

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

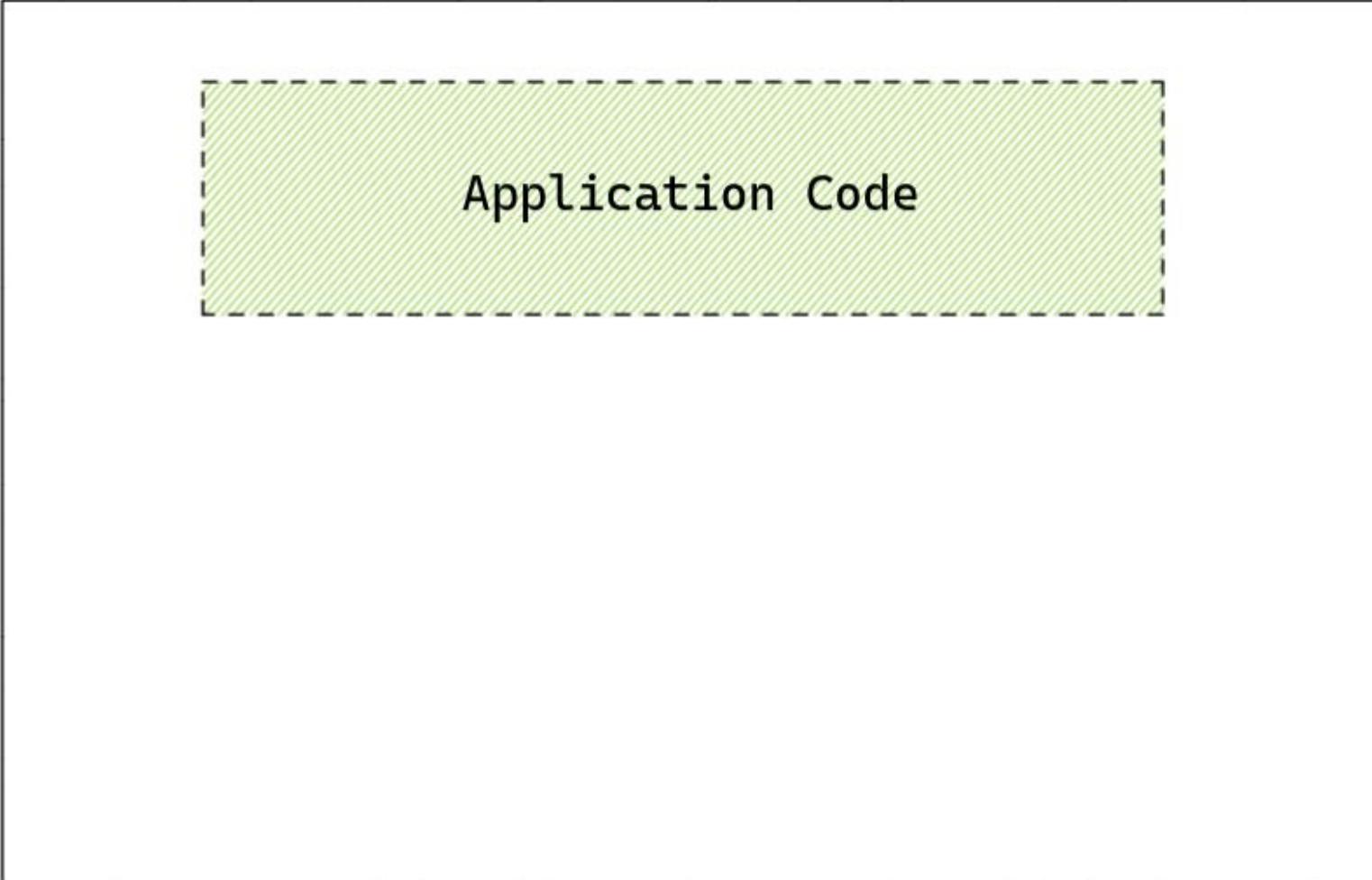
```
// An abridged main.go
func main() {
    go doSomething()
    doAnotherThing()
}
```

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

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

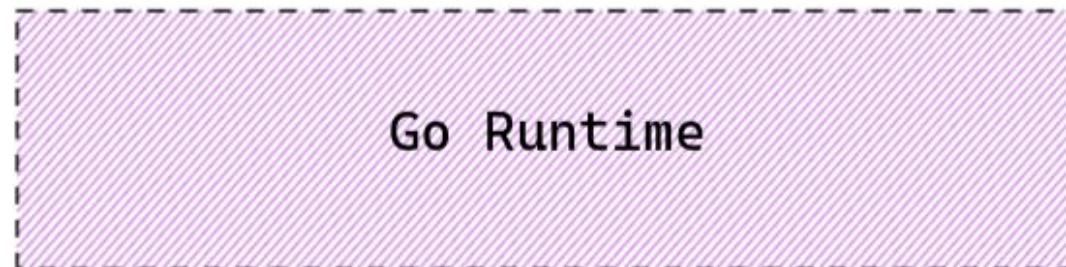
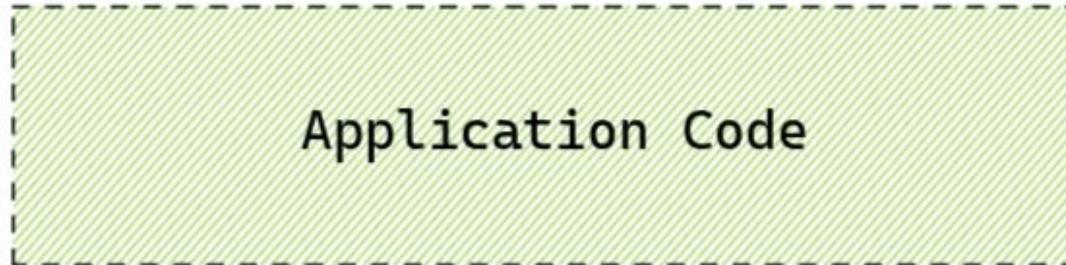
app

app

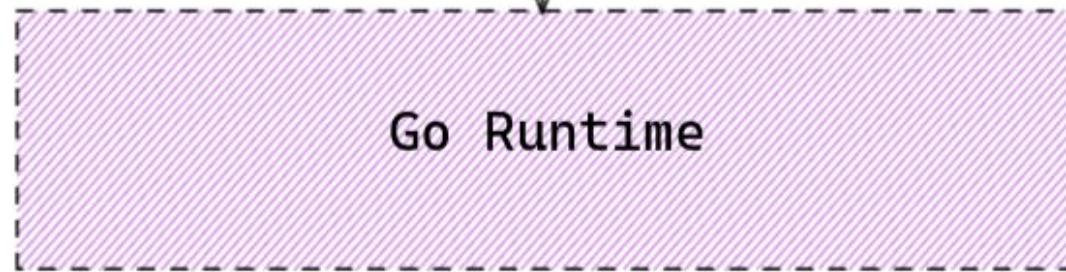
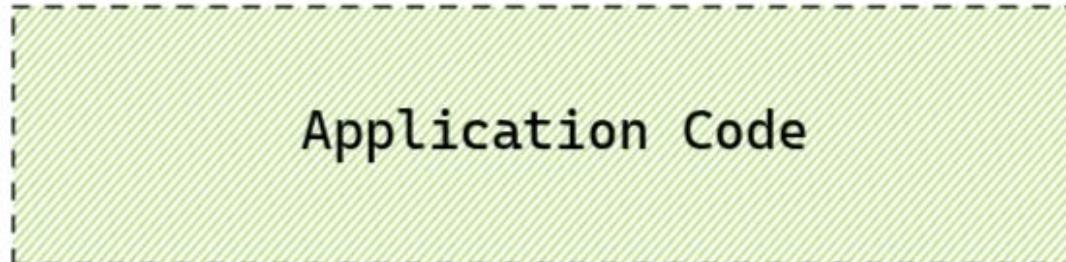


Application Code

app



app



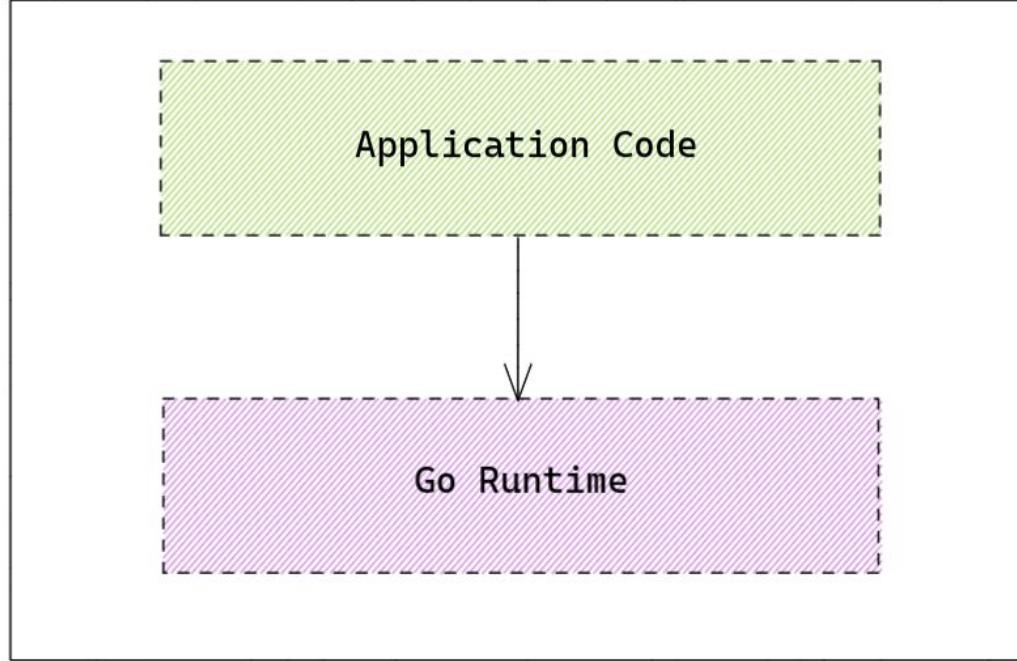
One such example of calling into the runtime

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



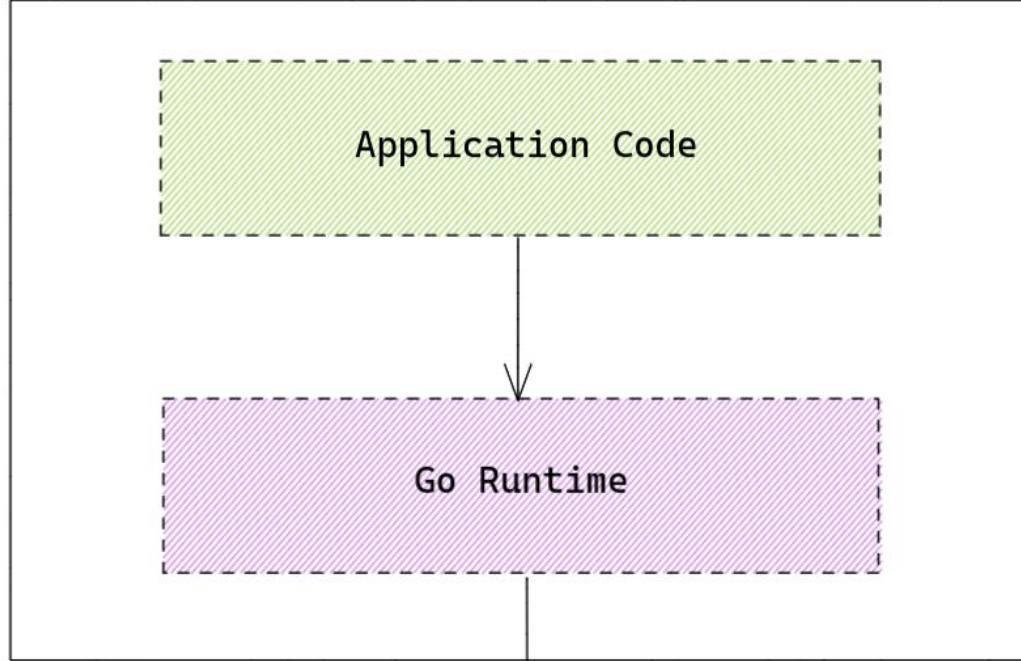
```
func main() {  
    runtime.newProc(...)  
    doAnotherThing()  
}
```

app



Operating System

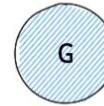
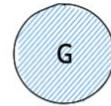
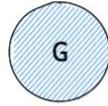
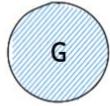
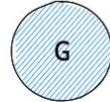
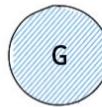
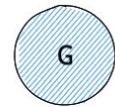
app

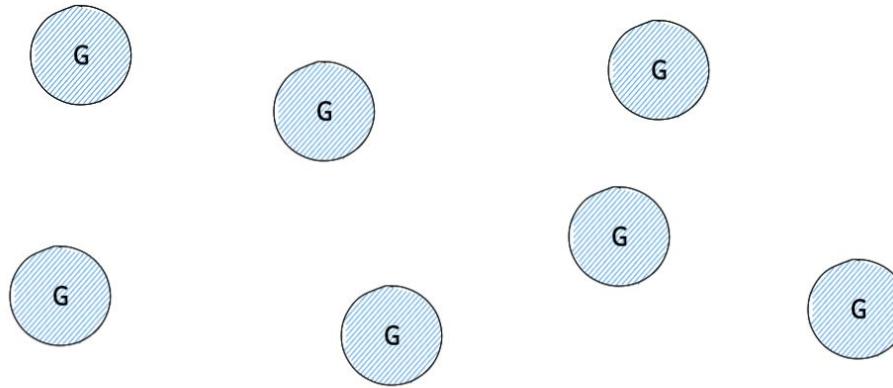


Operating System

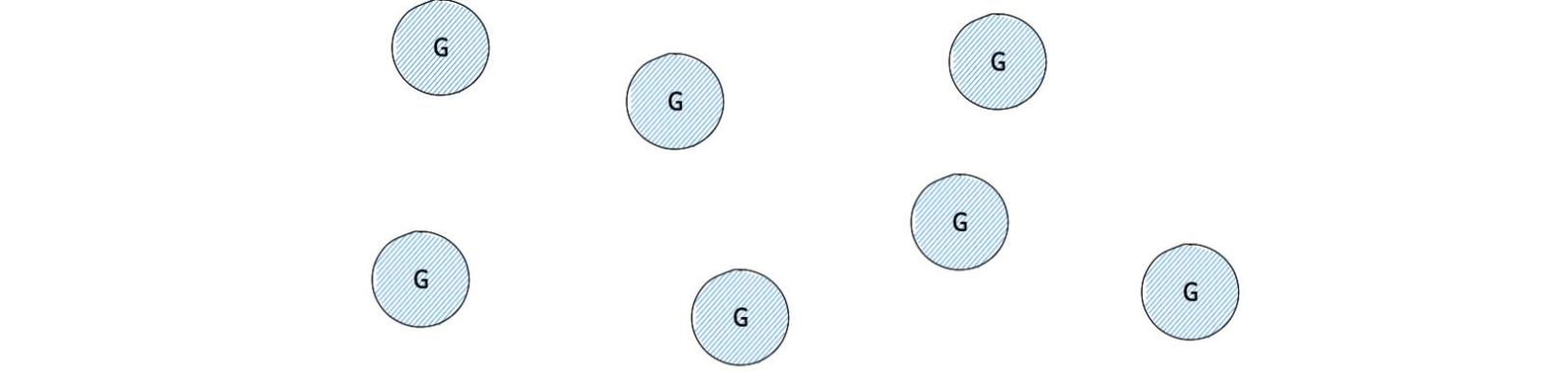
Let's actually run our code.

./app



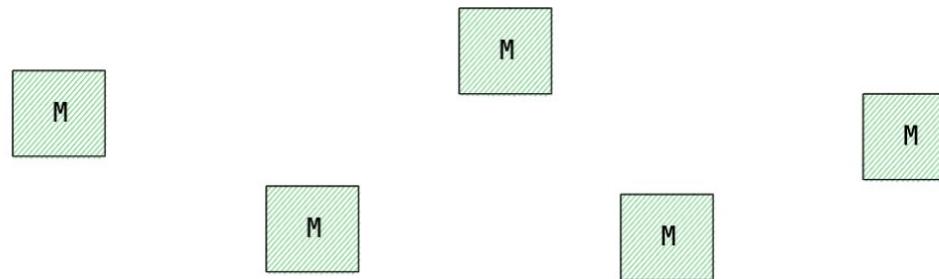


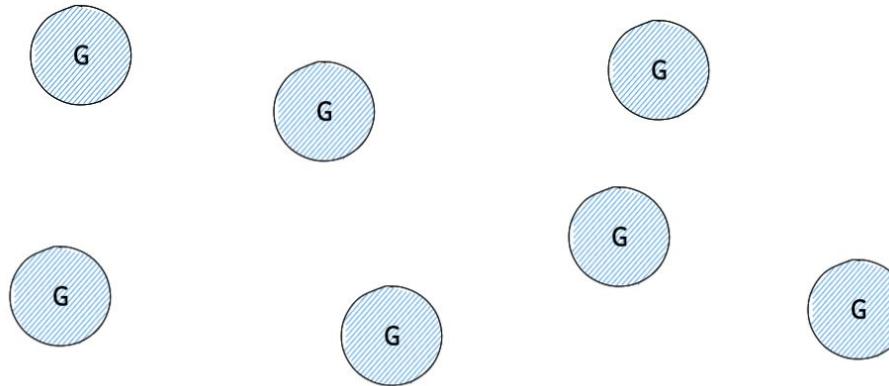
User space



User space

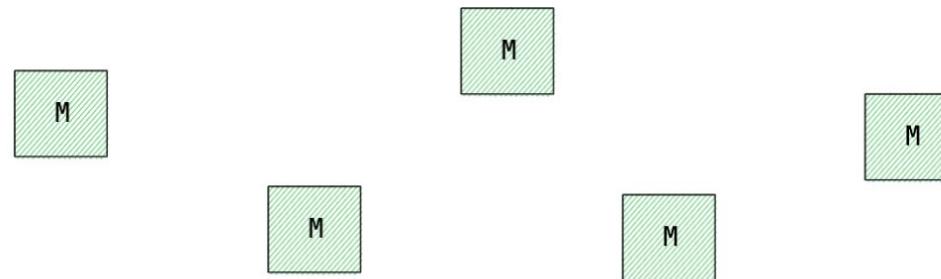
Kernel space





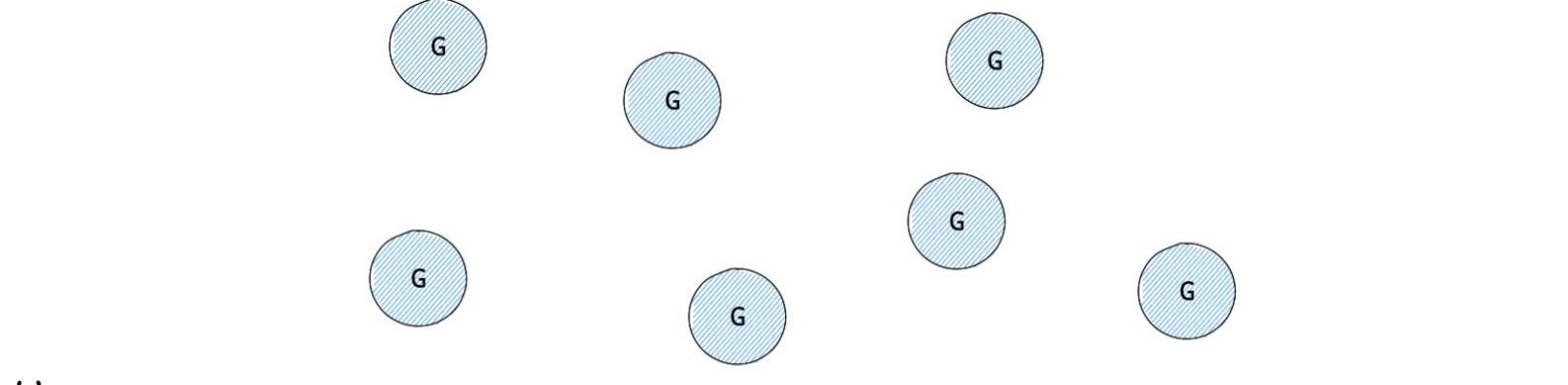
User space

Kernel space



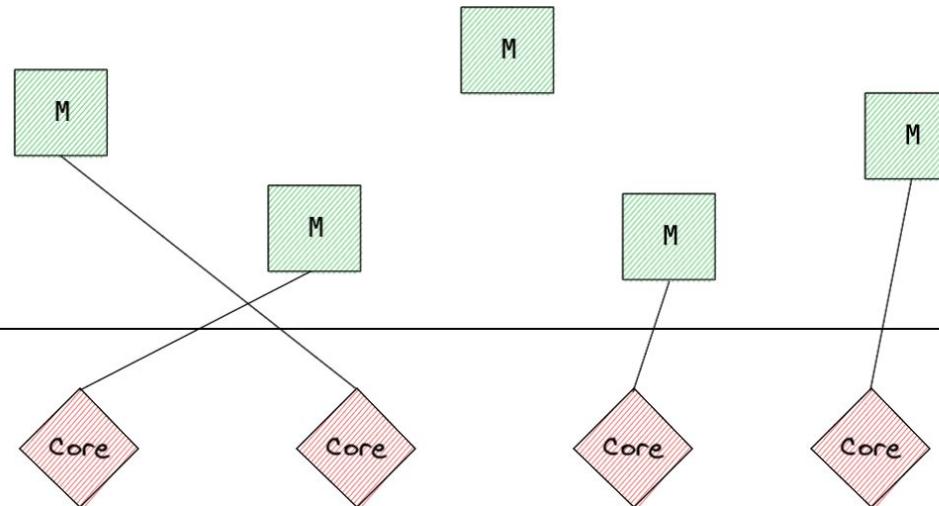
Hardware





User space

Kernel space



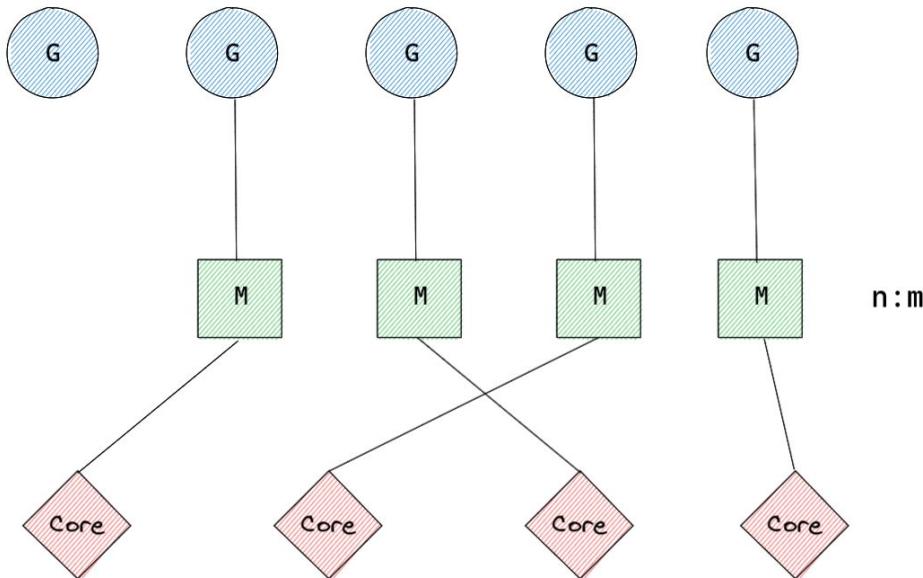
Hardware

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

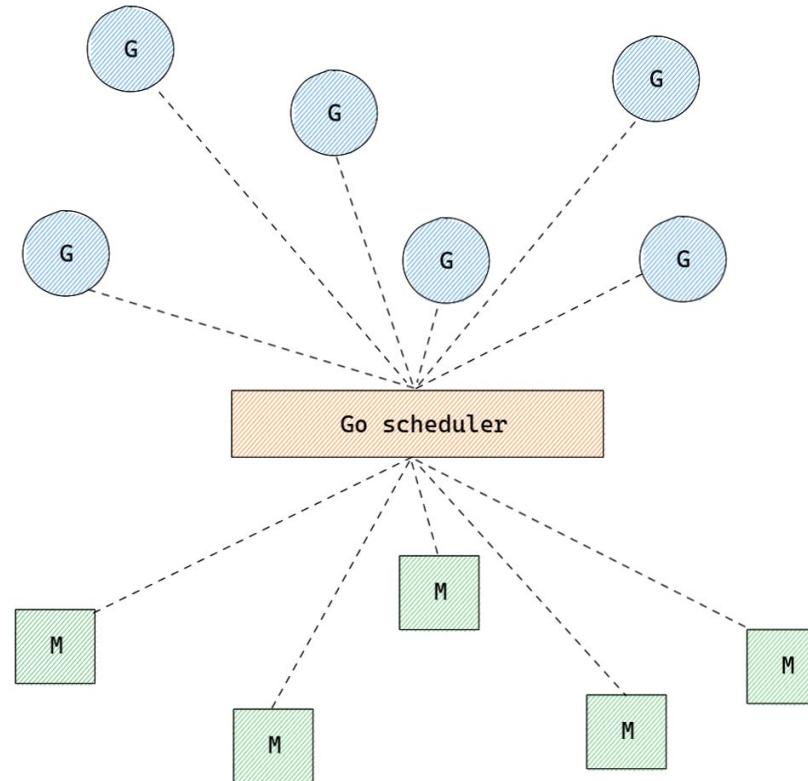
We need some way to map Goroutines to OS threads - user-space scheduling!

$n:m$ 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.

