goroutine scheduler

guohailong

你所看到的一切，都在以并发方式运行。

因为Goroutine，Go才与众不同。


go scheduler evolution history
单线程调度器
多线程调度器
任务窃取调度器
抢占式调度器
基于协作的抢占式调度器
基于信号的抢占式调度器

go scheduler evolution history

单线程调度器 · 0.x

- 只包含 40 多行代码。
- 程序中只能存在一个活跃线程，由 G-M 模型组成。

多线程调度器 · 1.0

- 允许运行多线程的程序。
- 全局锁导致竞争严重。

任务窃取调度器 · 1.1

- 引入了处理器 P，构成了目前的 G-M-P 模型。
- 在处理器 P 的基础上实现了基于工作窃取的调度器。
- 在某些情况下，Goroutine 不会让出线程，进而造成饥饿问题。
- 时间过长的垃圾回收（Stop-the-world，STW）会导致程序长时间无法工作。

抢占式调度器 · 1.2 ~ 至今

基于协作的抢占式调度器 - 1.2 ~ 1.13

- 通过编译器在函数调用时插入抢占检查指令，在函数调用时检查当前 Goroutine 是否发起了抢占请求，实现基于协作的抢占式调度。
- Goroutine 可能会因为垃圾回收和循环长时间占用资源导致程序暂停。

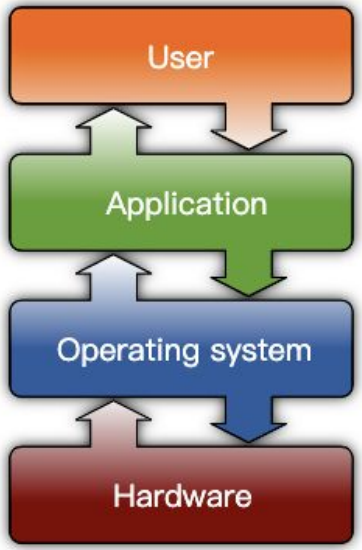
基于信号的抢占式调度器 - 1.14 ~ 至今

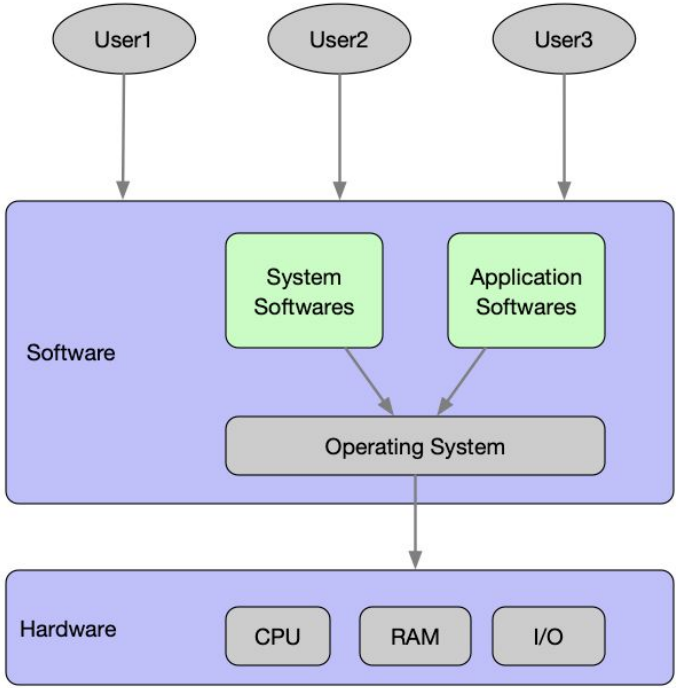
- 实现基于信号的真抢占式调度
- 垃圾回收在扫描栈时会触发抢占调度。
- 抢占的时间点不够多，还不能覆盖全部的边缘情况。

Operating system

An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Other





Process & thread

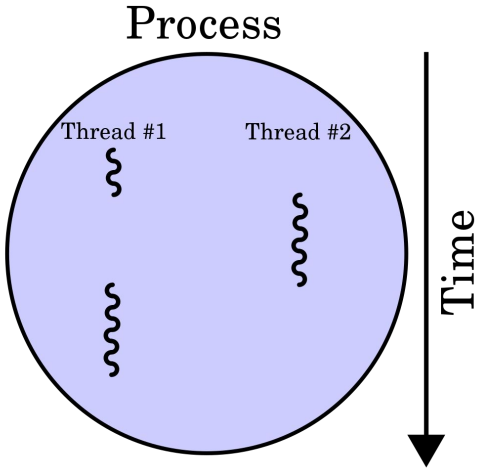
Processes

A process is the instance of a computer program that is being executed by one or many threads.

At the kernel level, a process contains one or more kernel threads, which share the process's resources, such as memory and file handles – a process is a unit of resources, while a thread is a unit of scheduling and execution.

Kernel threads

A kernel thread is a "lightweight" unit of kernel scheduling. At least one kernel thread exists within each process. If multiple kernel threads exist within a process, then they share the same memory and file resources.



Co-routine

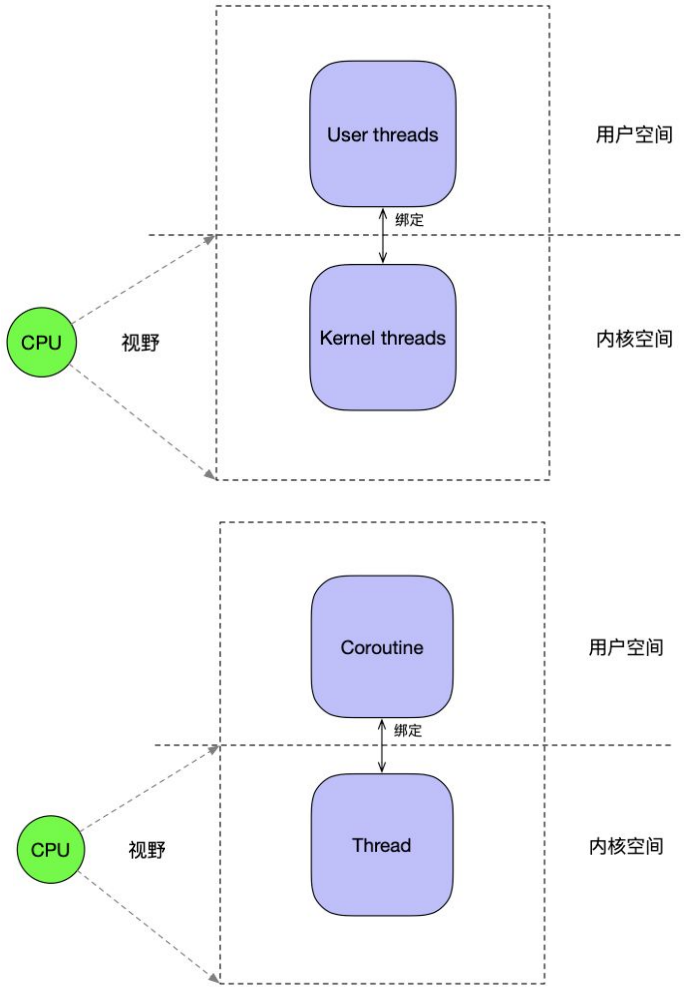
- **User threads**
- **Coroutine**

User threads

Threads are sometimes implemented in userspace libraries, thus called user threads.

