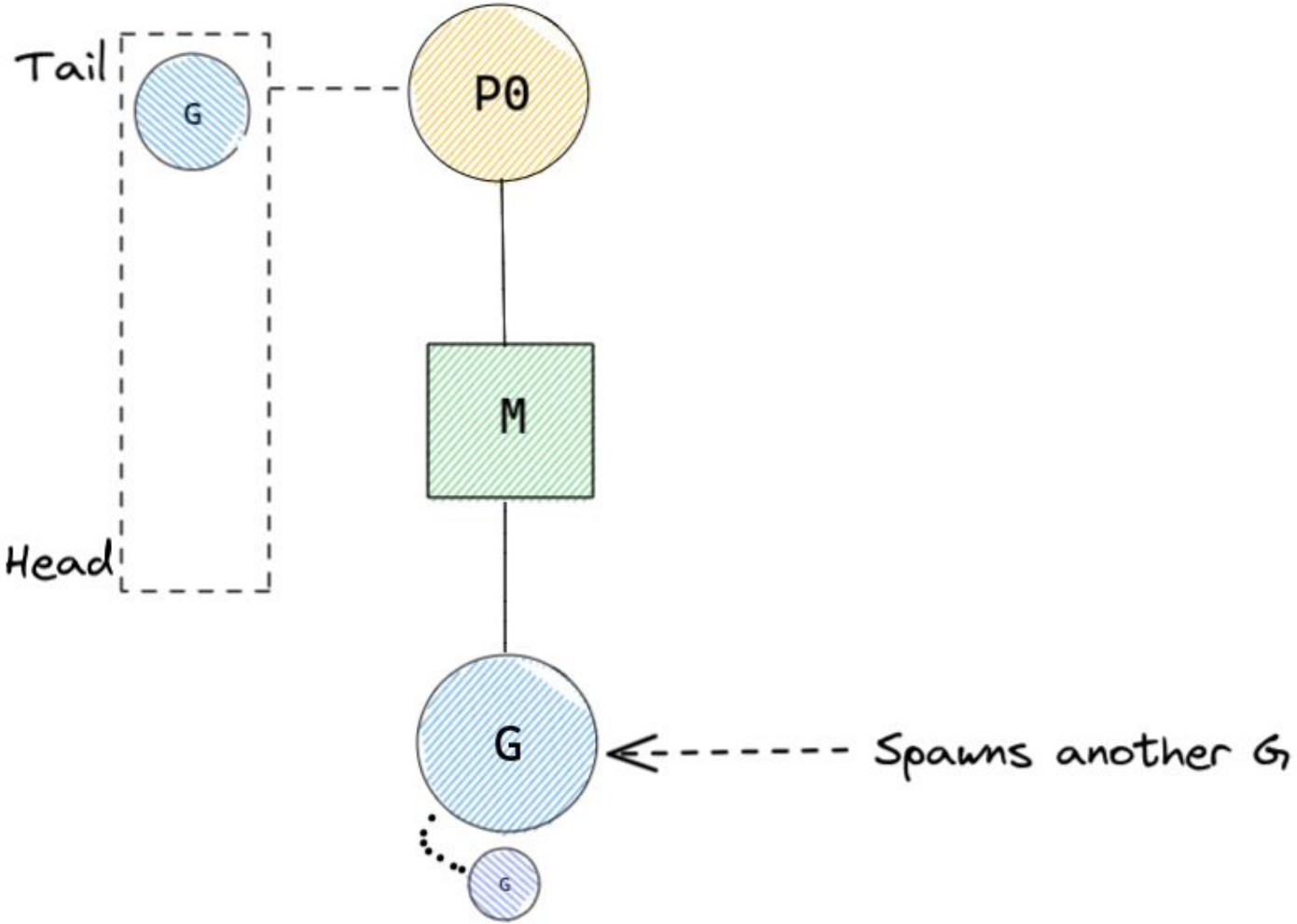
Awesome! Now that we have a way of handling resource hogs, let's revisit our code snippet.

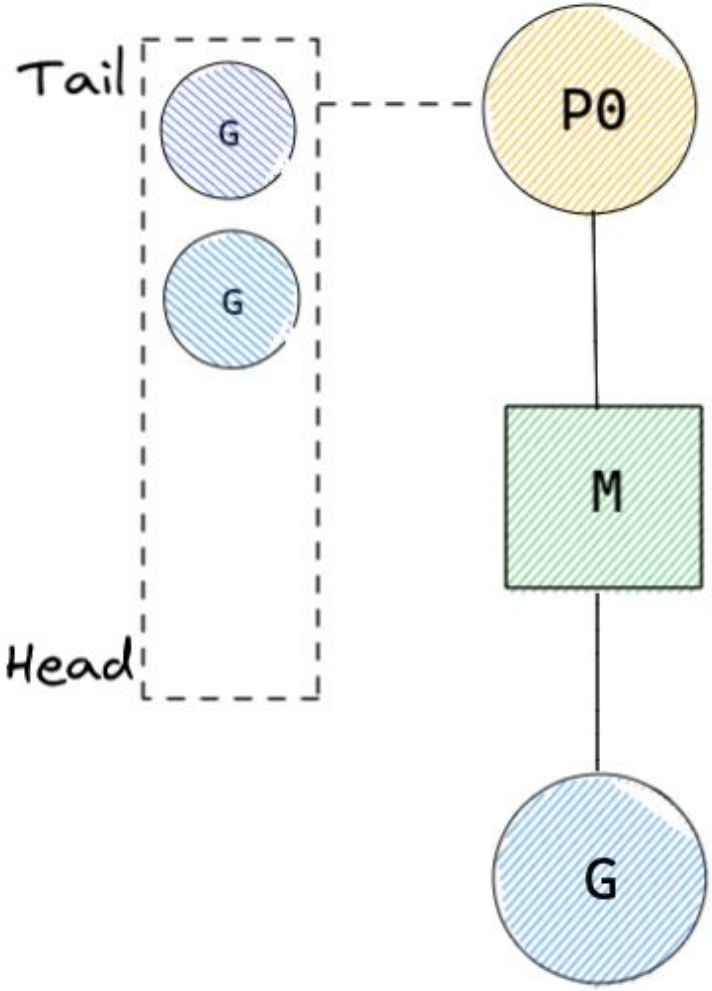
```
1   func main() {
2       var wg sync.WaitGroup
3       wg.Add(10)
4       for i := 0; i < 10; i++ {
5           go func() {
6               defer wg.Done()
7               // each matrix of size 1024
8               multiplyLargeMatrices()
9           }()
10      }
11      wg.Wait()
12  }
```

Here we have the main Goroutine spawning multiple Goroutines. Where do these spawned Goroutines go in the scheduler and when are they run?

- Spawned Goroutines are put in the local runqueue of the processor that is responsible for their creation.
- Considering FIFO brings fairness to the table, should we put the spawned Goroutines at the tail of the queue?
- Let’s see what this looks like.

```
1  func main() {
2      var wg sync.WaitGroup
3      wg.Add(10)
4      for i := 0; i < 10; i++ {
5          go func() {
6              defer wg.Done()
7              // each matrix of size 1024
8              multiplyLargeMatrices()
9          }()
10     }
11     wg.Wait()
12 }
```