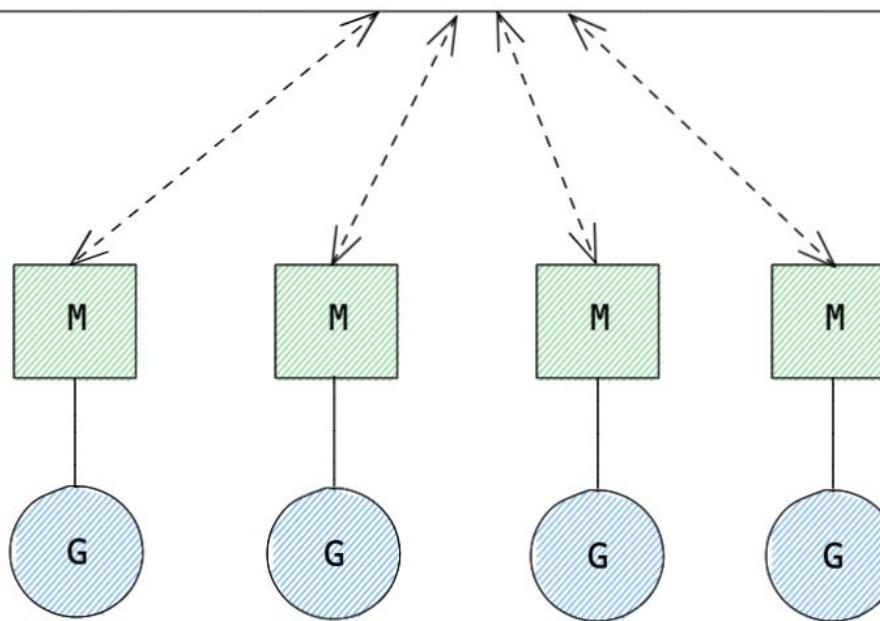
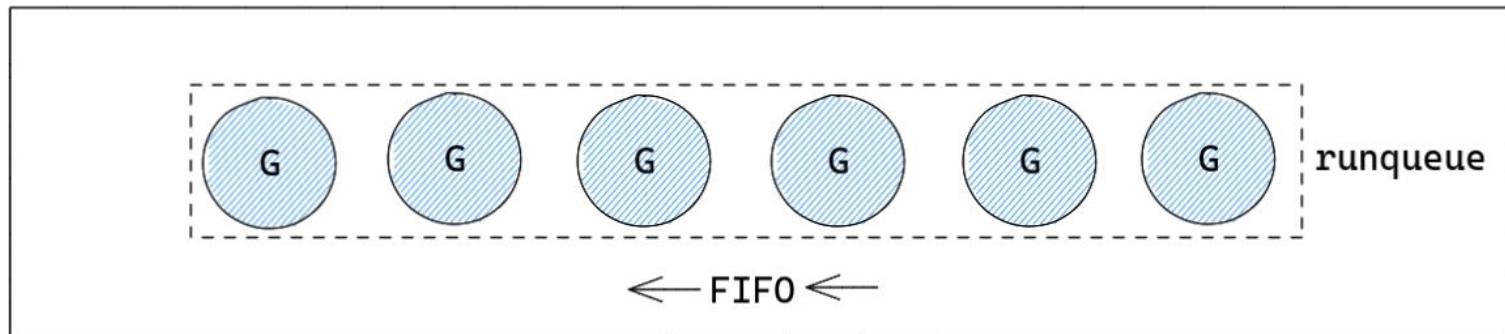


The Go scheduler does N-M scheduling.



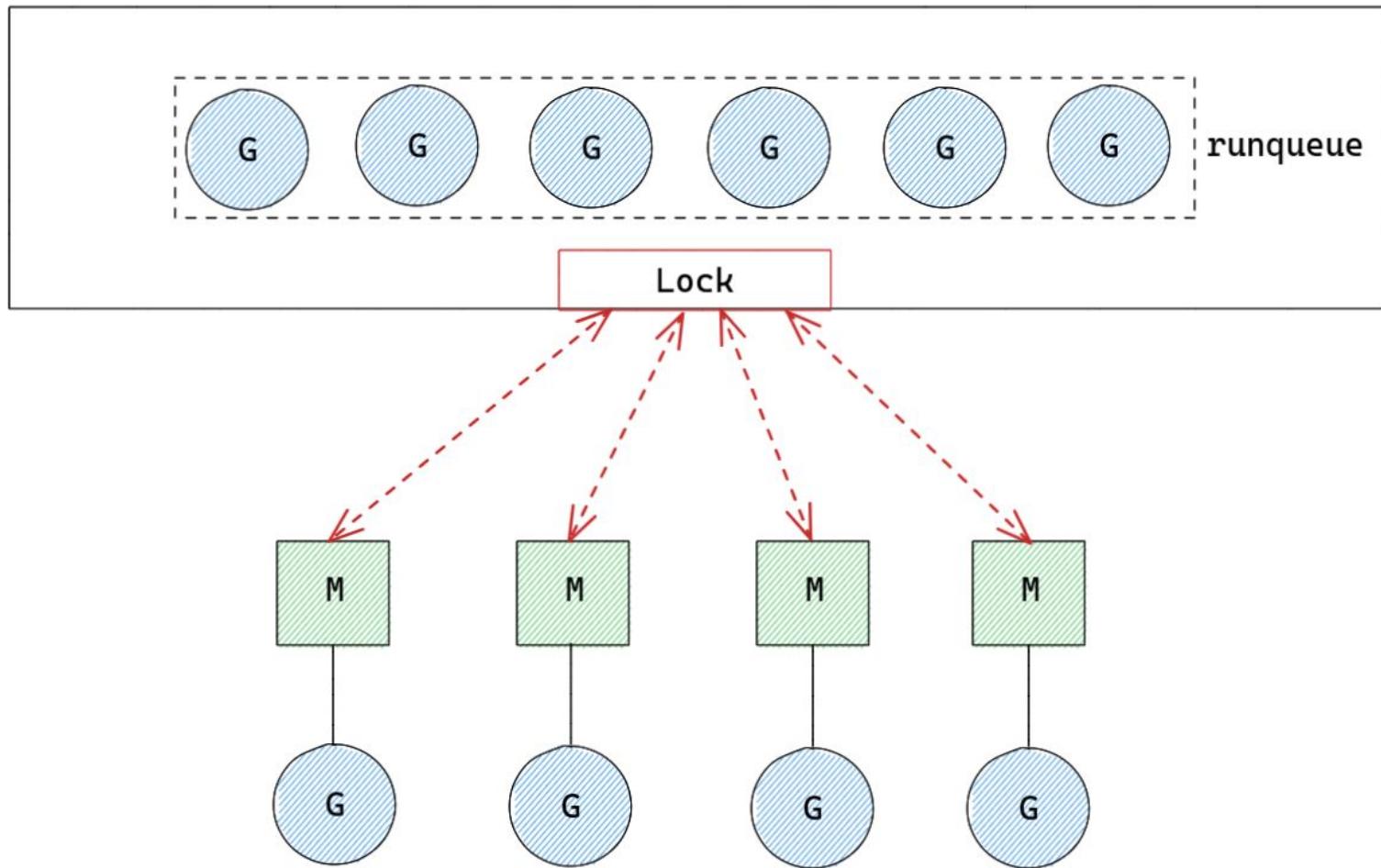
How do we keep track of Goroutines that are yet to be run?

Go scheduler



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

Go scheduler



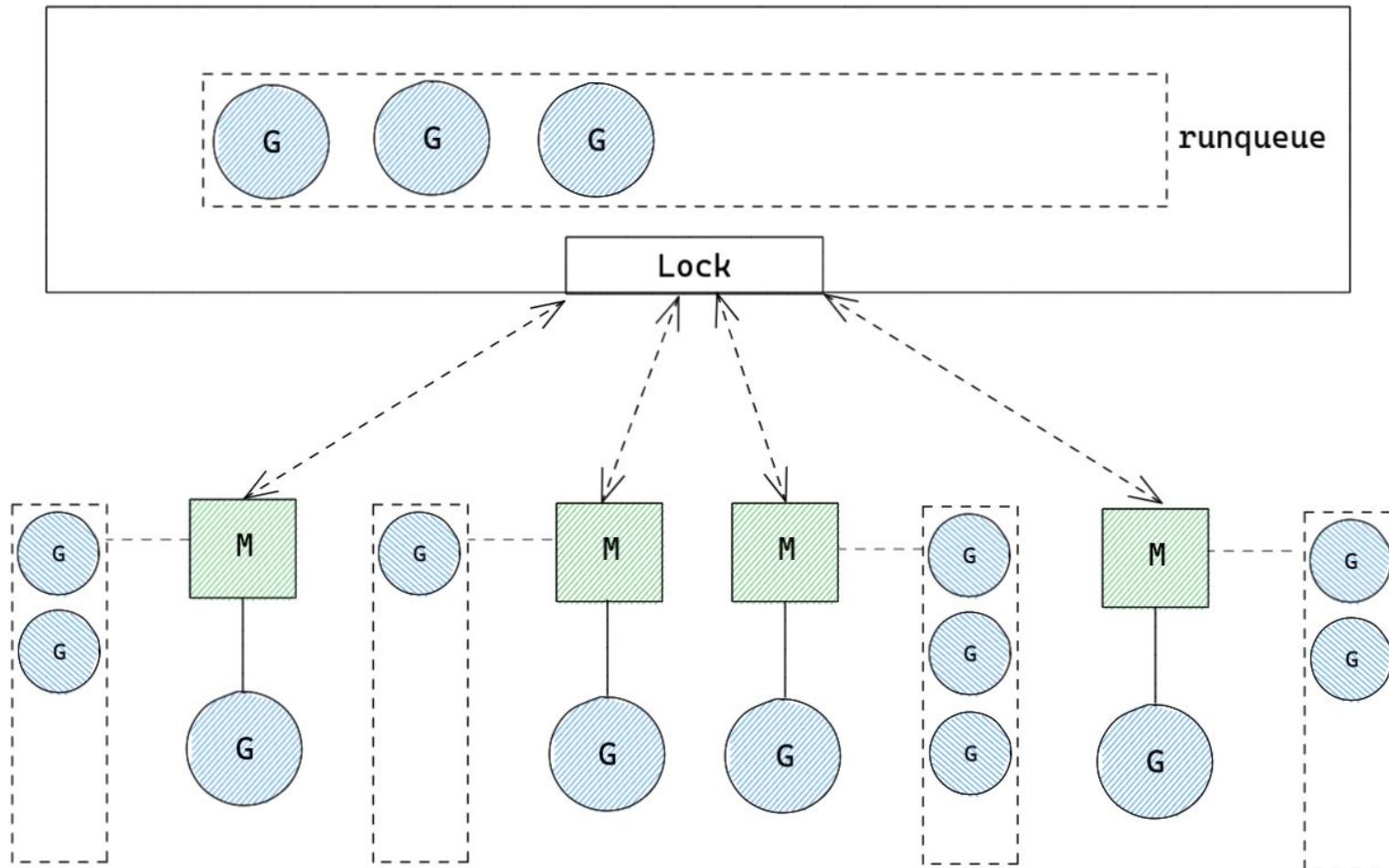
What if the running Goroutines happen to be long-running tight loops?

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

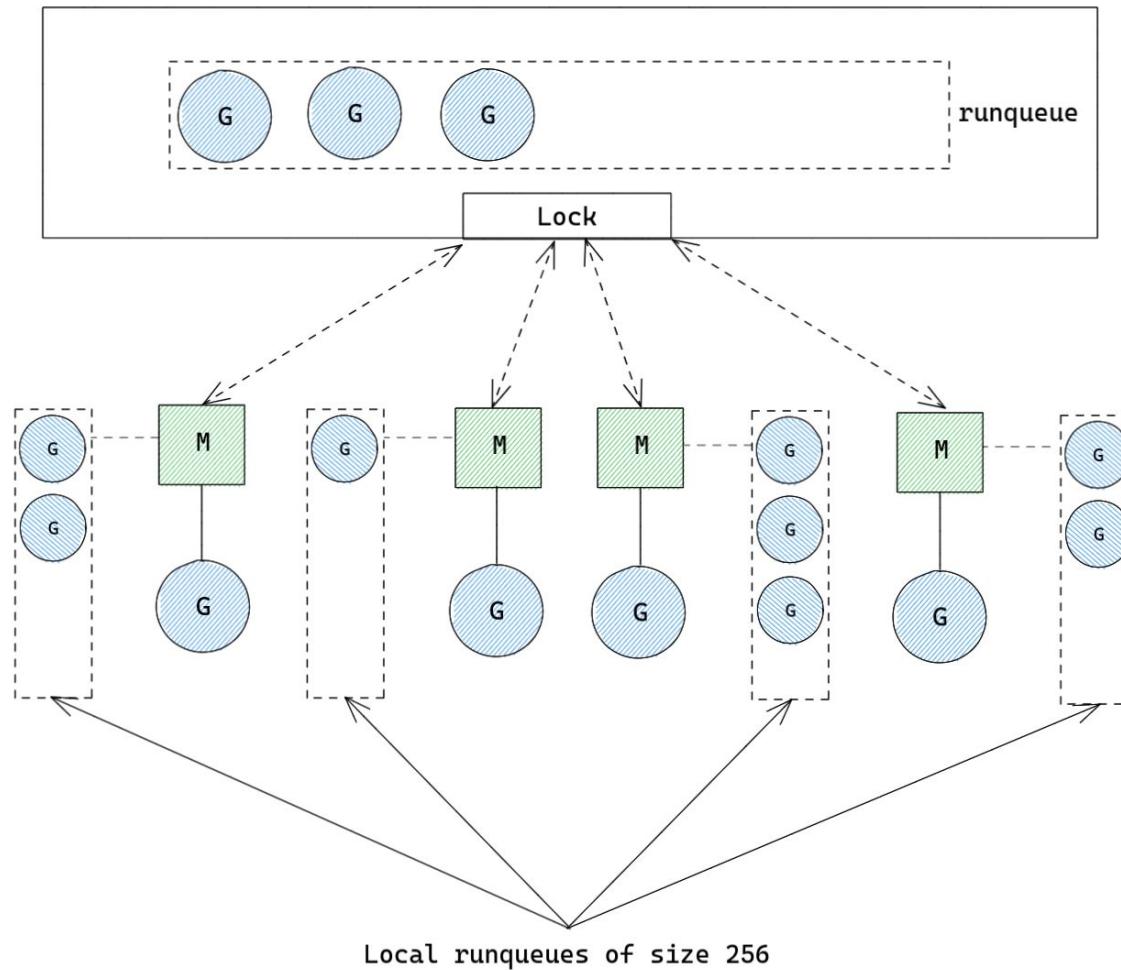
- We could try time-multiplexing.
 - But then with just one global runqueue, each Goroutine would end up getting a short slice of time - leading to poor locality and excessive context switching.

To begin addressing these challenges, the notion of *distributed runqueues* was introduced.

Go scheduler



Go scheduler



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

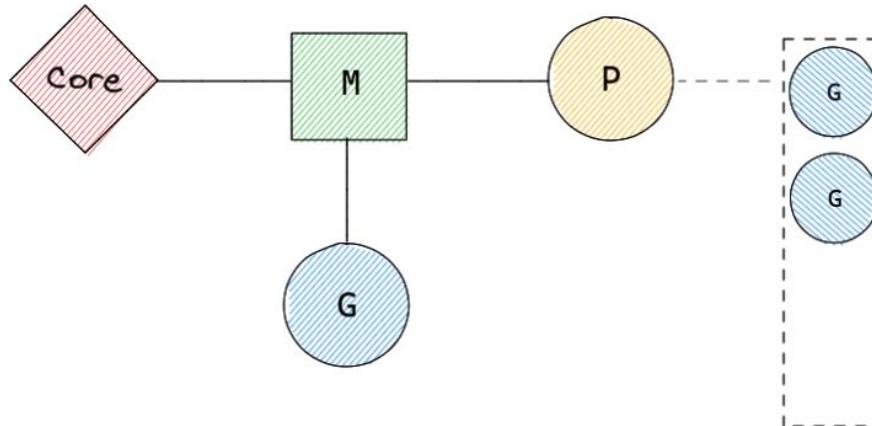
- This implies that there could be a significantly large number of threads that are blocked and not actively executing Goroutines.
- But we maintain some amount of per-thread state (the local `runqueue` being part of it).
 - We don't really need all this state for threads that are not actually executing 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?

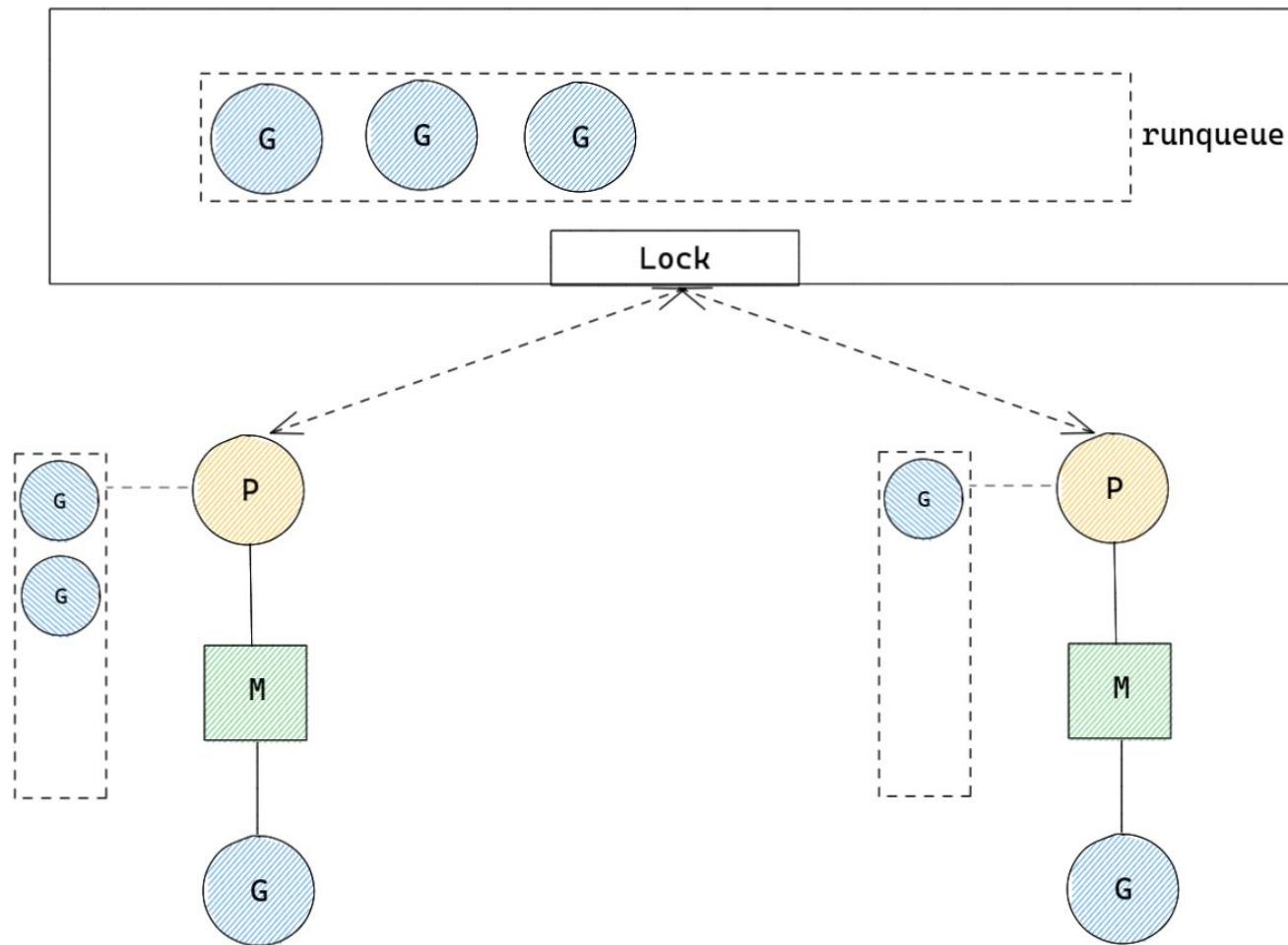
✨ Indirection ✨

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



“Go scheduler: Implementing language with lightweight concurrency”

by Dmitry Vyukov

Interlude: Fairness

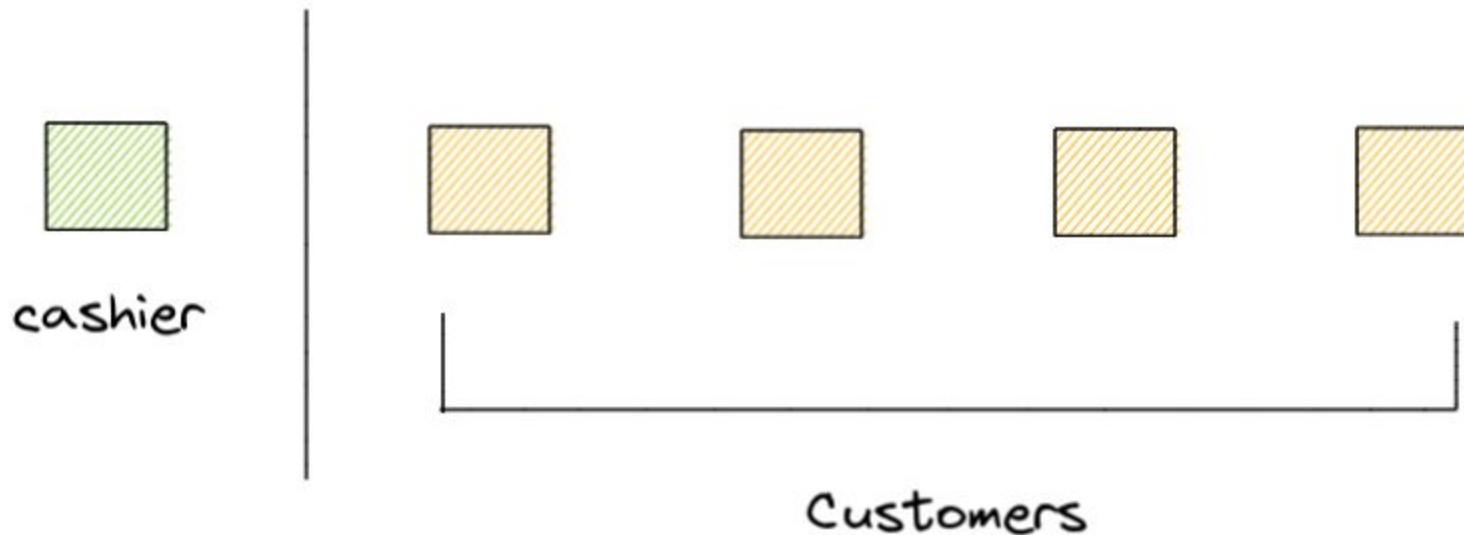
Let's consider the following scenario.

Supermarket

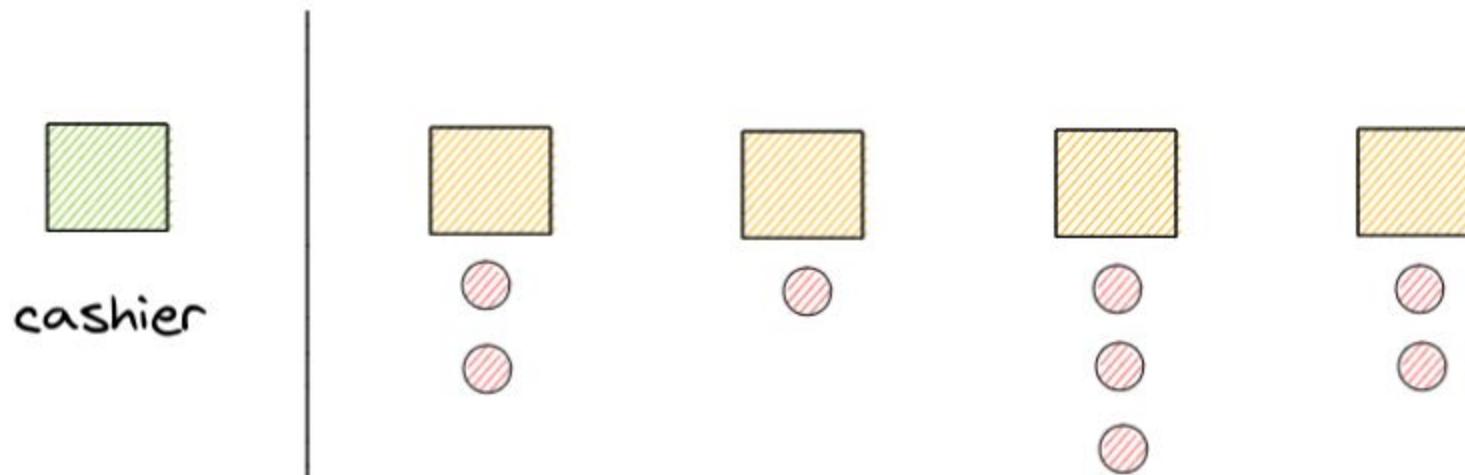


cashier

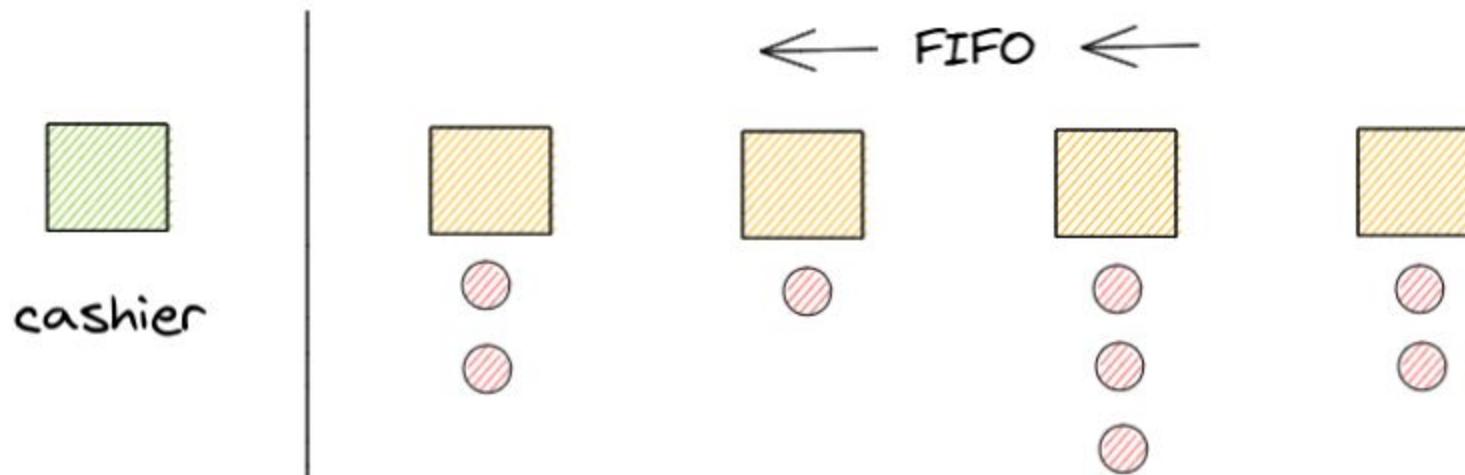
Supermarket



Supermarket



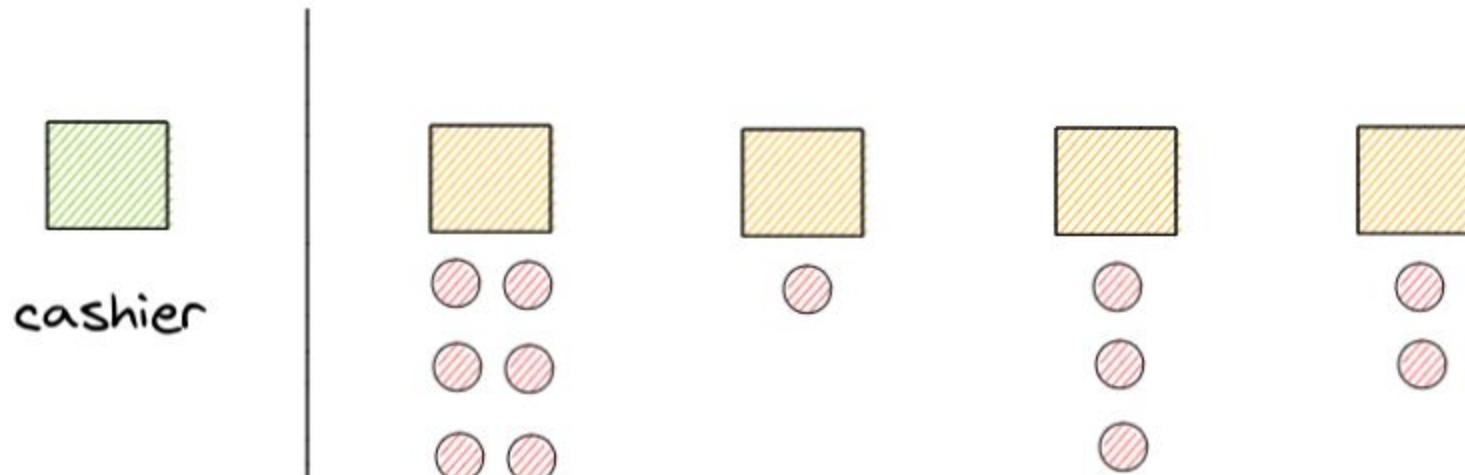
Supermarket



This looks fine.

