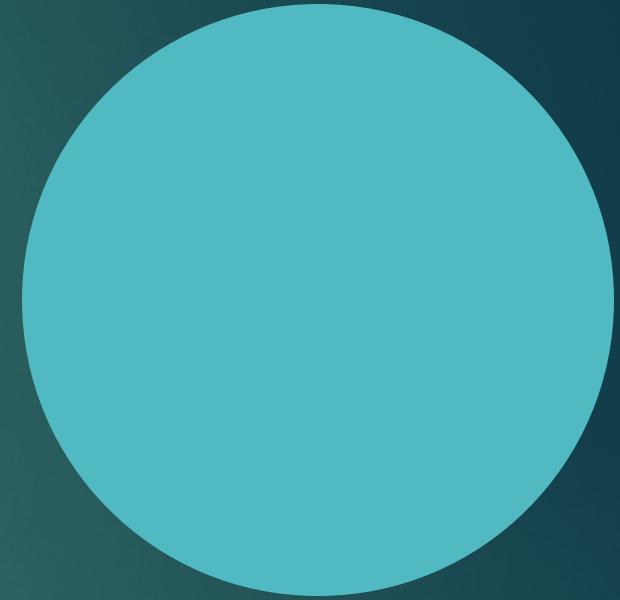


Go Scheduler



线程和进程

- ▶ 一个进程至少需要一个线程作为它的指令执行体，进程管理着资源（比如cpu、内存、文件等等），而将线程分配到某个cpu上执行。
- ▶ 进程是资源管理的最小单位，线程是程序执行的最小单位。

Linux内核实现

- ▶ 创建进程的方式
 - ▶ fork
 - ▶ vfork
 - ▶ clone
- ▶ 内部实现: do_fork
 - ▶ 签名: long do_fork(unsigned long clone_flags, ...)
 - ▶ fork - SIGCHLD
 - ▶ vfork - CLONE_VM | CLONE_VFORK | SIGCHLD
 - ▶ clone – 自定义flags, 可用于创建线程

Linux内核实现

- ▶ 创建线程的方式
 - ▶ clone
- ▶ 内部实现: do_fork
 - ▶ 签名: long do_fork(unsigned long clone_flags, ...)
 - ▶ clone – CLONE_VM | CLONE_FS | CLONE_FILES | ...

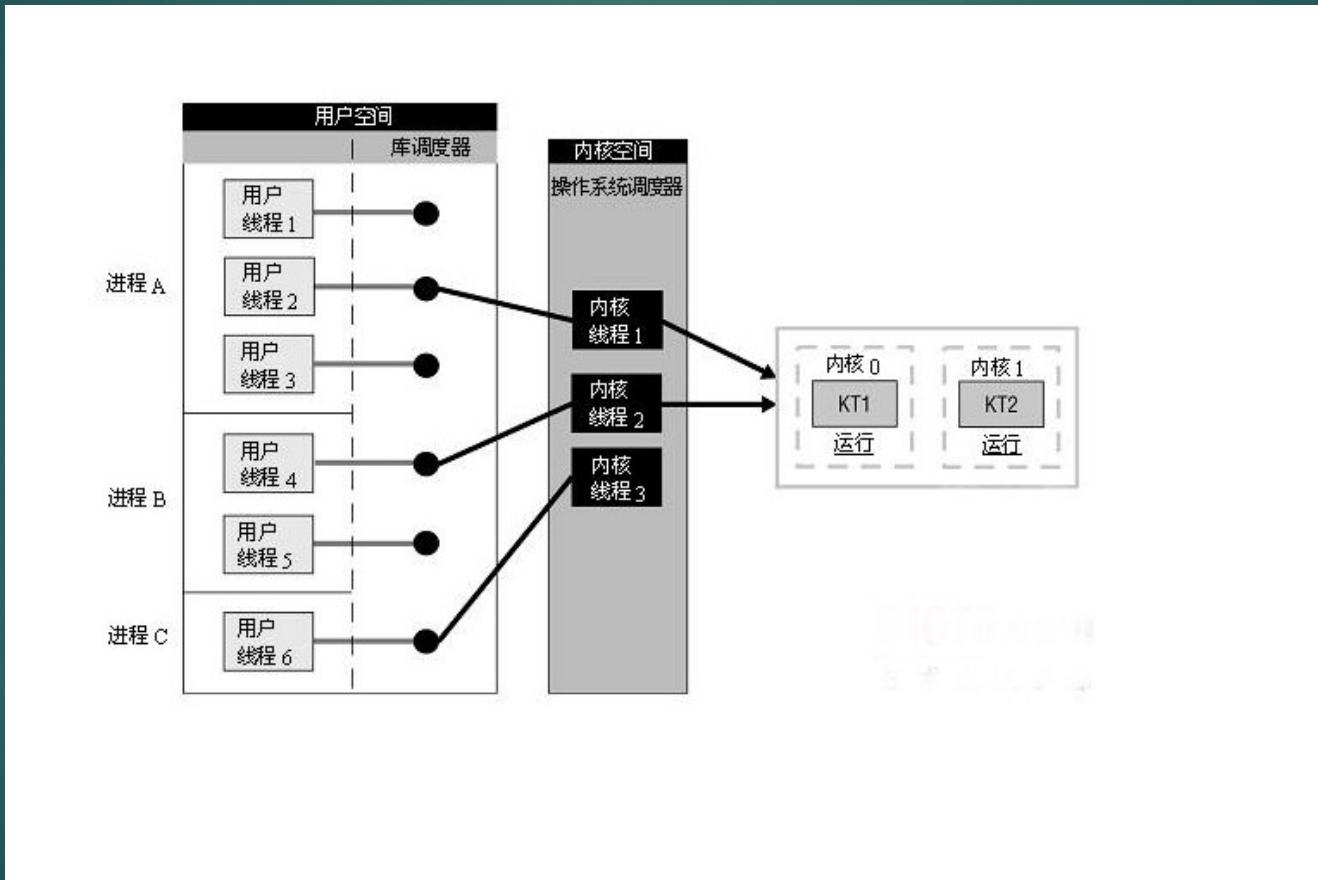
- ▶ pthread_create:

```
const int clone_flags = (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SYSVSEM  
| CLONE_SIGHAND | CLONE_THREAD  
| CLONE_SETTLS | CLONE_PARENT_SETTID  
| CLONE_CHILD_CLEARTID  
| 0);  
  
TLS_DEFINE_INIT_TP (tp, pd);  
  
if (__glibc_unlikely (ARCH_CLONE (&start_thread, STACK_VARIABLES_ARGS,  
clone_flags, pd, &pd->tid, tp, &pd->tid)  
== -1))  
    return errno;
```