Tail
G
P0
M
Head
G
Spawns another G
G

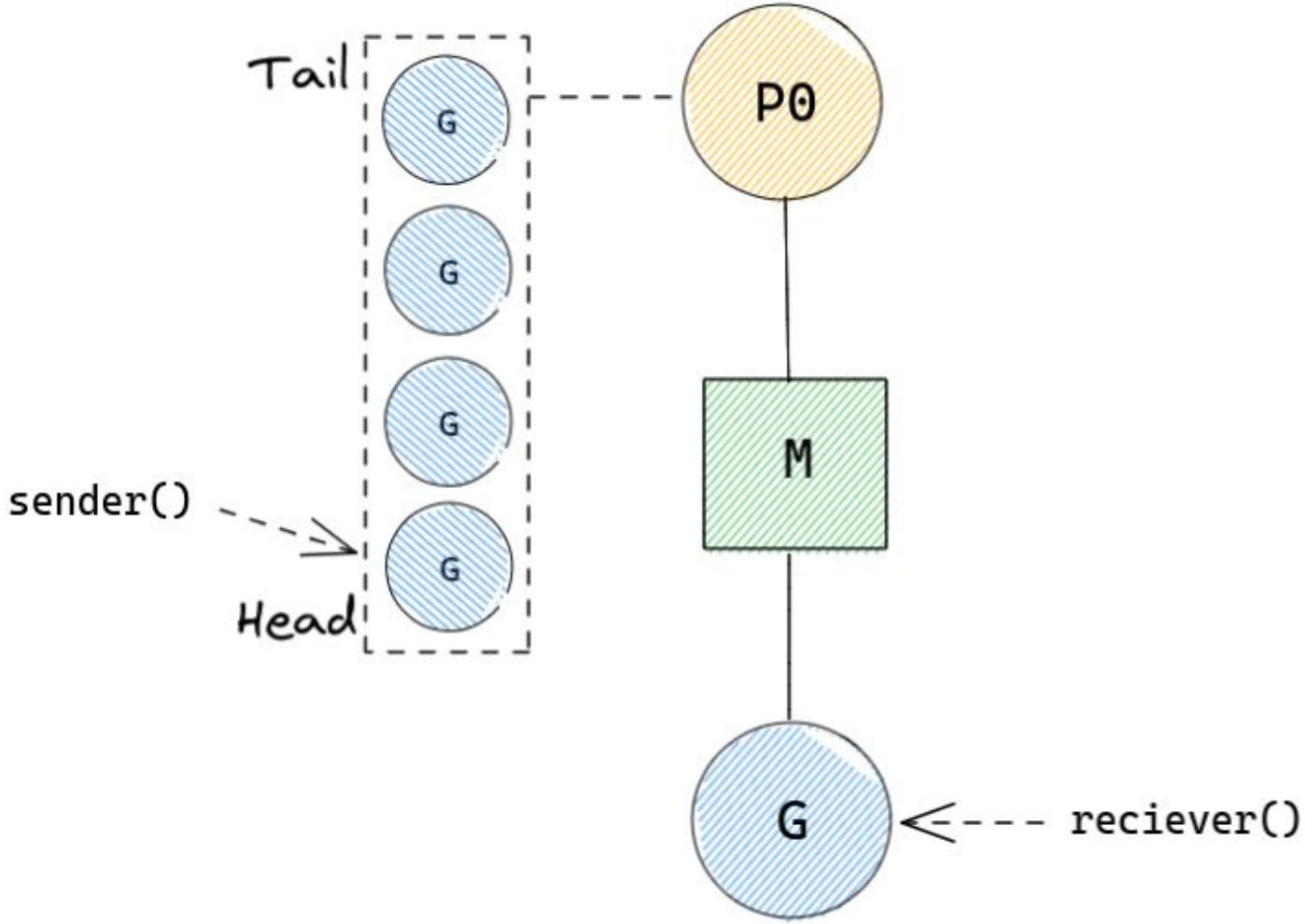

Tail
G
G
Head
P0
M
G

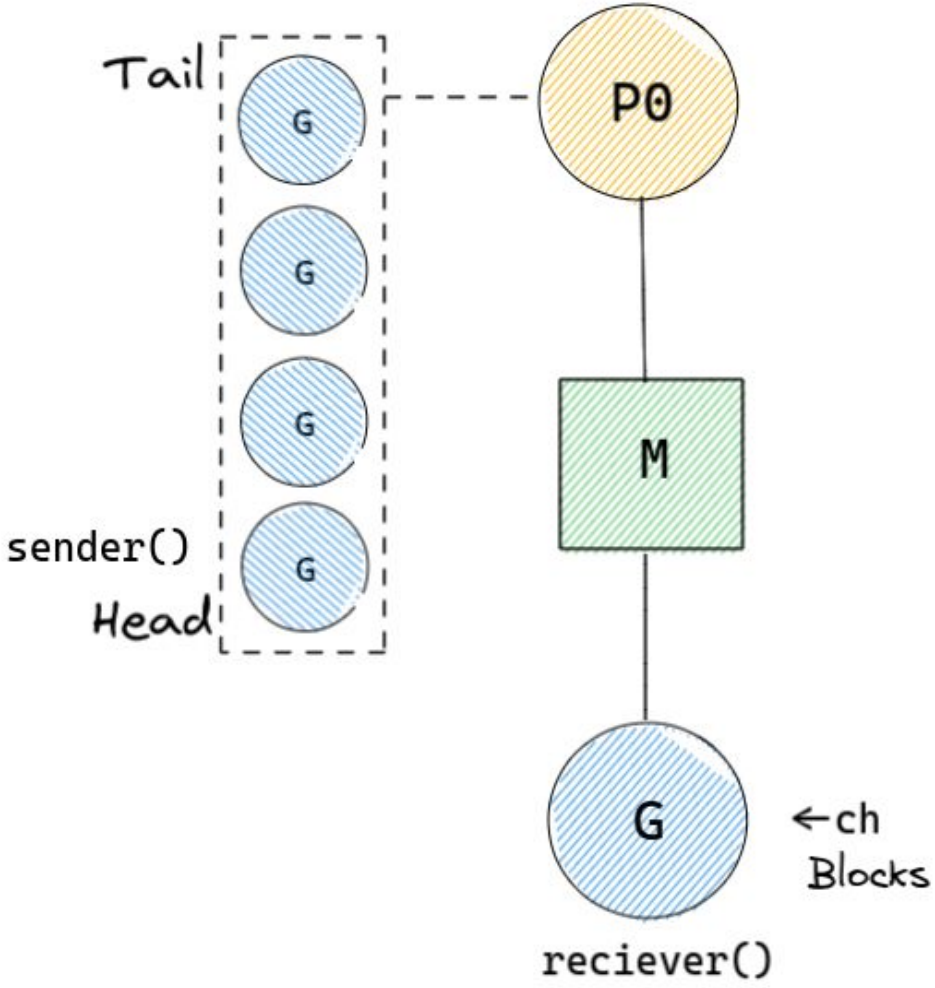
That looks good, but can we maybe do better?

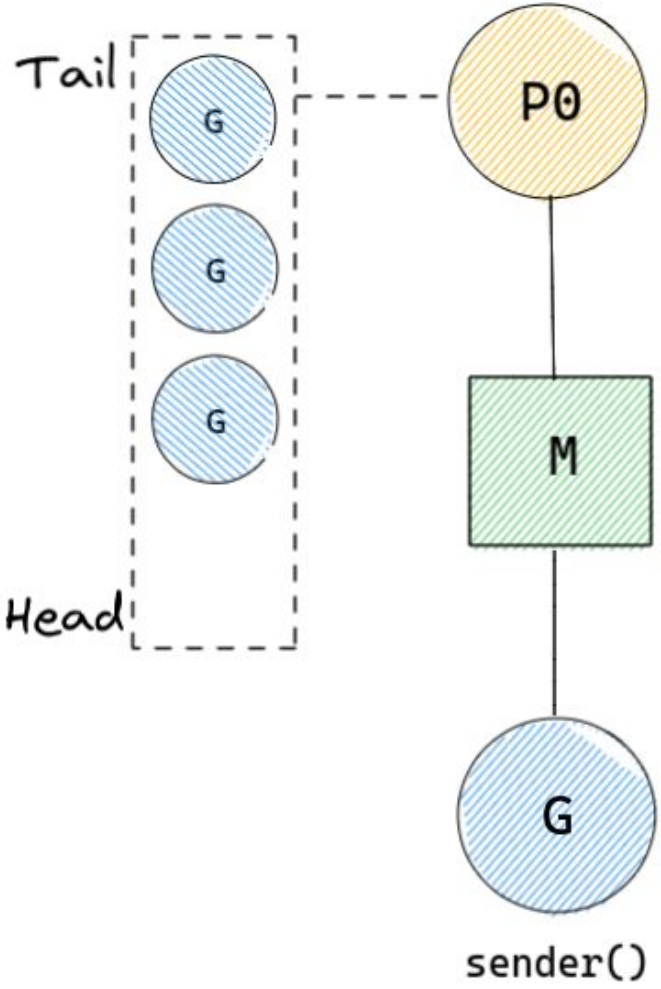
- FIFO is good for fairness, but not good for locality.
  - LIFO on the other hand is good for locality but bad for fairness.
- Maybe we can look at Go specific practices and see if we can optimize for commonly used patterns?
  - Channels are very often used in conjunction with Goroutines, be it for synchronization or communication.

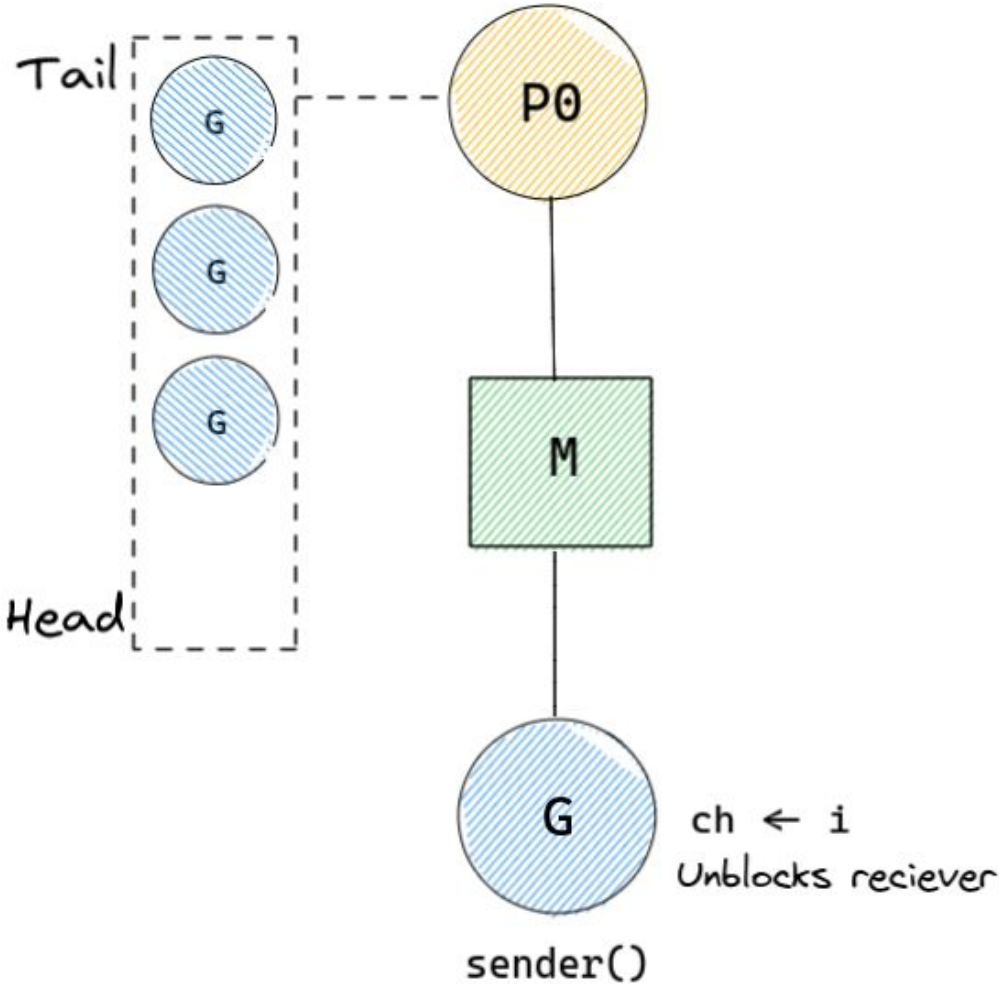
```go
func sender(ch chan int) {
    for i := 0; i < 10; i++ {
        ch <- i
    }
    close(ch)
}

func reciever(ch chan int) {
    for {
        msg, ok := <-ch
        if !ok {
            break
        }
        process(msg)
    }
}

func process(msg int) {
    fmt.Println("Recieved:", msg)
}

func main() {
    ch := make(chan int)
    go sender(ch)
    reciever(ch)
}
```

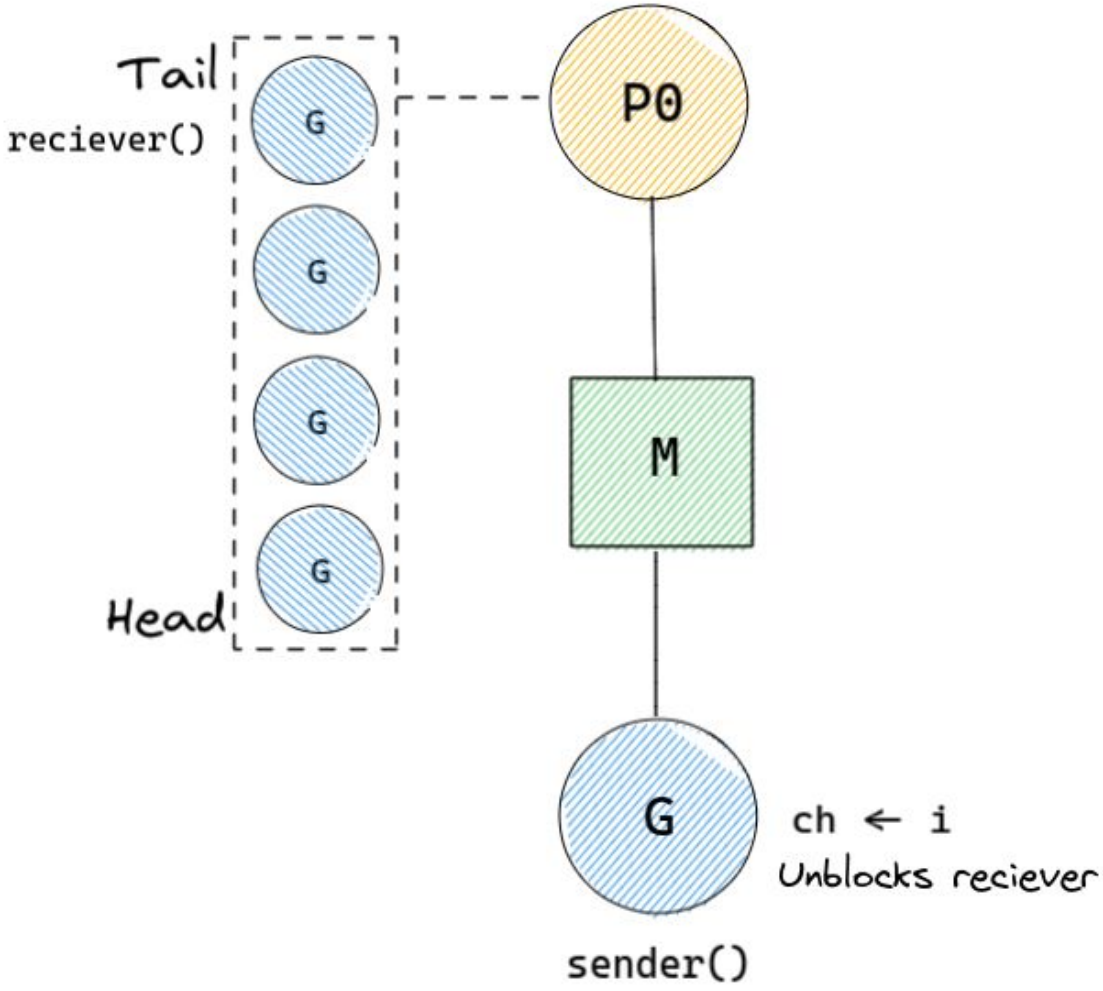
Let’s try and illustrate what happens in the scheduler.