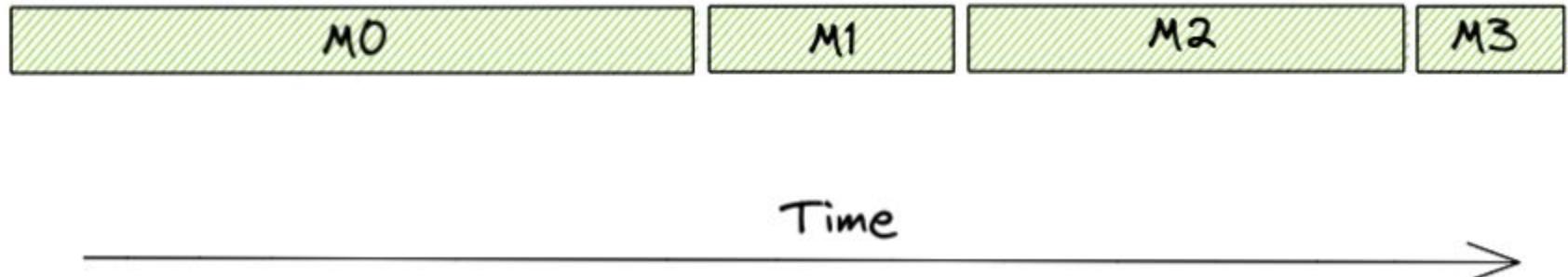
But, what if...

Supermarket



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

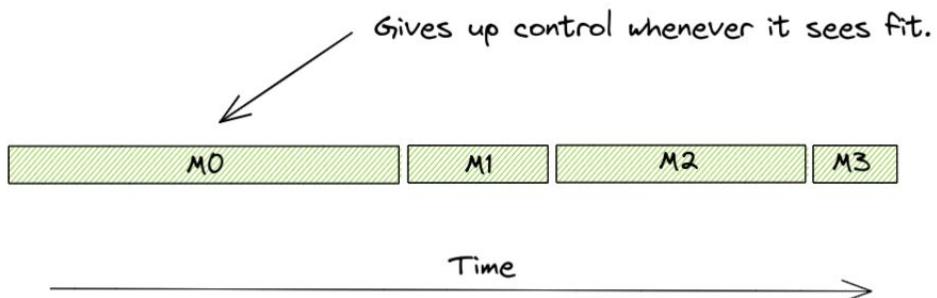
This is a common problem to deal with while considering fairness in scheduling.



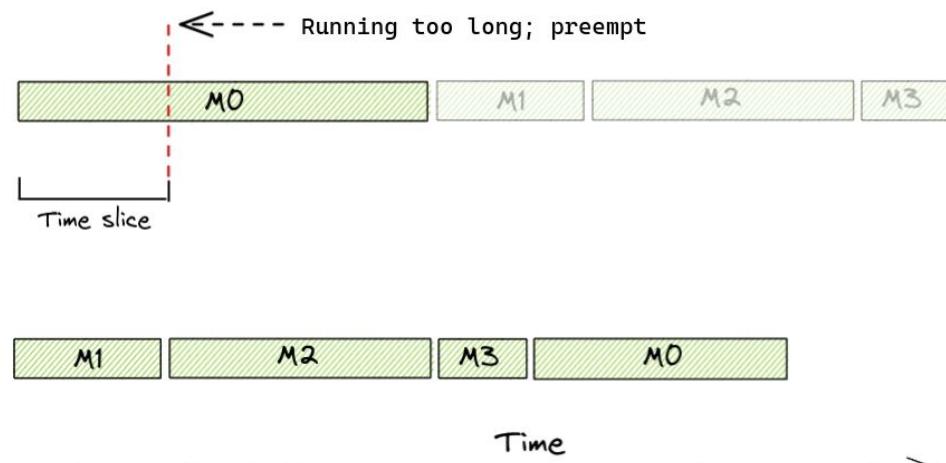
How do we deal with this in scheduling?

- 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!

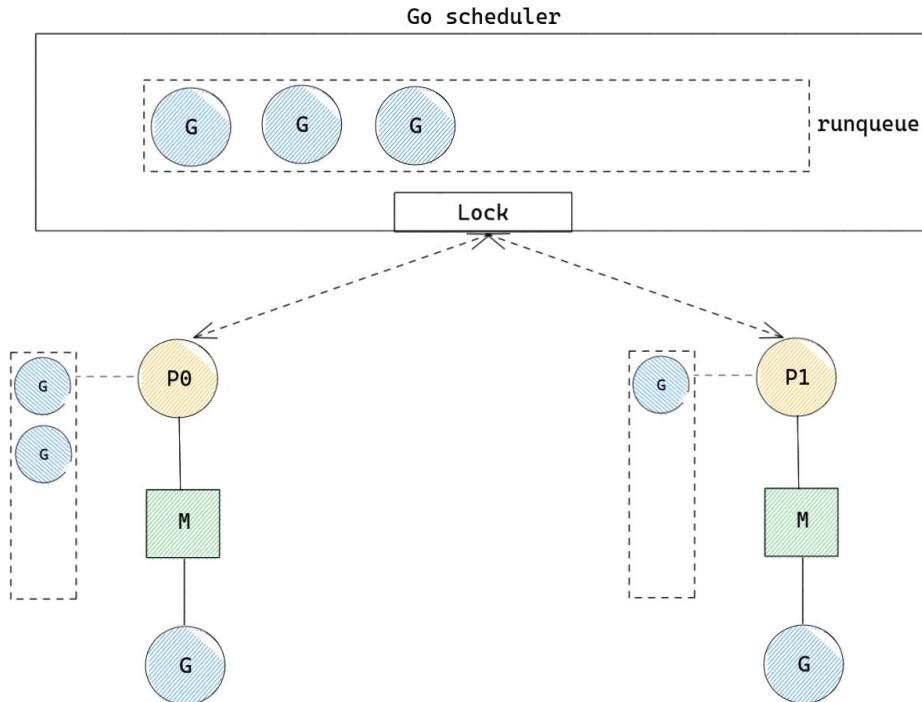
Co-operative Pre-emption



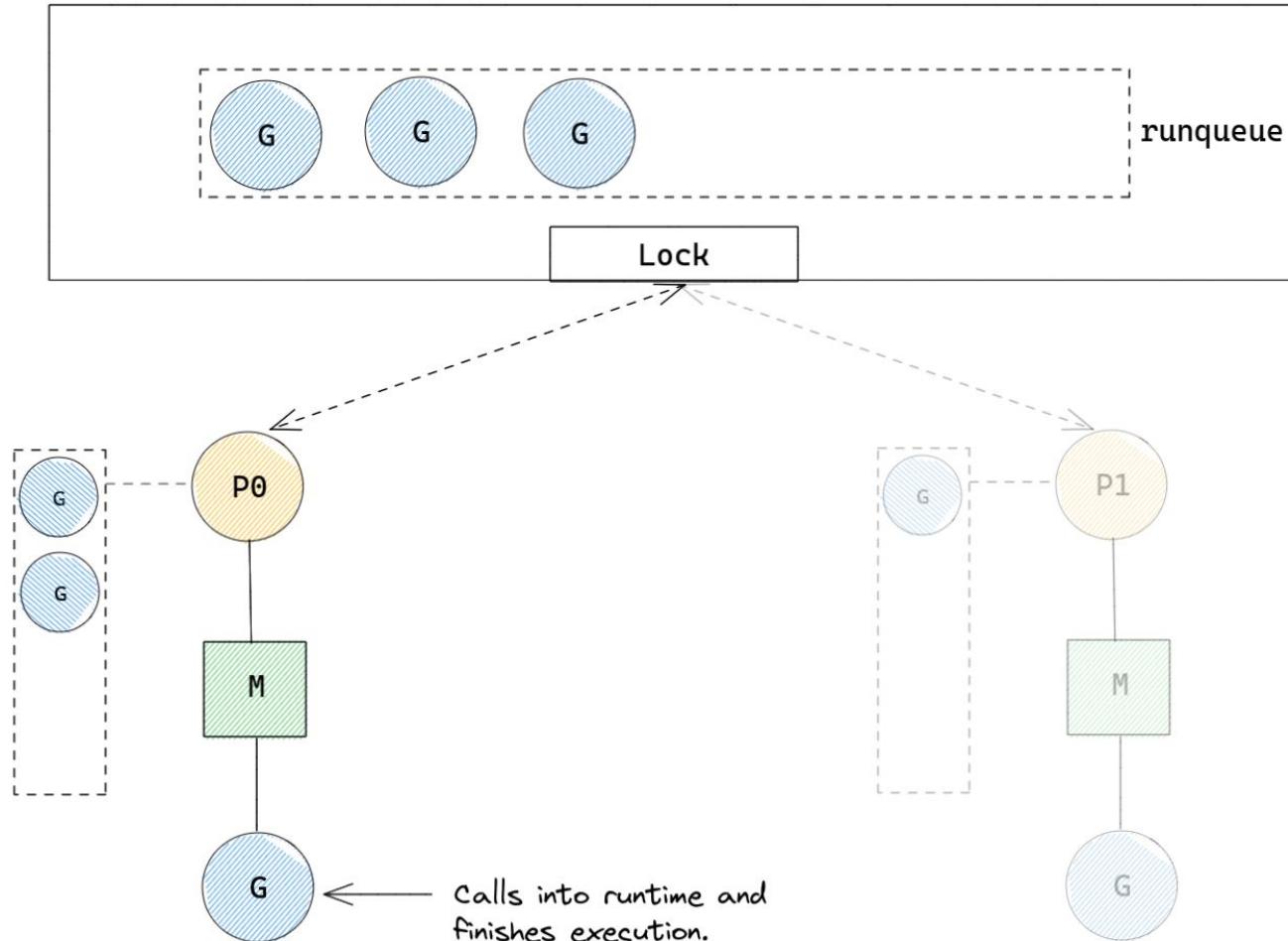
Non Co-operative Pre-emption



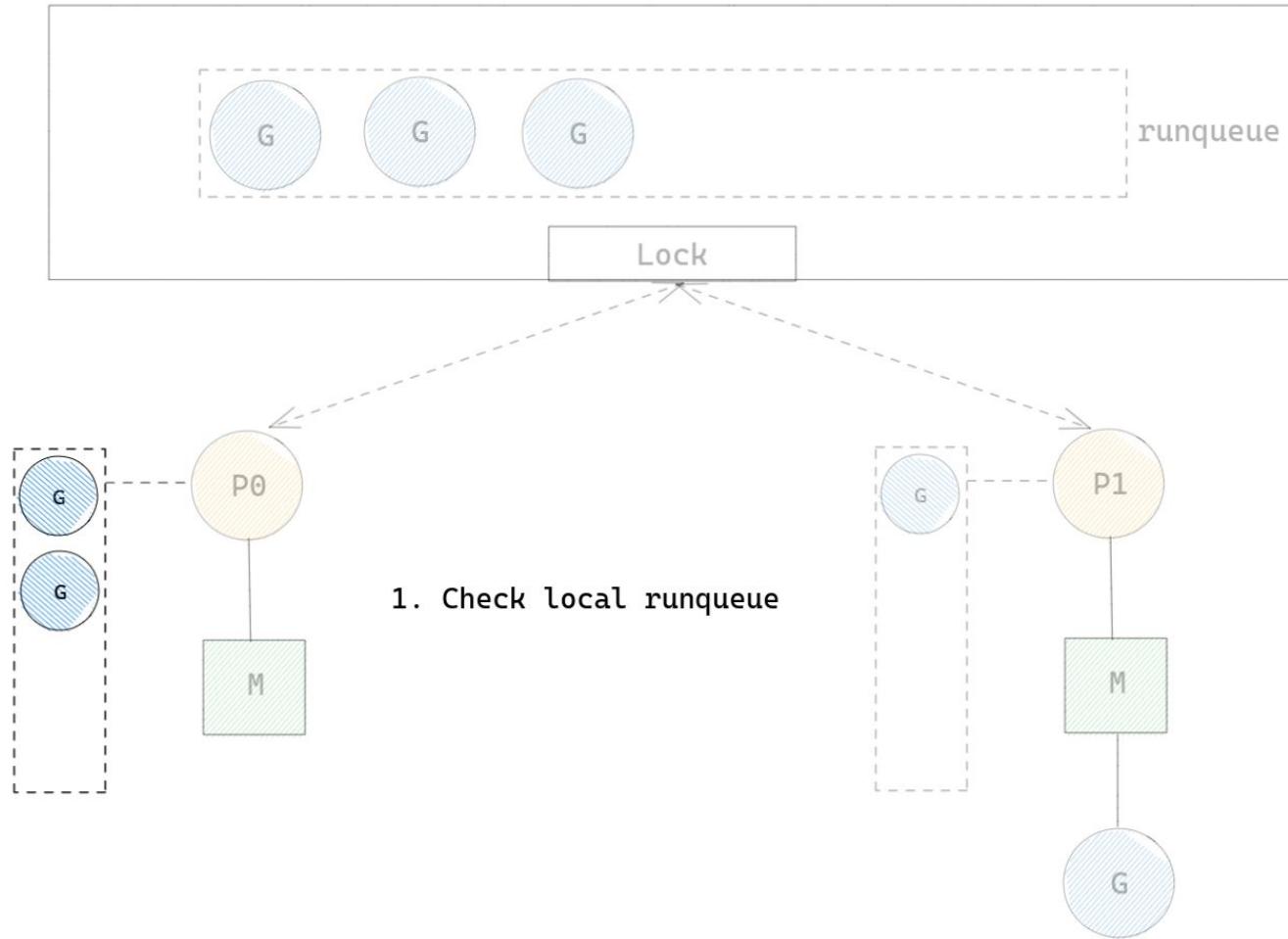
Alright, now that we have enough context, the first thing to ask ourselves is “how do we choose which Goroutine to run?”



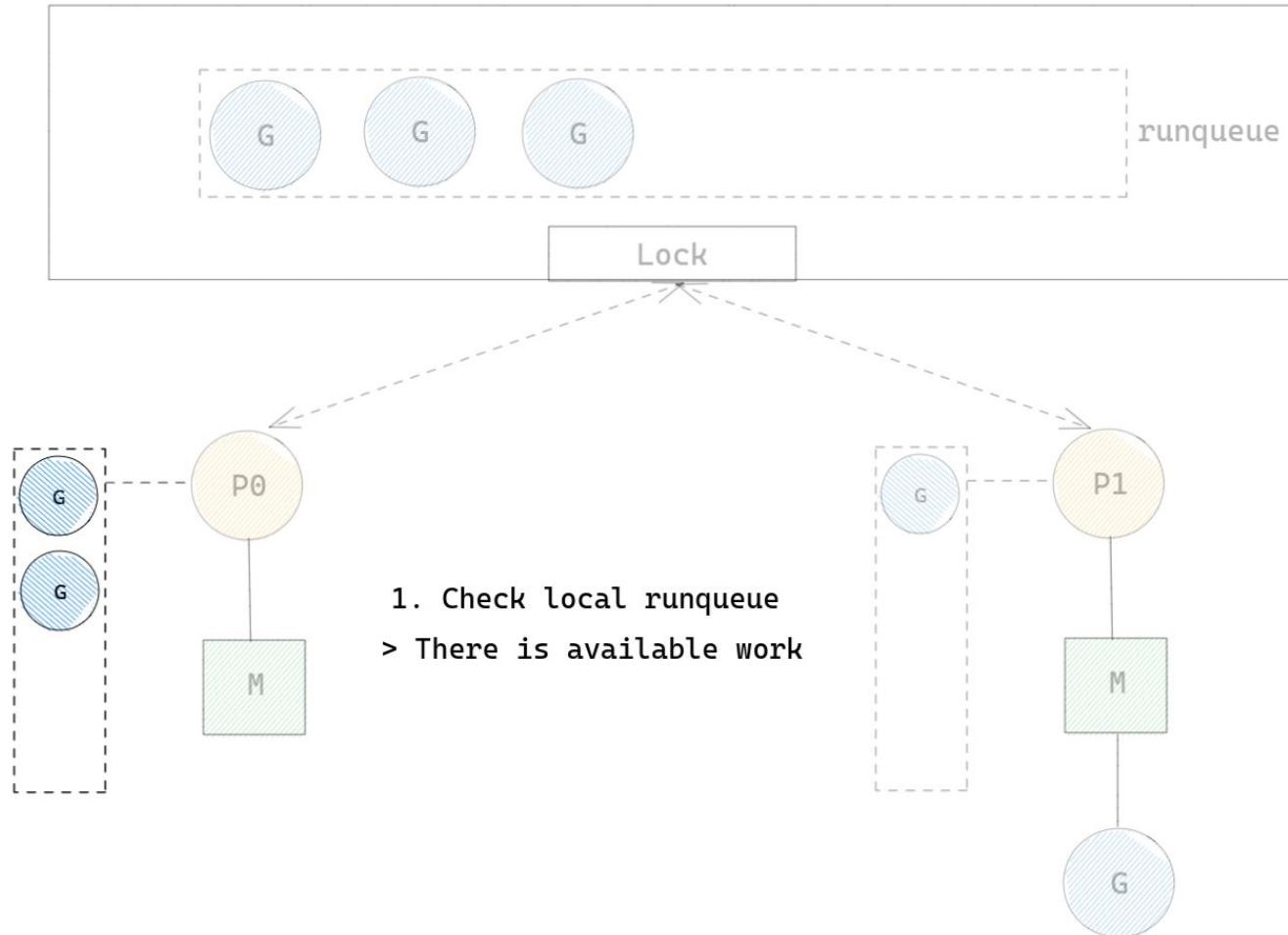
Go scheduler



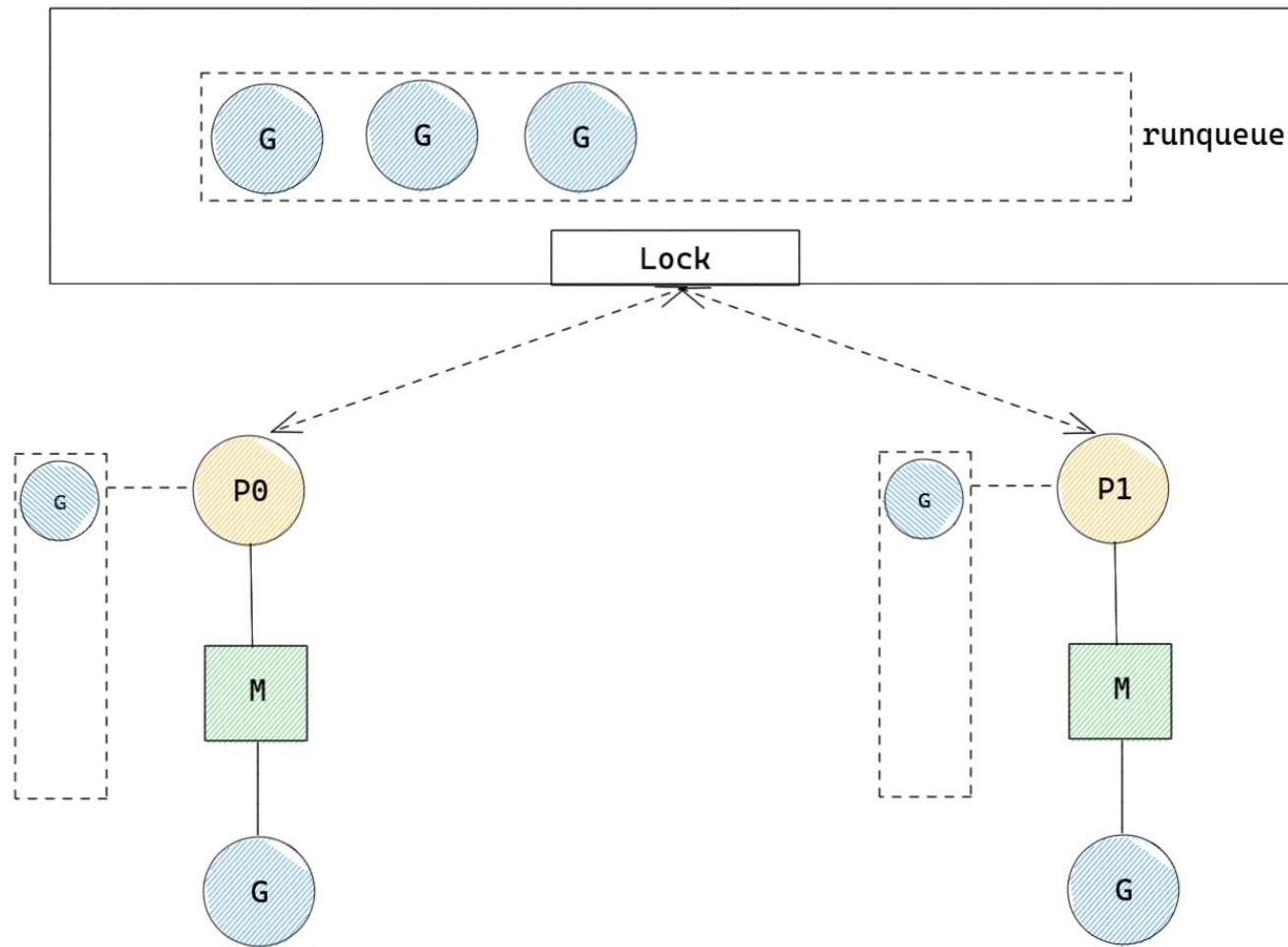
Go scheduler



Go scheduler

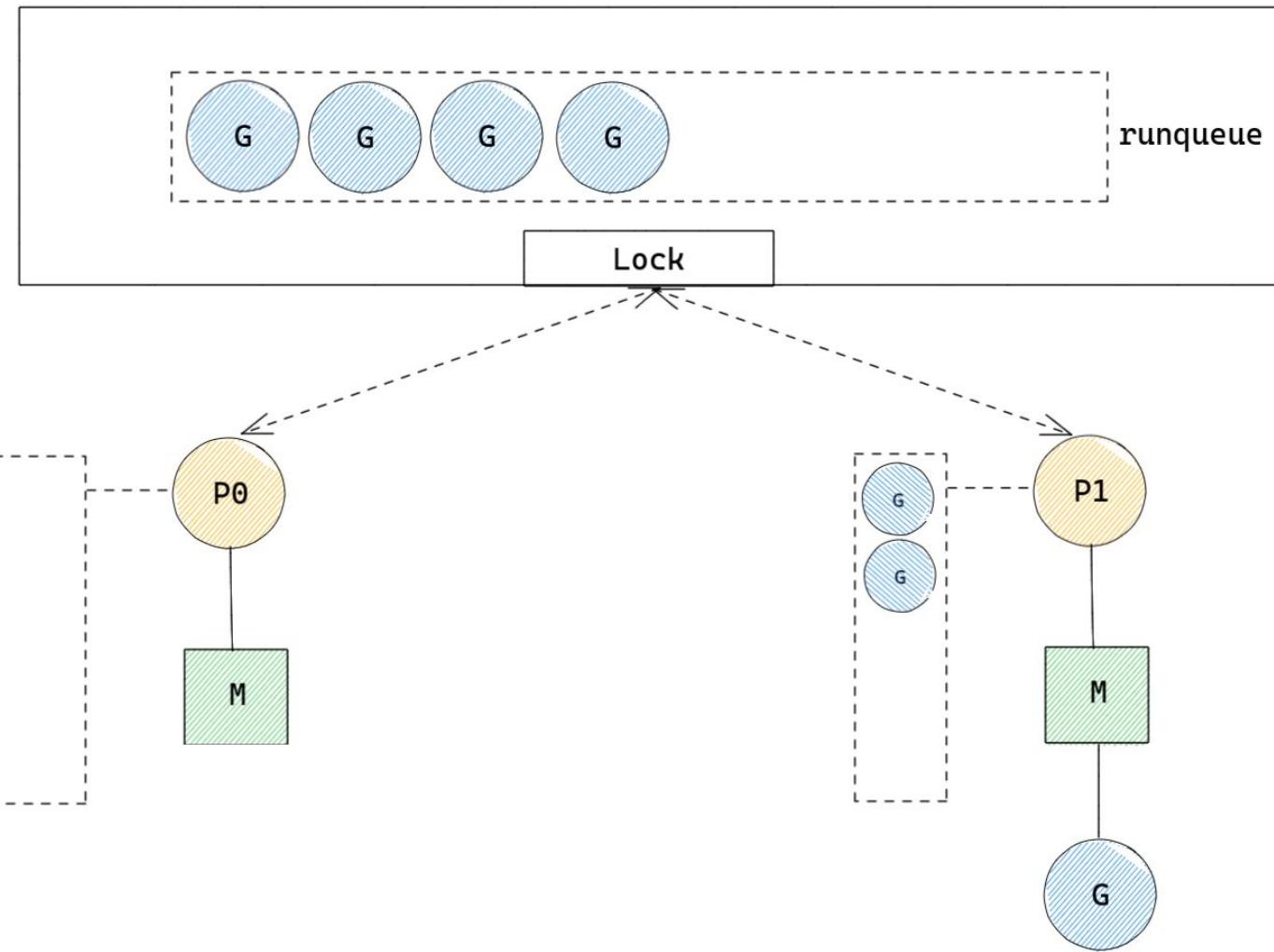


Go scheduler

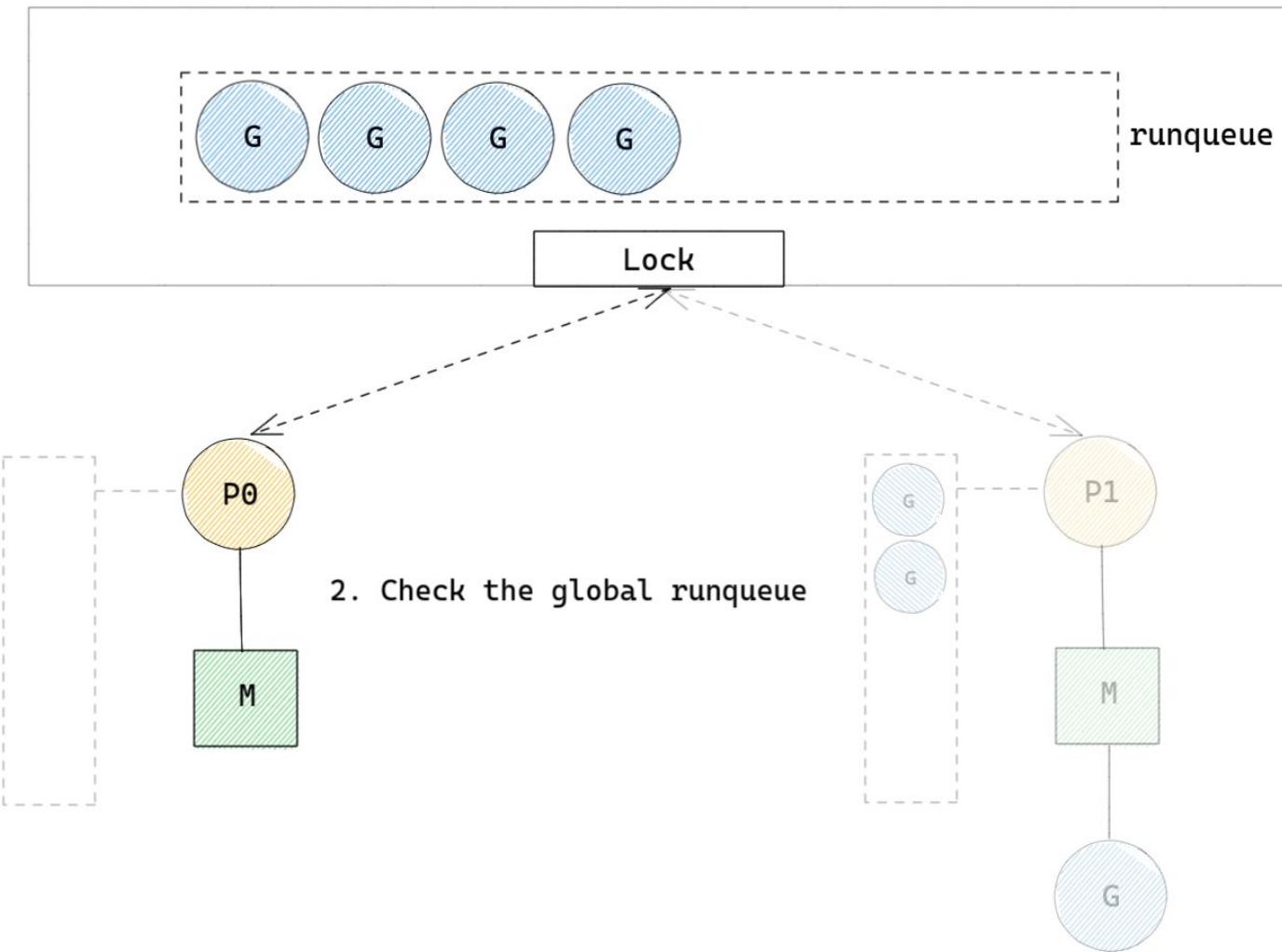


What if the situation is something like this?

