the scheduler

调度什么?

- process, thread

- coroutine -> goroutine

```go
package main

import "os"

func main() {
    var (
        ch     = make(chan struct{})
        images []string
    )

    for _, i := range images {
        go process(i)
    }

    <-ch
}

func process(image string) {
    go reportMetrics()

    f, err := os.Open(image)

    // ...
    _ = f
    _ = err
}

func reportMetrics() {

}
```

创建 goroutines

暂停 goroutine

系统调用阻塞

1 为什么需要调度器

2 如何实现一个调度器

3 一些关键问题

为什么需要调度器

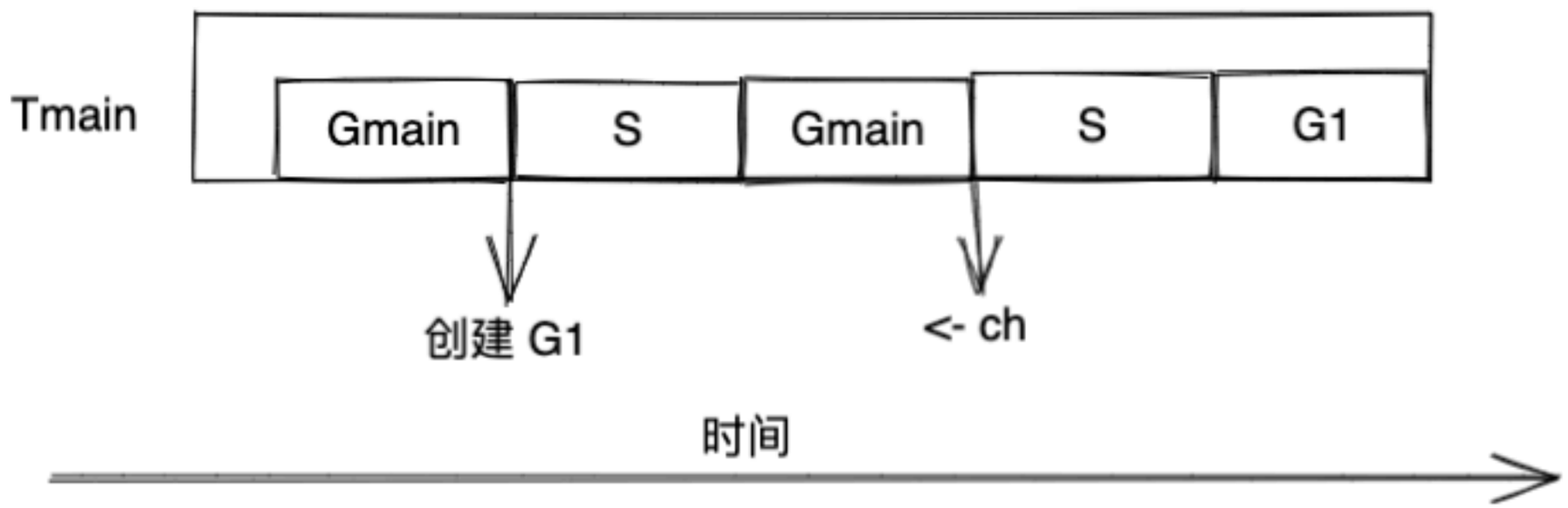
- goroutine -> 用户空间的线程
- initial goroutine stack = 2KB
- default thread stack = 8Kb
- 更快的创建、销毁、上下文切换