Tail
G
G
G
G
Head
sender()
P0
M
G
reciever()


Tail
G
G
G
G
sender()
Head
P0
M
G
←ch
Blocks
reciever()


Tail
G
G
G
Head
P0
M
G
sender()


Tail
G
G
G
Head
P0
M
G
ch ← i
Unblocks reciever
sender()

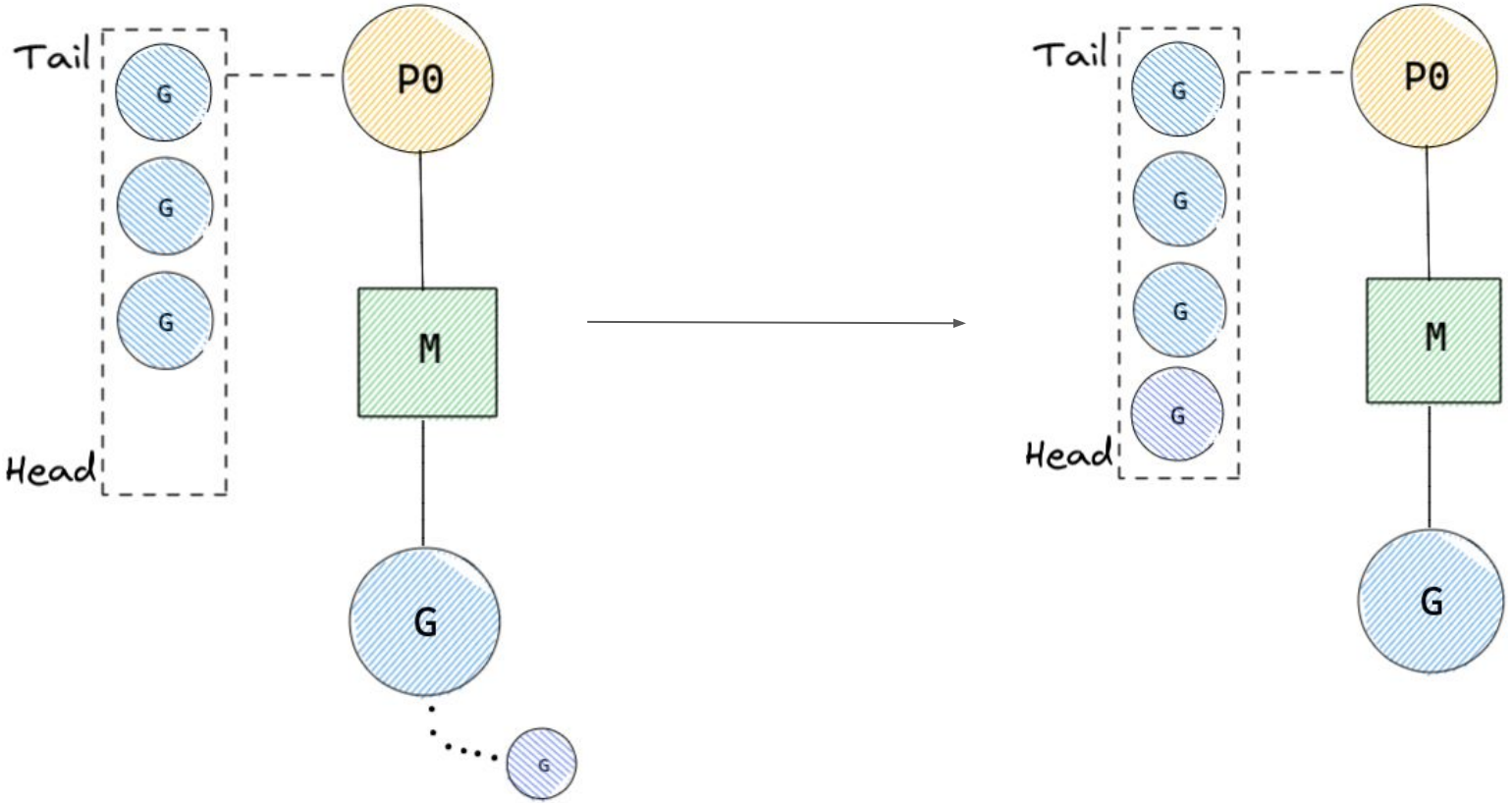

Tail
reciever()
G
G
G
G
Head
P0
M
G
ch ← i
Unblocks reciever
sender()

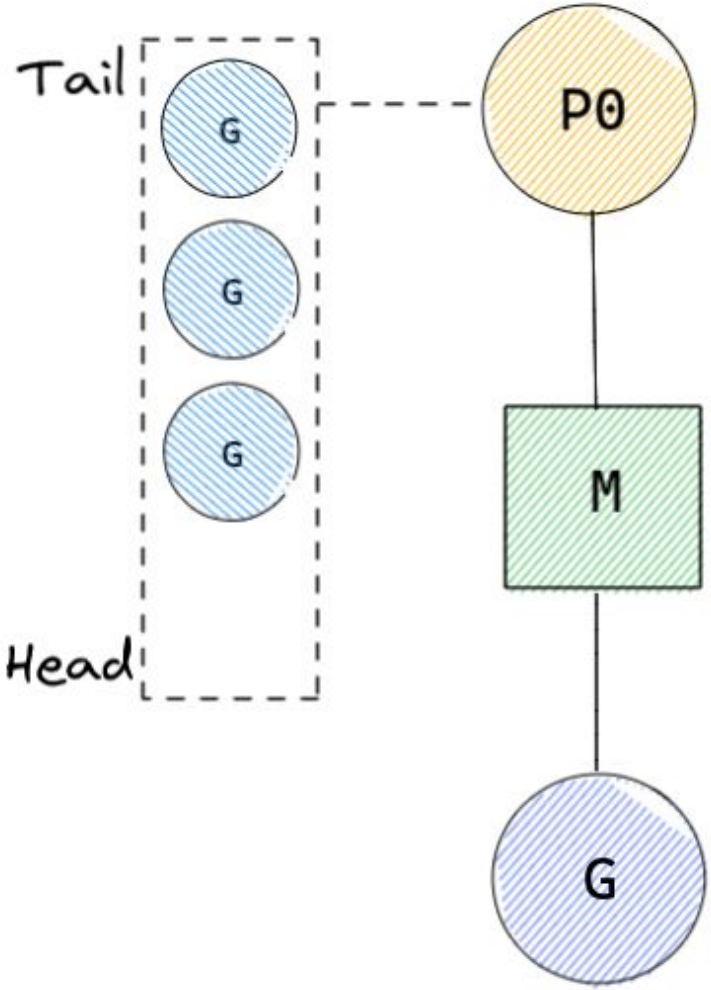
Impact On Performance

- This sending and receiving is a prolonged process - if this happens every time then each of them will have to wait for the other Goroutines to complete or be pre-empted.
- If they are long running ones - pre-emption takes ~ 10ms
    - Length of each local runqueue is fixed at 256.
    - Worst case - all are long-running, leading to one of them being blocked for ~ 255 * 10ms
- This essentially is an issue of poor locality.
- Can we combine LIFO and FIFO approaches to try and achieve better locality?

Improving Locality

- Whenever a Goroutine is spawned, it is put at the head of the local `runqueue` rather that the tail.



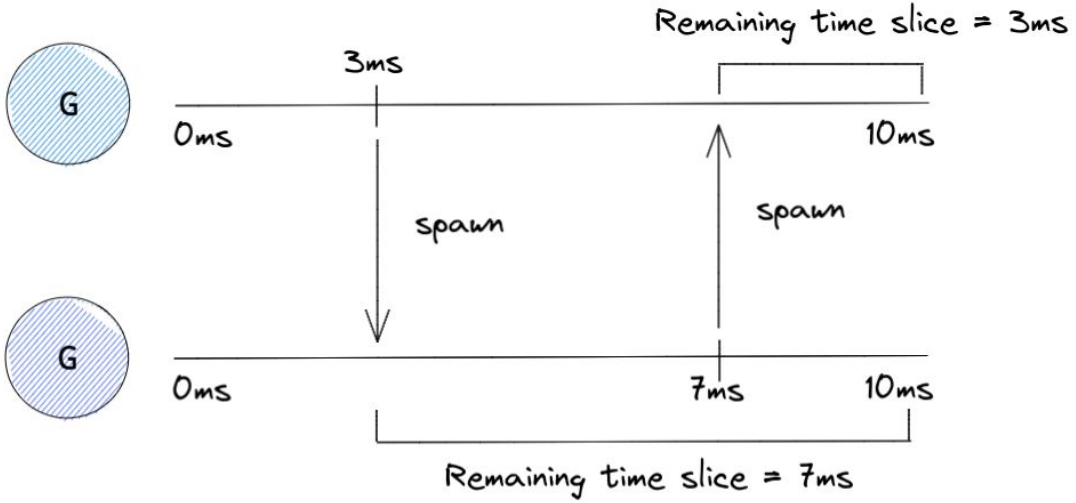

Tail
G
G
G
Head
P0
M
G

The issue now is that the 2 can constantly re-spawn each other and starve the remaining routines in the queue.

The way the Go scheduler solves this problem is by doing something known as *time slice inheritance*.

Time Slice Inheritance

- The spawned Goroutine that is put at the head of the queue inherits the remaining time slice of the Goroutine that spawned it.
- This effectively gives a time slice of 10ms to the spawner-spawnee pair post which one of them will be preempted and put in the global `runqueue`.



Impact On Performance

- From the commit that implemented this change:

```
benchmark                old ns/op     new ns/op     delta
BenchmarkPingPongHog     684287        825           -99.88%

On the x/benchmarks suite, this change improves the performance of
garbage by ~6% (for GOMAXPROCS=1 and 4), and json by 28% and 36% for
GOMAXPROCS=1 and 4. It has negligible effect on heap size.
```

Things look good!

Buuuut… could lead to starvation of Goroutines in the global `runqueue`.

- Right now, our only chance of polling the global `runqueue` is when we try and look for a Goroutine to run (after verifying that the local `runqueue` is empty).
- If our local `runqueue`s are always a source of work, we would never poll the global `runqueue`.

Things look good!

Buuuut… could lead to starvation of Goroutines in the global `runqueue`.