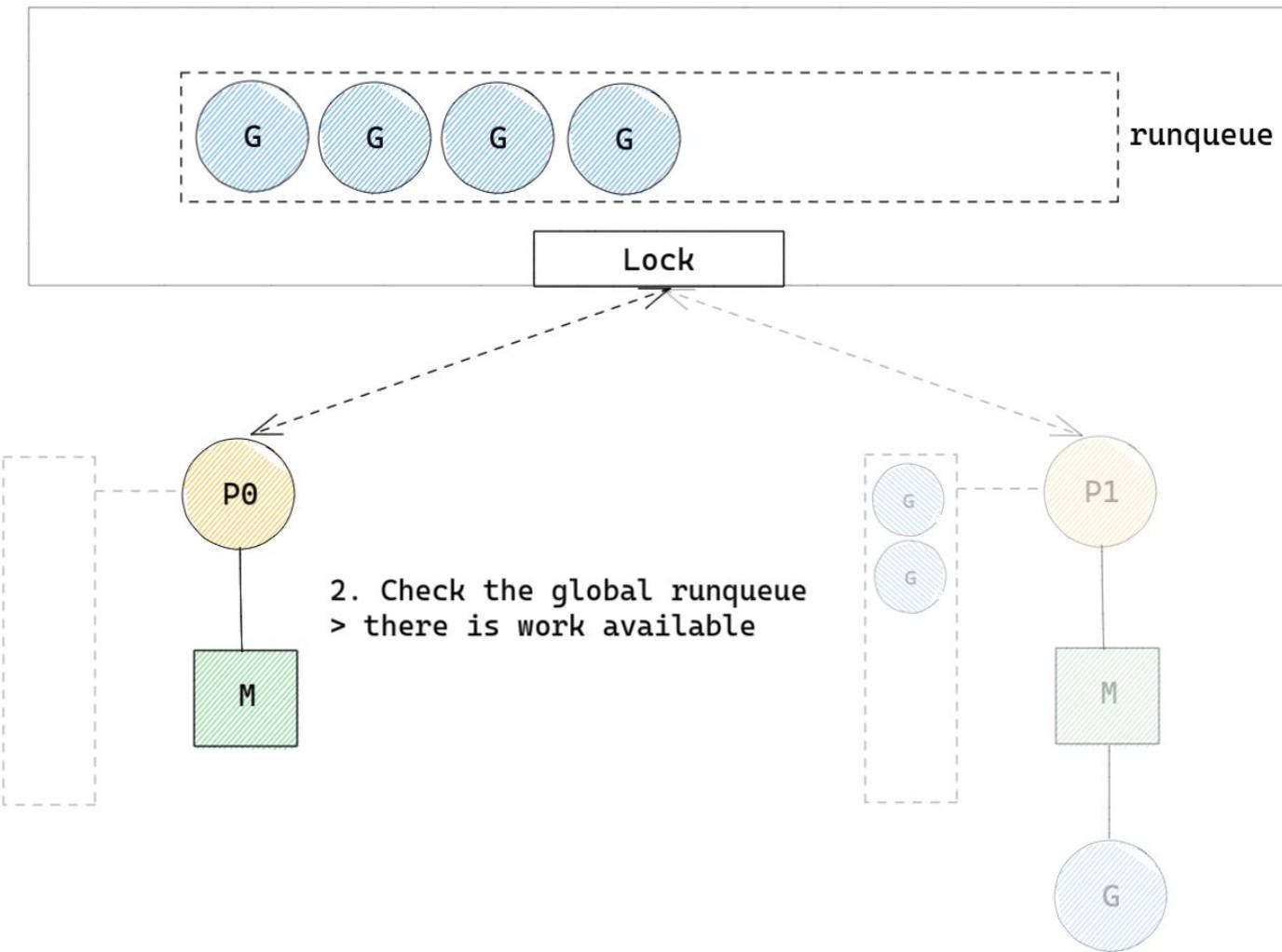
Go scheduler



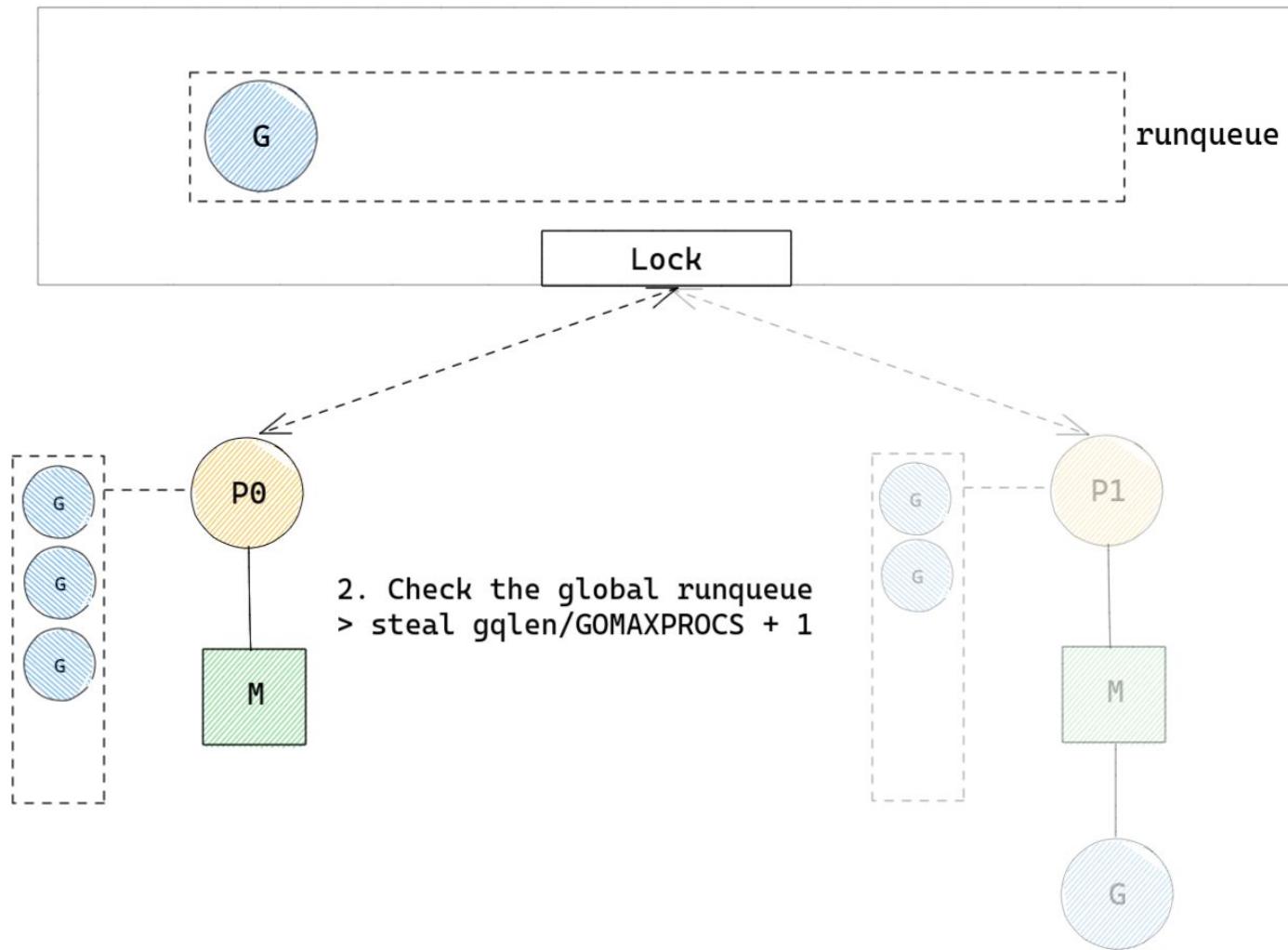
Go scheduler



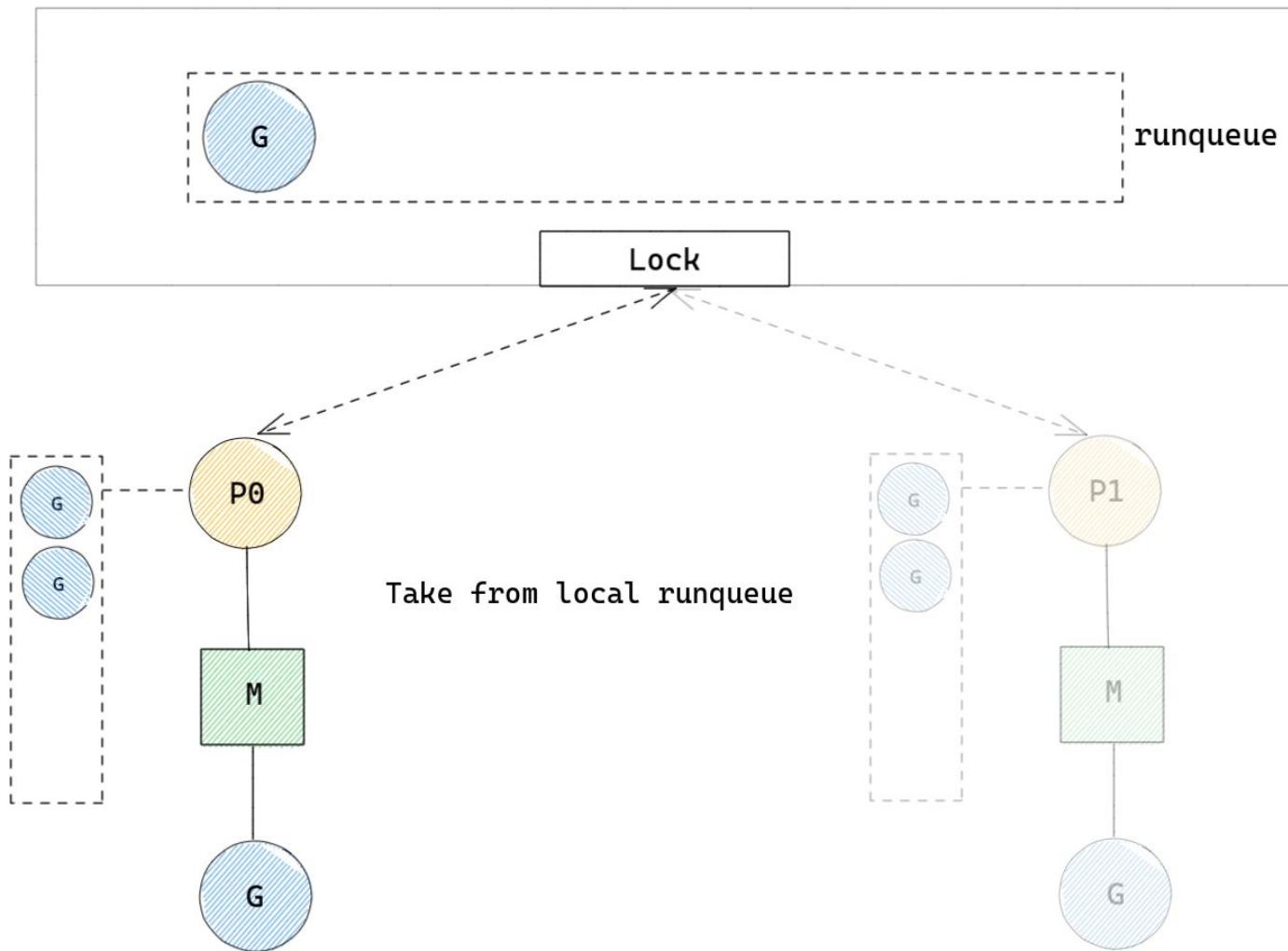
Go scheduler



Go scheduler

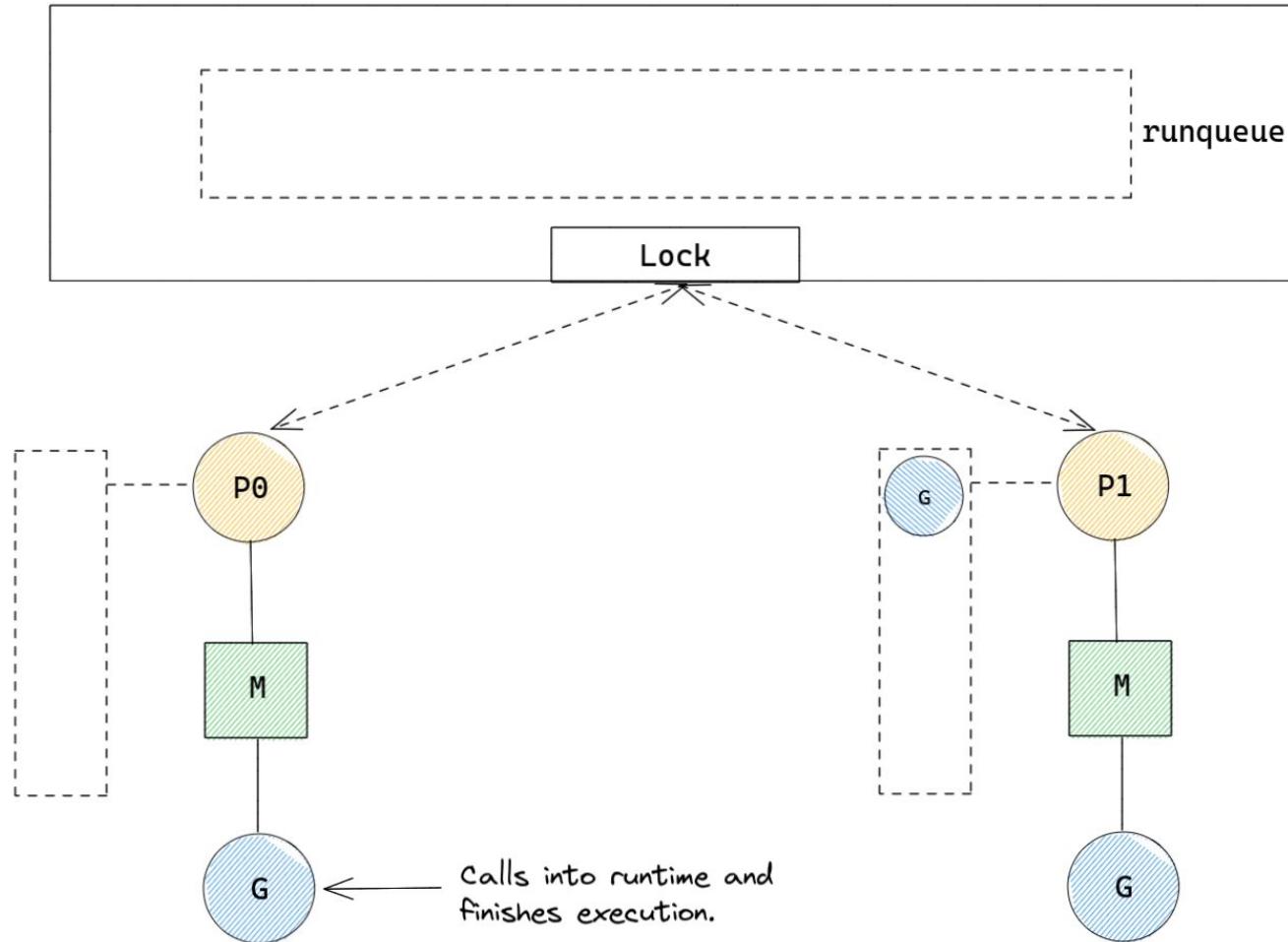


Go scheduler

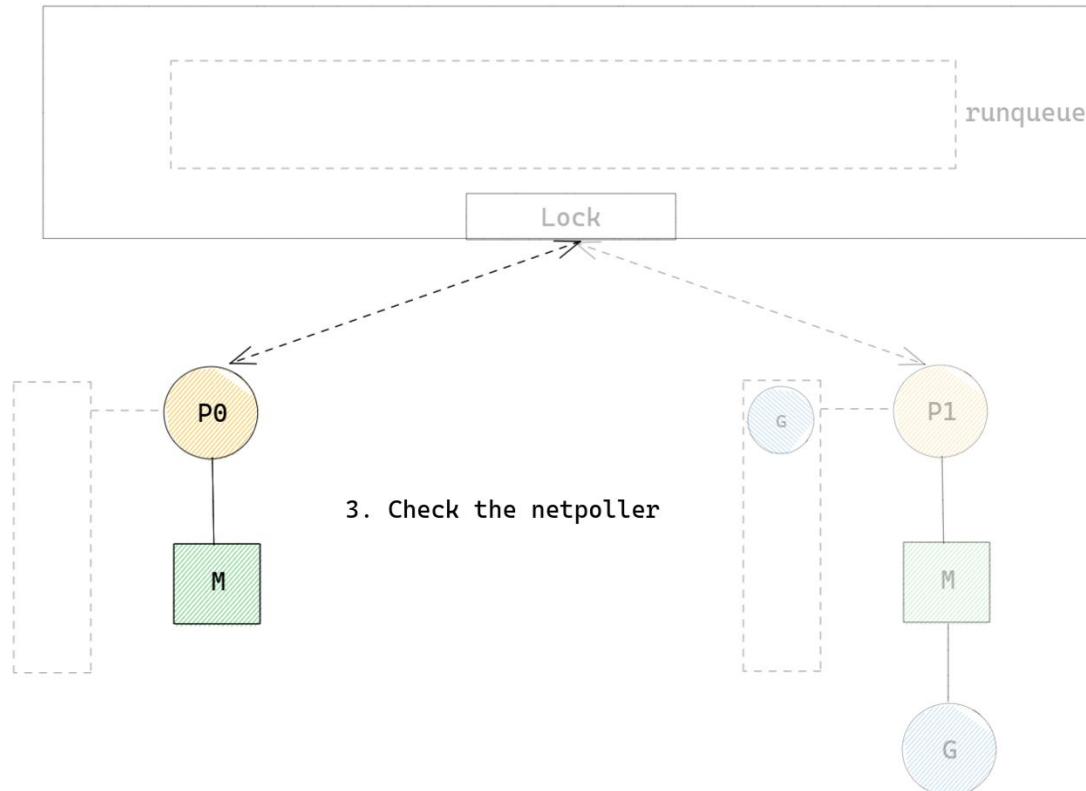


Global runqueue empty too?

Go scheduler



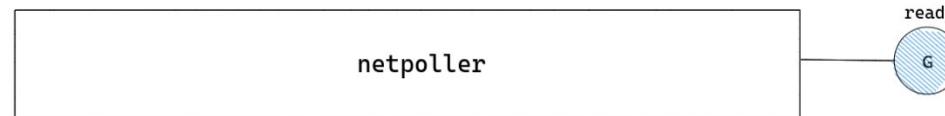
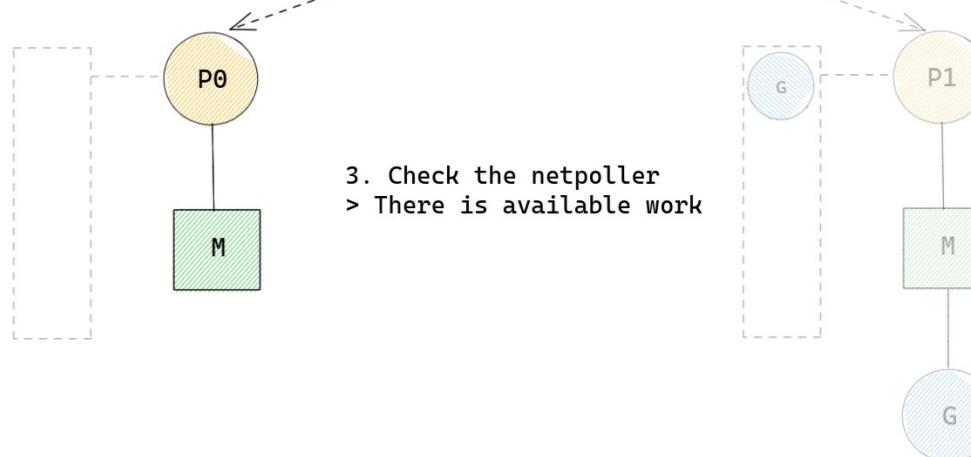
Go scheduler



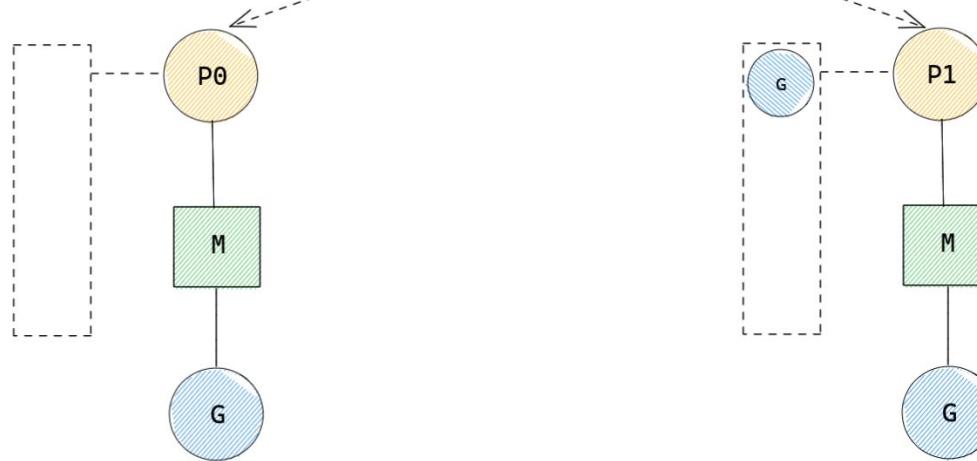
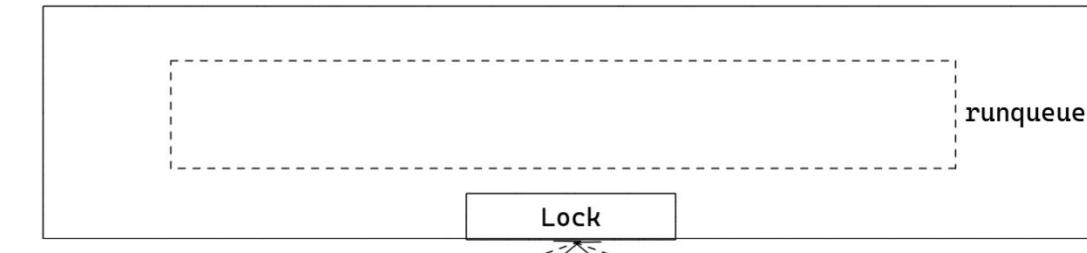
netpoller

ready

Go scheduler



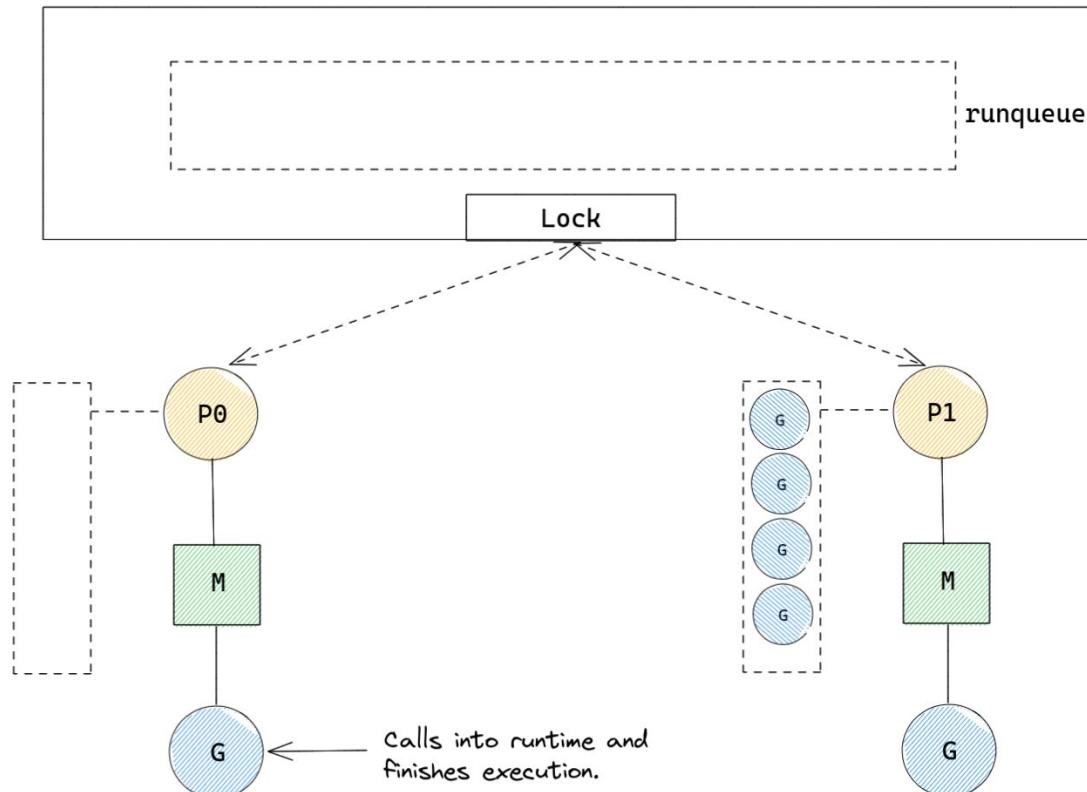
Go scheduler



netpoller

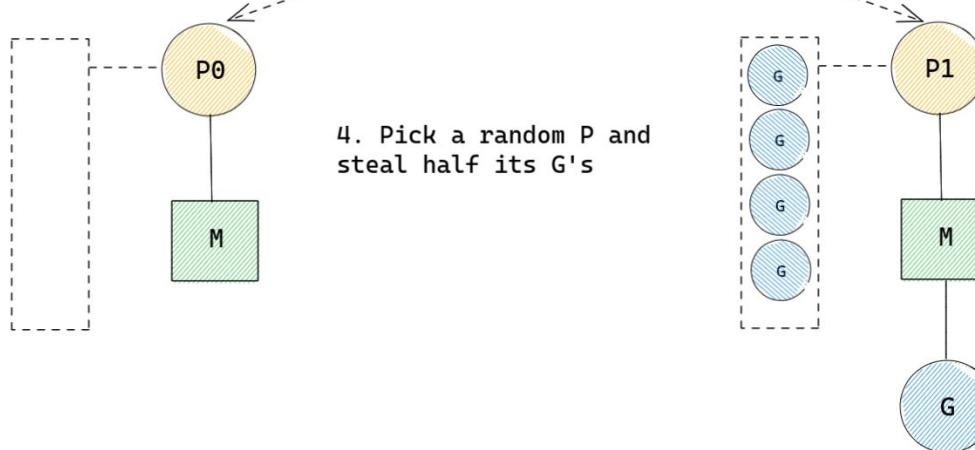
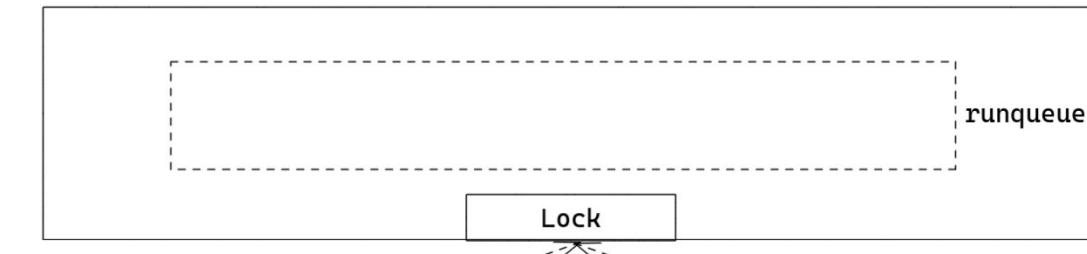
What if `netpoller` doesn't have any ready Goroutines and let's assume the situation looked something like this.