目标

- 使用小部分内核线程
- 支持高并发
- 将负载均摊到多个 CPU 核心上

何时调度

- 在影响 goroutine 执行的时候

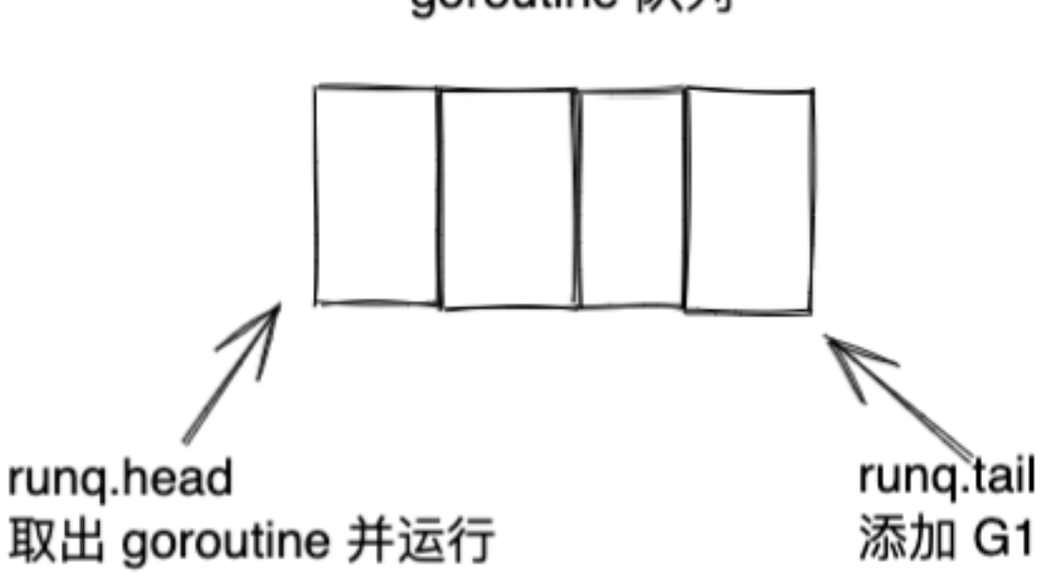


如何复用线程

- 何时创建线程
- 如何将 goroutine 分配到线程

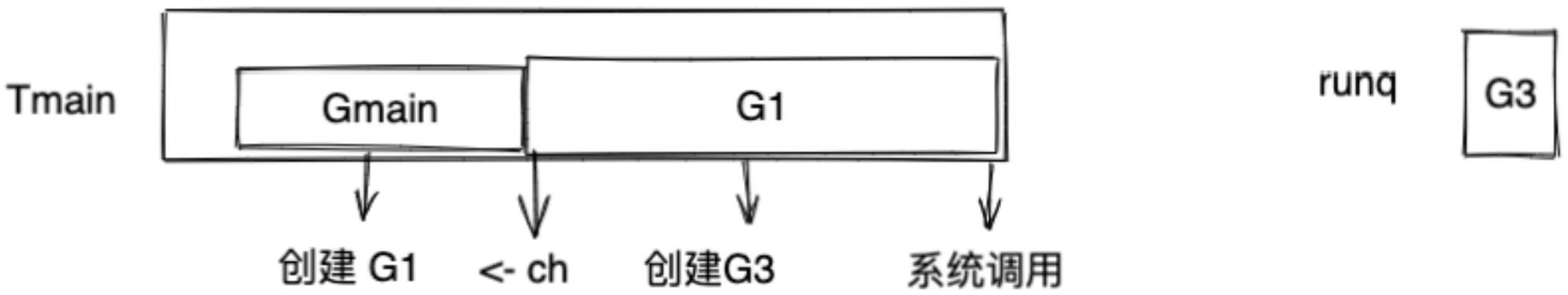