- Right now, our only chance of polling the global `runqueue` is when we try and look for a Goroutine to run (after verifying that the local `runqueue` is empty).
- If our local `runqueue`s are always a source of work, we would never poll the global `runqueue`.
- To try and address this corner case - the Go scheduler polls the global queue *occasionally.*

```
if someCondition {
     getFromGlobal()
} else {
     doThingsAsBefore()
}
```

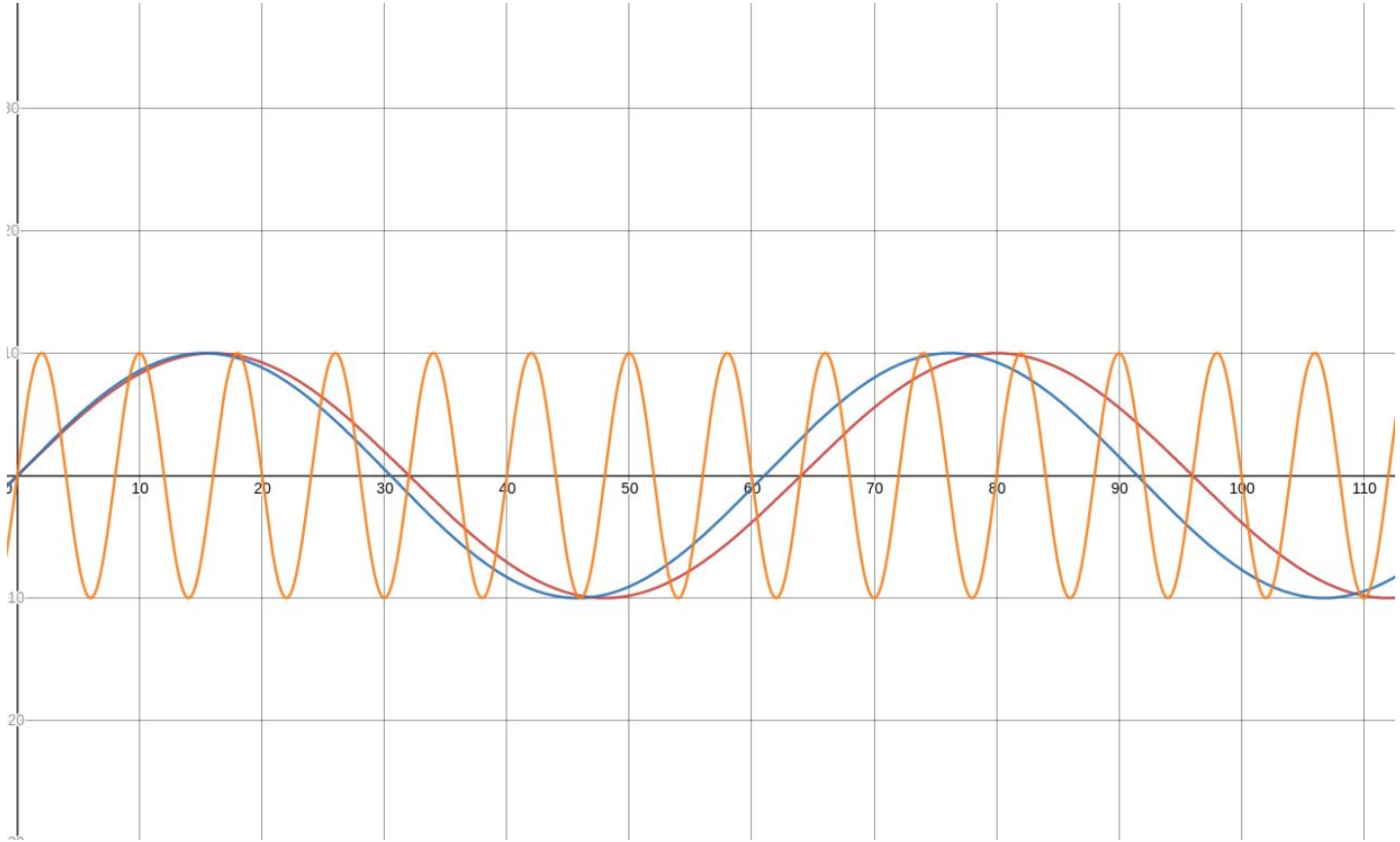
- Should be efficient to compute.
- While implementing this, a few things initially considered:
  - Have the condition be a function of the local queue length.
    - Every `4q + 16` th scheduling round where `q` is the length of the local queue.
    - Requires an explicit new counter.
  - Everytime `schedtick & 0x3f == 0` is true
    - This is too simple a check and there can still be cases where the global queue is never polled.
    - There exists a test (`TestTimerFairness2`) in the `runtime` package that verifies this.
- So, how is this condition finally computed?

```
if schedtick % 61 == 0 {
     getFromGlobal()
} else {
     doThingsAsBefore()
}
```

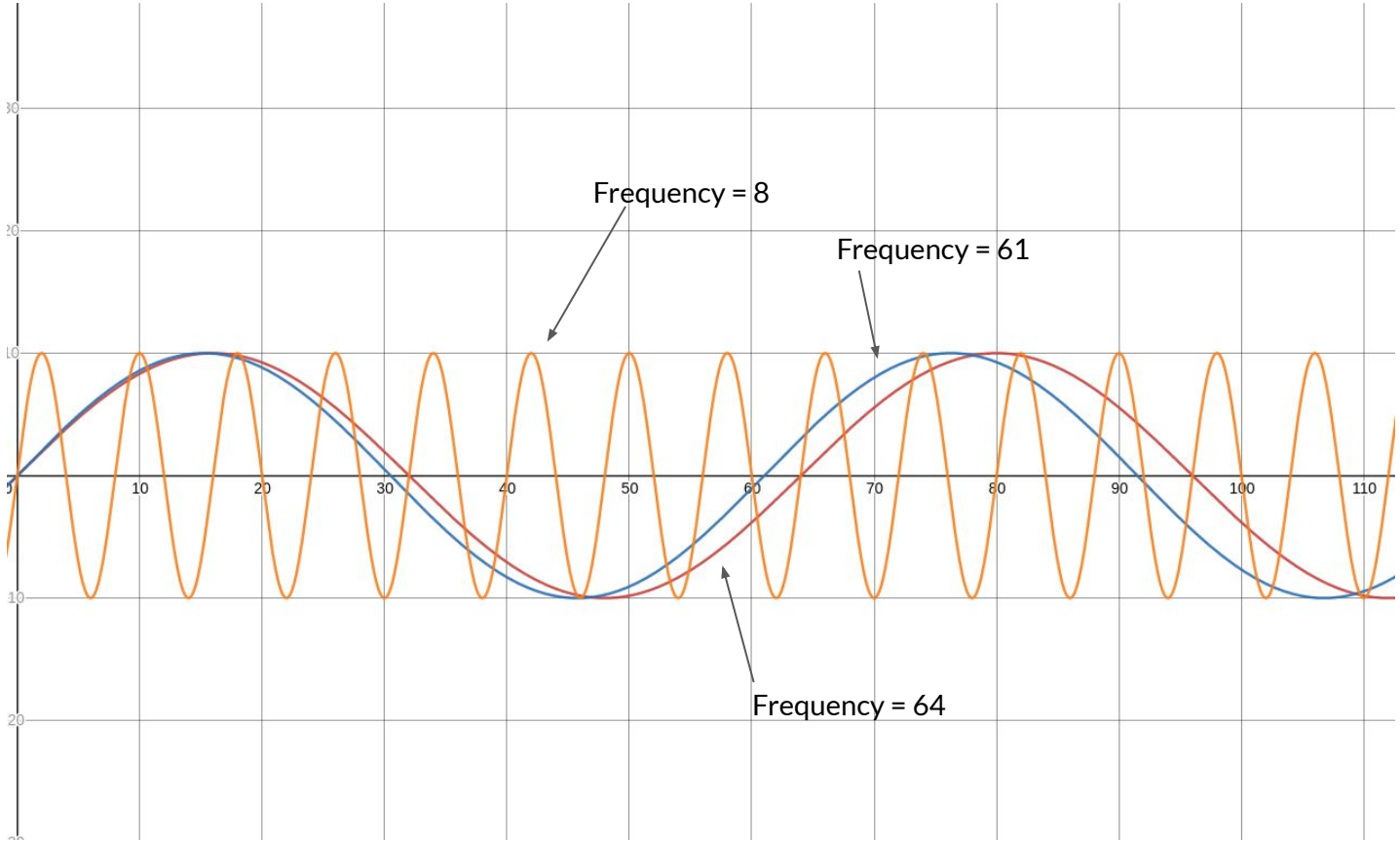
- This check is efficient to perform - requires an already maintained counter and “%” is optimized away to a `MUL` instruction which is better than `DIV` on modern processors.
- Check could be more efficient if we just went with a power of 2, we could compute a bit mask.
- So, why 61?
    - Not too big
    - Not too small
    - Prime

- Is 61 chosen keeping fairness in mind?
  - Yes.

- Is 61 chosen keeping fairness in mind?
  - Yes.

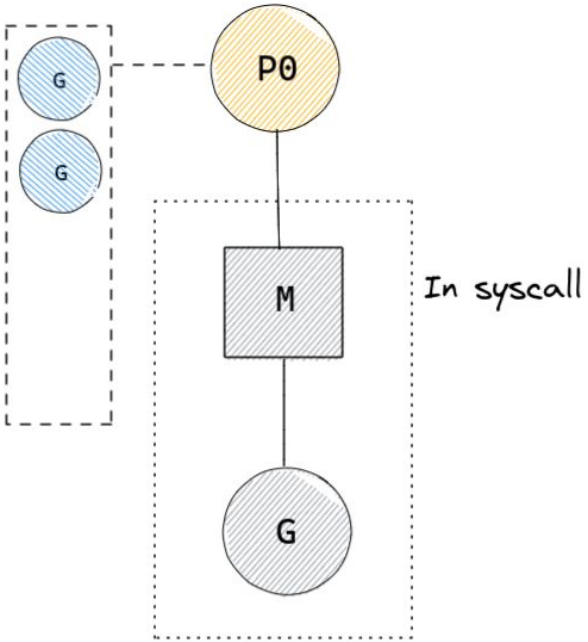


- Is 61 chosen keeping fairness in mind?
  - Yes.

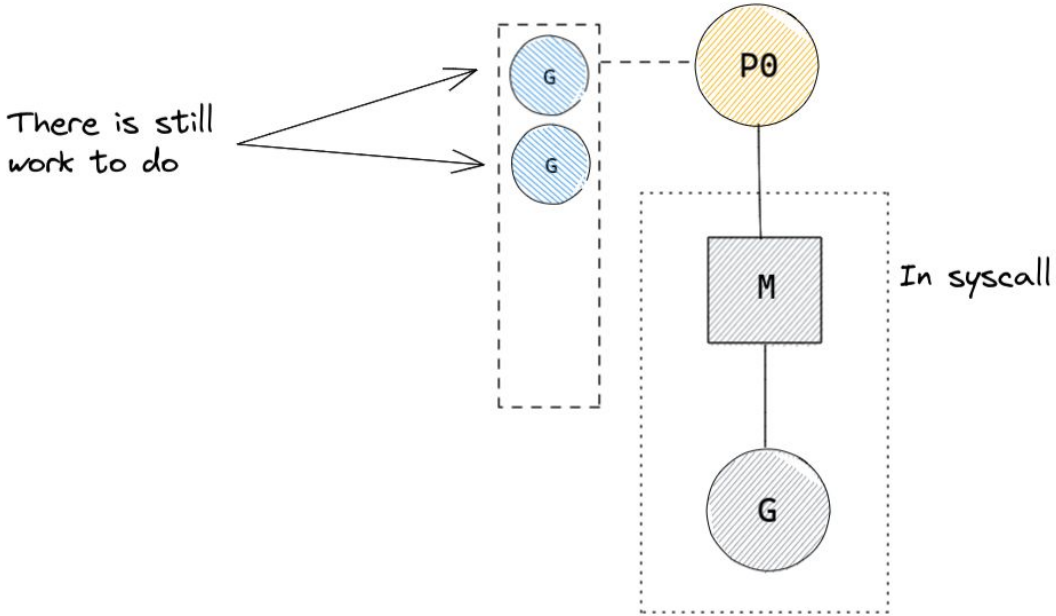


We’ve seen how Goroutines effectively end up running on threads, but what happens if the thread itself blocks in something like a `syscall`?