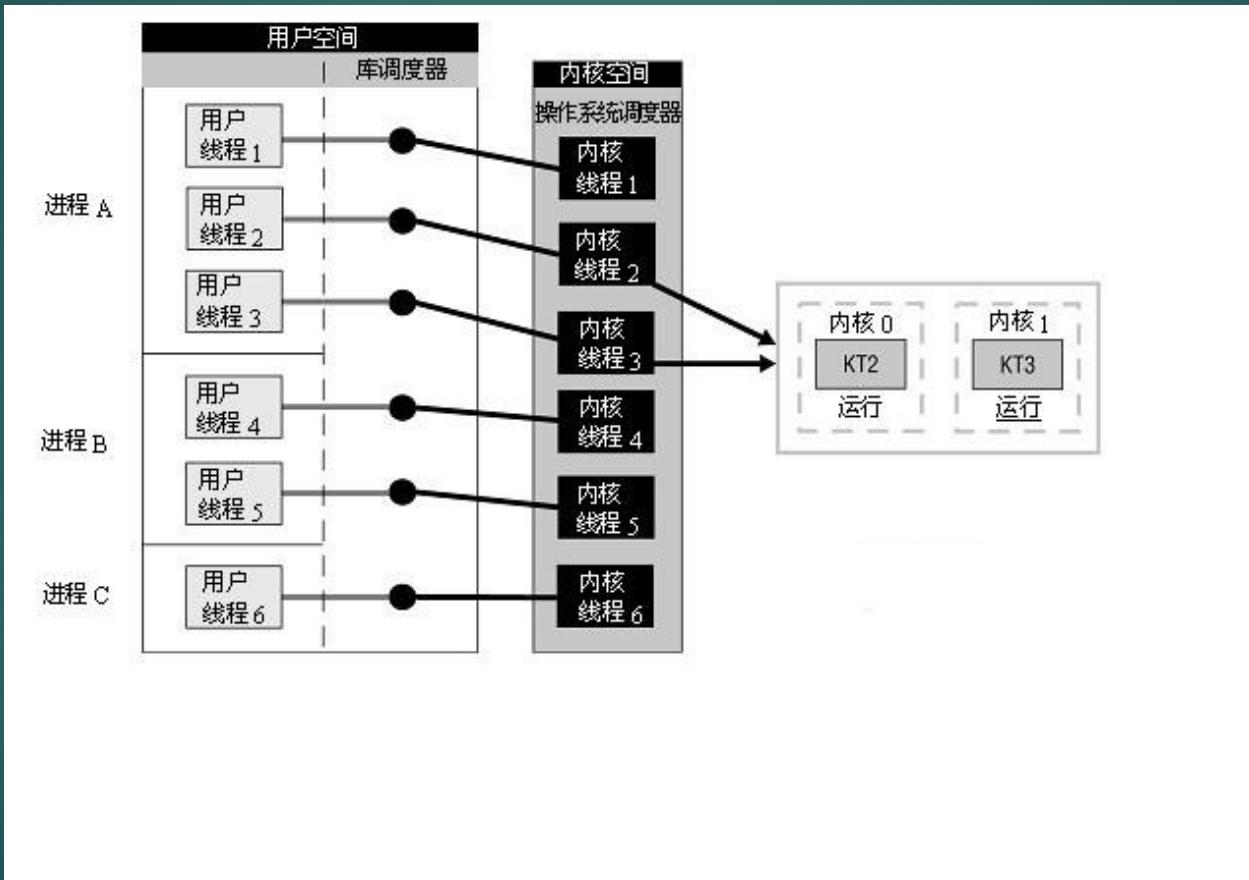
用户级线程和内核级线程

- ▶ 用户级线程：由应用程序实现的线程，内核无感知。
- ▶ 内核级线程：由内核实现的线程，受内核调度。

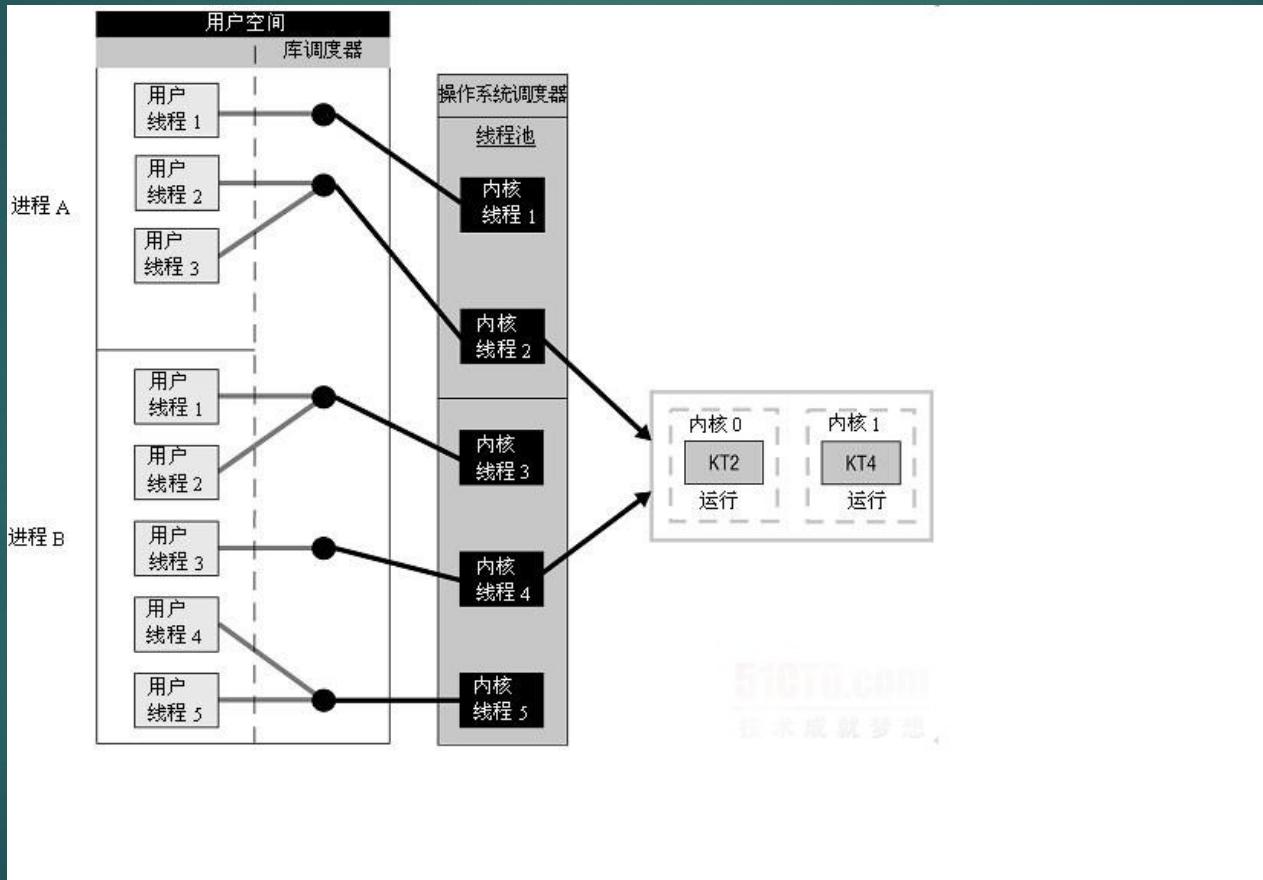
多对一模型



一对一模型



多对多模型



Go调度模型 —— GMP

- ▶ G (goroutine) : 协程
 - ▶ main + go
 - ▶ runtime.main
 - ▶ go func() {}

```
type g struct {
    stack      stack      // 栈空间: hi <-> lo
    stackguard0 uintptr   // 检验是否需要扩展栈, g 使用
    stackguard1 uintptr   // 检验是否需要扩展栈, g0 使用
    m          *m         // 对应的 m
    sched      gobuf     // 上下文: pc、sp 等
    goid      int64
    preempt    bool       // 是否被抢占中
}
```

```
type gobuf struct {
    sp  uintptr
    pc  uintptr
    g   guintptr
    bp  uintptr // for GOEXPERIMENT=framepointer
}
```

Go调度模型 —— GMP

- ▶ M (machine) : 系统线程
 - ▶ 所有的go程序都要在M上执行
 - ▶ goroutine (需要P)
 - ▶ syscall (同步需要P, 异步不需要)
 - ▶ sysmon (不需要P)

```
type m struct {
    g0          *g           // 调度栈
    mstartfn   func()       // 启动函数
    curg        *g           // 当前正在执行的 g
    p           puintptr     // 绑定的 p
    nextp      puintptr     // 唤醒 m 时, 会绑定该 p
    park        note         // m 休眠时使用的型号量
}
```