The kernel is unaware of them, so they are managed and scheduled in userspace.

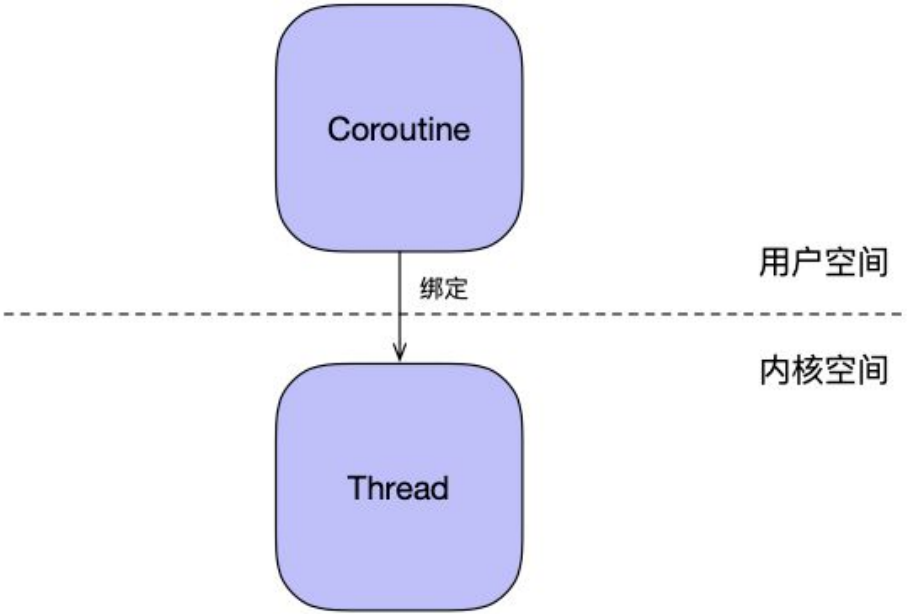
Some implementations base their user threads on top of several kernel threads, to benefit from multi-processor machines (M:N model). User threads as implemented by virtual machines are also called green threads.



Threading models

1:1 (kernel-level threading)

Threads created by the user in a 1:1 correspondence with schedulable entities in the kernel are the simplest possible threading implementation.

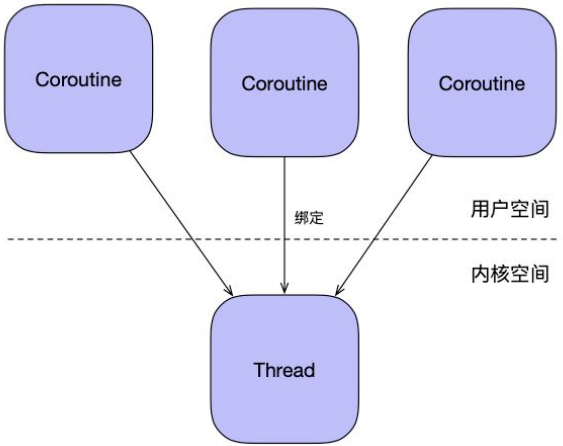


Threading models

N:1 (user-level threading)

An N:1 model implies that all application-level threads map to one kernel-level scheduled entity; the kernel has no knowledge of the application threads.
With this approach, context switching can be done very quickly.

One of the major drawbacks, however, is that it cannot benefit from the hardware acceleration on multithreaded processors or multi-processor computers: there is never more than one thread being scheduled at the same time.



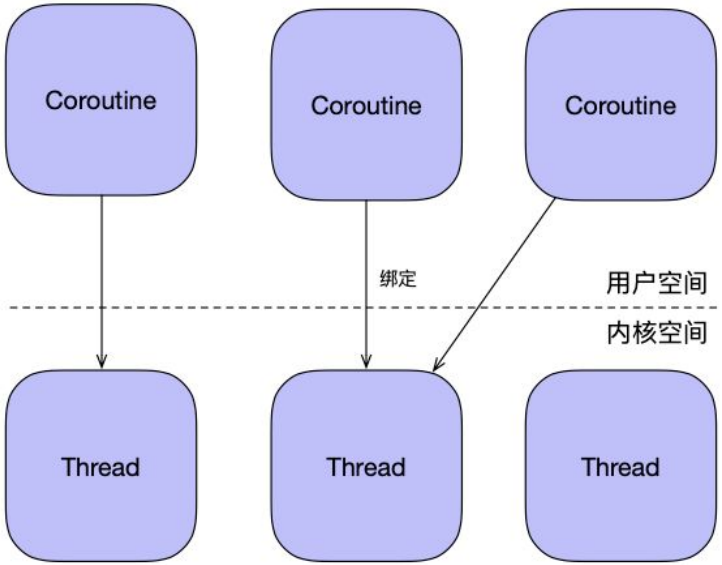
Threading models

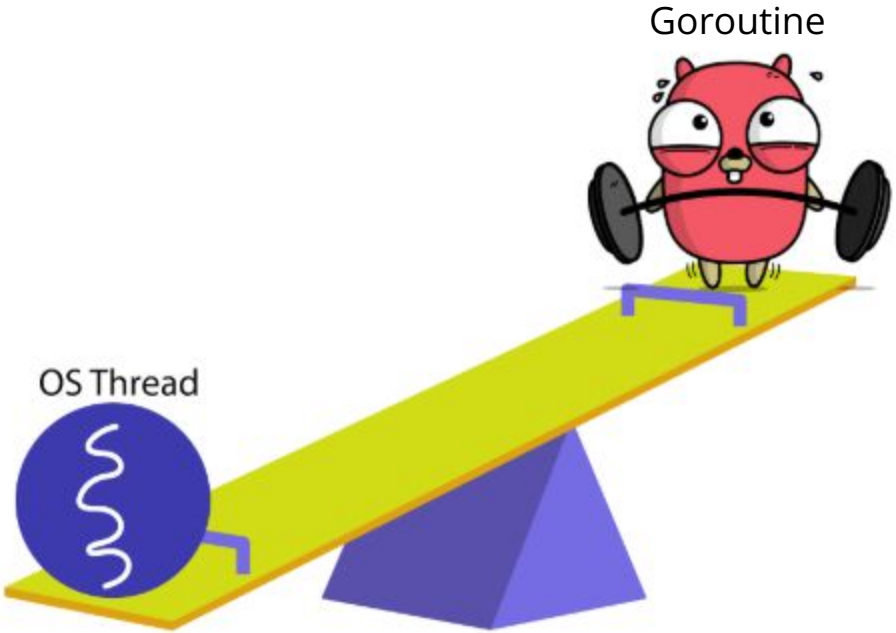
M:N (hybrid threading)

M:N maps some M number of application threads onto some N number of kernel entities.