We’ve seen how Goroutines effectively end up running on threads, but what happens if the thread itself blocks in something like a `syscall`?

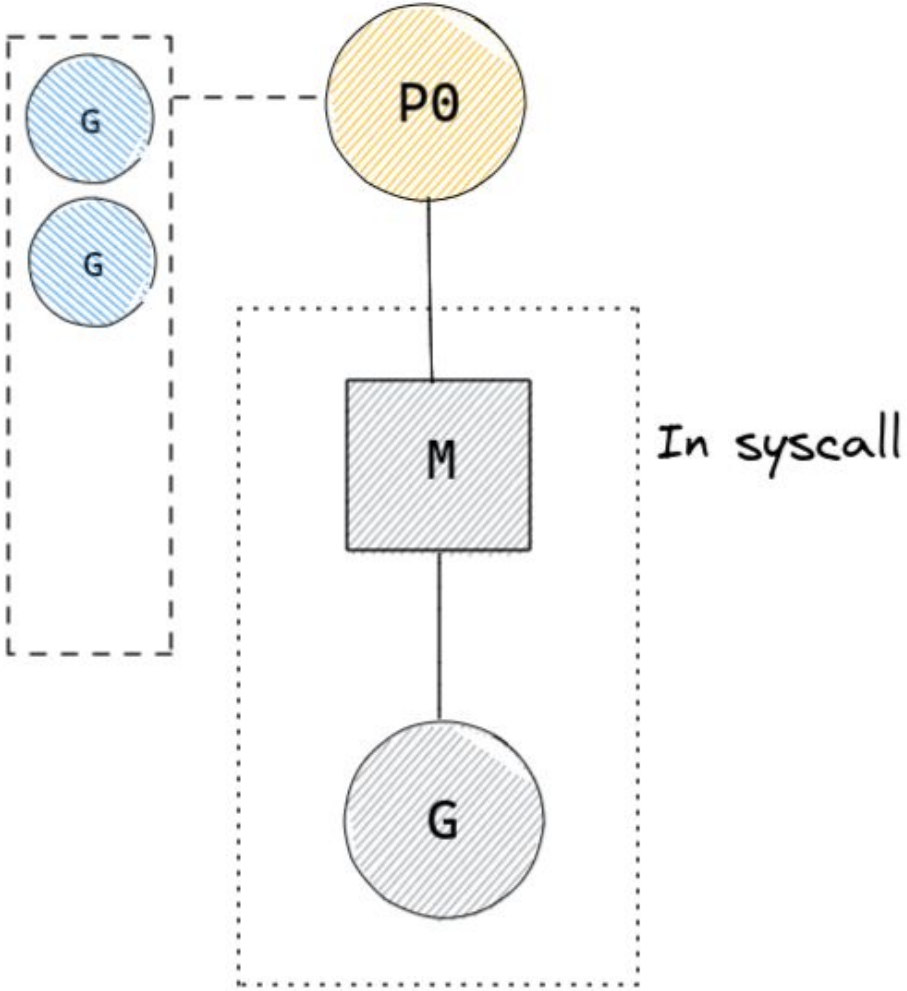


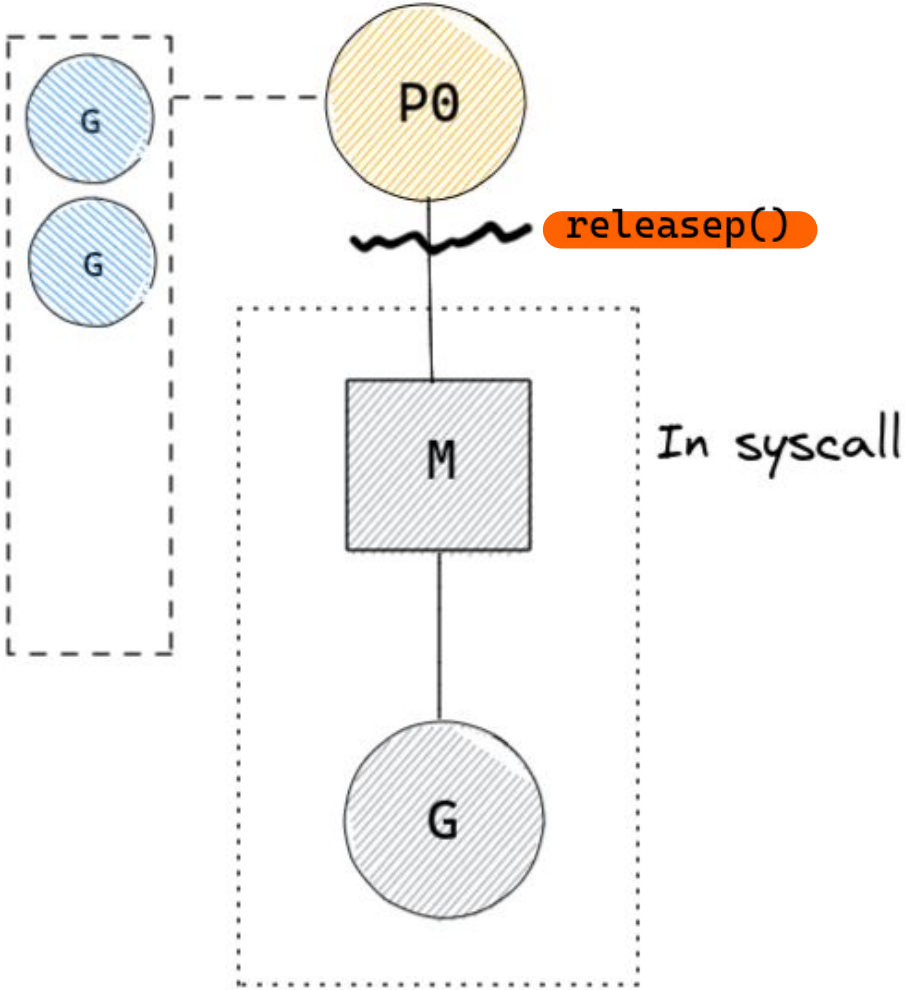
We've seen how Goroutines effectively end up running on threads, but what happens if the thread itself blocks in something like a `syscall`?

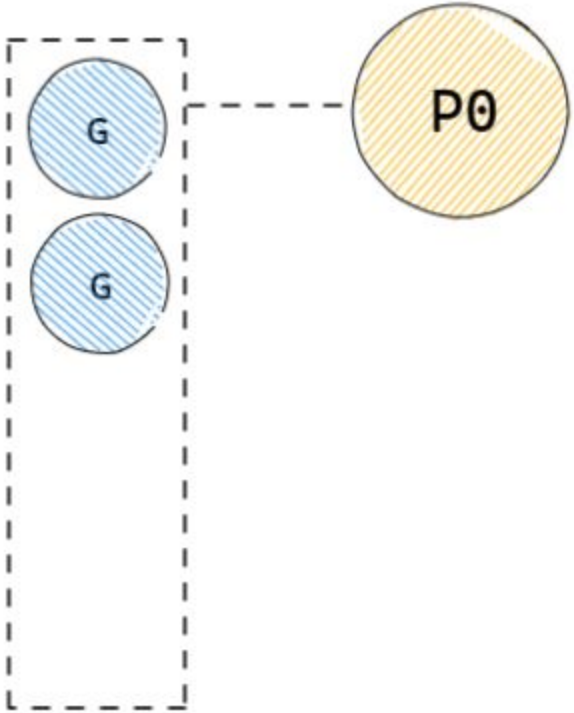


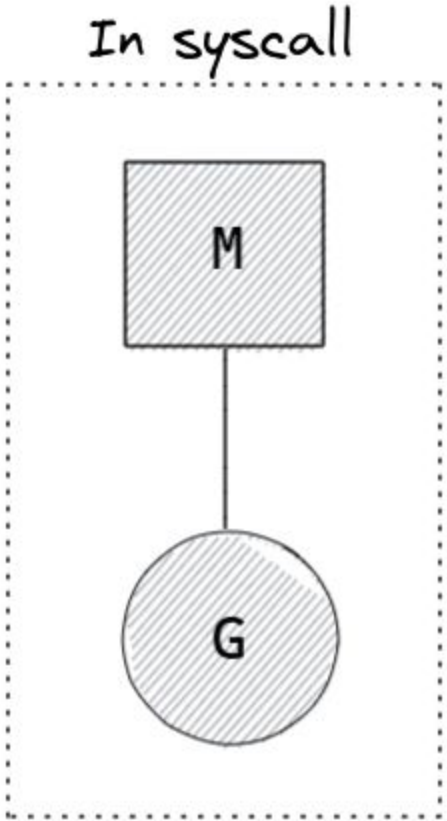
https://www.pinterest.com/pin/981221837533328912/

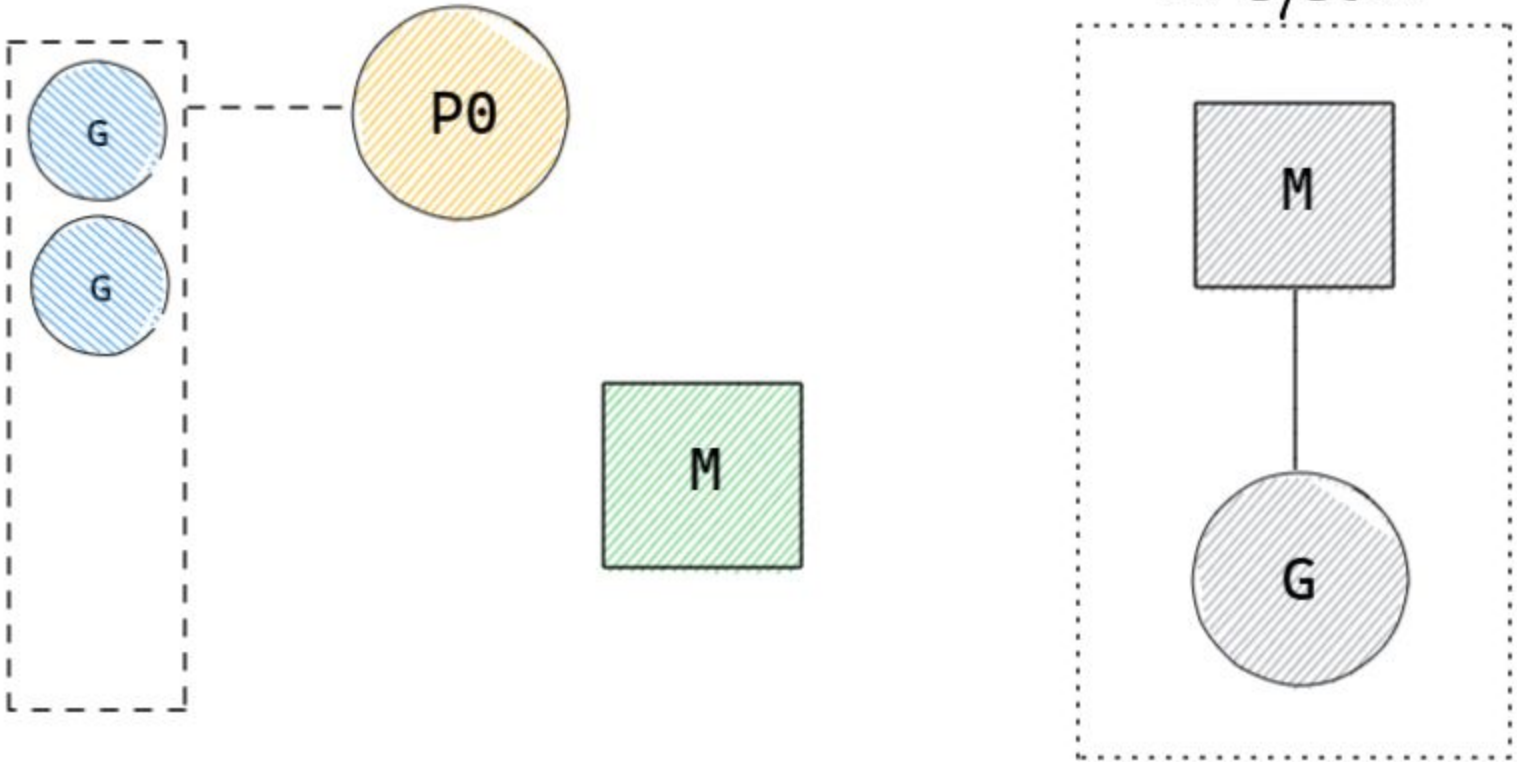
We perform something known as the `handoff` to deal with this.