runqueue

- 使用 FIFO runq 来记录



初始版本

- 使用单线程运行全部 Goroutine



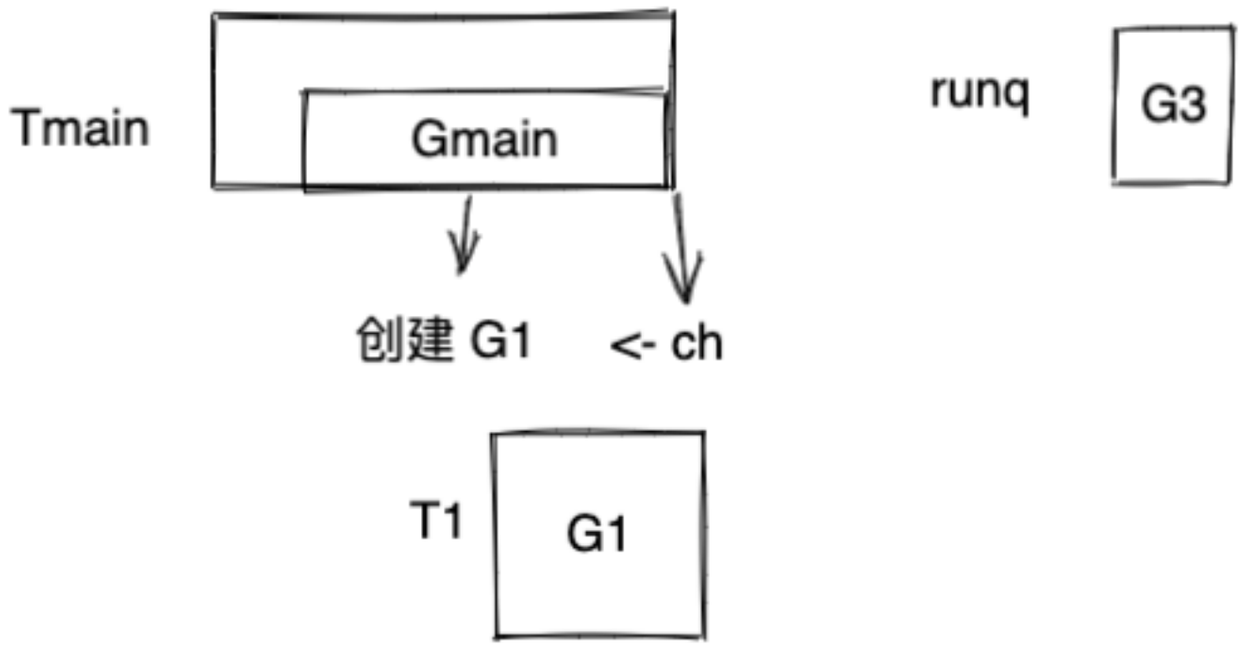
- 一个 goroutine 会阻塞线程，没有其他 goroutine 被调度

- 只利用了单个 cpu 核心

为每个 goroutine 创建线程?