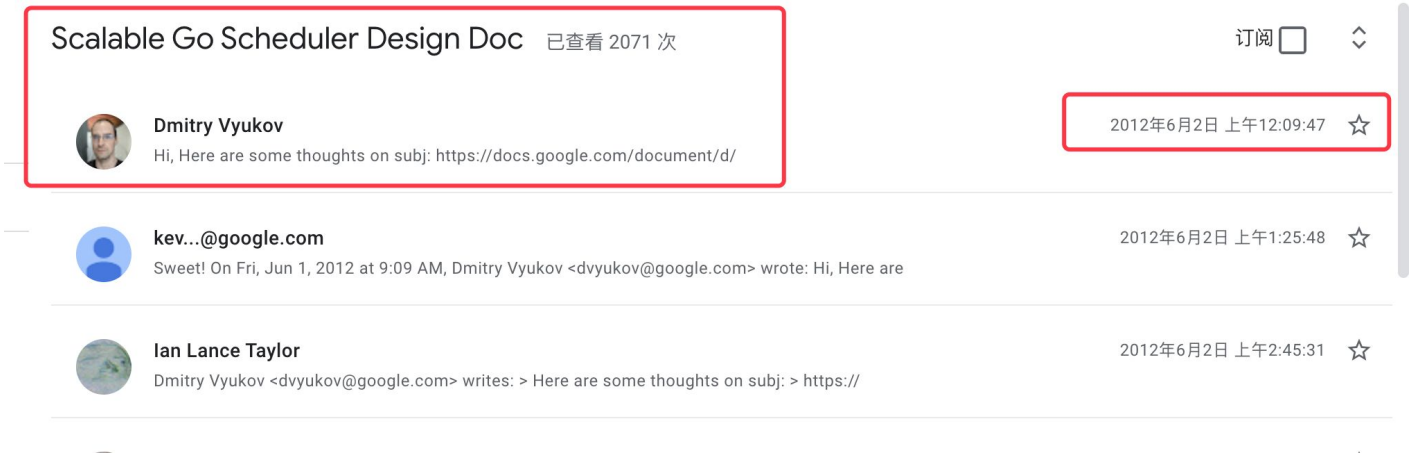
In the M:N implementation, the threading library is responsible for scheduling user threads on the available schedulable entities; this makes context switching of threads very fast, as it avoids system calls.

This is a compromise between kernel-level ("1:1") and user-level ("N:1") threading. In general, "M:N" threading systems are more complex to implement than either kernel or user threads, because changes to both kernel and user-space code are required.





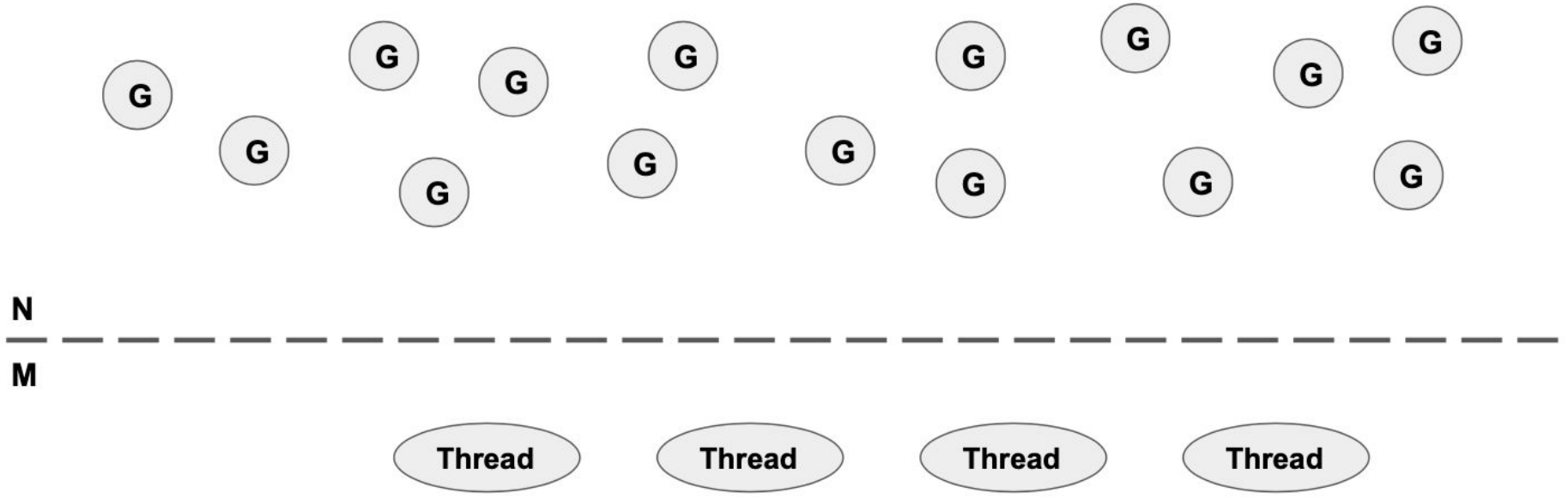
A *goroutine* is a **lightweight thread** (logically a thread of execution) managed by the Go runtime.



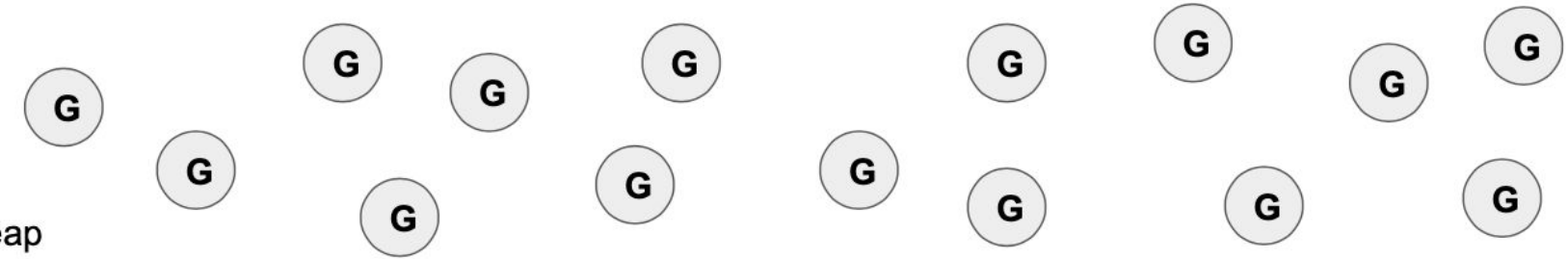
https://groups.google.com/g/golang-dev/c/_H9nXe7jG2U?pli=1