Go scheduler



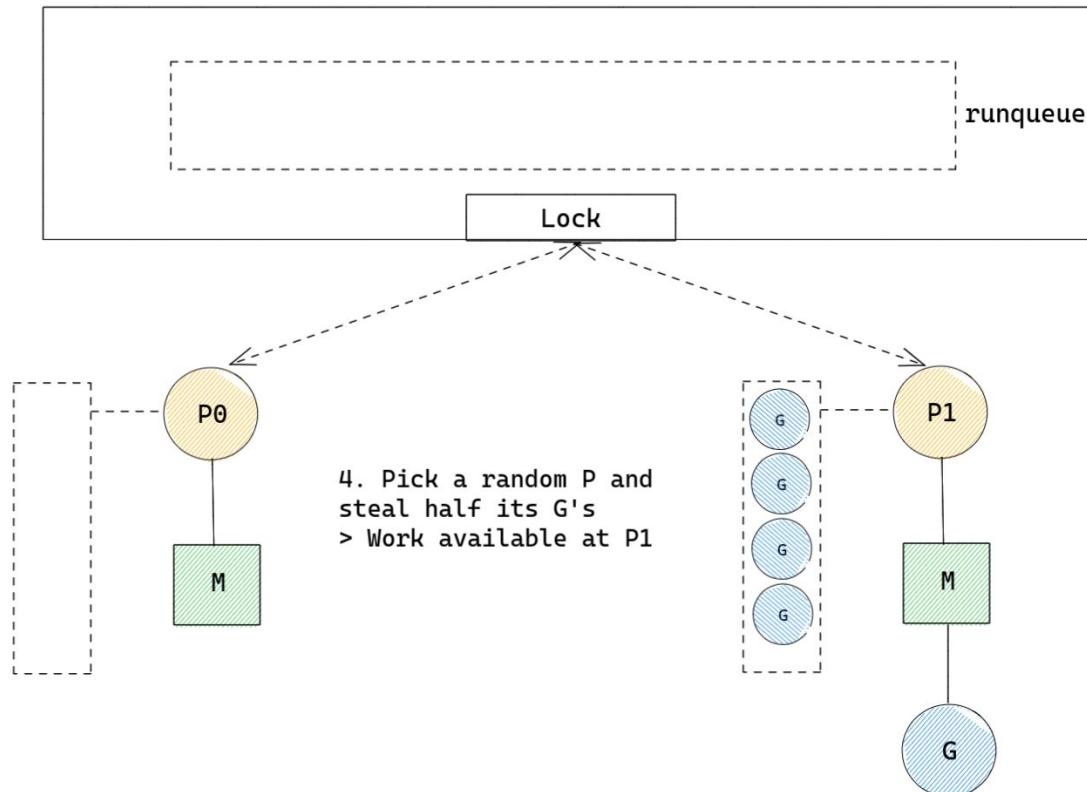
netpoller

Go scheduler



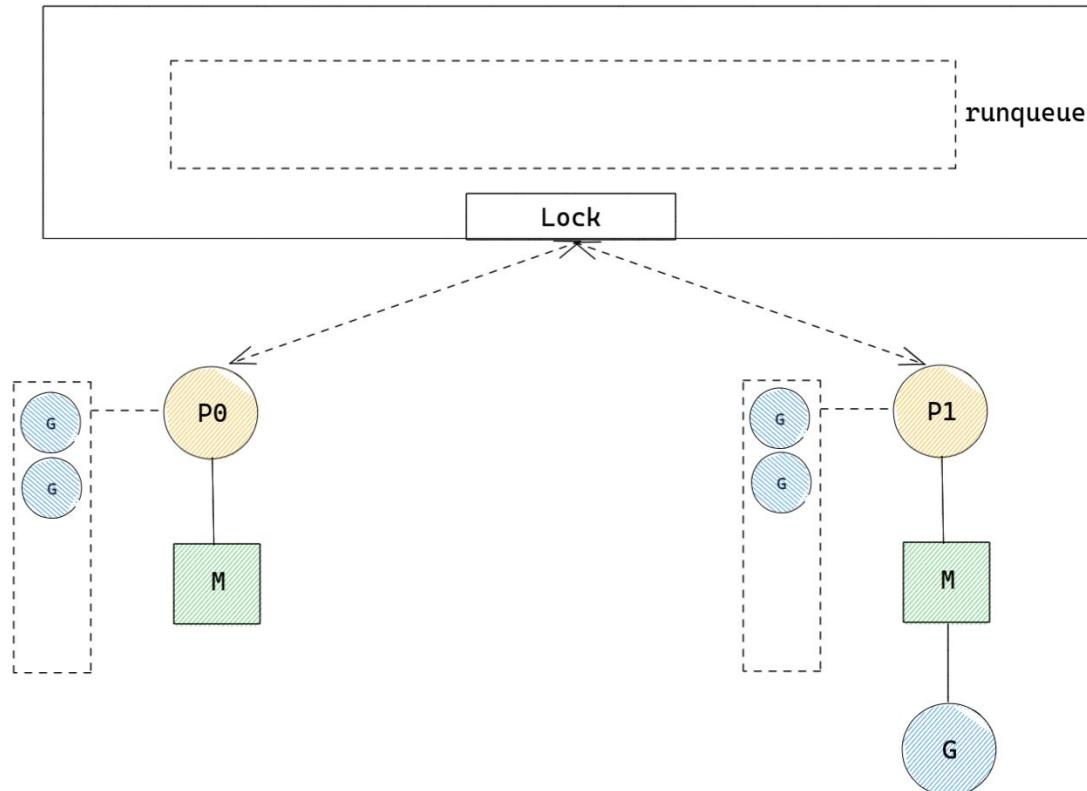
netpoller

Go scheduler



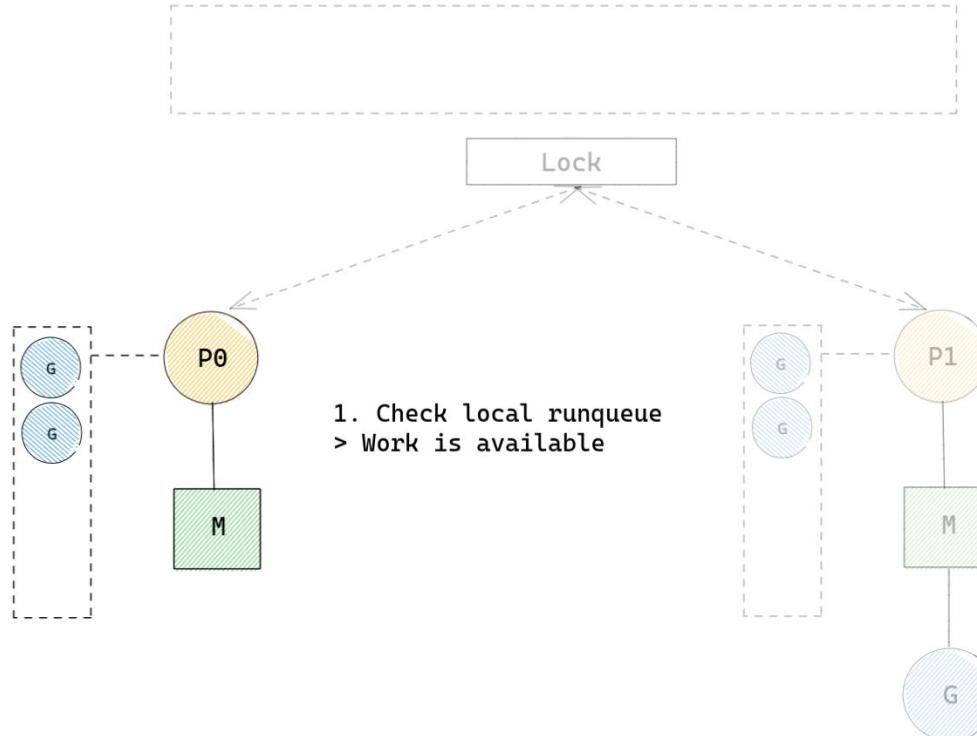
netpoller

Go scheduler



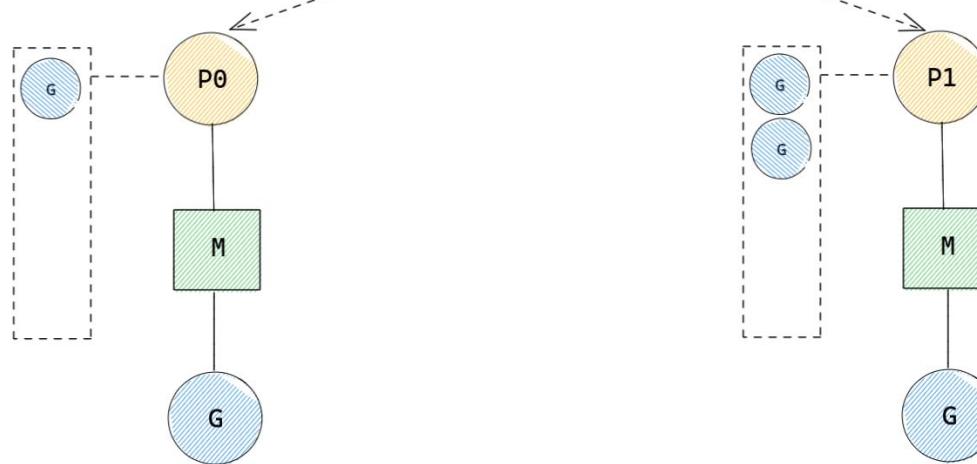
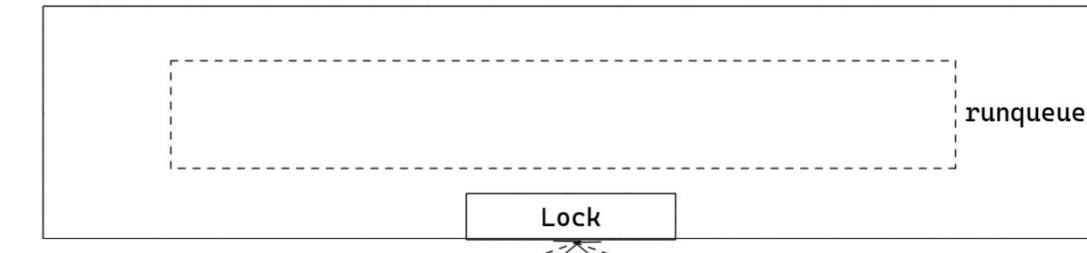
netpoller

Go scheduler



netpoller

Go scheduler



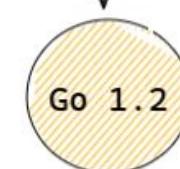
netpoller

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

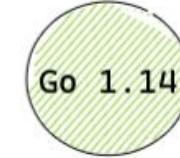
We spoke about pre-emption earlier, let's see how Go did it then and now.



Purely co-operative pre-emption



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



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

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

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

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

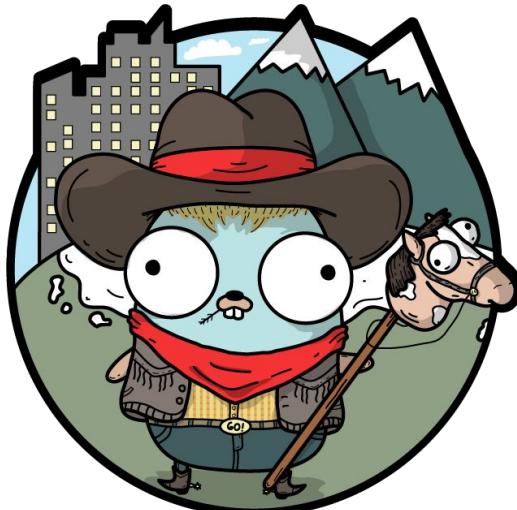
- Who sends this signal?

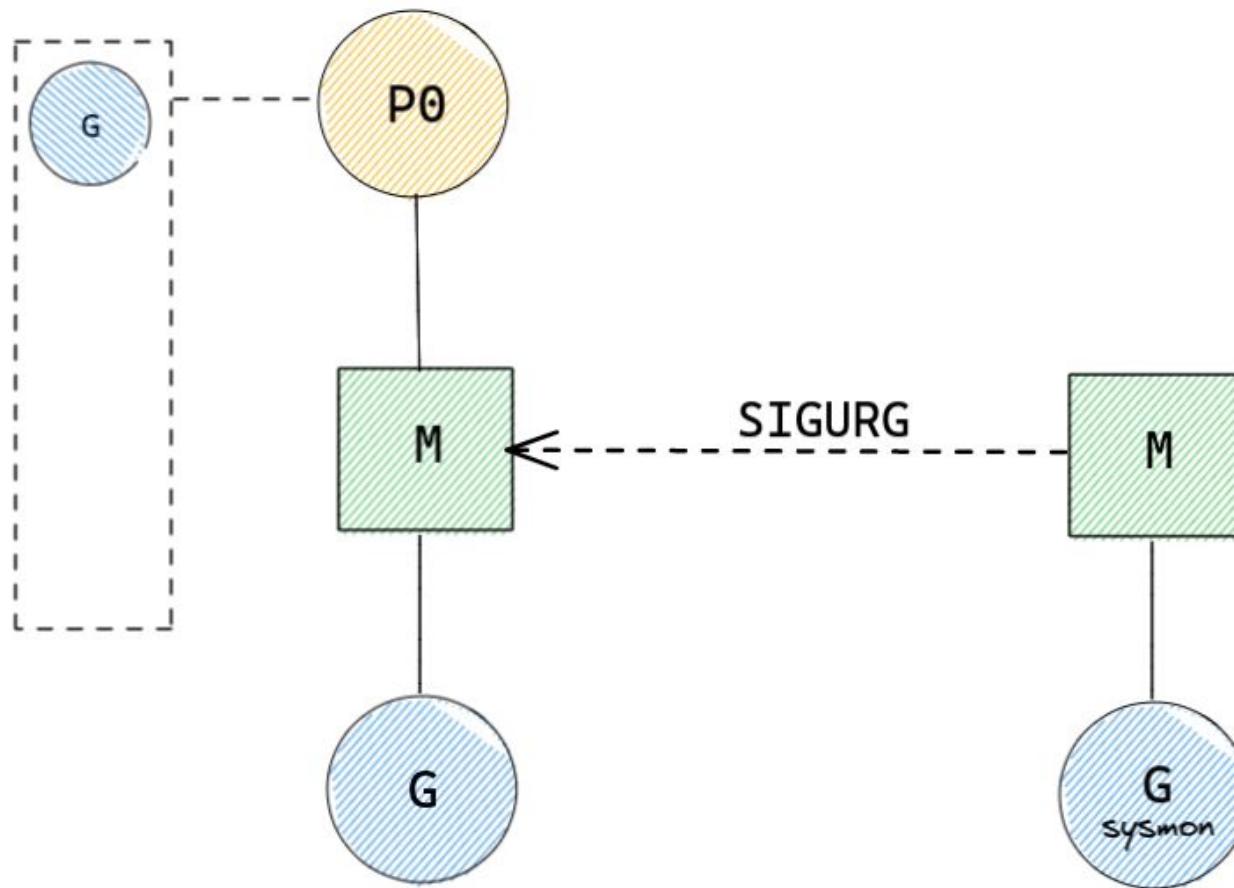
sysmon



Non Co-operative Pre-emption

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

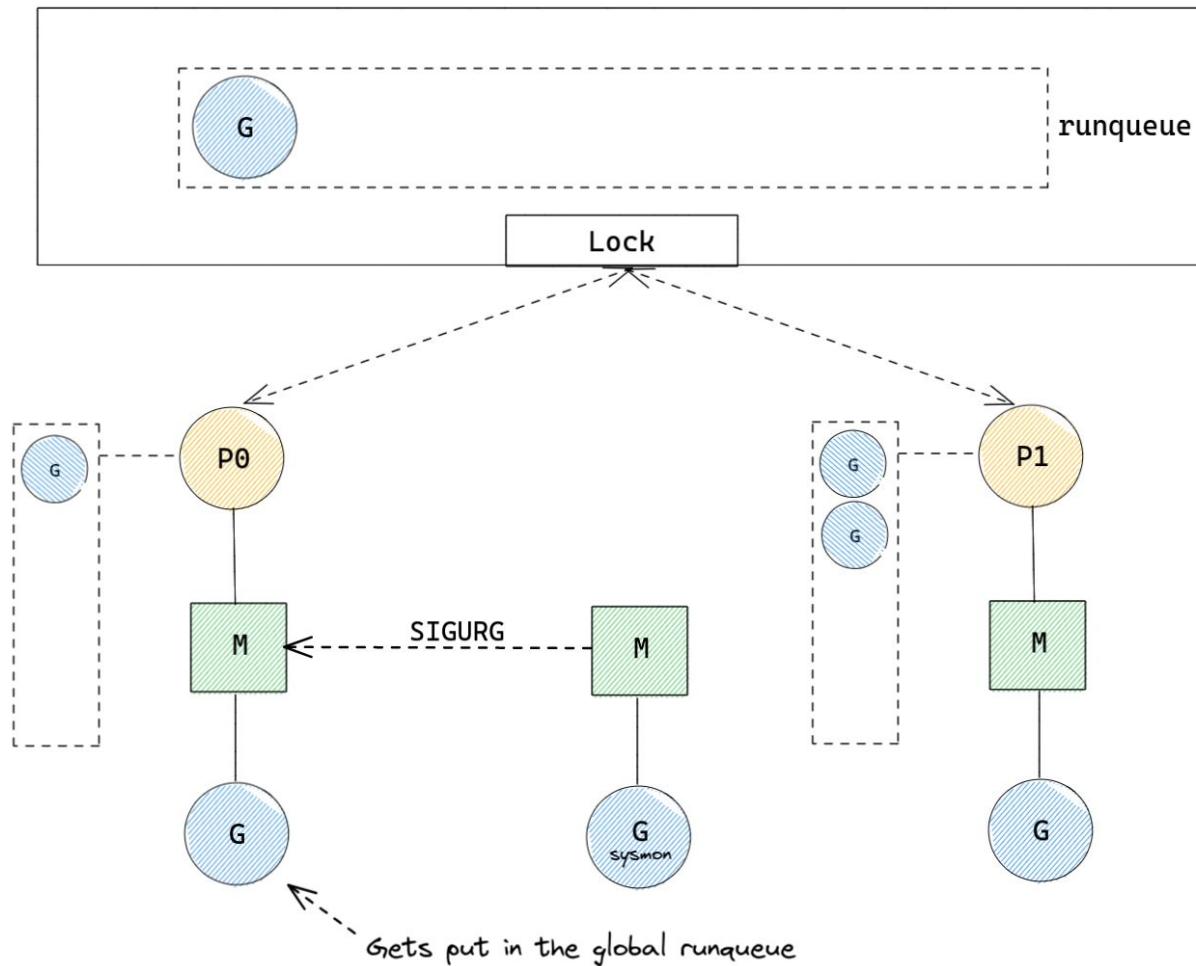




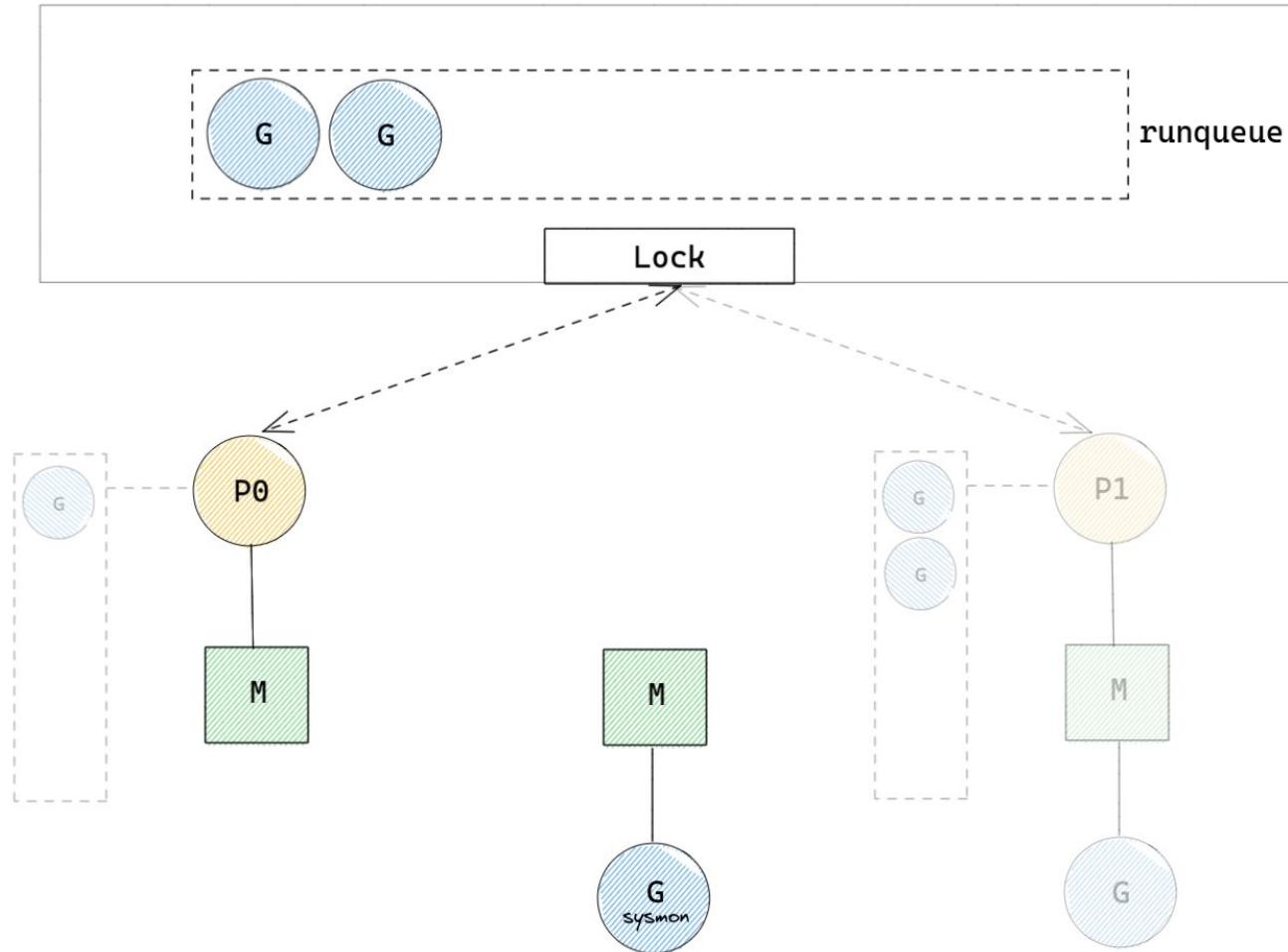
Been running for 10ms

Where does the pre-empted Goroutine go?