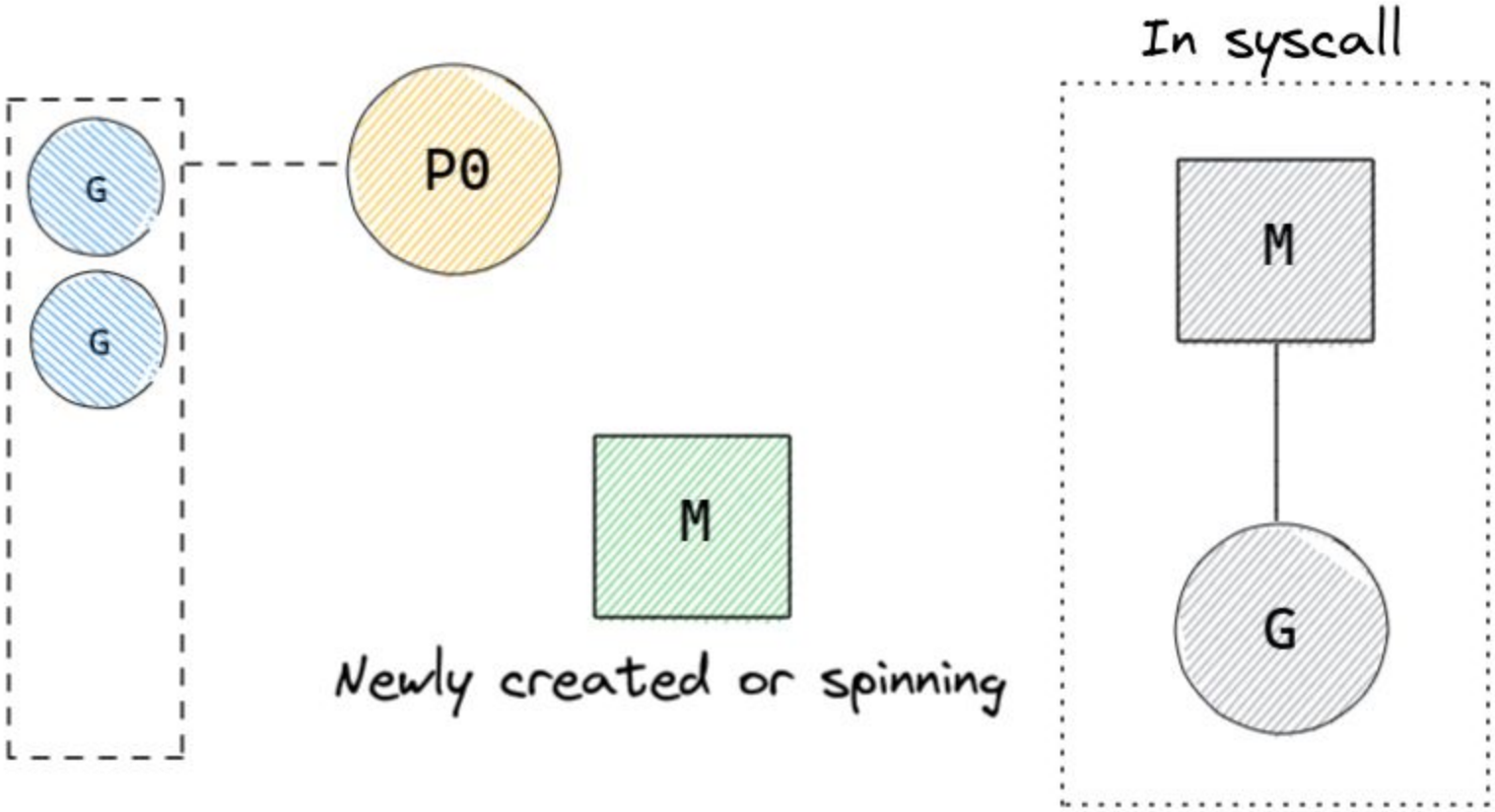

G
G
P0
M
G
In syscall

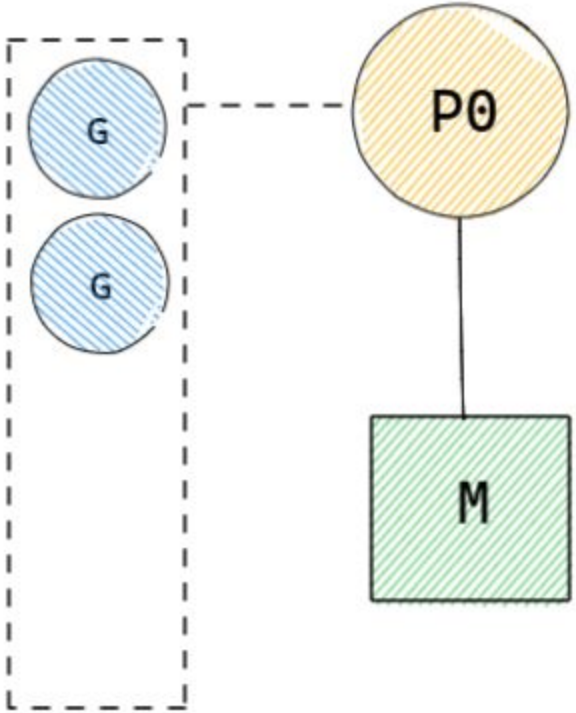

G
G
P0
releasep()
M
In syscall
G

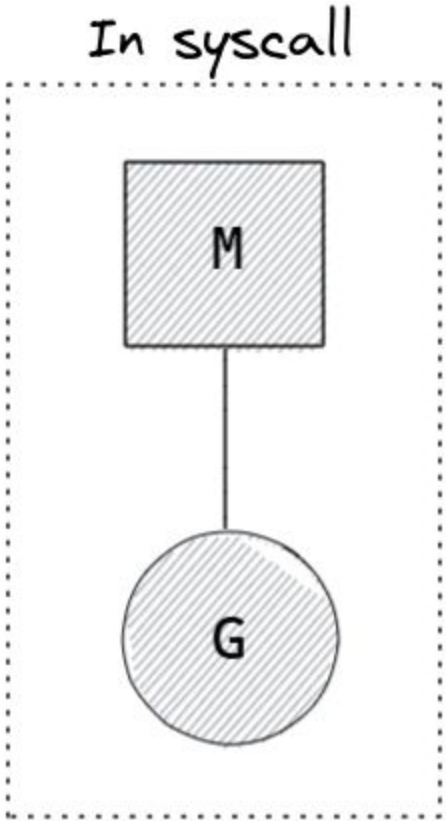

G
G
P0

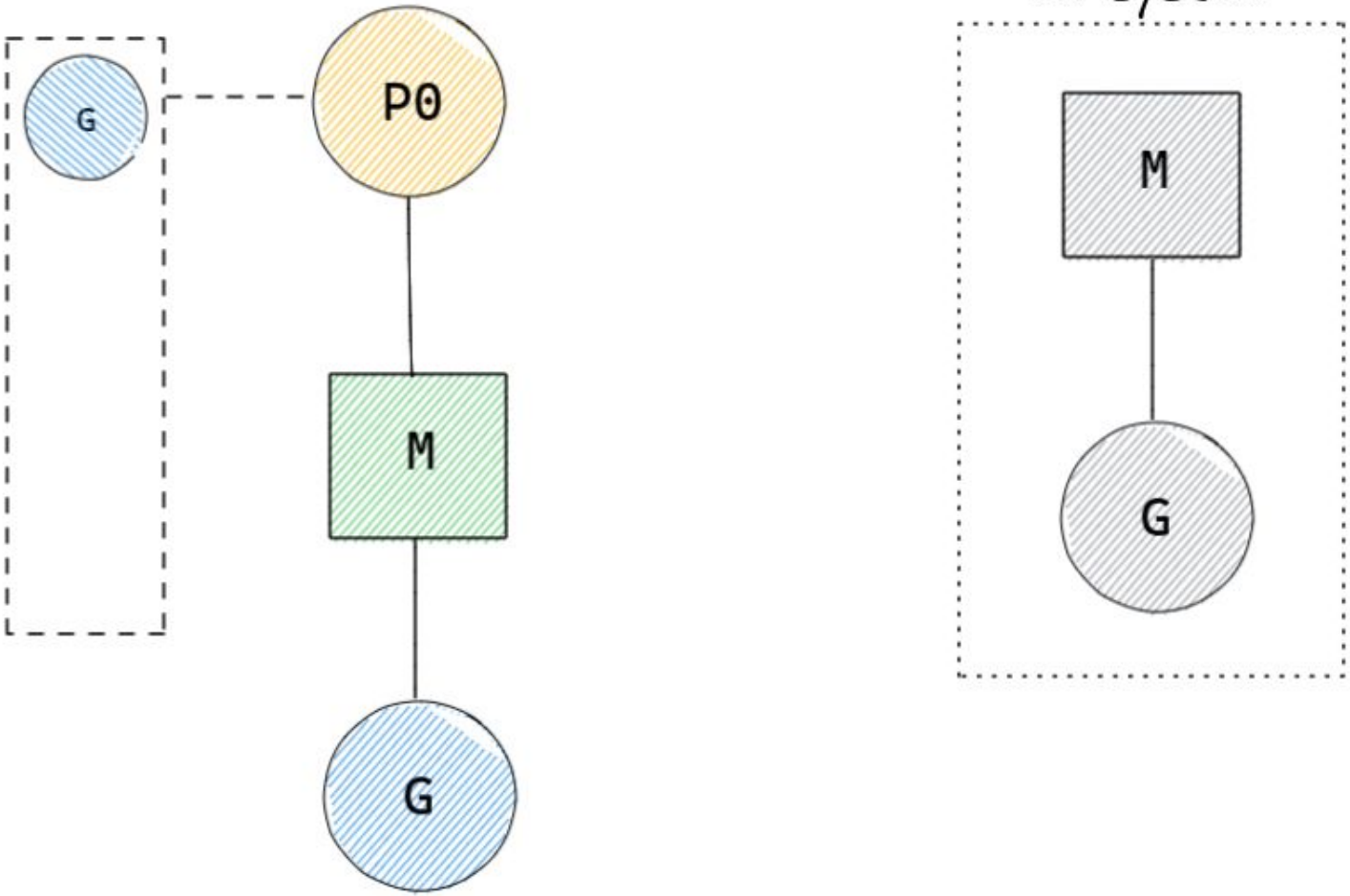

In syscall
M
G


G
G
P0
M
In syscall
M
G


In syscall
G
G
P0
M
Newly created or spinning
M
G

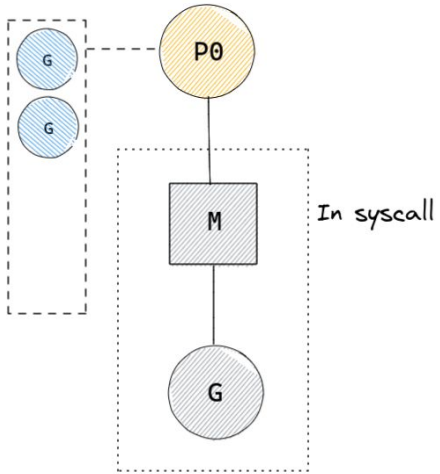

G
G
P0
M


In syscall
M
G


G
P0
M
G
In syscall
M
G

`handoff` can get expensive

- Especially when you have to create a new thread.
- And some `syscall`s aren't blocking for a prolonged period of time and doing a `handoff` for every `syscall` might be significantly expensive.
- To optimize for this, the scheduler does handoff in a slightly more intelligent manner.
  - Do handoff immediately only for some syscalls and not all.
  - In other cases let the p block as well.