Go调度模型 —— GMP

- ▶ P (process) : 系统资源
 - ▶ P 控制程序的真实并发度
 - ▶ 默认创建 CPU 核心数个 P
 - ▶ runtime.GOMAXPROCS 可以调整 P 的数目

```
type p struct {  
    lock mutex          // 当其他 P 来偷 g 时，需要先抢占这个锁  
    status   uint32       // P 的状态  
    m       muintptr     // back-link to associated m (nil if idle)  
    runqhead uint32      // 本地 g 队列队列头编号  
    runqtail uint32      // 本地 g 队列队列尾编号  
    runq    [256]guintptr // 本地 g 队列  
}
```

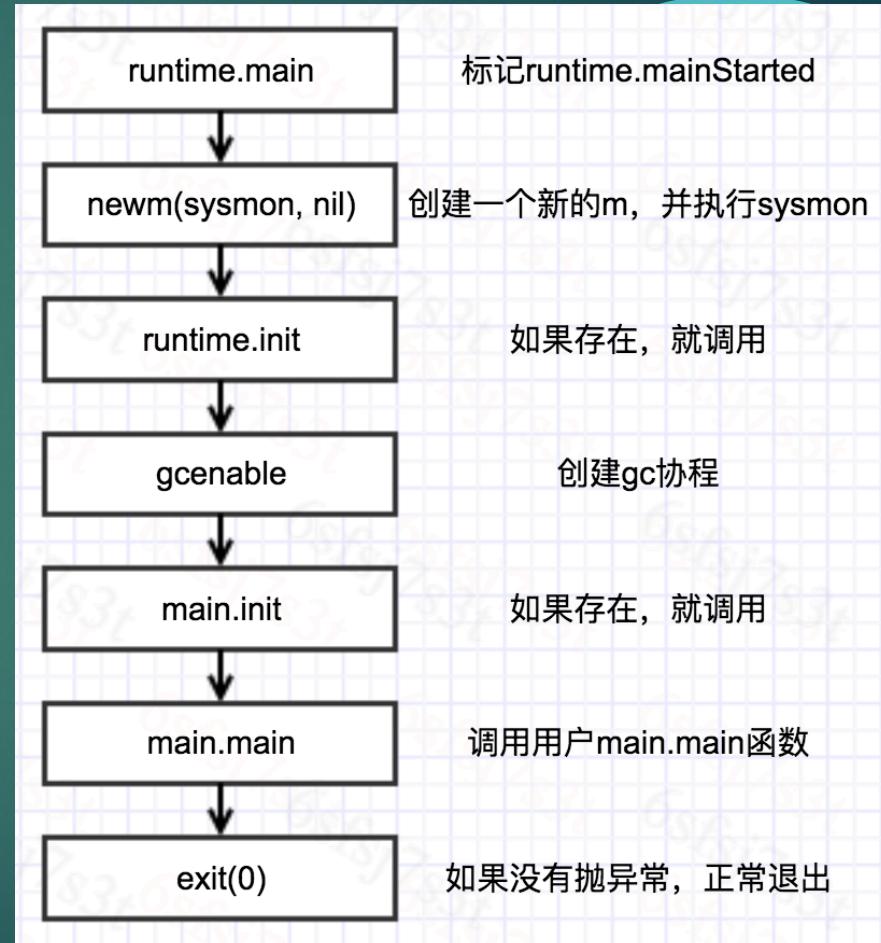
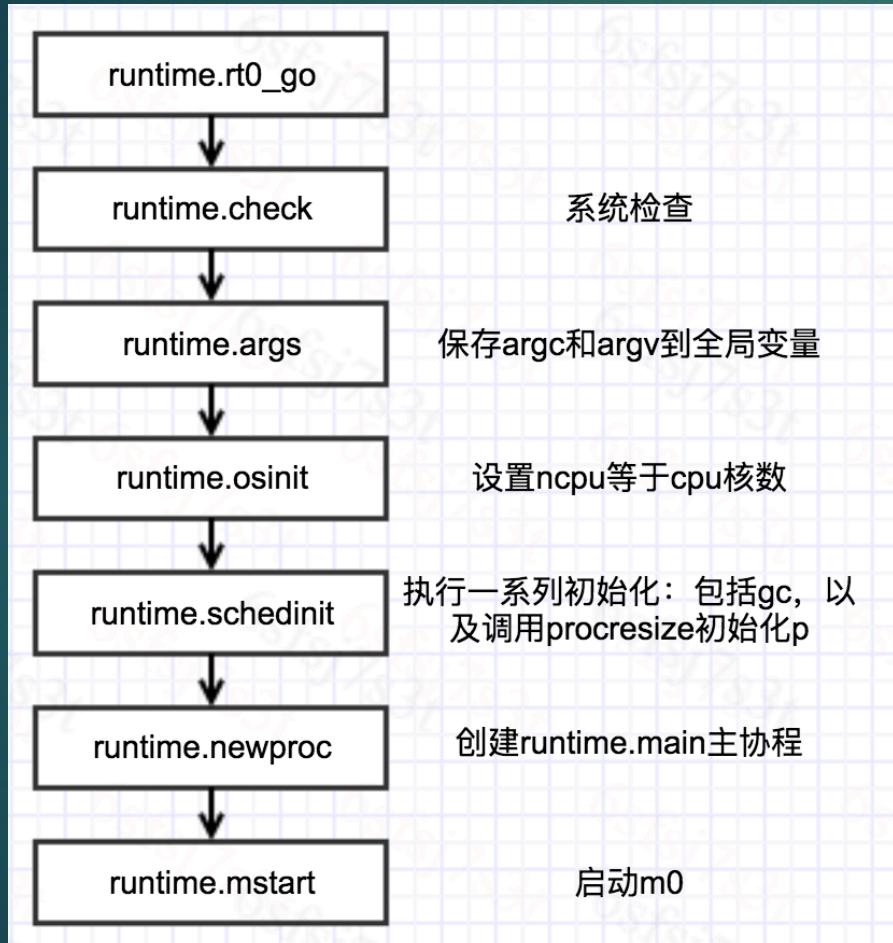
Go调度模型 —— 全局调度器