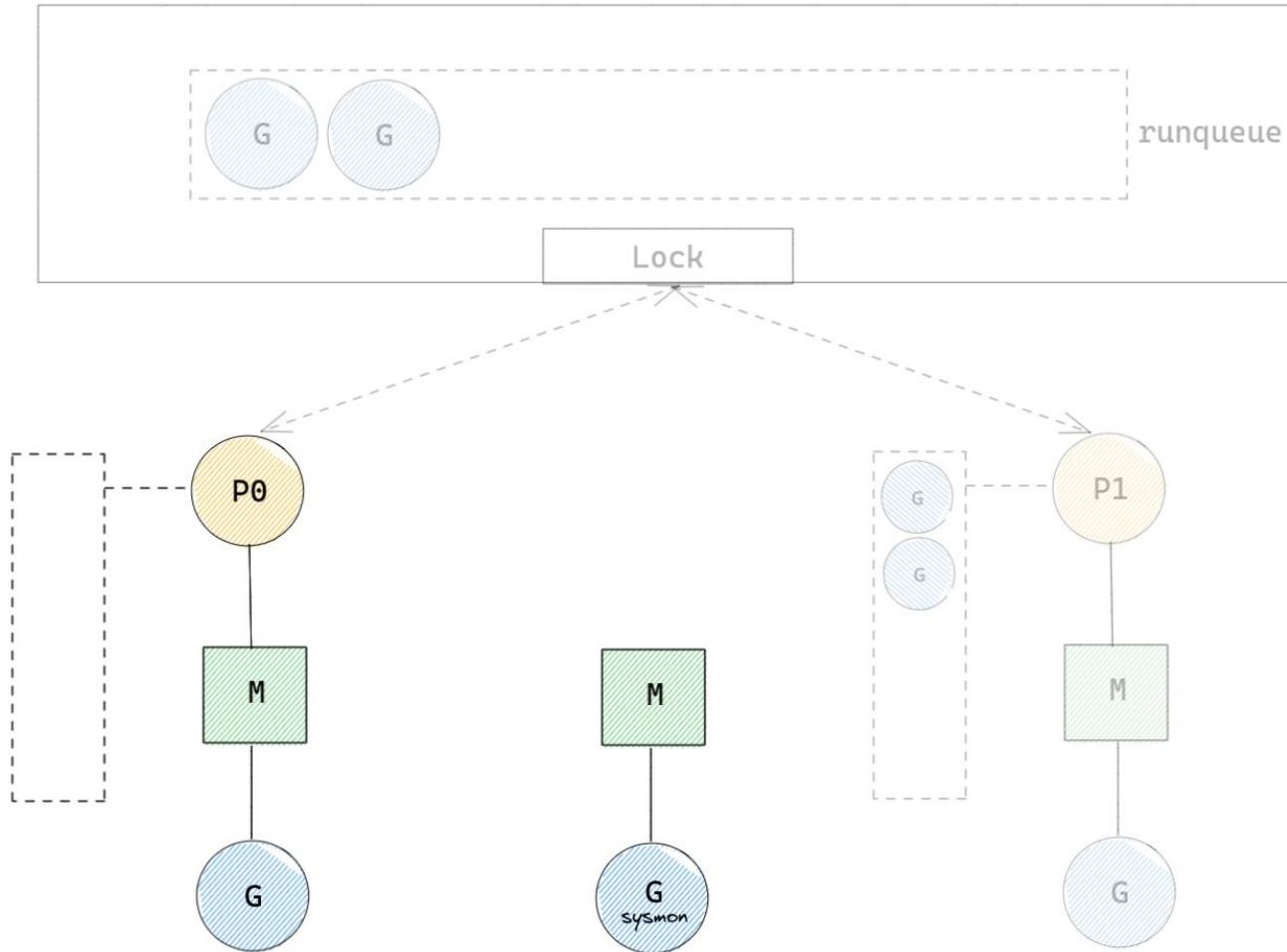
Go scheduler



Go scheduler



Go scheduler



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

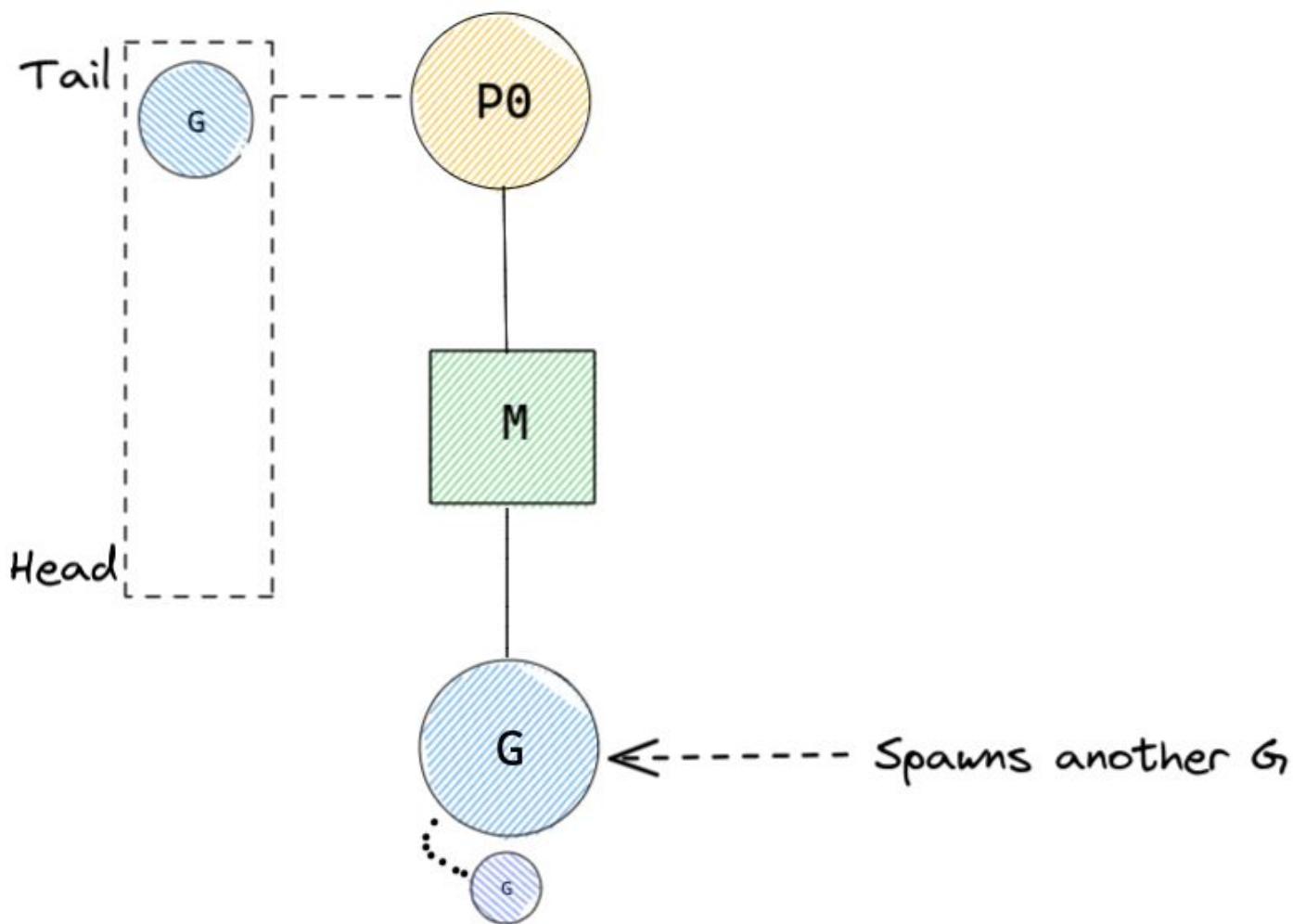
Awesome! Now that we have a way of handling resource hogs, let's revisit our code snippet.

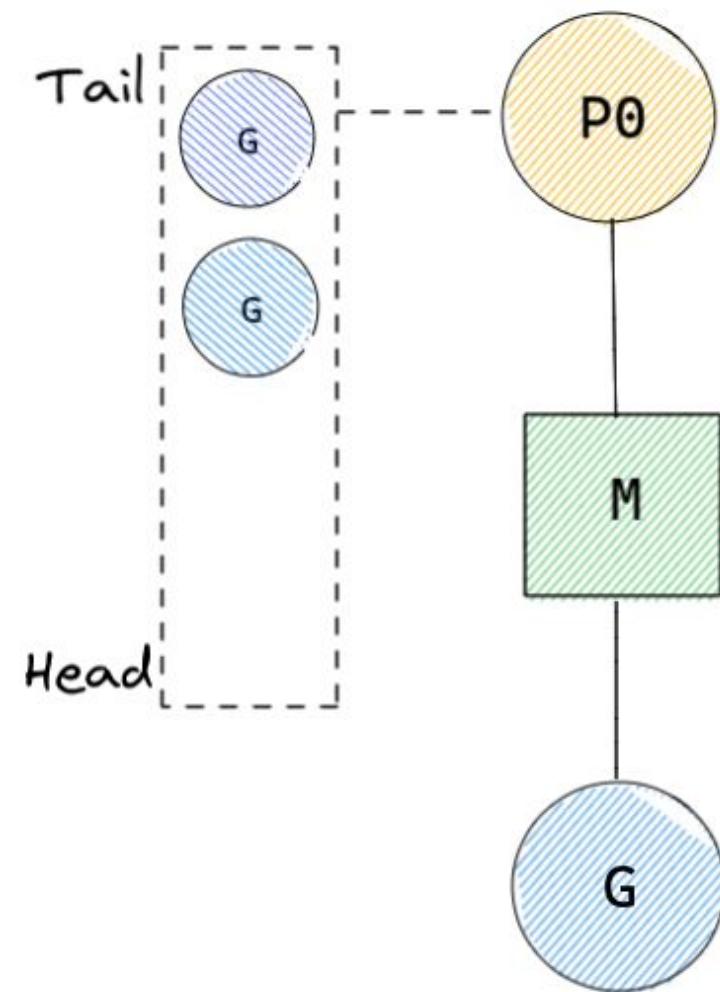
```
1  func main() {
2      var wg sync.WaitGroup
3      wg.Add(10)
4      for i := 0; i < 10; i++ {
5          go func() {
6              defer wg.Done()
7              // each matrix of size 1024
8              multiplyLargeMatrices()
9          }()
10     }
11     wg.Wait()
12 }
```