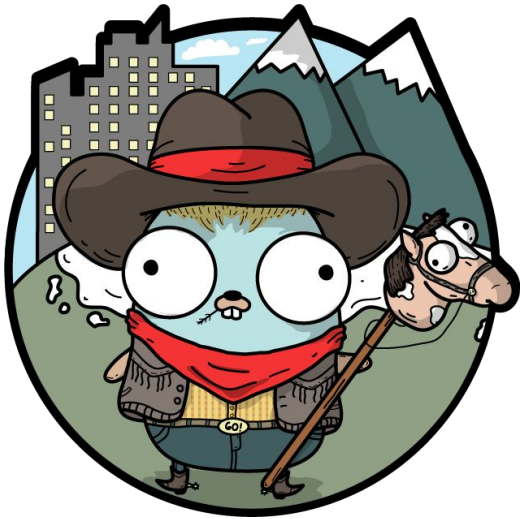
`handoff` can get expensive

- But what happens in cases when we don't perform `handoff` and the `p` still ends up being blocked for a non-trivial amount of time?

`handoff` can get expensive

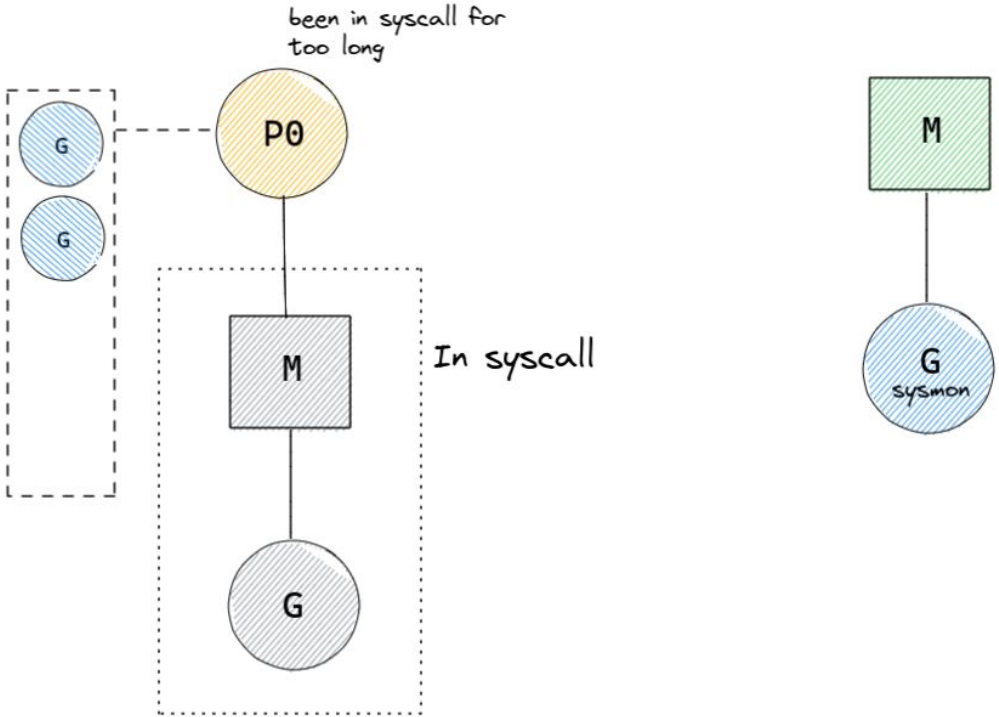
- But what happens in cases when we don't perform `handoff` and the `p` still ends up being blocked for a non-trivial amount of time?

sysmon



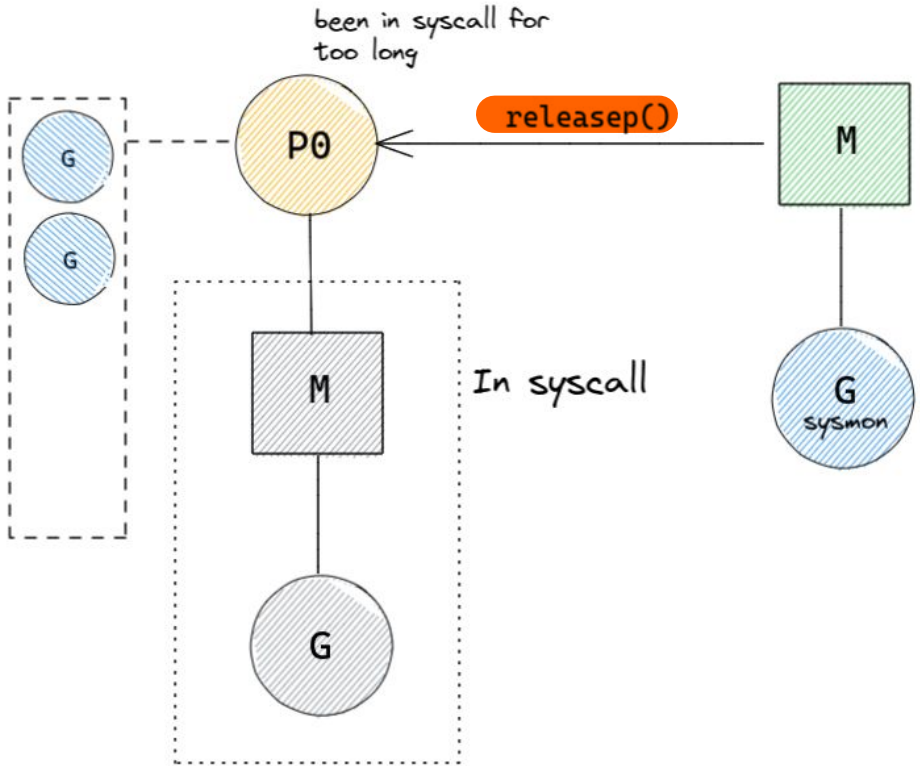
`handoff` can get expensive

- If sysmon sees that a p has been in the executing `syscall` state for too long, it initiates a `handoff`.



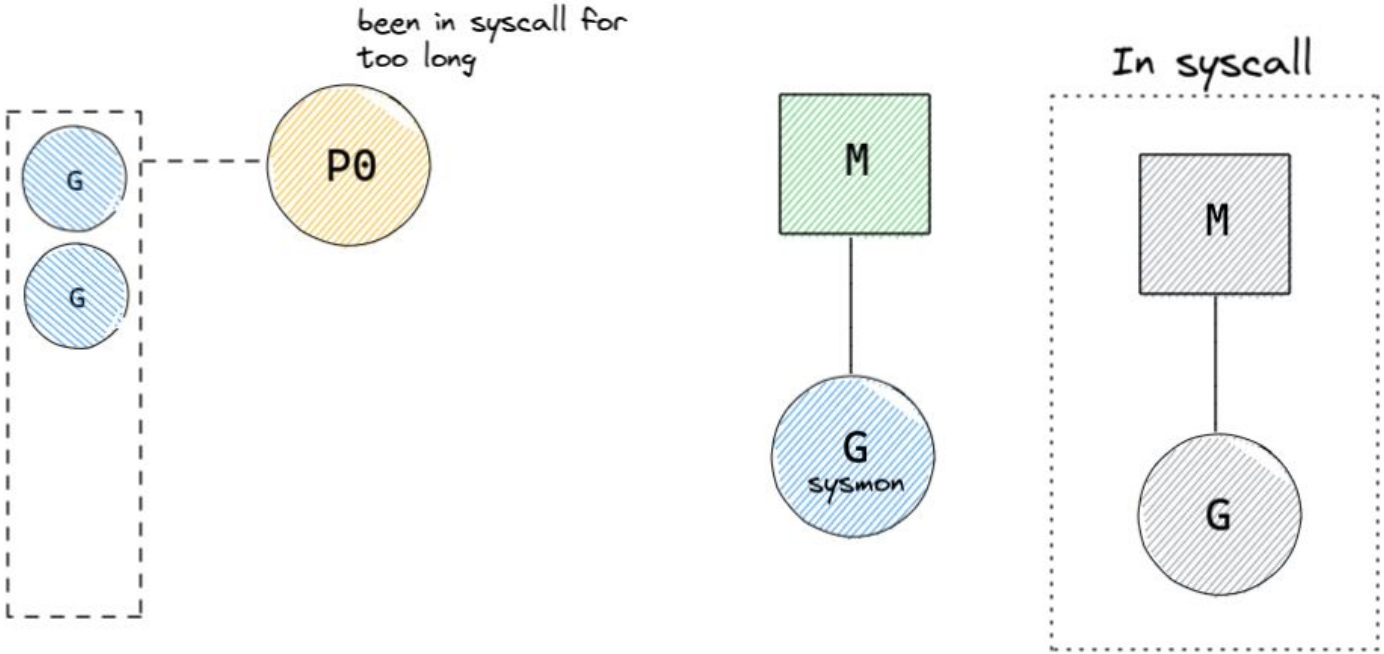
`handoff` can get expensive

- If sysmon sees that a p has been in the executing `syscall` state for too long, it initiates a `handoff`.