https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit

M:N Threading



M:N Threading



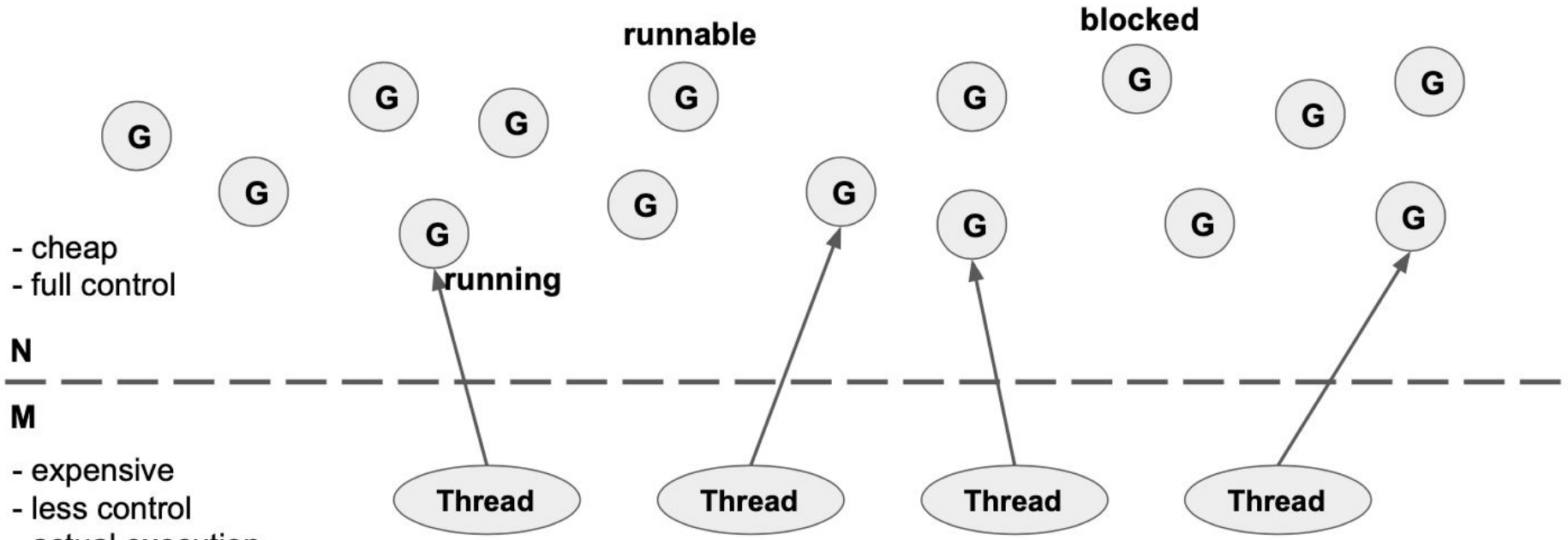
- cheap
- full control

N

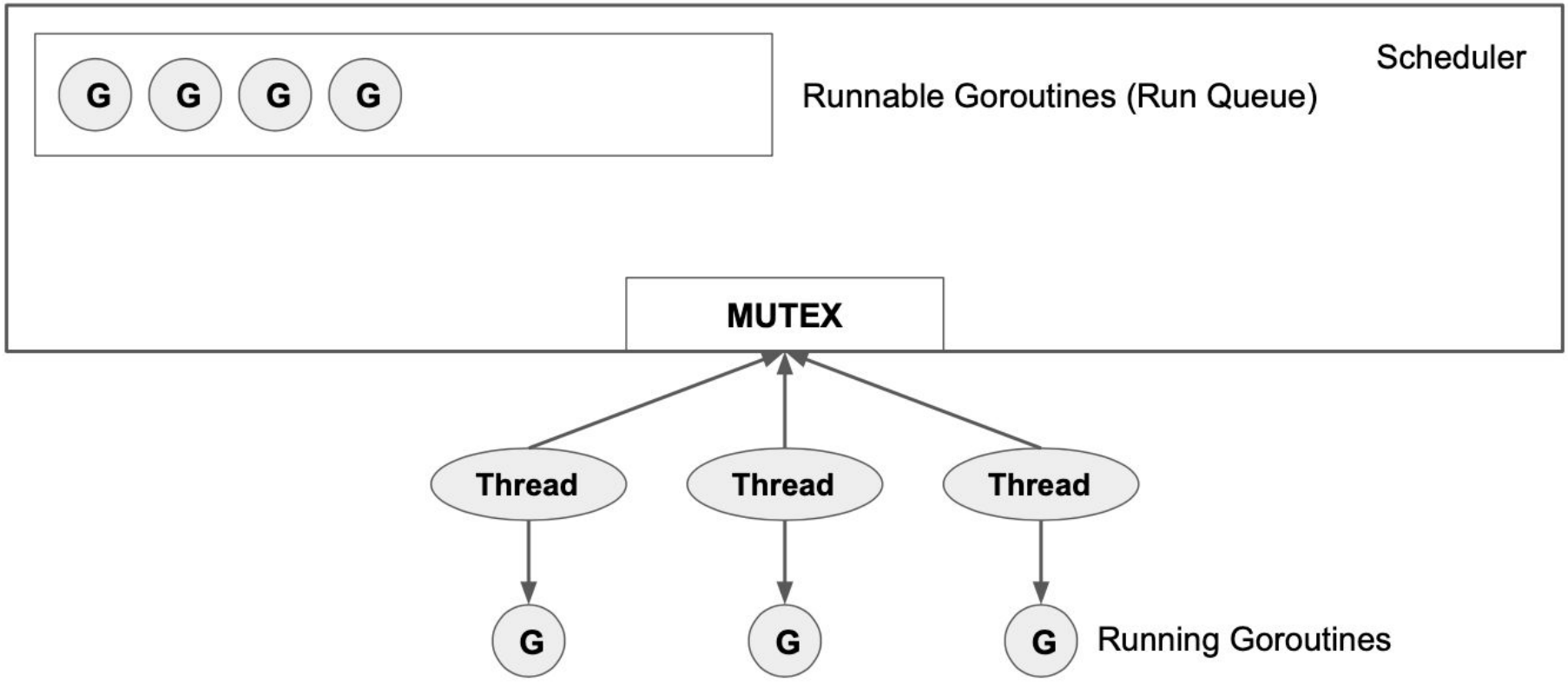
M

- expensive
- less control
- actual execution
- parallelism