► Schedt

- 全局只有一个实例
- 全局管理 G、M 和 P

```
type schedt struct {  
    lock mutex          // 全局锁  
    midle   muintptr    // idle 状态的 m  
    nidle   int32        // idle 状态的 m 的数量  
    nmidlelocked int32  // locked 状态的 m 的数量  
    mnext   int64        // 已经创建的 m 的总数  
    maxmcount int32      // m 允许的最大数  
    pidle   puintptr     // 空闲 p 列表  
    npidle  uint32       // 空闲 p 的数目  
    nmspinning uint32    // 处于自旋的 m 的数量  
    runqhead guintptr   // 全局队列队列头编号  
    runqtail guintptr   // 全局队列队列尾编号  
    runqsize int32       // 全局队列大小  
    freem   *m           // 空闲的 m 列表  
}
```

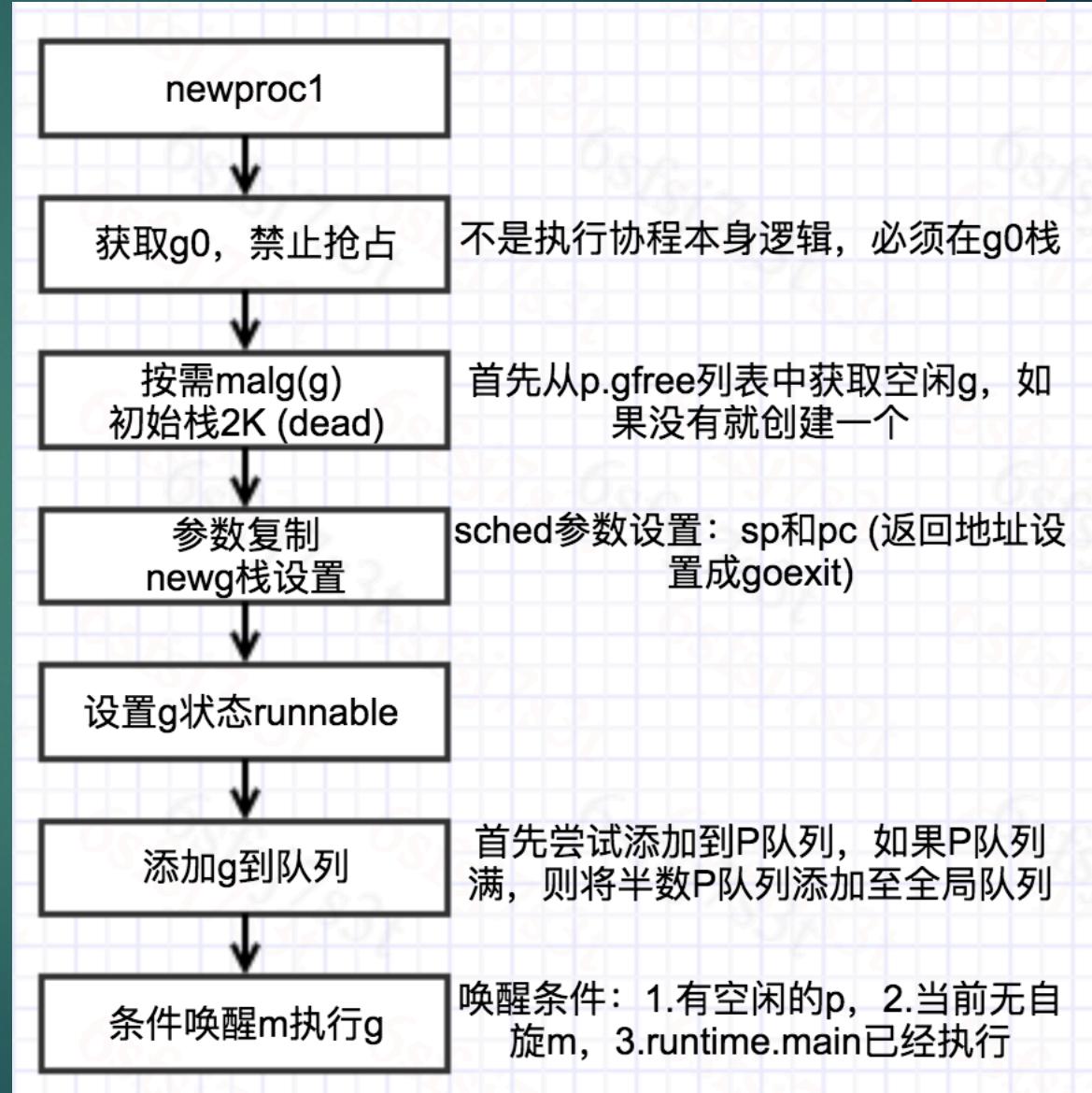
启动程序 rt0_go



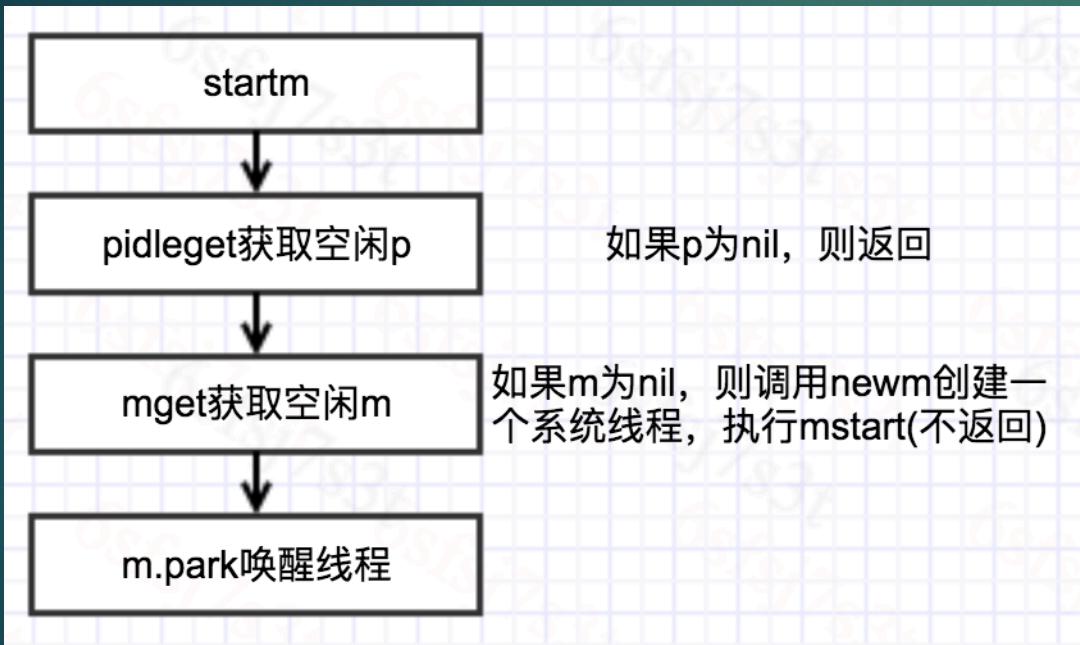
创建协程 newproc

```
func main() {
    go func() {
        fmt.Println("Hello, World")
    }()
}
```

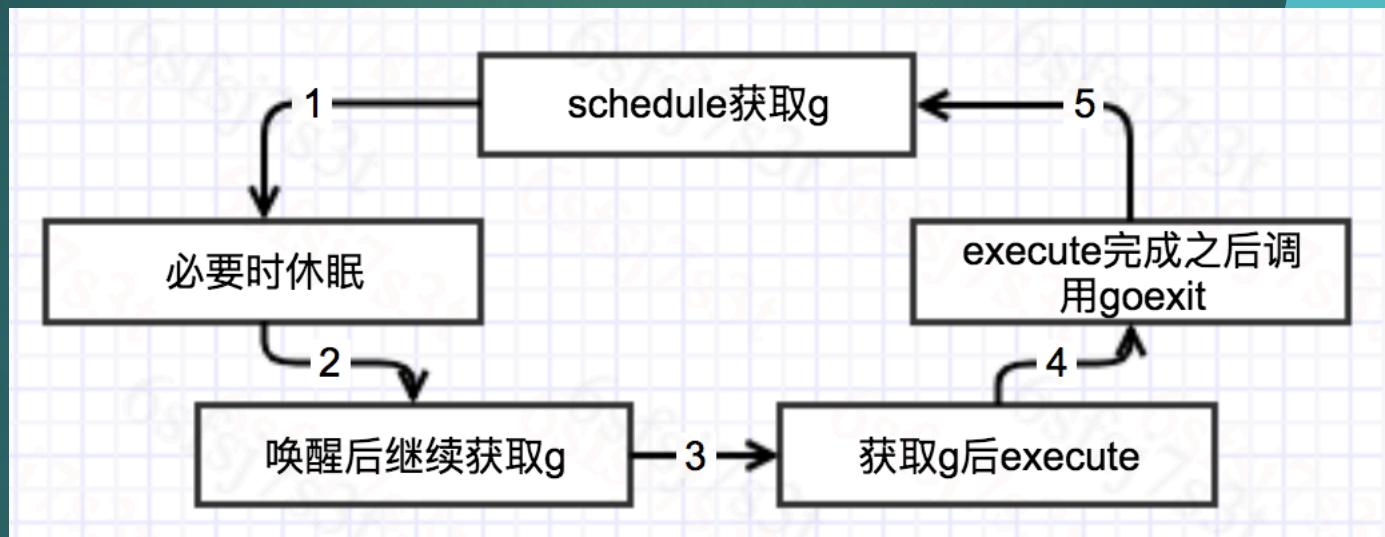
```
func newproc(siz int32, fn *funcval) {
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    pc := getcallerpc()
    systemstack(func() {
        newproc1(fn, (*uint8)(argp), siz, pc)
    })
}
```