复用线程

- 需要时创建新线程
- 空闲时，保留线程，以供之后使用

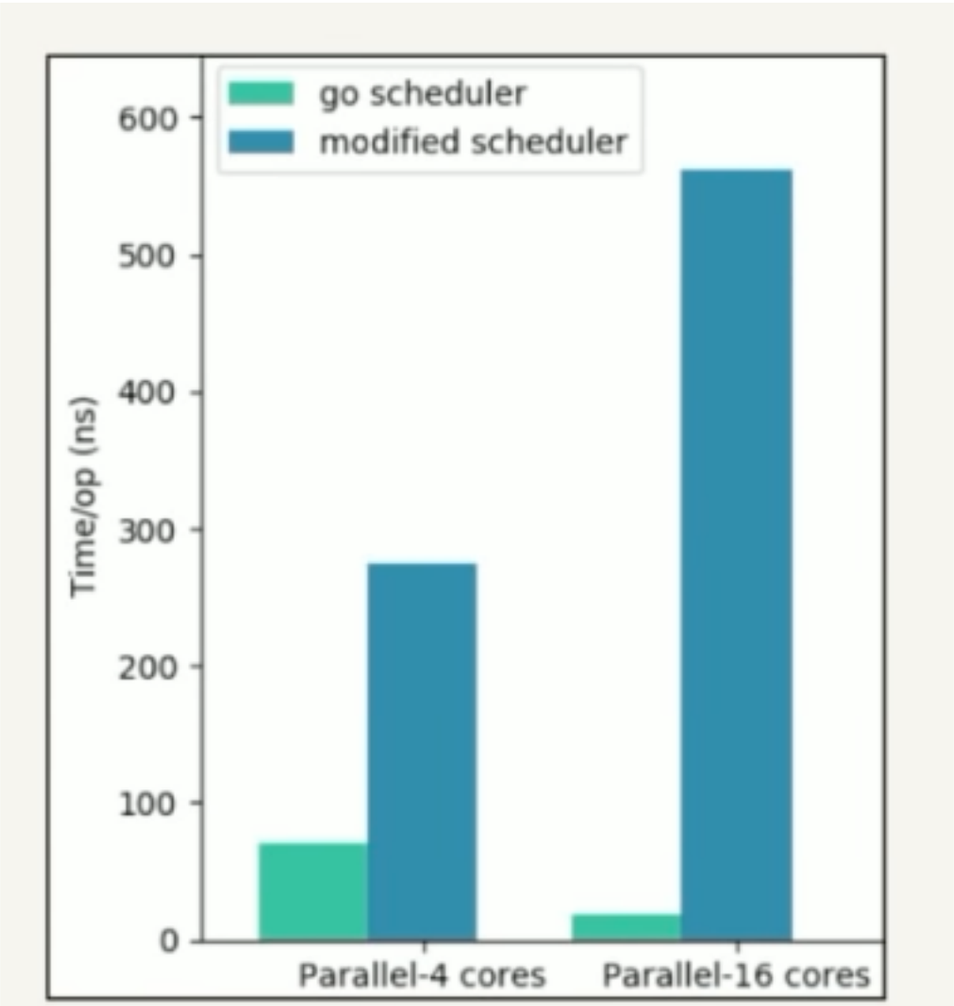


- runq 需要 lock
- 线程数过多

限制线程数

- 限制访问 runq 的线程数
- 处于 syscall 等其他阻塞的线程，不计入总数
- 线程最大数 = GOMAXPROC

多核心 cpu



scheduler benchmarks

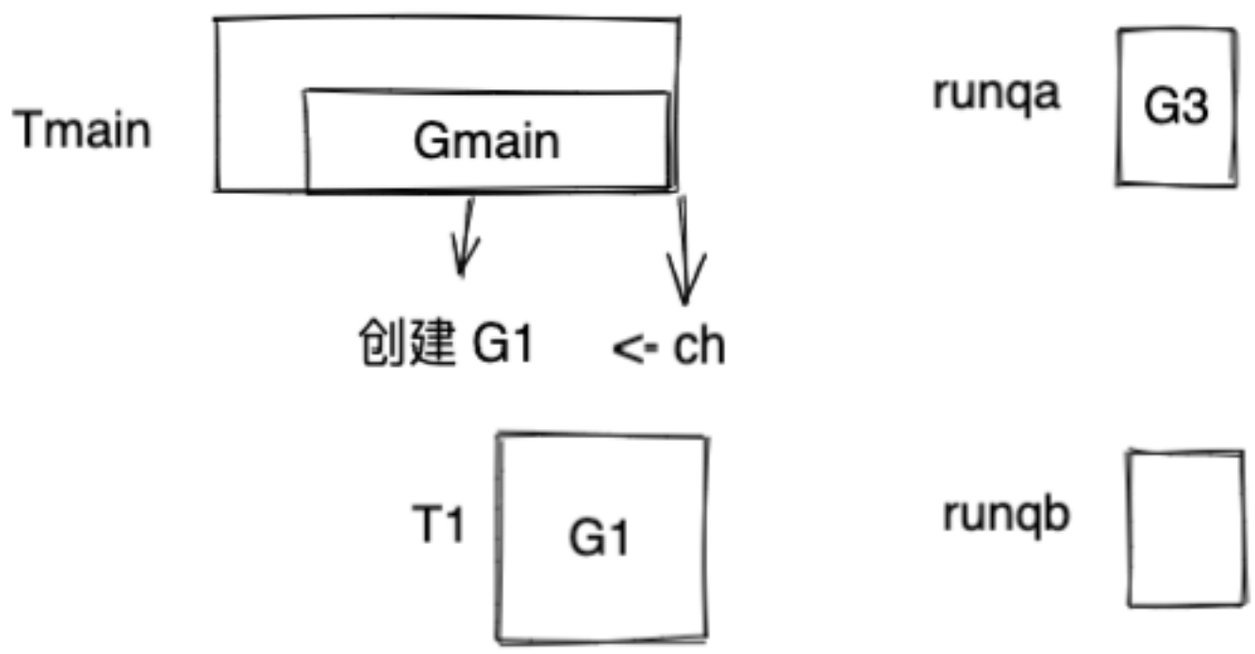
(`CreateGoroutineParallel`)