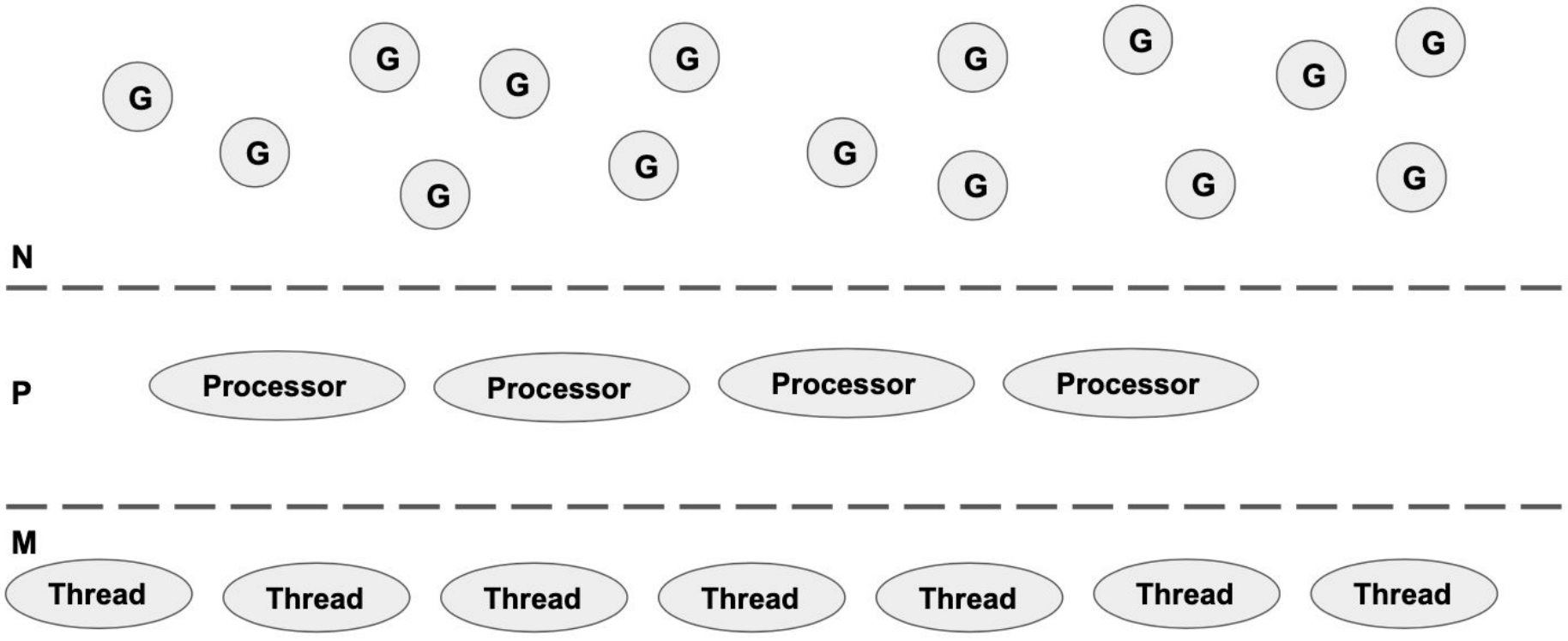
Goroutine States



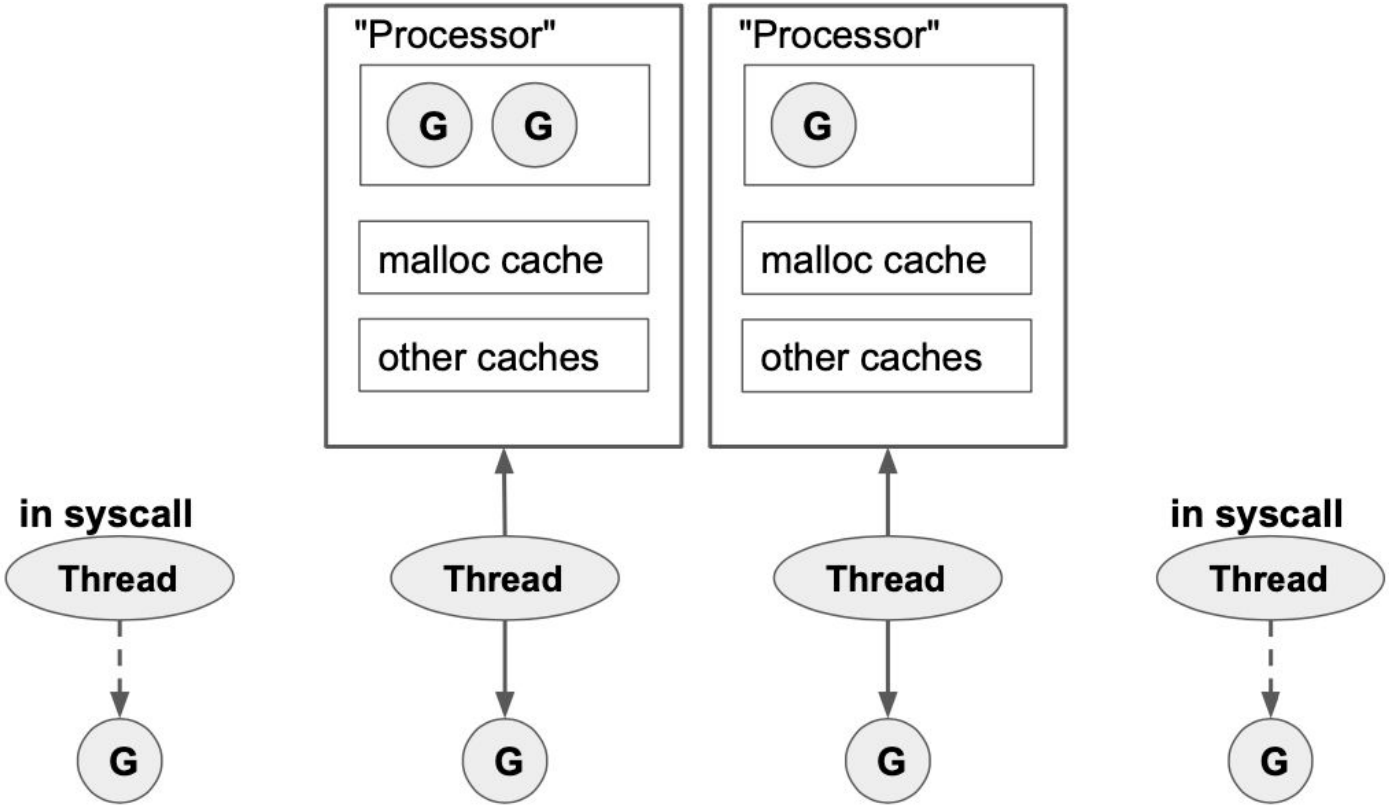
Simple M:N Scheduler



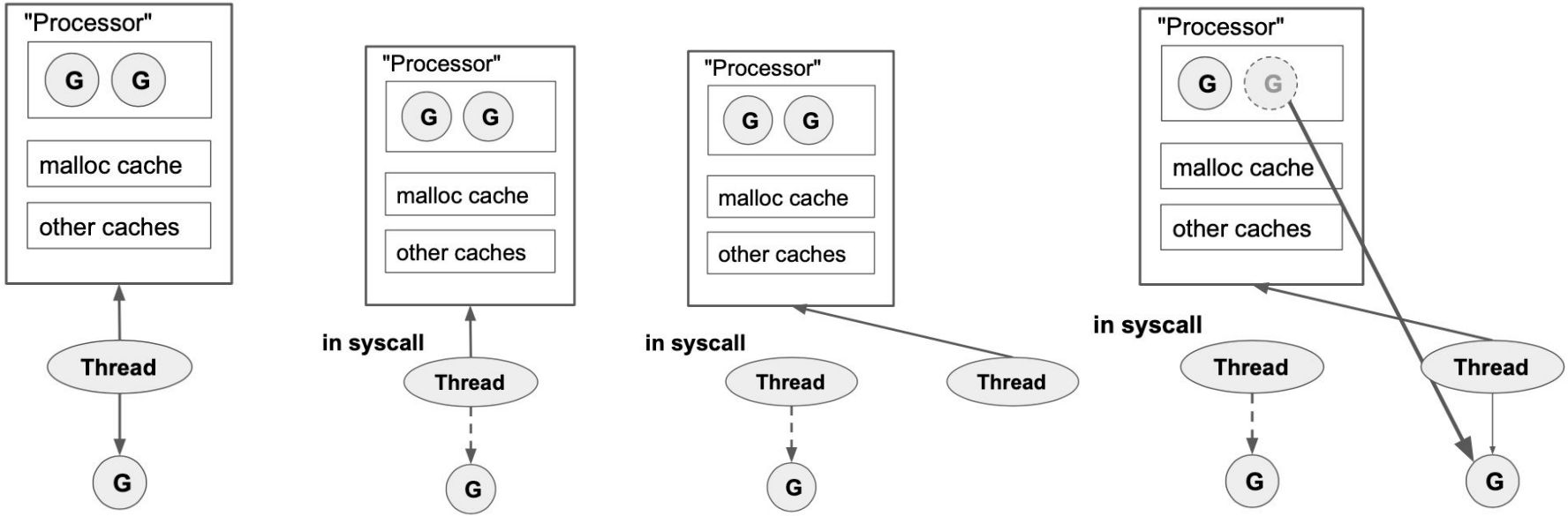
M:P:N Threading



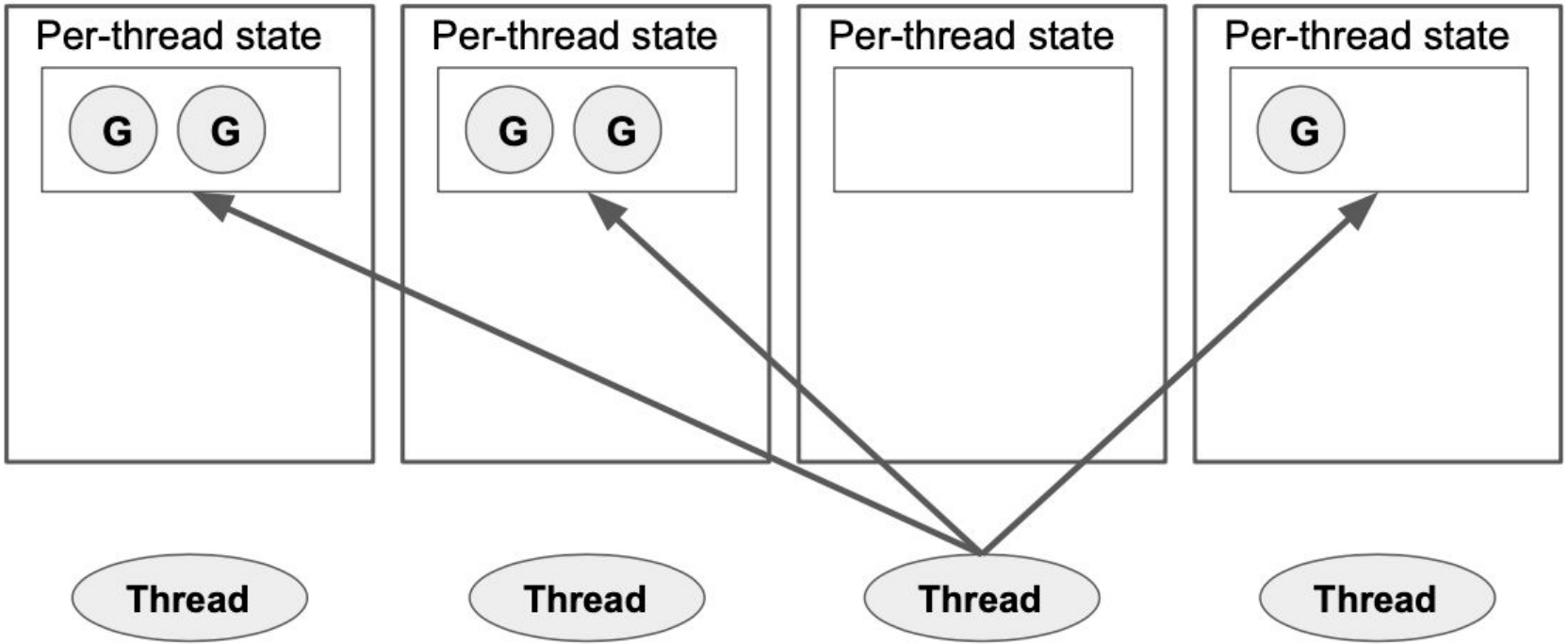
Distributed 3-Level Scheduler