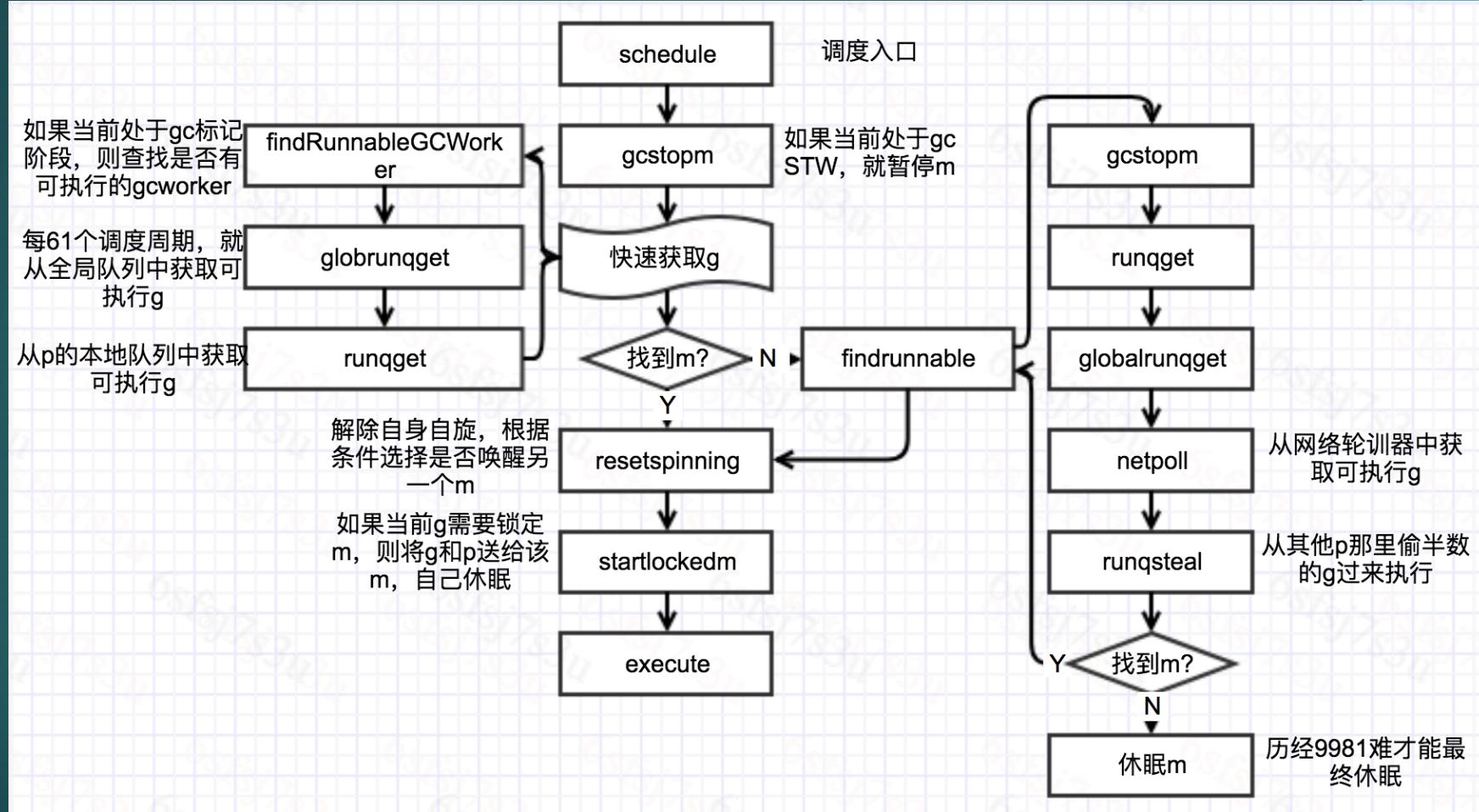
创建线程 mstart



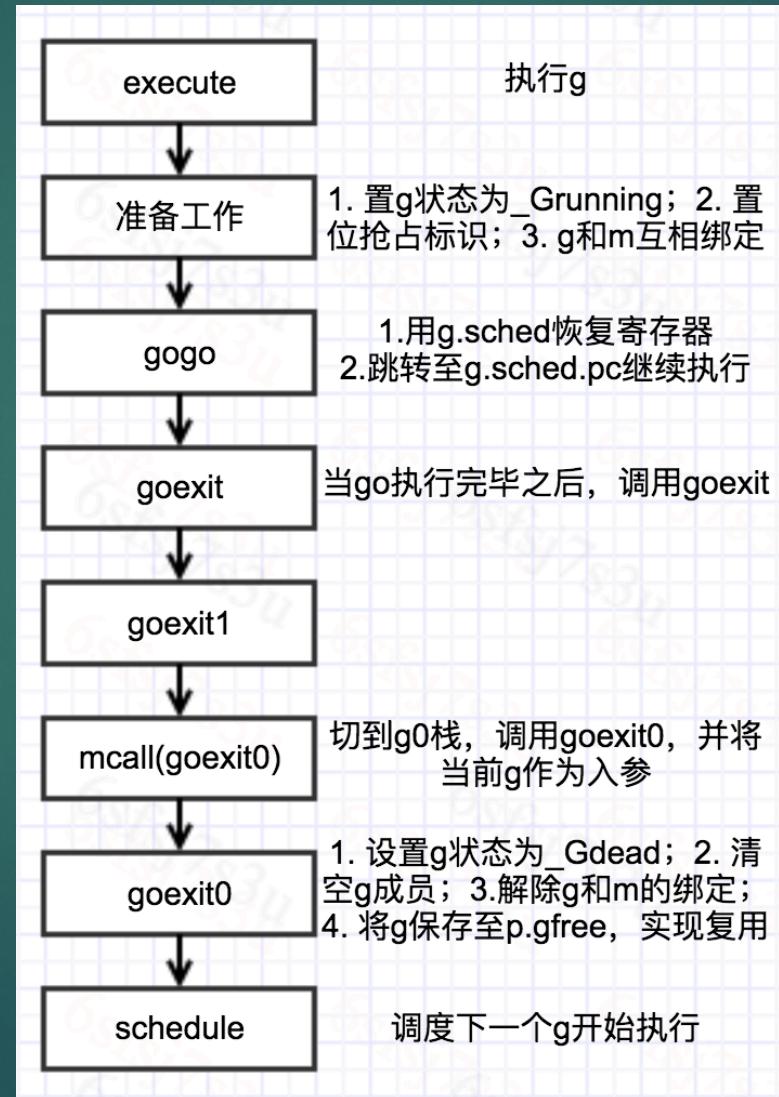
调度循环 schedule