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

- Spawns 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 }
```





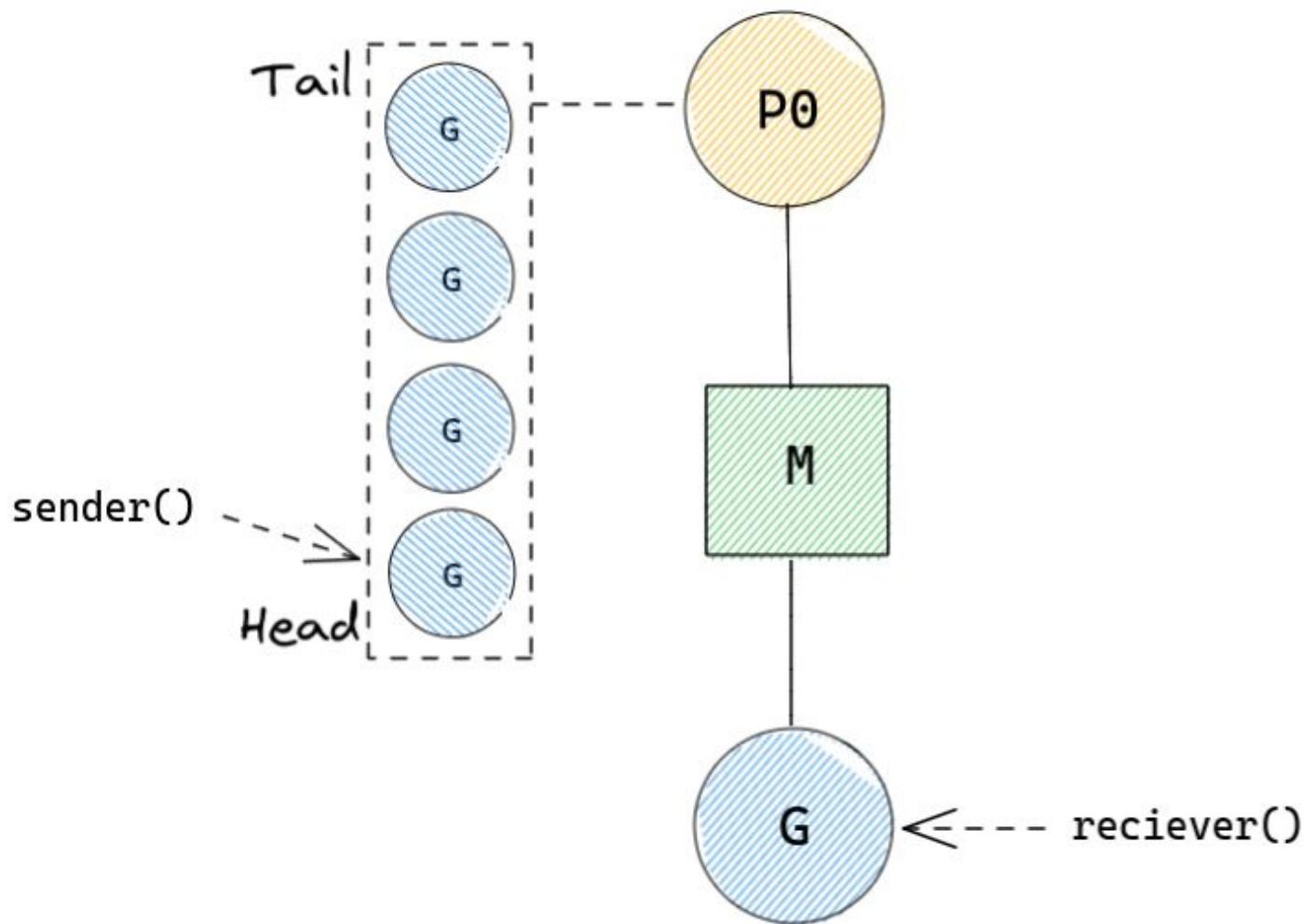
That looks good, but can we maybe do better?

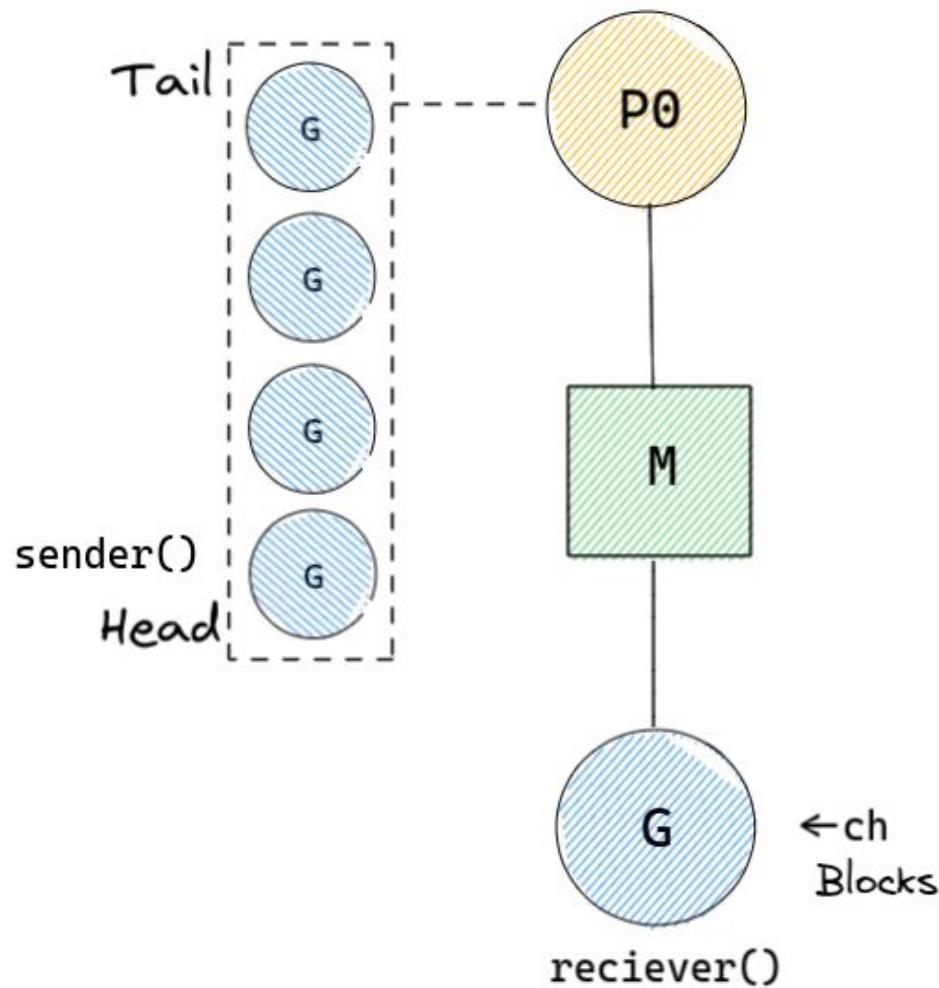
- FIFO is good for fairness, but not good for locality.
 - LIFO on the other hand is good for locality but bad for fairness.
- Maybe we can look at Go specific practices and see if we can optimize for commonly used patterns?
 - Channels are very often used in conjunction with Goroutines, be it for synchronization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

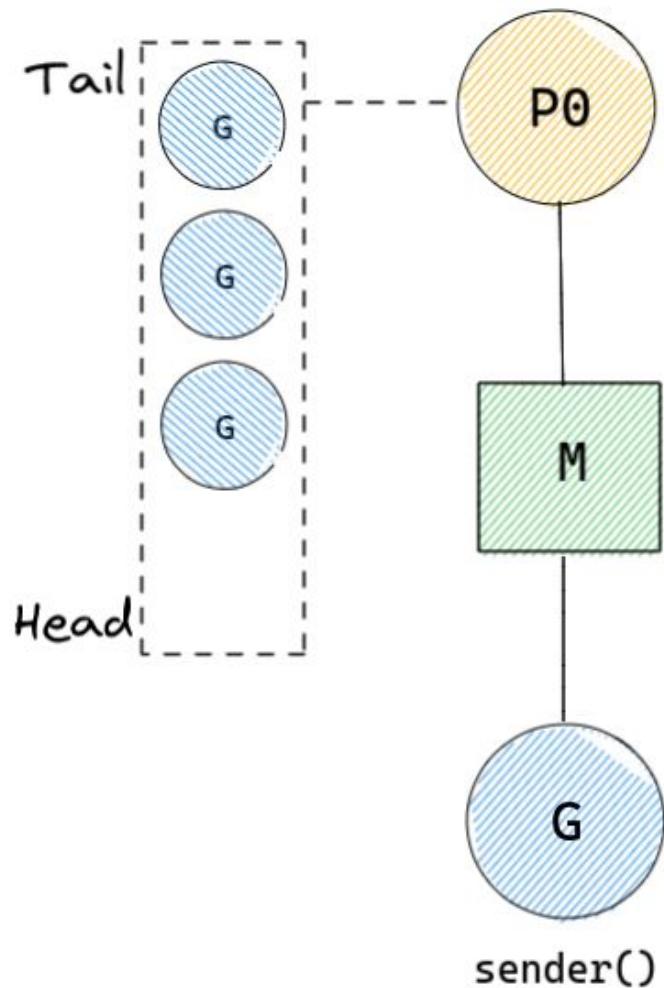


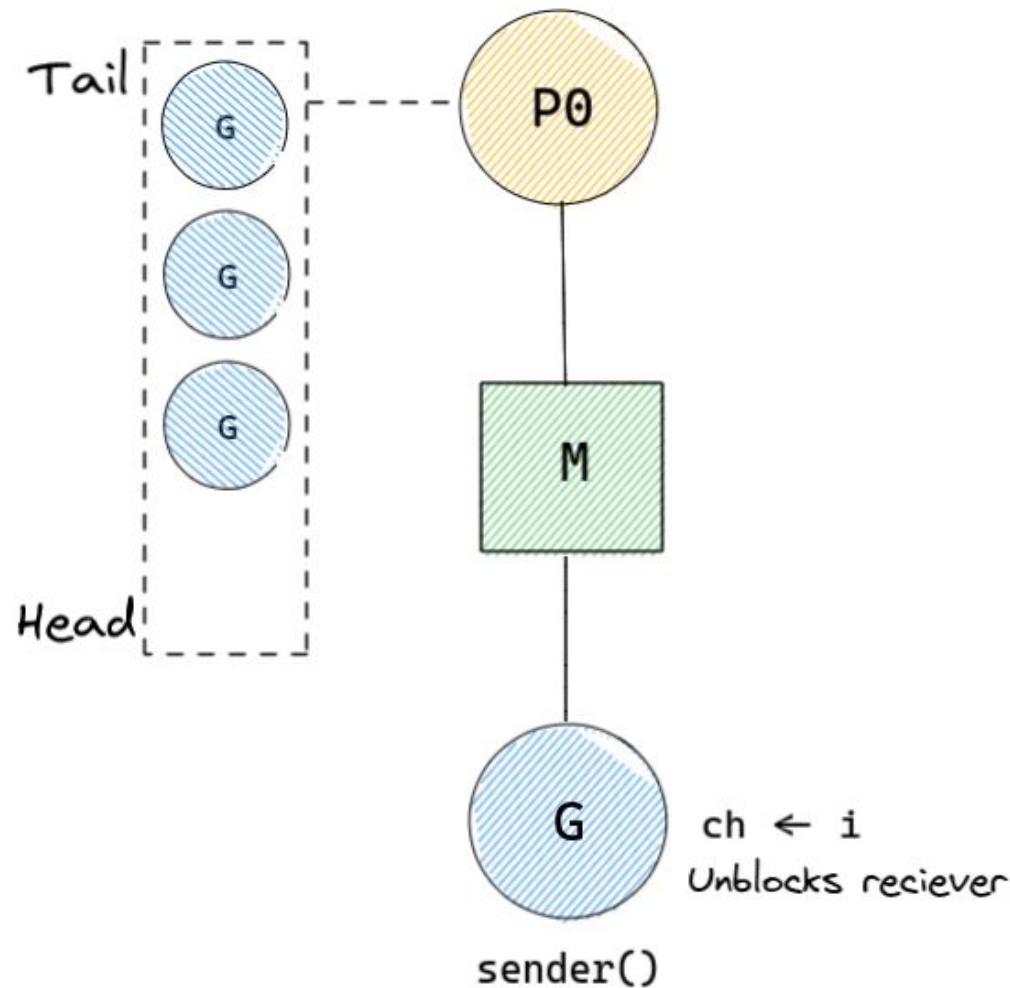
```
1  func sender(ch chan int) {
2      for i := 0; i < 10; i++ {
3          ch <- i
4      }
5      close(ch)
6  }
7
8  func receiver(ch chan int) {
9      for {
10          msg, ok := <-ch
11          if !ok {
12              break
13          }
14          process(msg)
15      }
16  }
17
18 func process(msg int) {
19     fmt.Println("Received:", msg)
20 }
21
22 func main() {
23     ch := make(chan int)
24     go sender(ch)
25     receiver(ch)
26 }
```

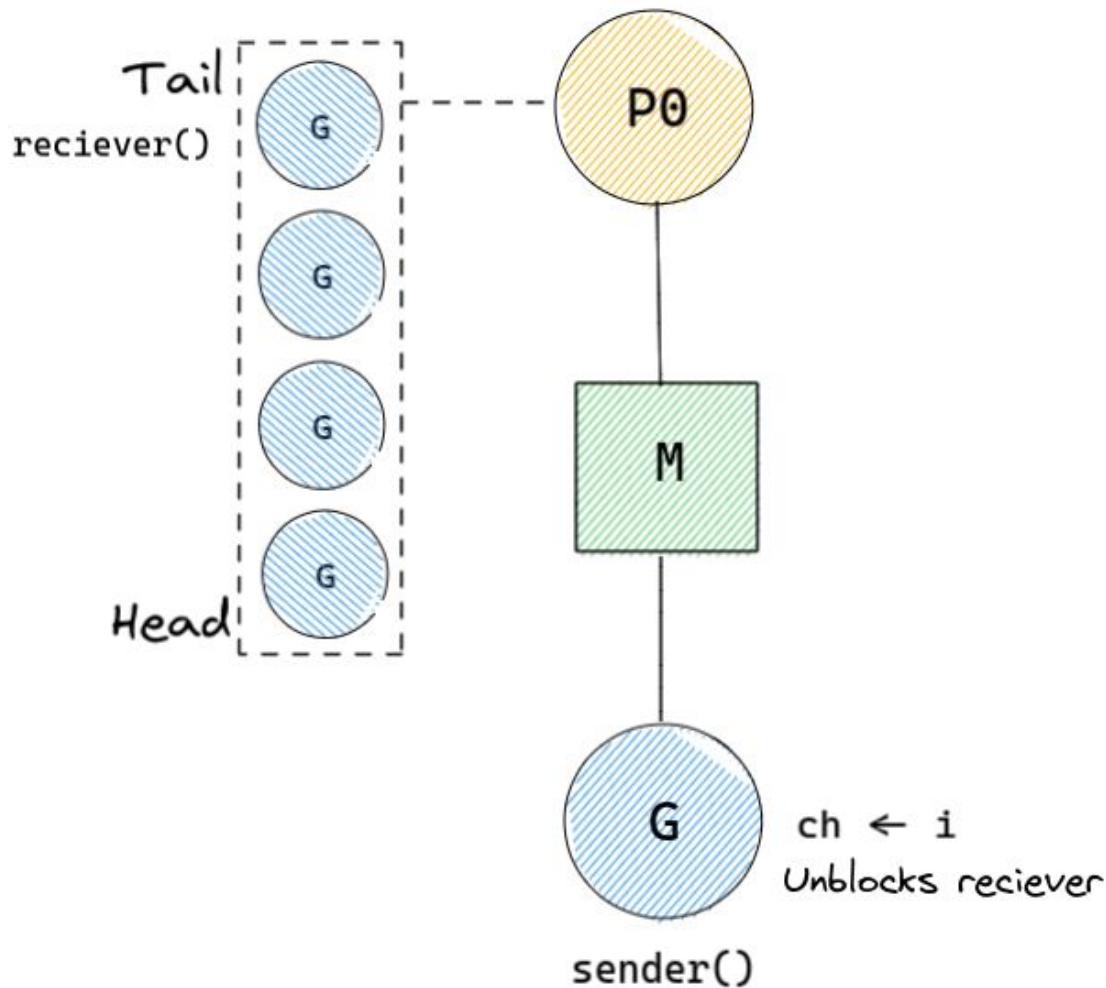
Let's try and illustrate what happens in the scheduler.









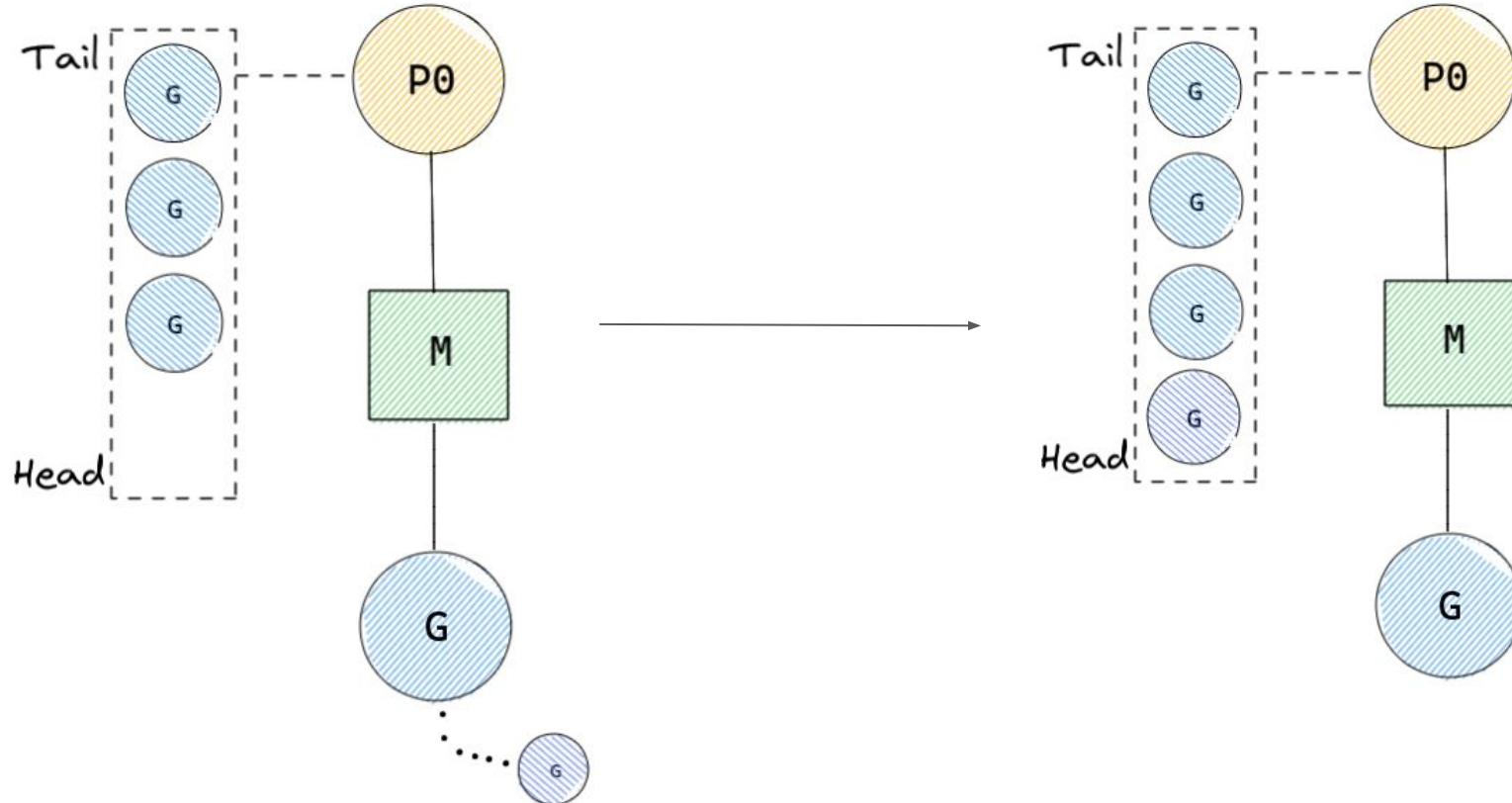


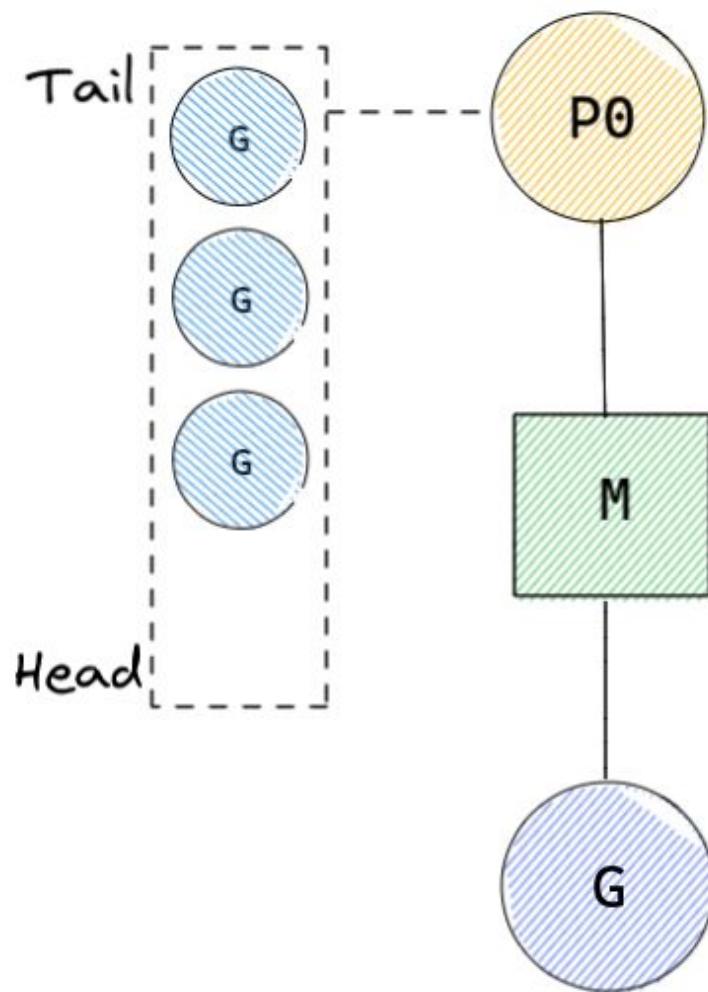
Impact On Performance

- 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 $\sim 10\text{ms}$
 - Length of each local runqueue is fixed at 256.
 - Worst case - all are long-running, leading to one of them being blocked for $\sim 255 * 10\text{ms}$
- 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 than the tail.



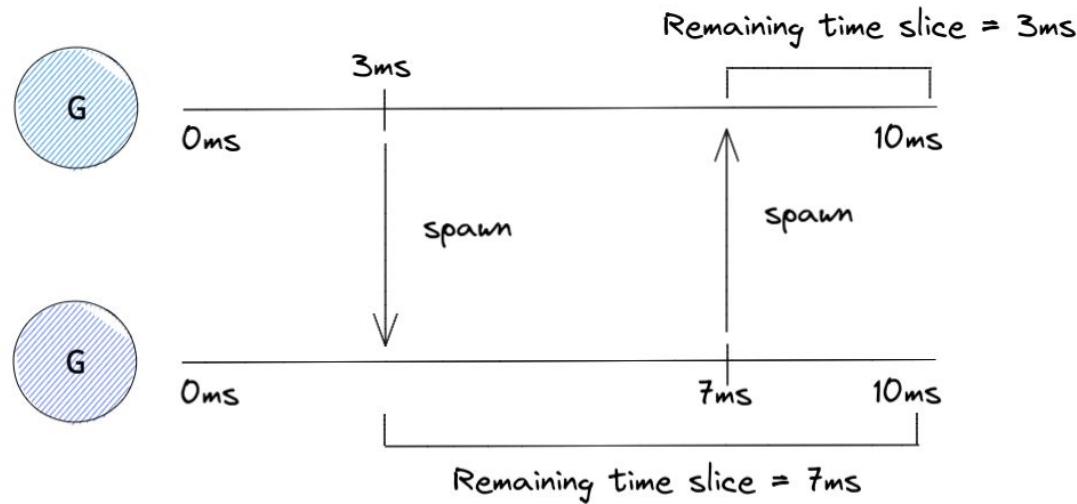


The issue now is that the 2 can constantly re-spawn each other and starve the remaining routines in the queue.

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 `runqueues` 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 `runqueues` 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?

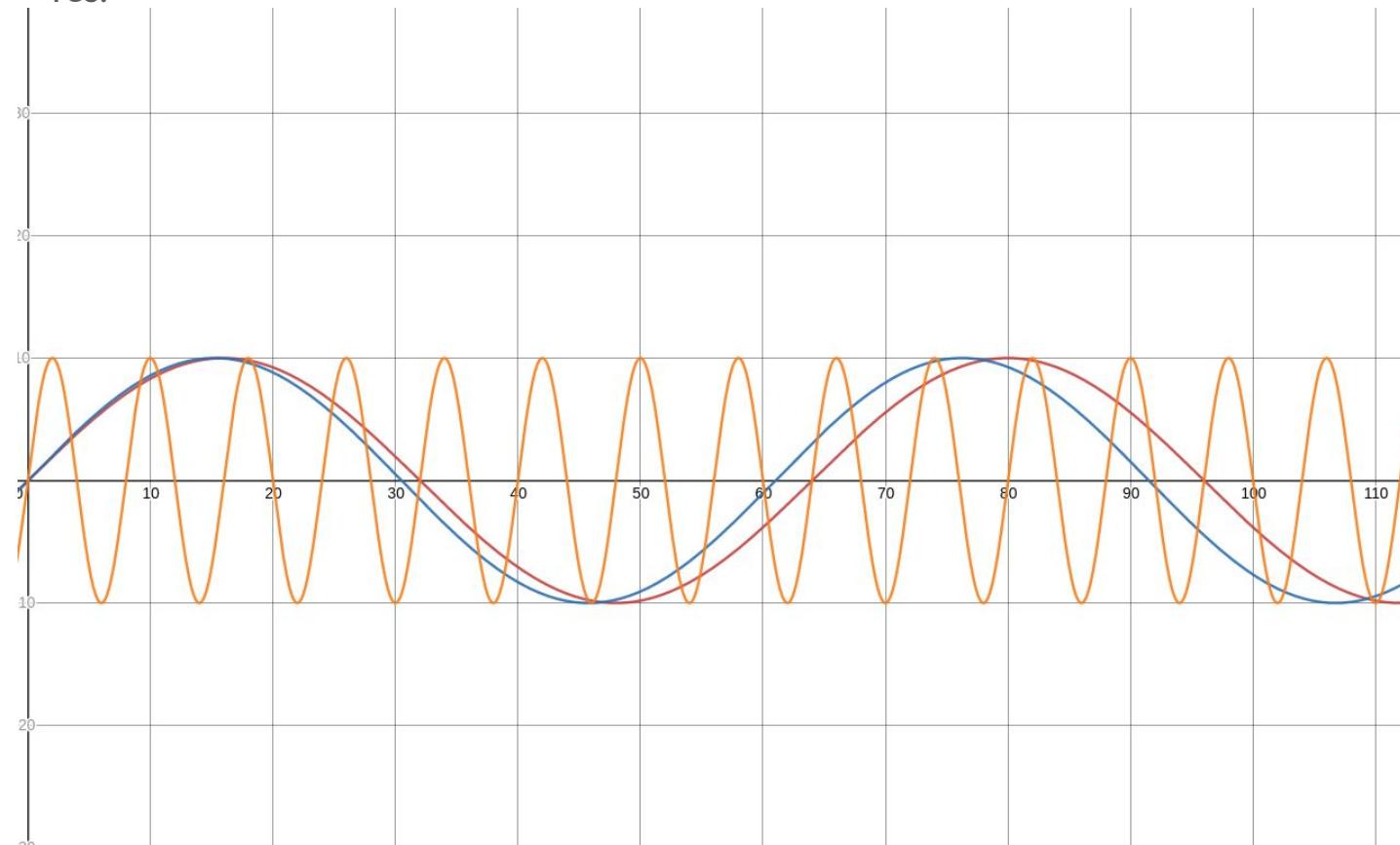
```
if schedtick % 61 == 0 {  
    getFromGlobal()  
} else {  
    doThingsAsBefore()  
}
```

- 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.

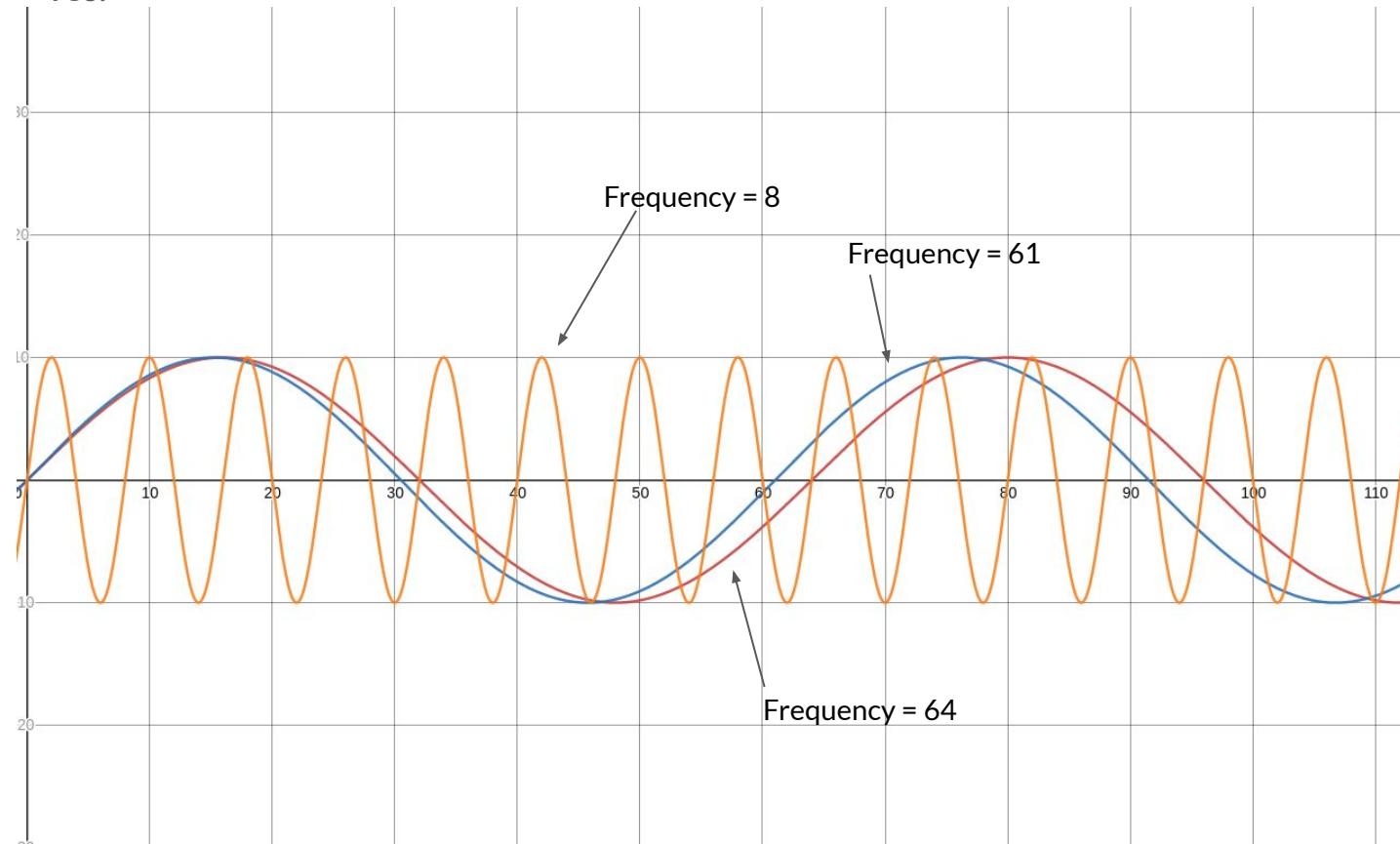
- Is 61 chosen keeping fairness in mind?

○ Yes.



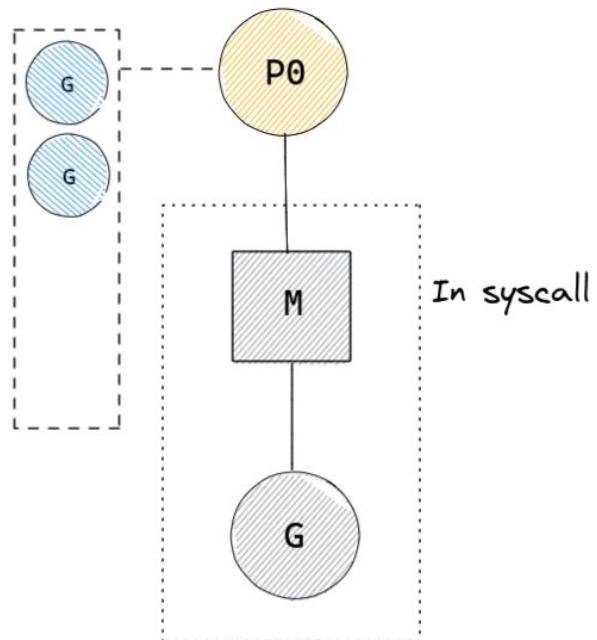
- Is 61 chosen keeping fairness in mind?

Yes.

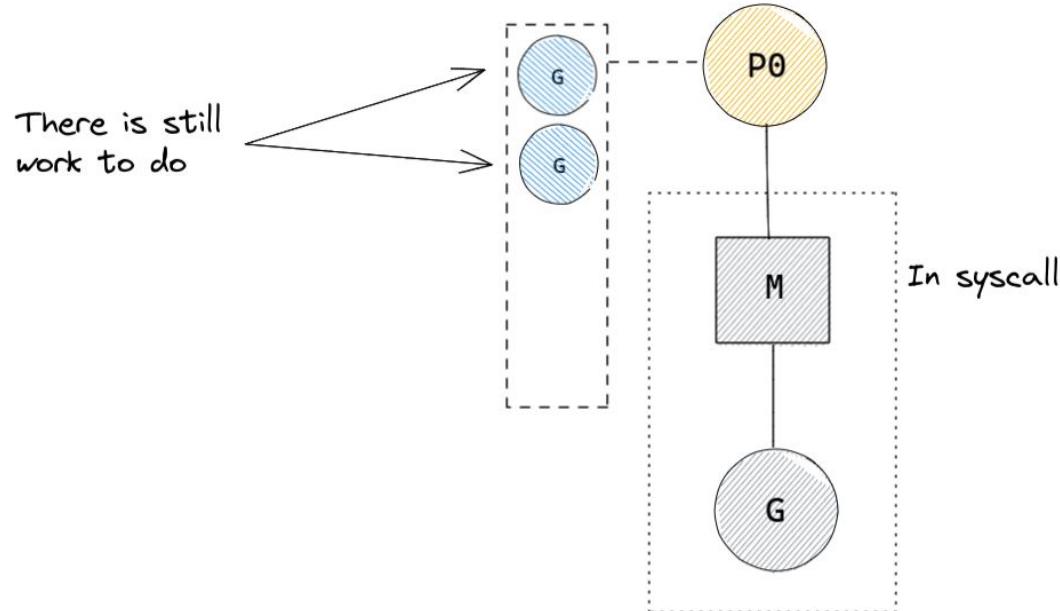


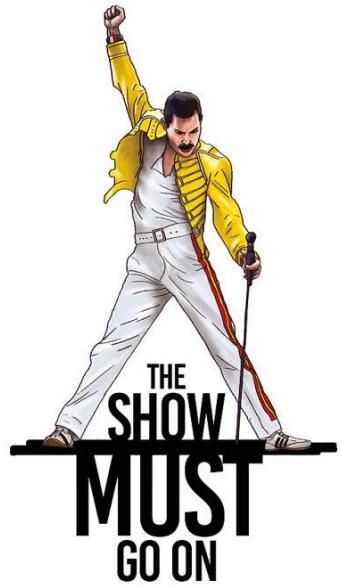
We've seen how Goroutines effectively end up running on threads, but what happens if the thread itself blocks in something like a `syscall`?

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

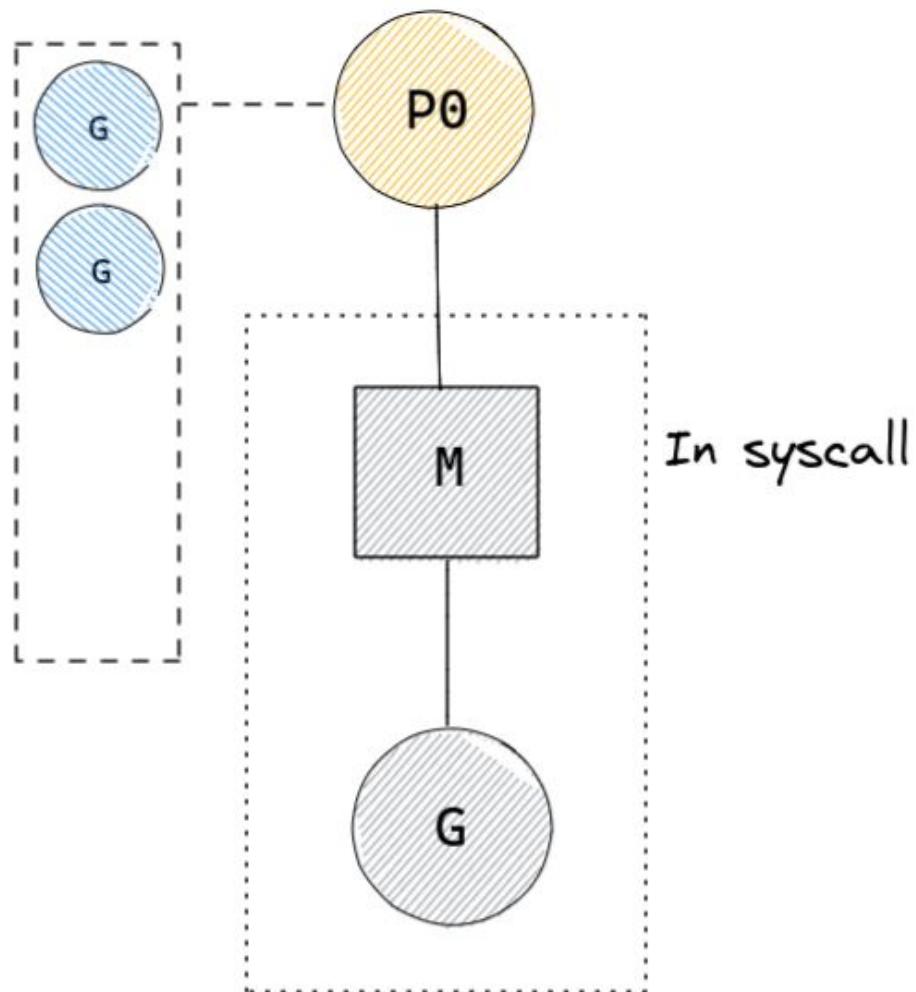