Syscall handling: Handoff



Work Stealing

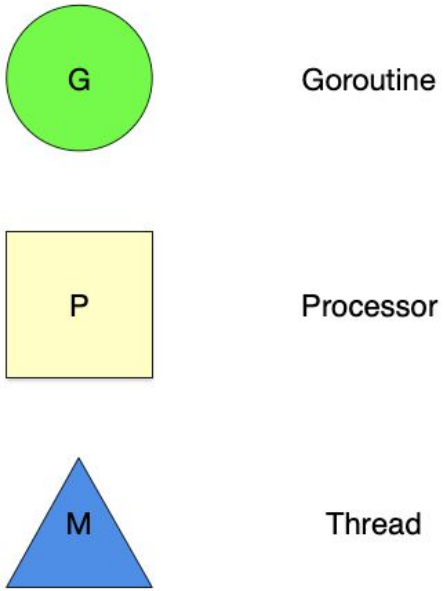


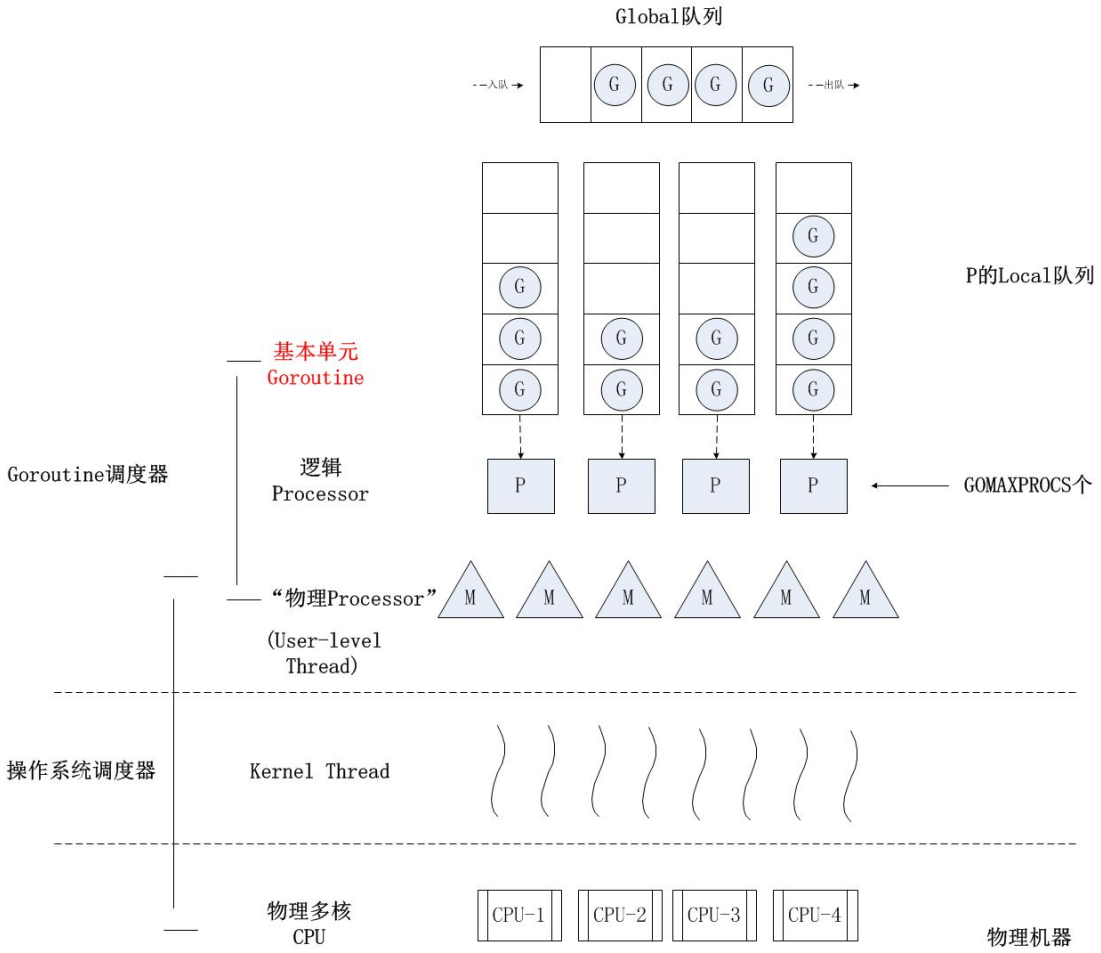
Poll Order

Main question: what is the next goroutine to run?

1. Local Run Queue
2. Global Run Queue
3. Network Poller
4. Work Stealing