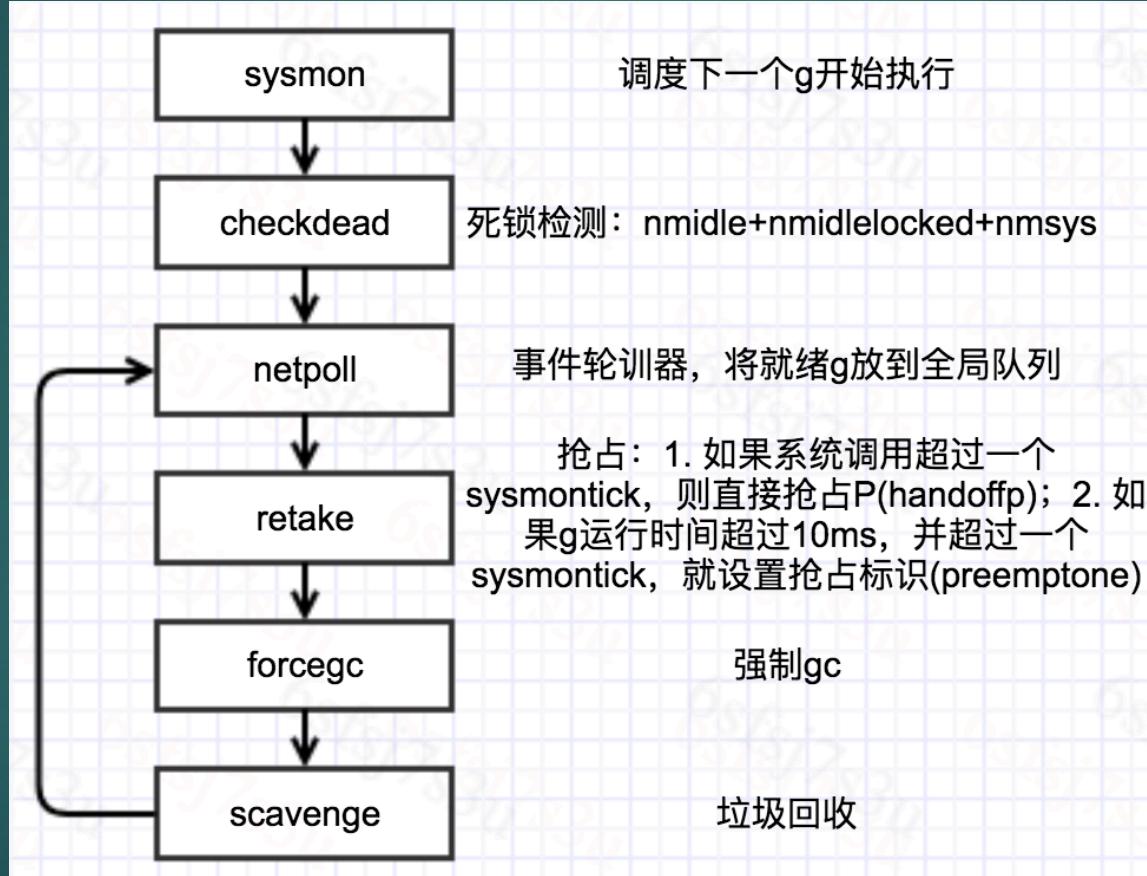
调度流程 schedule



执行流程 execute



后台流程 sysmon



抢占流程 retake