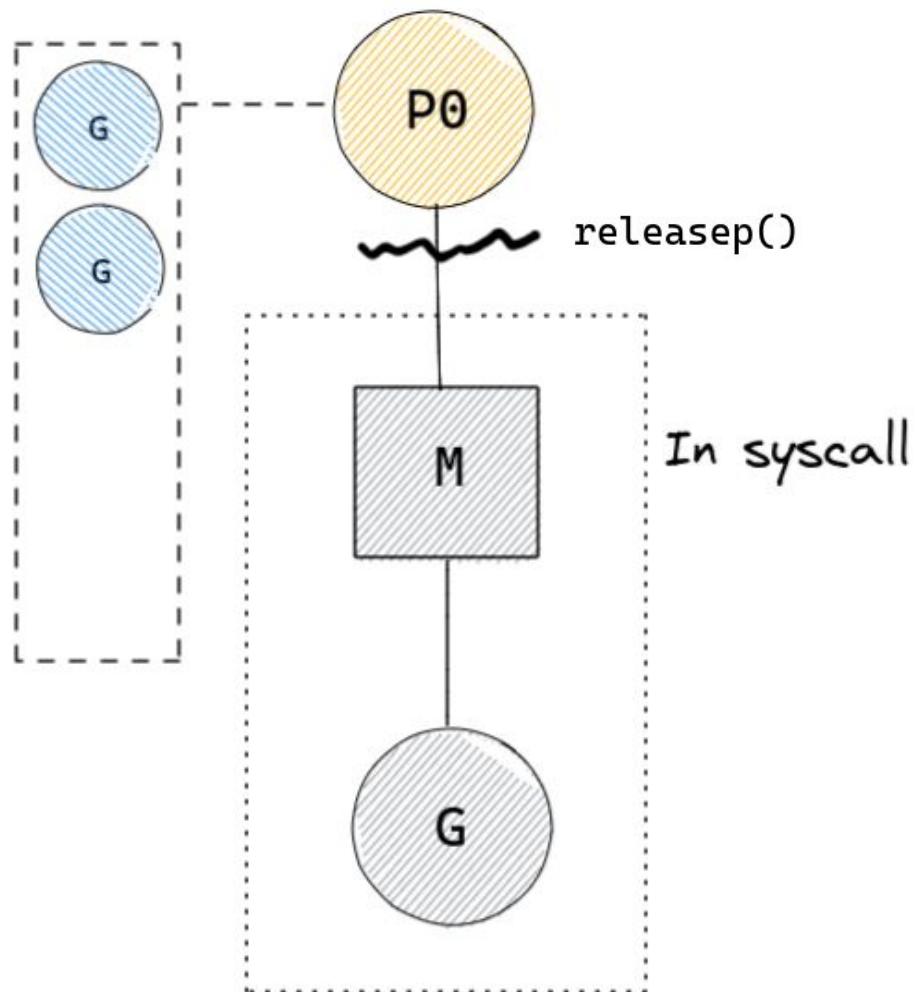
We've seen how Goroutines effectively end up running on threads, but what happens if the thread itself blocks in something like a syscall?

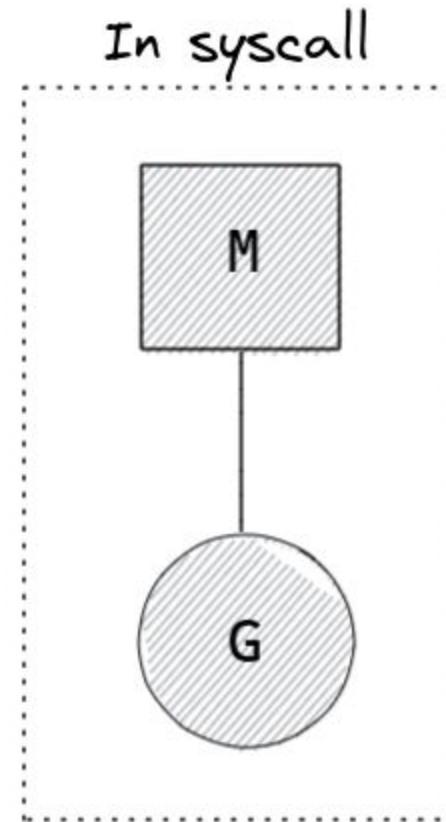
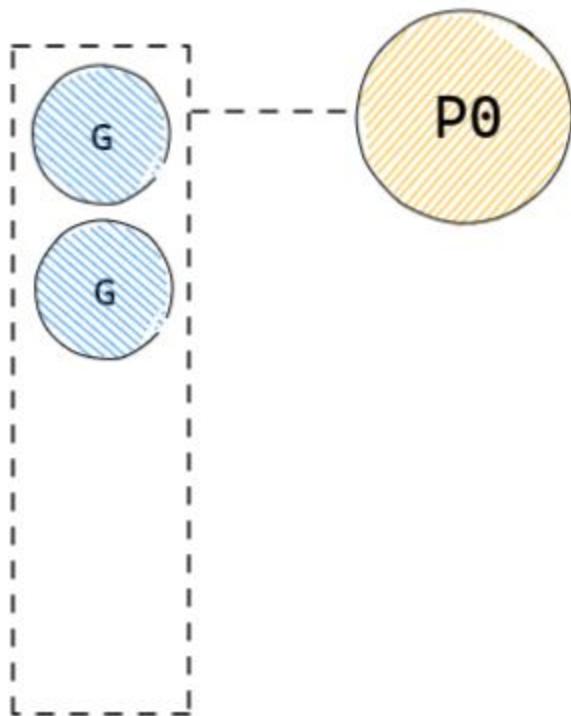


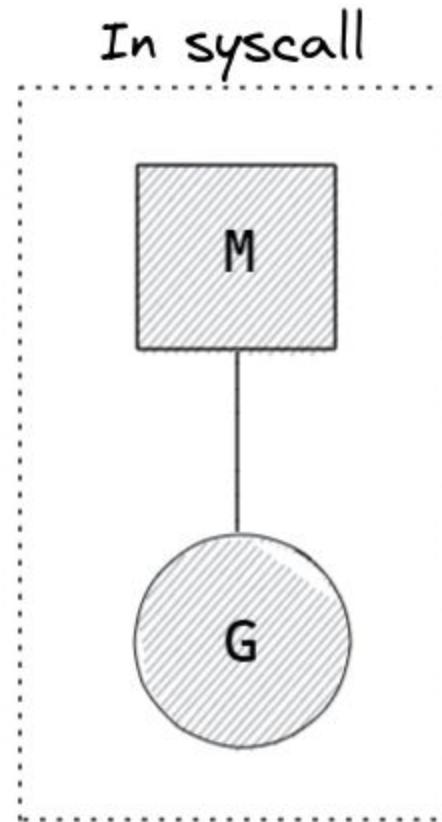
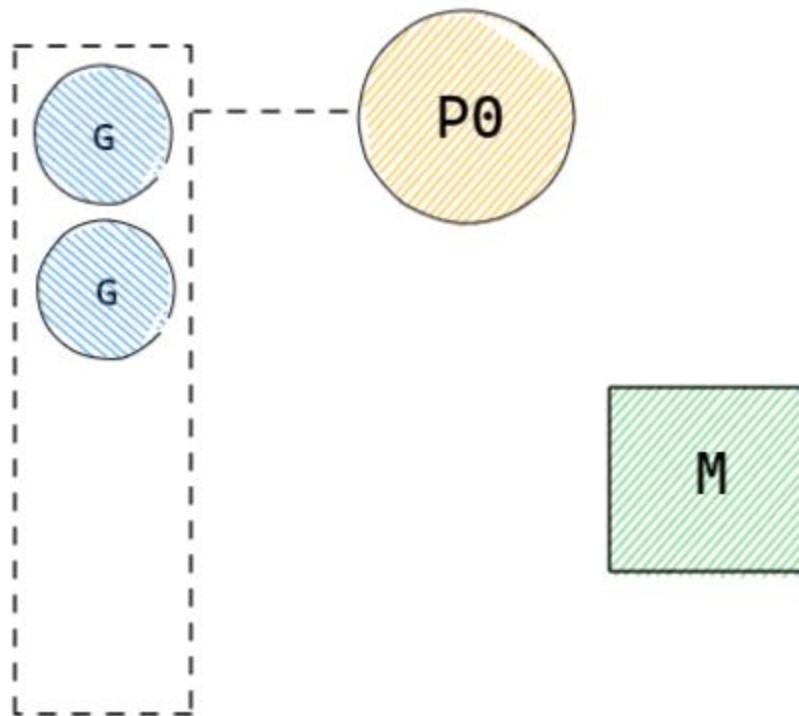


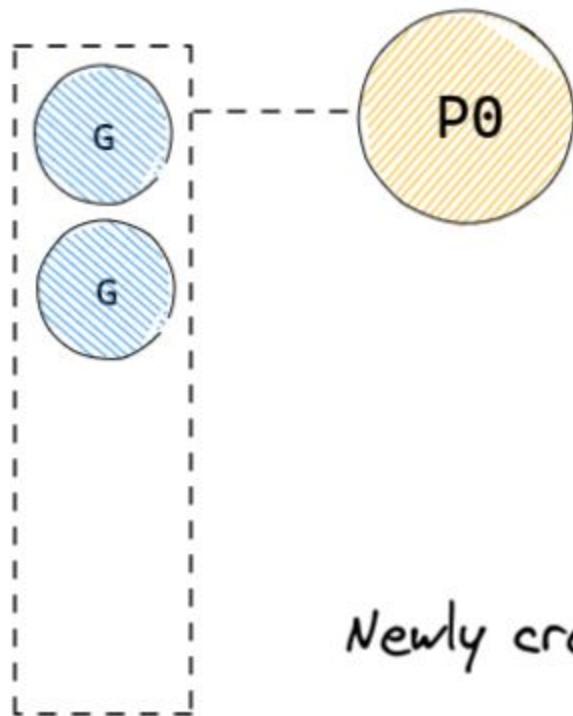
We perform something known as the handoff to deal with this.



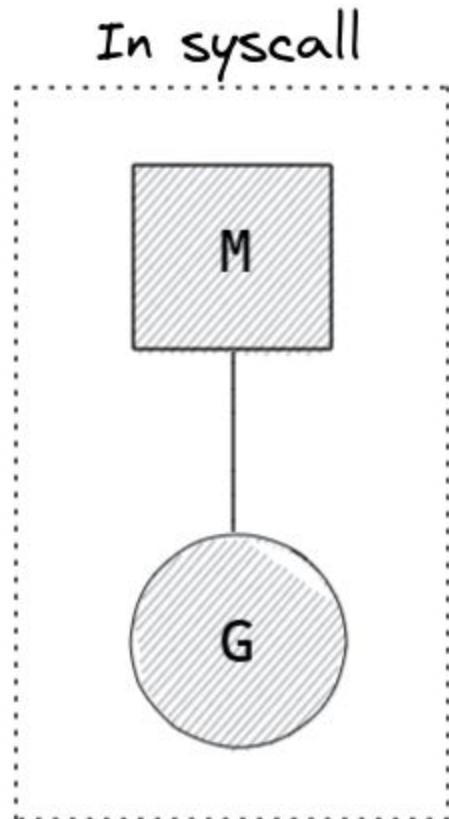


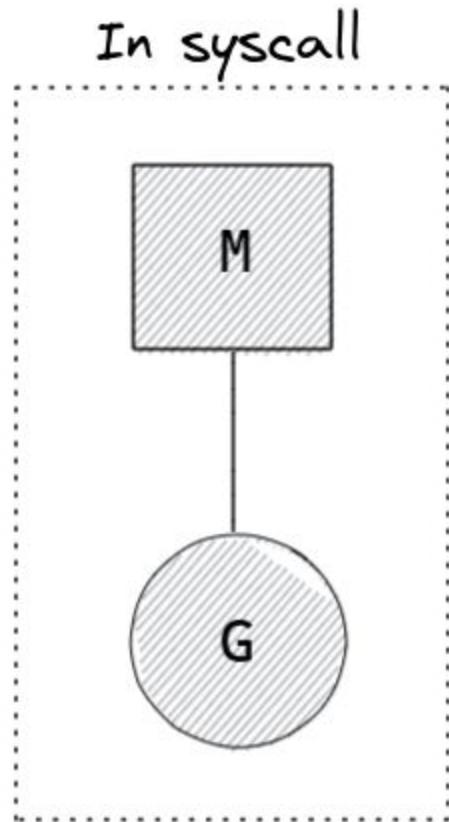
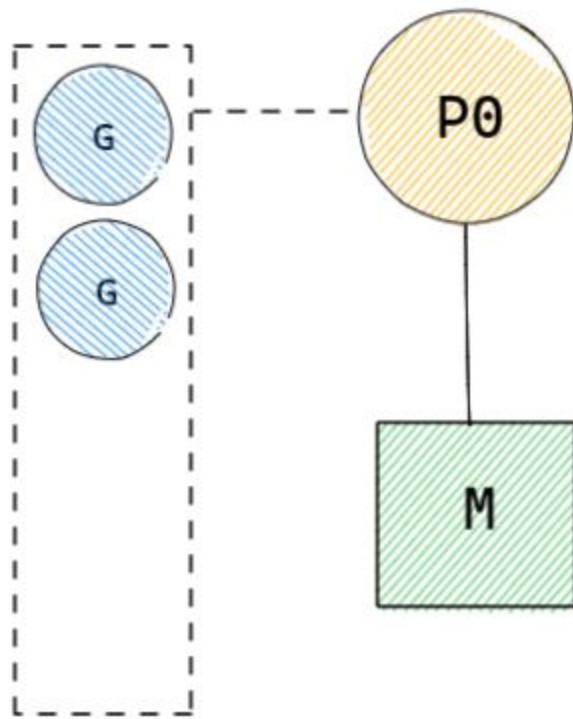


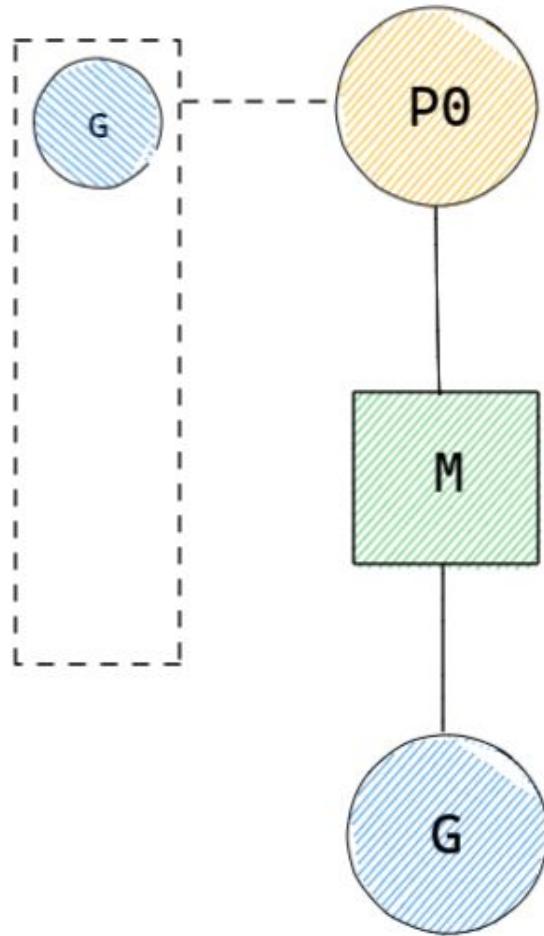




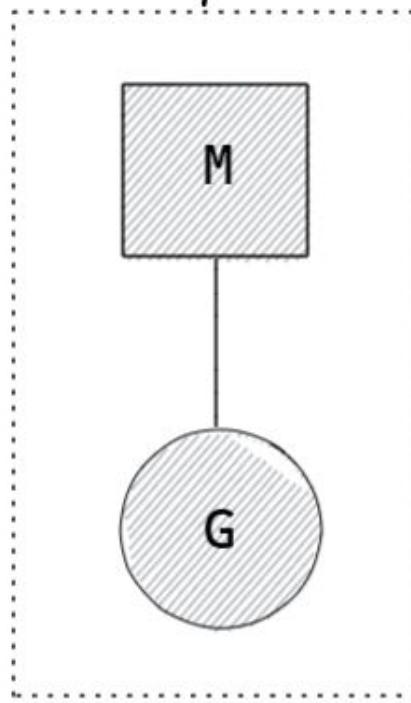
Newly created or spinning





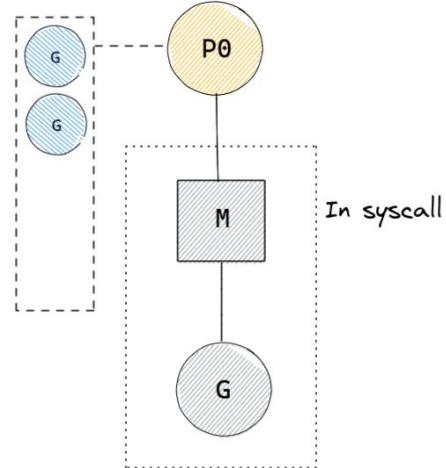


In syscall



handoff can get expensive

- Especially when you have to create a new thread.
- And some syscalls 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.



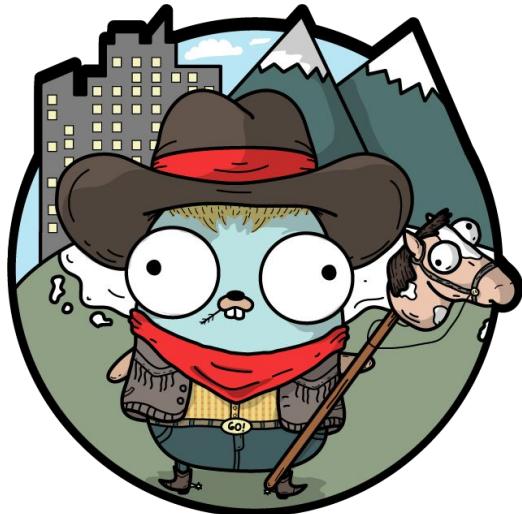
handoff can get expensive

- 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

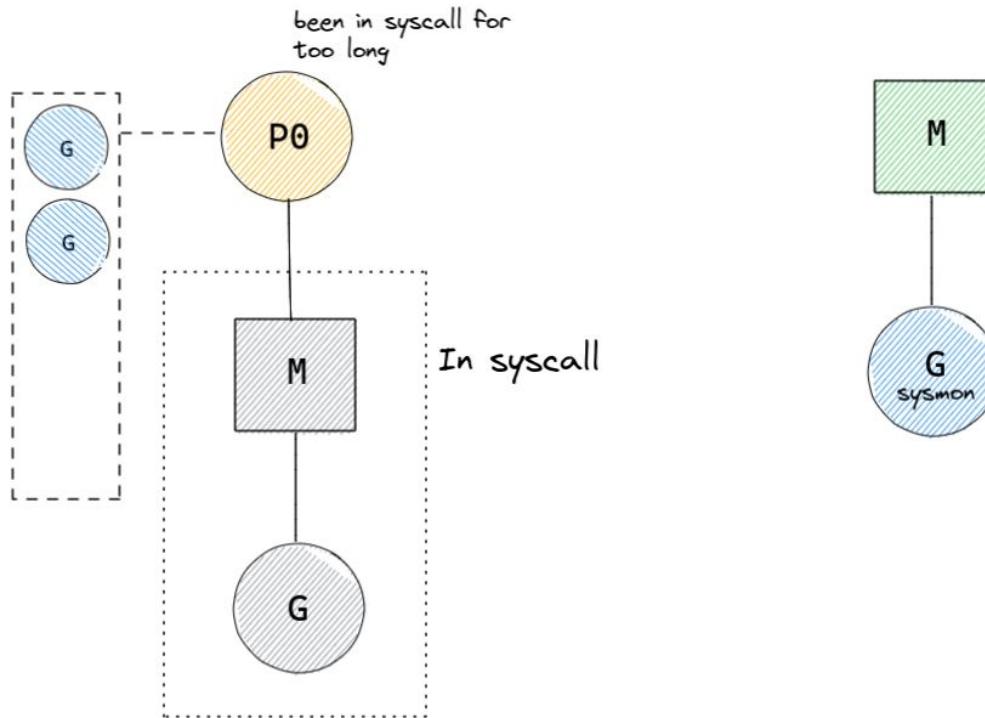
- But what happens in cases when we don't perform handoff and the p still ends up being blocked for a non-trivial amount of time?

sysmon



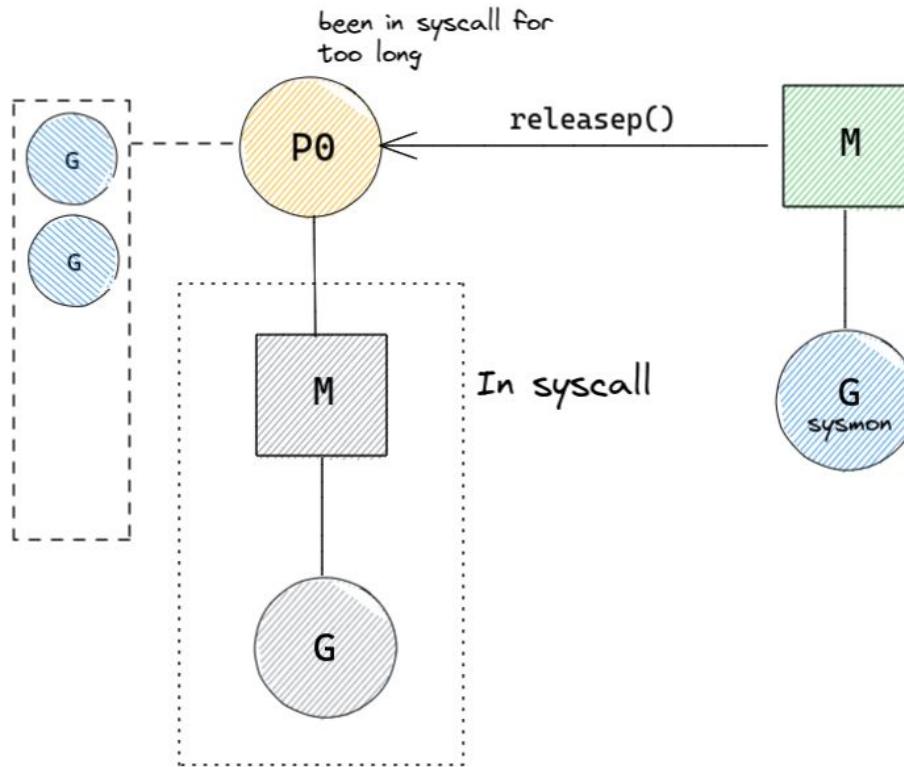
handoff can get expensive

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



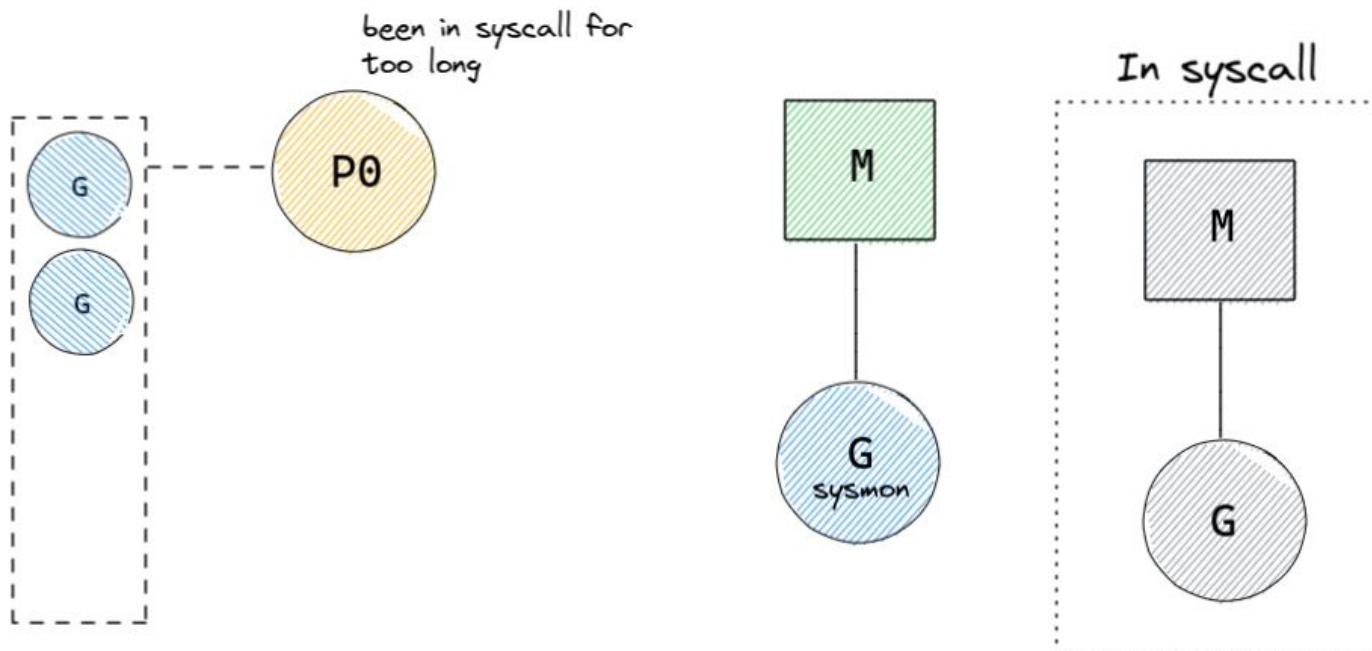
handoff can get expensive

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



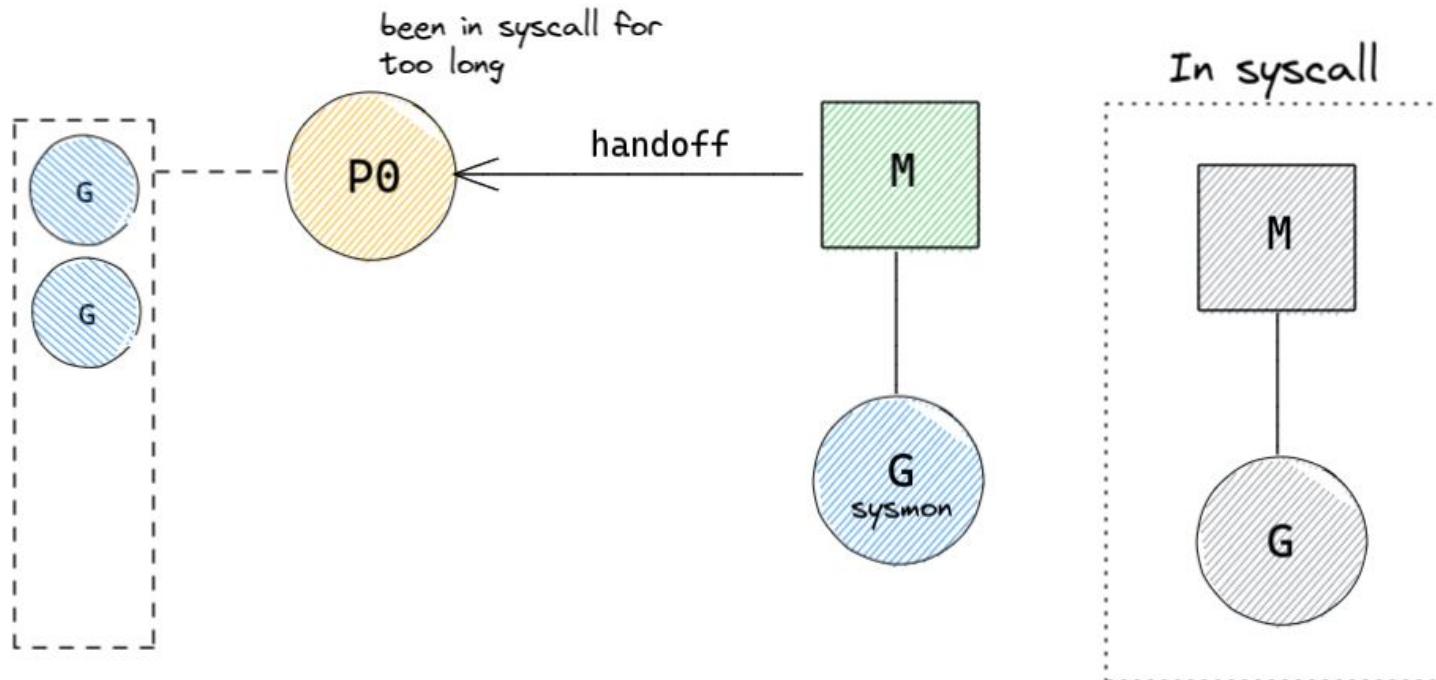
handoff can get expensive

- 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 its 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()

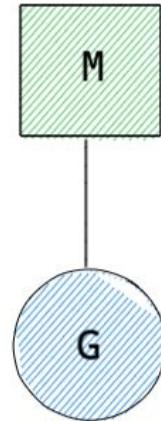
- Yields the processor.
- Calling Goroutine is sent to global queue.
- If you plan to use it for performance reasons, it's likely that improvement can be done in your implementation itself.
- Use only if absolutely necessary!

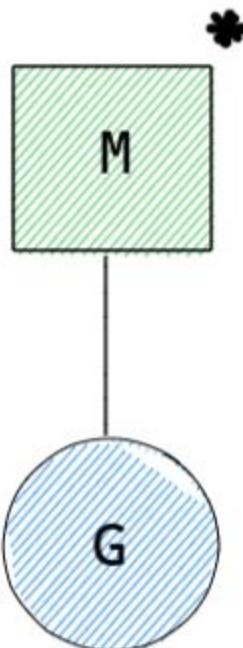
Goexit()

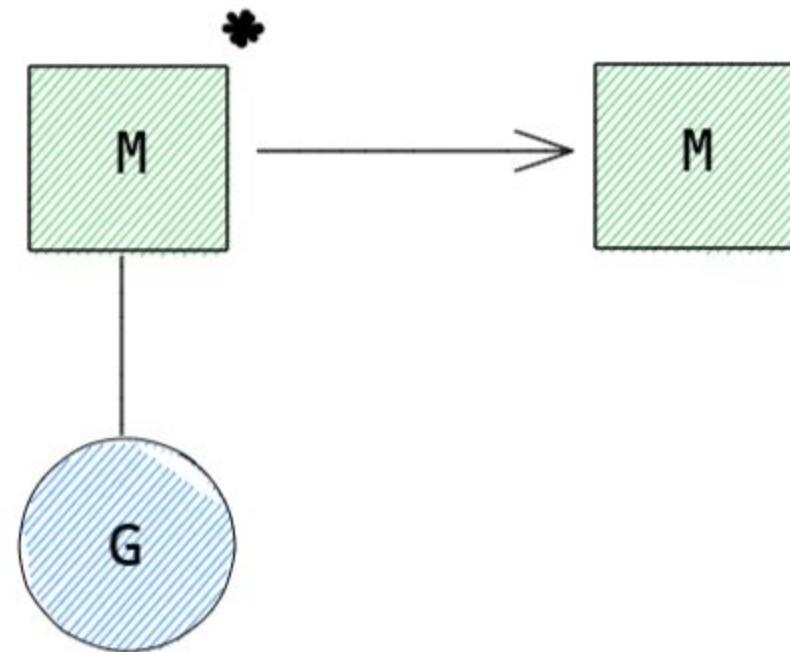
- Terminates (only) calling Goroutine.
- If called from main Goroutine, it terminates, and other Goroutines continue to run.
 - Program crashes once Goroutines finish because main Goroutine does not return.
- Used in testing (`t.Fatal()`)

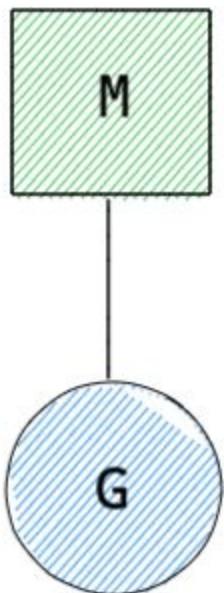
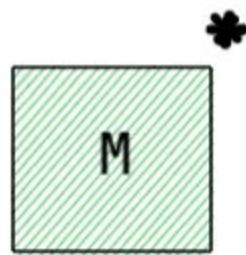
LockOSThread() / UnlockOSThread()

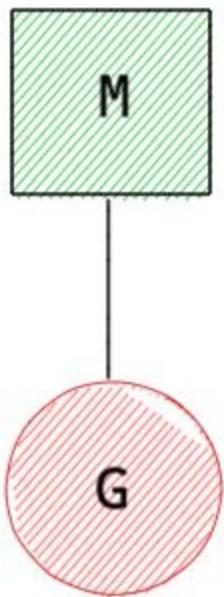
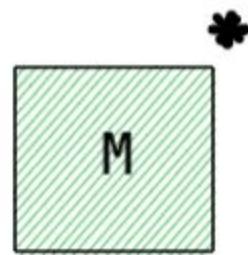
- Wires calling Goroutine to the underlying OS Thread.
- Primarily used when the Goroutine changes underlying thread's state.











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 fingerprint.

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 blames
- [Go's work-stealing scheduler](#)

Thank you!