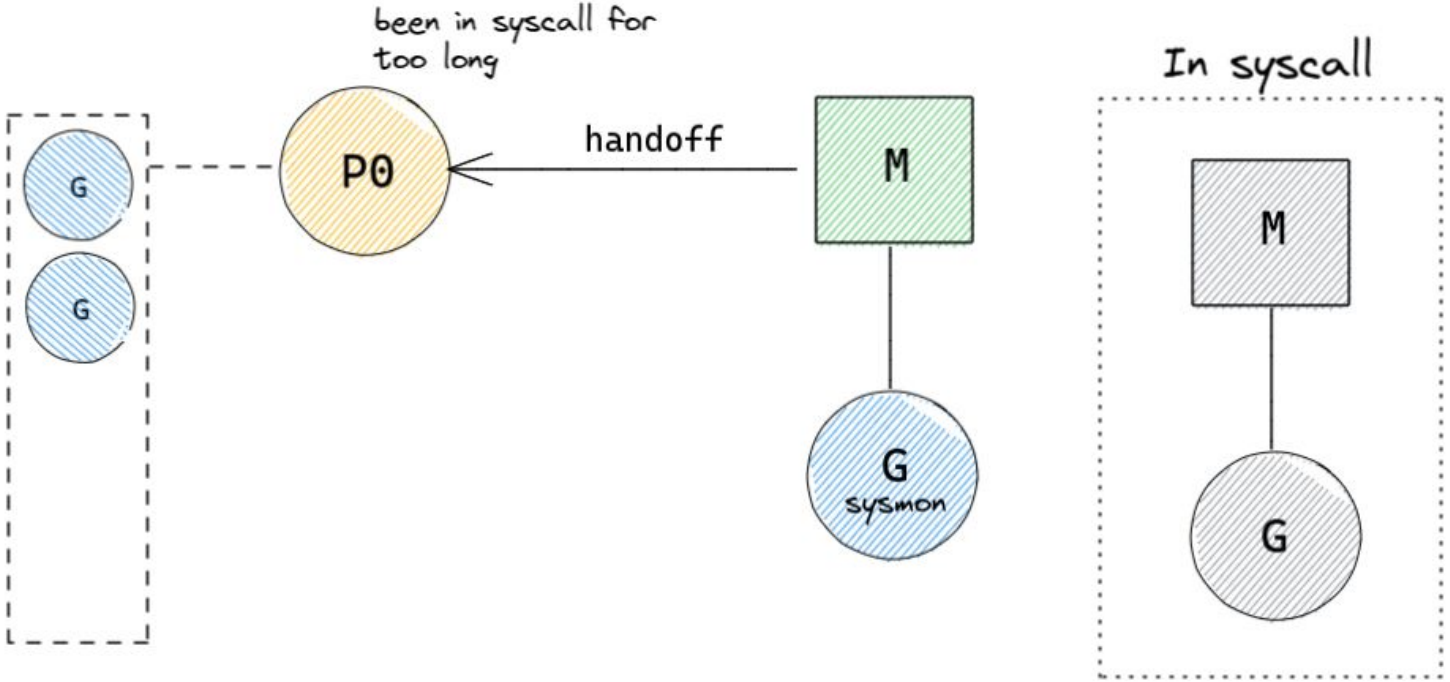
`handoff` can get expensive

- If sysmon sees that a p has been in the executing `syscall` state for too long, it initiates a `handoff`.



`handoff` can get expensive

- If sysmon sees that a p has been in the executing `syscall` state for too long, it initiates a `handoff`.



What happens when the `syscall` returns?

- The scheduler tries to schedule this Goroutine on it's old `p` (the one it was on before going into a `syscall`).
- If that is not possible, it tries to get an available idle `p` and schedule it on there.
- If no idle `p` is available, the scheduler puts this Goroutine on the global queue.
    - Subsequently it also parks the thread that was in the `syscall`.

Awesome! We now have a fairly good idea about what happens under the hood. Yay!

But all this is taken care of by the runtime itself, are there any knobs we can turn to try and control some of this behaviour?

`runtime` APIs to interact with the scheduler

- Try and treat the runtime as a blackbox as much as possible!
- (It's a good thing that) Not a lot of exposed knobs to control the runtime.
- Whatever *is* available should be understood thoroughly before using in code.

`runtime` APIs to interact with the scheduler

- `NumGoroutine()`
- `GOMAXPROCS()`
- `Gosched()`
- `Goexit()`
- `LockOSThread()/UnlockOSThread()`

`GOMAXPROCS()`

- Sets the value of GOMAXPROCS
- If changed after program is started, it will lead to a **stop the world** operation!

`Gosched()`

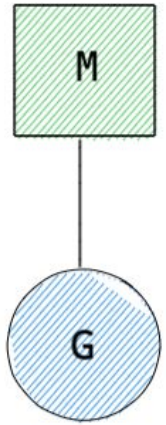
- Yields the processor.
- Calling Goroutine is sent to global queue.
- If you plan to use it for performance reasons, it’s likely that improvement can be done in your implementation itself.
- Use only if absolutely necessary!

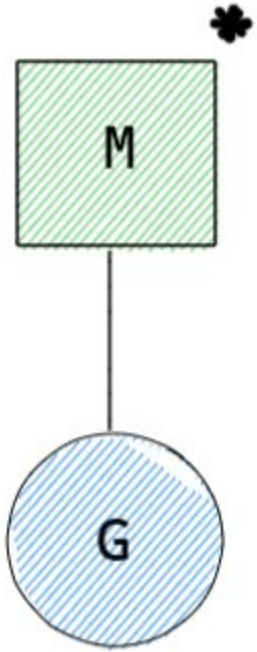
`Goexit()`

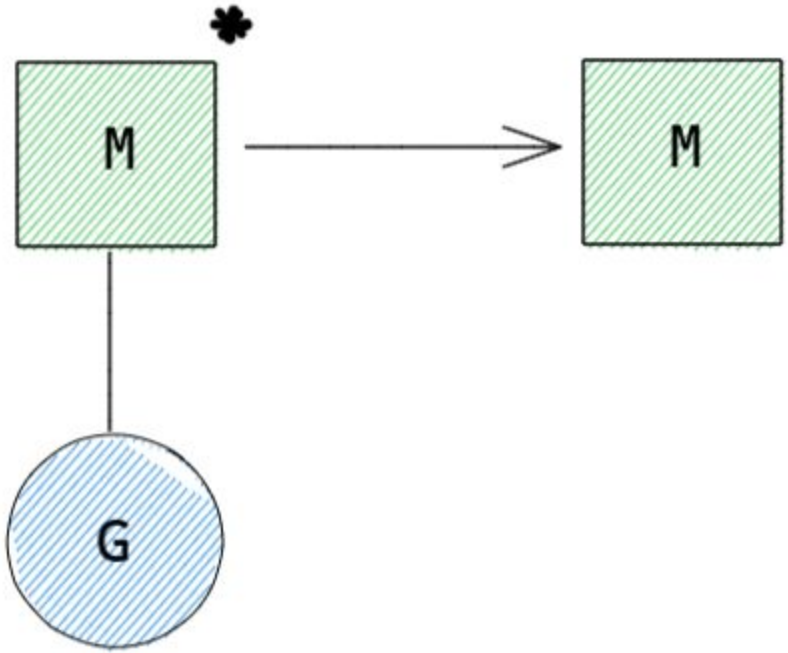
- Terminates (only) calling Goroutine.
- If called from main Goroutine, it terminates, and other Goroutines continue to run.
    - Program crashes once Goroutines finish because main Goroutine does not return.
- Used in `testing` (`t.Fatal()`)

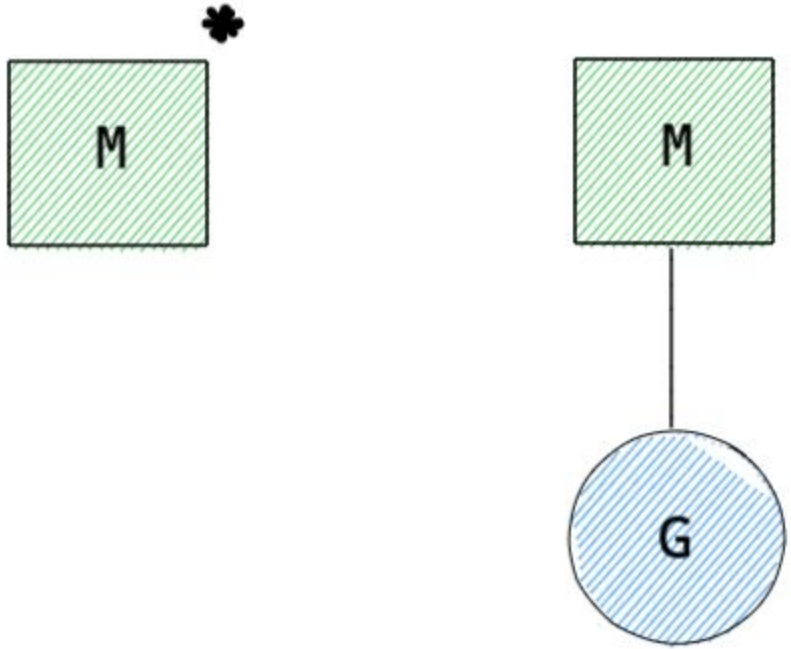
`LockOSThread()/UnlockOSThread()`

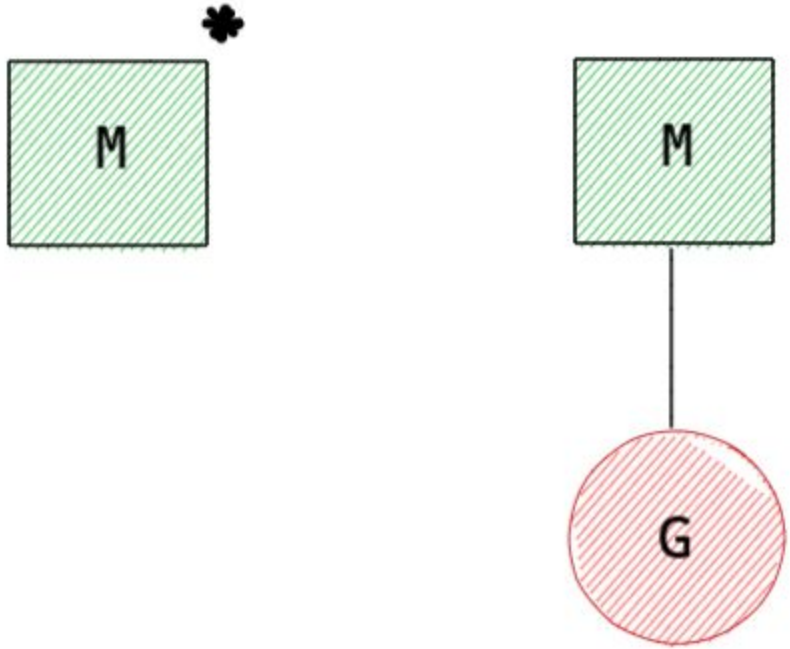
- Wires calling Goroutine to the underlying OS Thread.
- Primarily used when the Goroutine changes underlying thread's state.




M
G


M
M
G


M
M
G


M
M
G

`LockOSThread()/UnlockOSThread()`

- Weaveworks has an excellent case-study on this:
  - https://www.weave.works/blog/linux-namespaces-and-go-don-t-mix
  - https://www.weave.works/blog/linux-namespaces-golang-followup
- Let's look at the fineprint.

`LockOSThread()/UnlockOSThread()`

- Acts like a “taint” indicating thread state was changed.
- No
  - Goroutine can be scheduled on this thread till `UnlockOSThread()` is called the same number of times as `LockOSThread()`.
  - Thread can be created from a locked thread.
- Don’t create Goroutines from a locked one that are expected to run on the modified thread state.
- If a Goroutine exits before unlocking the thread, the thread is gotten rid of and is not used for scheduling anymore.

Phew - that's a lot of information, but congratulations on making it this far, you're awesome!

https://github.com/MadhavJivrajani/gse

Conclusion

- Go’s scheduler is distributed and not centralized.
- Fairness is kept at the forefront of the design next to scalability.
- Scheduler design factors in domain specific knowledge along with language specific patterns.
- Understand runtime APIs well before using them - use only if necessary.
- Be especially careful when changing thread state.

References

- Scalable Go Scheduler Design Doc
- Go scheduler: Implementing language with lightweight concurrency
- The Scheduler Saga
- Analysis of the Go runtime scheduler
- Non-cooperative goroutine preemption
  - Pardon the Interruption: Loop Preemption in Go 1.14
- `go/src/runtime/{proc.go, proc_test.go, preempt.go, runtime2.go, ...}`
  - And their corresponding `git blame`s
- Go's work-stealing scheduler

Thank you!