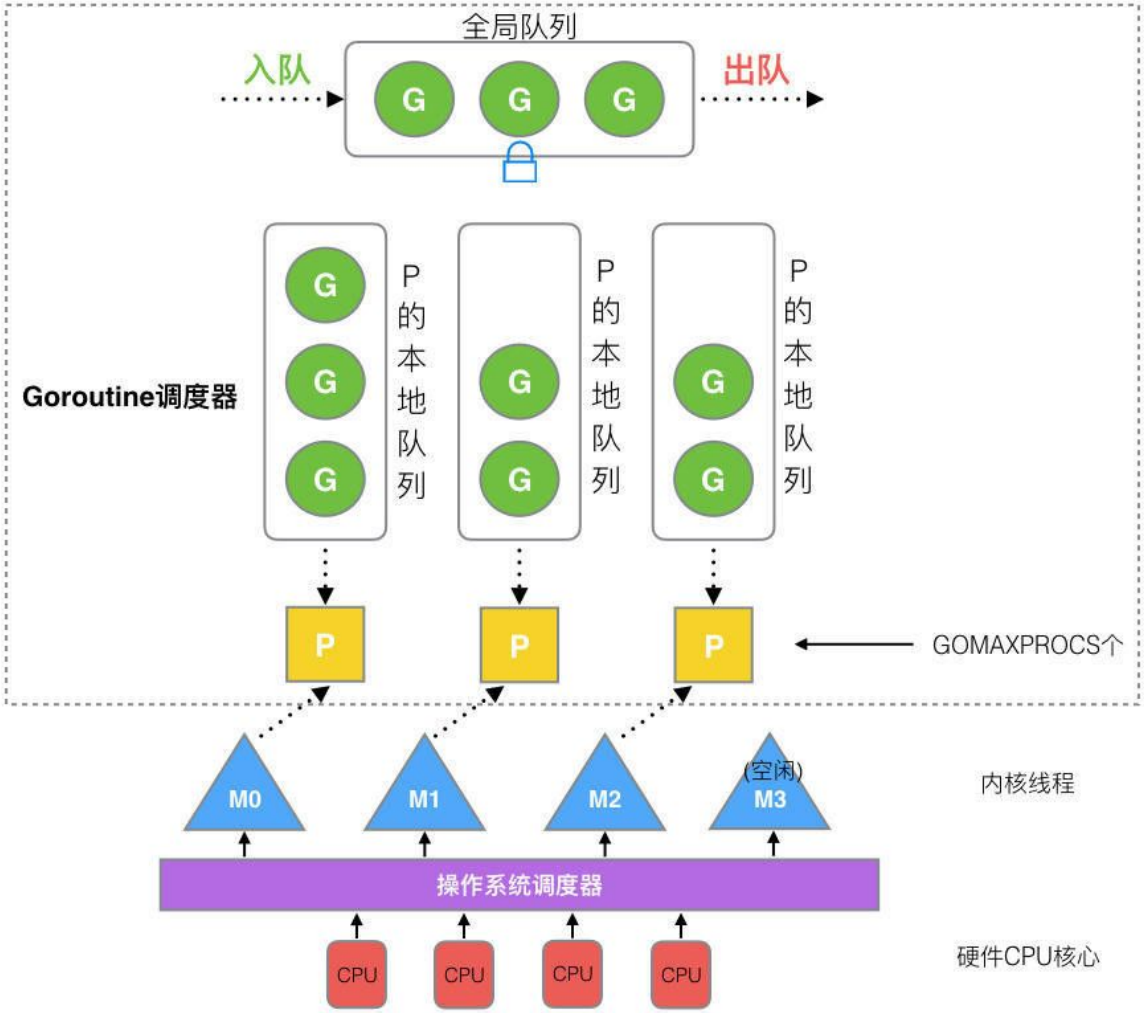
- G: goroutine
- M: OS thread (machine)
- P: processor





Goroutine调度原理图

看一个例子


全局队列
入队
出队
G
G
G
Goroutine调度器
G
G
G
P
的
本
地
队
列
G
G
P
的
本
地
队
列
G
G
P
的
本
地
队
列
P
P
P
GOMAXPROCS个
M0
M1
M2
(空闲)
M3
内核线程
操作系统调度器
CPU
CPU
CPU
CPU
硬件CPU核心

