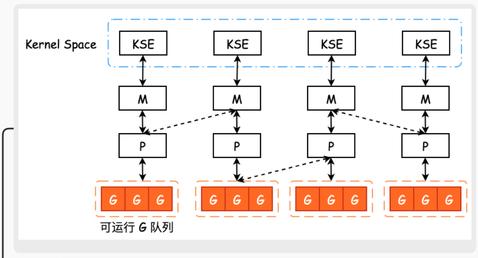
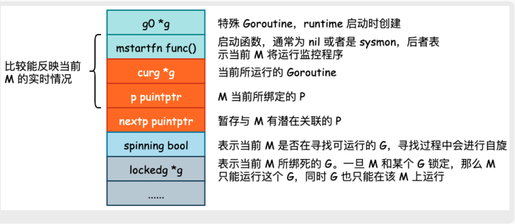
GMP

概述



- M
  - Machine，Goroutine 实际运行的载体，一个 M 表示一个内核线程，有时候也会叫它工作线程
  - 每一个 M 都是勤劳的工作者，不仅仅只是运行现有的任务，还会主动地到其它各个地方（队列）寻找可运行的 G，尽可能地不让自己闲下来
- P — Processor，简单的来说就是管理 G 对象和为其运行提供"上下文"，其内部包含两个队列，一个是可运行的 G 队列，另一个则是自由 G 列表
- G — Goroutine，对一段需要并发执行代码的封装

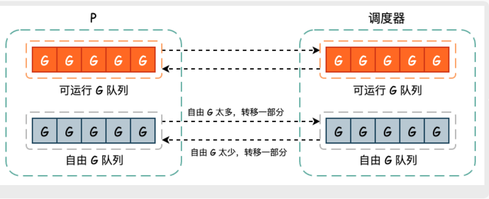
M



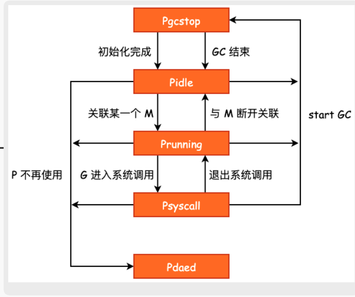
- 一个 M 对应了一个内核线程，在大多数情况下，创建一个 M 的原因都是没有足够的 M 去运行 G。另外一种情况就是 runtime 需要执行一些监控任务或者是 GC 时，也会有 M 的创建
- M 的最大值默认为 10000，即单个 go 程序最多能够创建 10000 个内核线程，但是实际上我们根本不可能创建这么多的线程，所以这个上限值是绝对够用的。另外，我们也可以通过 debug.SetMaxThreads() 来设置 M 的最大值
- 另外需要注意的是，M 和内核调度实体（KSE）之间的绑定关系非常稳固，也就是说，在 M 的生命周期内，不会和其它 KSE 进行绑定

P

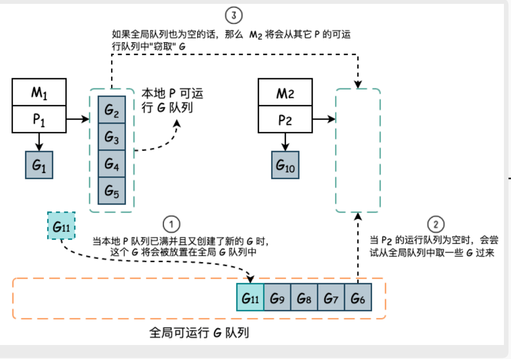
- P 是 G 能够在 M 中运行的关键，runtime 会在合适的时候让 P 绑定到另外的 M 上，使得 P 中保存的可运行 Goroutine 能够得到运行。也就是说，P 和 M 之间的关系并不牢固，正所谓"有奶便是娘"



- 当 P 中的自由 G 队列太长时，就会给调度器分一部分；当 P 的自由 G 队列为空时，又会从调度器中匀一部分过来
- 前面我们提到了 P 中有两个队列，一个是可运行 Goroutine 队列，另一个则是自由 G 队列，自由 G 队列其实就是保存那些已经运行完毕的 Goroutine。当程序调用 go func() 需要新建一个 Goroutine 时，就会从自由 G 队列中取出一个 G，并对其进行初始化，从而减少创建的 G 的时间。
- P 的状态转移