多 runqueue

- N 核心 CPU 机器 -> N runqueue

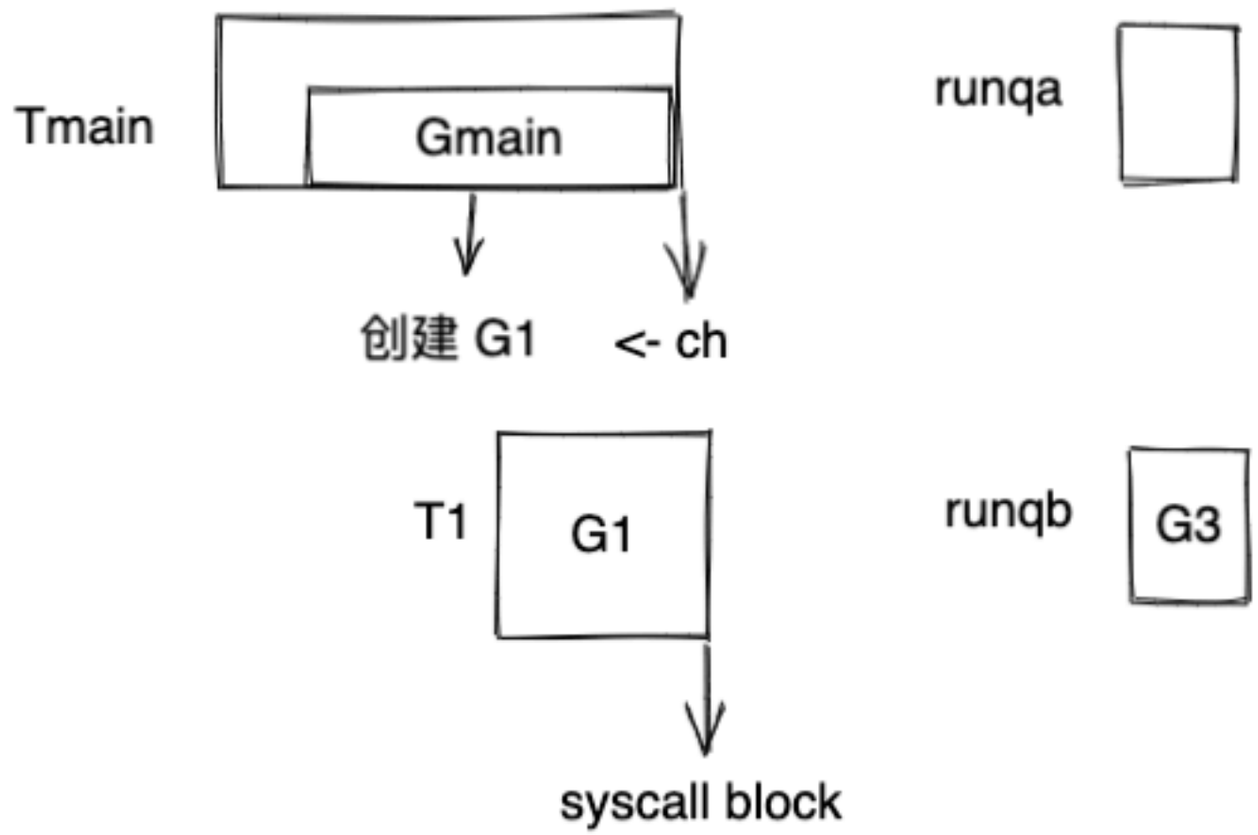
- 一个 runqueeue -> 一个线程

多 runqueue



- 能够从其他 runq 偷取 Goroutine

系统调用阻塞



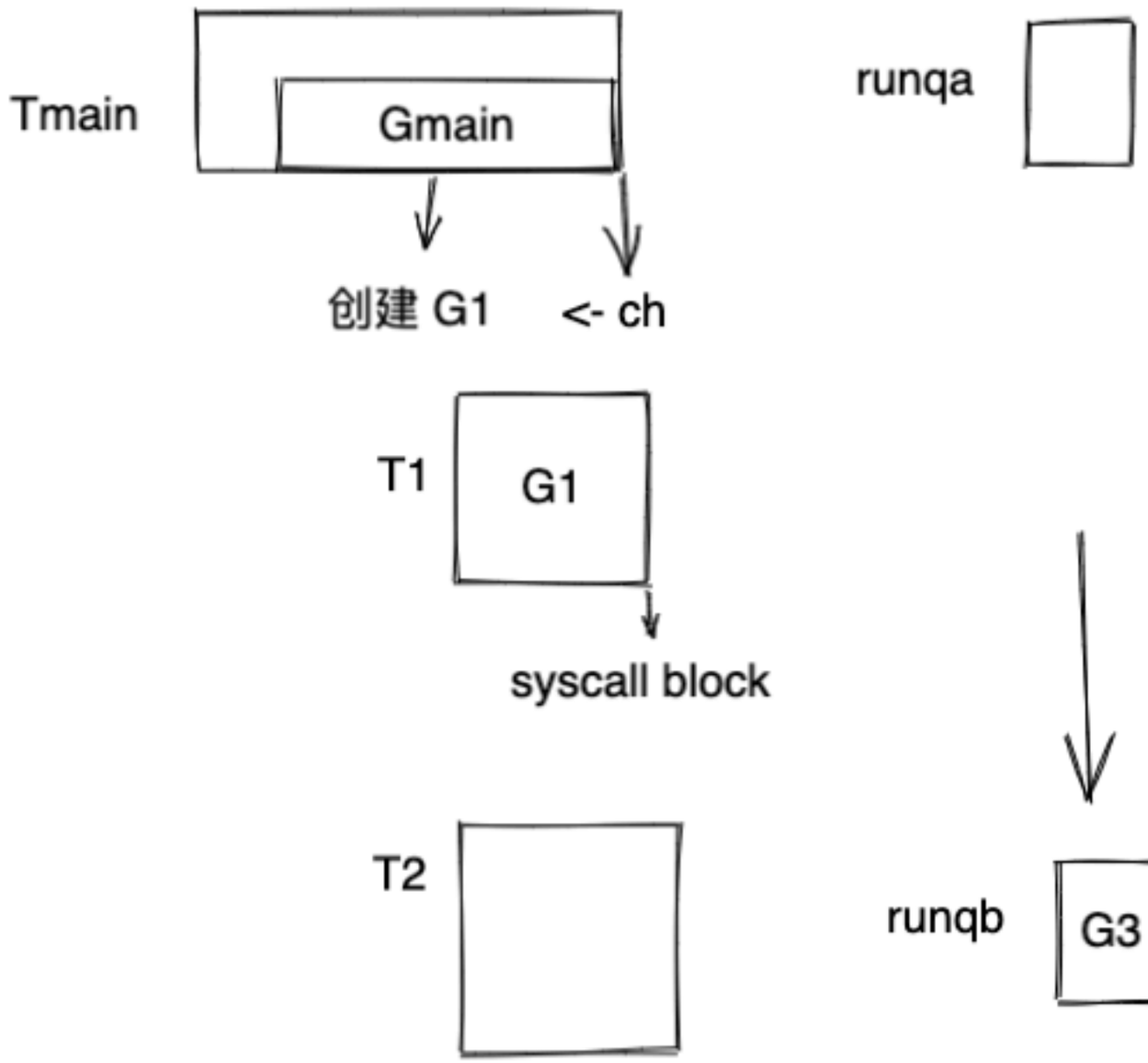
runq 转交

- monitor 监测到线程阻塞一定时间后，将 runq 转交给其他线程

- 如果在进入 syscall 前转交，可能会降低效率，因为 syscall 可能快速返回

- 创建新线程，并将 runq 转交

runq 转交



协作调度

- 协作式：goroutine 主动陷入调度器，在创建 goroutine、channel 操作、系统调用等时间进行调度

```go
func comlicatedAlgorithm(image string) {
    //
    for i := 0; i < math.MaxInt64; i++ {
        //
    }
}
```

抢占

- 后台线程: sysmon
- 运行时间超过 10ms 的 Goroutine
- 调度走 Goroutine, 调入全局低优先级 runq

小结

- 线程持续检查可运行任务，如全局 runq, 网络资源，gc，goroutine 窃取等
- runq 存储在结构体 P 中
- GMP, G: Goroutine, M: Thread, P: Runque

历史版本

- 0.x: 单线程调度，只存在一个活跃线程, G-M 模型
- 1.0: 多线程，全局锁竞争严重
- 1.1: 引入 P，实现了 Goroutine 窃取

历史版本

- 1.2 - 1.13: 基于协作的抢占调度

- 1.14 - 至今：基于信号的抢占调度

优势

- 少量内核线程
- 高并发支持
- 将负载均摊到多核心

限制

- FIFO，没有优先级
- NUMA 支持