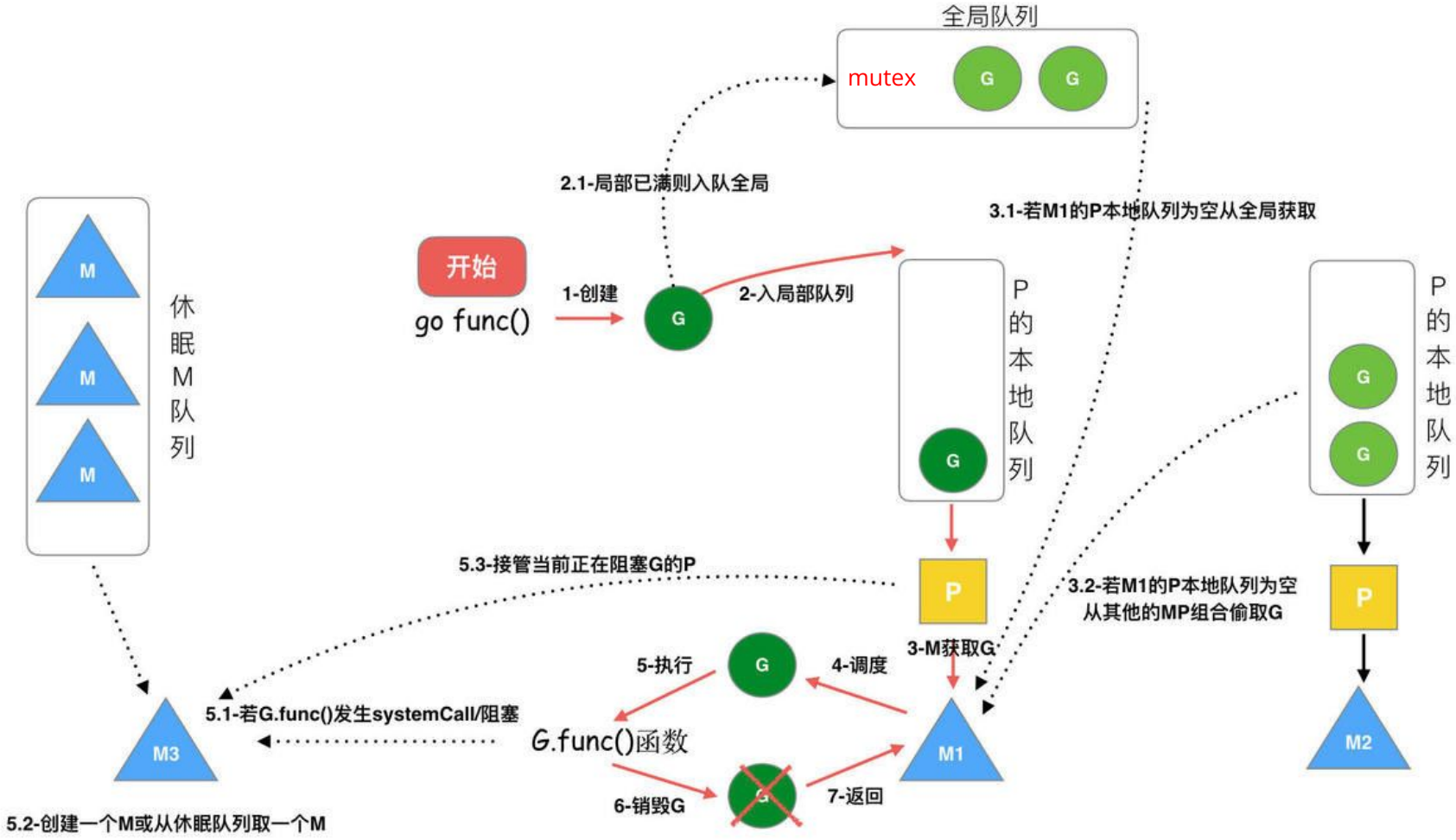
P、M 的数量

P 的个数：

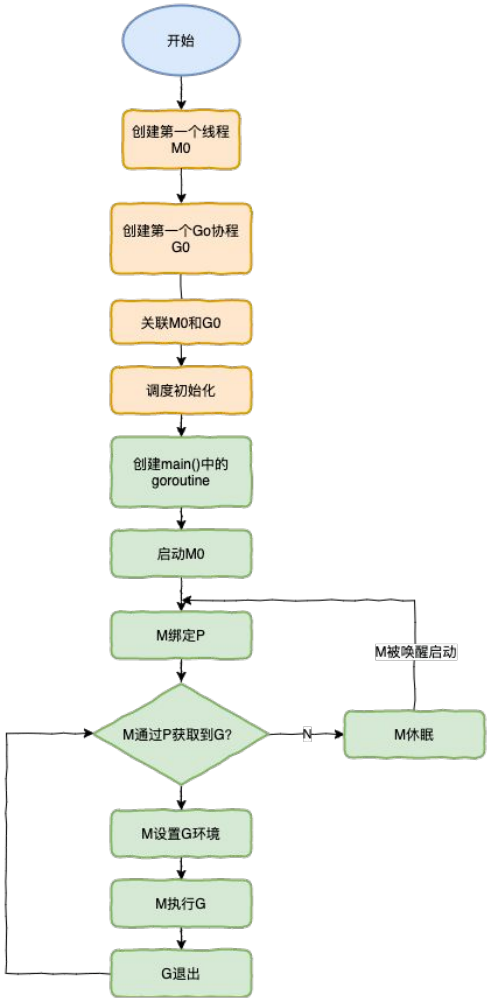
由启动时环境变量 $GOMAXPROCS 或者是由 runtime 的方法 GOMAXPROCS() 决定。

M 的个数：

- Go 程序启动时，会设置 M 的最大数量，默认 10000。
- runtime/debug 中的 SetMaxThreads 函数，设置 M 的最大数量。
- 一个 M 阻塞了，会创建新的 M。


全局队列
mutex
G
G
2.1-局部已满则入队全局
3.1-若M1的P本地队列为空从全局获取
开始
go func()
1-创建
G
2-入局部队列
M
M
M
休眠M队列
P的本地队列
G
P
P的本地队列
G
G
P
5.3-接管当前正在阻塞G的P
3.2-若M1的P本地队列为空
从其他的MP组合偷取G
3-M获取G
5-执行
G
4-调度
5.1-若G.func()发生systemCall/阻塞
G.func()函数
M3
M1
M2
6-销毁G
7-返回
5.2-创建一个M或从休眠队列取一个M

goroutine scheduler-生命周期




```go
package main

import "fmt"

func main() {
    fmt.Println("Hello world")
}
```

Scheduler-trace

https://github.com/golang/go/wiki/Performance#scheduler-trace

```
GOMAXPROCS=1 GODEBUG=schedtrace=1000 ./example
GOMAXPROCS=1 GODEBUG=schedtrace=1000,scheddetail=1 ./example
```

```
M0: p=-1 curg=-1
```

Summary

Go 调度本质是把大量的 G 分配到少量线程上去执行，并利用多核并行，实现强大的并发。

G 的初始栈仅有2KB，且创建操作只是在用户空间分配对象，远比进入内核态分配线程要简单得多。调度器让多个M进行调度循环，不停获取并执行任务，所以我们才能创建成千上万个并发任务。

Fairness Hierarchy

Goroutine - preemption

Local Run Queue - time slice inheritance

Global Run Queue - check once in a while

Network Poller - background thread

= minimal fairness at minimal cost

Q&A

Thanks

参考资料

https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit

https://assets.ctfassets.net/oxjq45e8ilak/48lwQdnyDJr2O64KUsUB5V/5d8343da0119045c4b26eb65a83e786f/100545_516729073_DMITRII_VIUKOV_Go_scheduler_Implementing_language_with_lightweight_concurrency.pdf

https://www.youtube.com/watch?v=-K11rY57K7k&t=325s

https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit#heading=h.xzdeqfl4wfqo

https://draveness.me/golang/docs/part3-runtime/ch06-concurrency/golang-goroutine/

https://learnku.com/articles/41728

https://medium.com/@ankur_anand/illustrated-tales-of-go-runtime-scheduler-74809ef6d19b

https://en.wikipedia.org/wiki/Operating_system

https://en.wikipedia.org/wiki/Thread_(computing)#Threading_models

https://en.wikipedia.org/wiki/Green_threads

https://rakyll.org/scheduler/

《Go语言学习笔记》