  - P 的初始状态为 Pgcstop，虽然此时并没有 GC 进行
- Psyscall
  - 在 P 的状态转移中，Psyscall 是一个较为重要的状态，表示了当前 G 正在进行系统调用（system call），可能是在等待文件、socket I/O 等
  - 当 P 当前的 G 进入系统调用后，P 将进入 Psyscall 状态，并且解绑与当前 M 的关联。因为此时 M 已经被进入系统调用的 G 所阻塞了，而 P 的可运行 G 队列中还有其它的 G，不能饿着孩子，所以解除当前 M 的关联，去寻找其余空闲的 M。并且原来的 M 会保存这个 P，以期望系统调用返回时能够进行快速绑定
  - 当 G 的系统调用结束后，M 将尝试和原来的 P 进行重新绑定
    - 绑定成功 — 更新 Psyscall 为 Prunning，M、P 和 G 像原来一样继续运行
    - 绑定失败 — 说明此时 P 已经和其它的 M 进行了绑定，那么需要将当前 G 交给调度器的可运行 G 队列中，M 再找一个其它空闲的 P



  - 左图描述了关于 G 调度的一些内容，包括本地 P 可运行 G 队列已满、P 可运行队列为空，以及全局可运行 G 队列为空的情况
  - ❶ P 中的可运行队列长度是有限的（目前是 256），当某一个 Goroutine 创建了超出其容量的 G 时，超出的 G 将会被放置于全局可运行 G 队列中，等待被其它 M 所获取
  - ❷ 当某一个 P 中的可运行队列为空，但是全局队列不为空时，将会把全局队列中的一部分 G 挪过来进行运行
  - ❸ 除此之外，如果全局队列也为空的话，那么将会从其它 P 队列中"窃取"一部分的 G 过来
- Questions — 限制 M 的数量和限制 P 的数量分别通过什么方式？它们之间有什么区别？
  - 通过 debug.SetMaxThreads() 来限制 M 的数量，通过 runtime.GOMAXPROCS() 来限制 P 的数量
  - 限制 M 的数量实际上是限制了单个 go 程序所能创建的内核线程数，限制 P 则是限制了 Goroutine 的最大并发数，限制 P 的数量并不等同于限制 M 的数量，尽管它们是一对一的。也就是说，此时 M 的数量可能多于 P 的数量，尤其发生在 M 被阻塞时

G

- G 即 Goroutine，当用户使用 go func() 创建一个并发任务时，runtime 会在本地 P 的自由 G 列表以及调度器的自由 G 列表中获取一个可用的 G，然后对其进行初始化。如果没找到，那么就新建一个 G 对象，并初始化。初始化的过程主要包括运行函数的设置、状态的变更以及 Groutine ID 的生成
- G 本身并没有什么过于特殊的地方，当一个 G 运行结束后并不会被销毁，而是会被放在本地 P 的自由 G 列表或者是调度器的全局自由 G 列表中，以便后续进行复用

Scheduler 如何对 G 进行调度

- 基于协作的抢占调度（sysmon + runtime）
  - 基于 协作的抢占机制依赖于 sysmon，即 system monitor，系统监视器。当 Go 程序运行时，runtime 就会启动一个名为 sysmon 的 M，sysmon 主要的工作就是监控运行时的 GC 情况，收回长时间阻塞在系统调用的 P，以及向长时间运行的 G 发出抢占调度
  - 抢占调度的原理非常简单，只要 sysmon 发现一个 G 运行了超过 10ms，就将 G 的抢占标志位设置成 true，那么当 G 下一次调用函数或方法时，runtime 将会对其进行抢占，将 G 移入到本地 P 的可运行队列中
  - 但是，基于 sysmon 实现的抢占调度仍然无法解决死循环 G 的抢占，比如 for {}，因为抢占的时机发生在 G 调用函数或方法时，如果 G 根本就不调用任何的函数或者方式时，runtime 也没辙
- 基于信号的抢占调度
  - 这个信号也就是操作系统中的信号，即 signal，调度器主要使用了 SIGURG 信号
  - go 在启动时将所有的信号都注册了一遍，包括实时信号，那么 SIGURG 也将会对应一个信号处理函数，由 sigaction() 系统调用所指定。同样是由 sysmon 进行扫描，当发现某一个 G 运行时间超过 10ms 之后，向运行该 G 的 M 怼一个 SIGURG 信号，M 线程收到信号后将 G 扔到可运行 G 队列中，即完成抢占
  - 我们可以使用 pthread_kill() 向同一个进程下的具体线程发送信号，但是无法向进程信号那样，在外部向某一个具体的线程发送信号
- 使 G 均匀分布在各个 M 上 — 其实就是 work stealing，即工作窃取，上面儿也讨论过，M 从其它 P 的可运行 G 队列中拿一些到自己这儿来运行