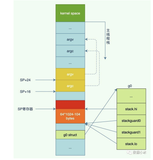
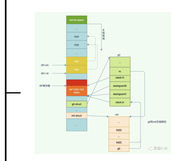
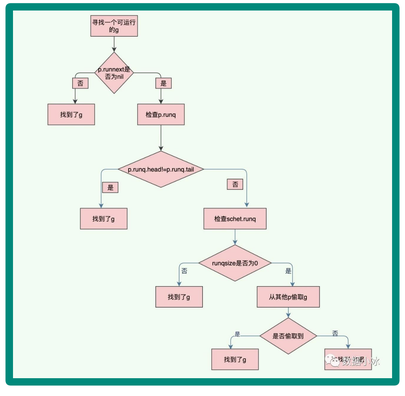
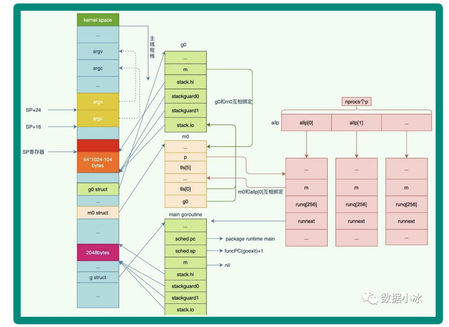
go语言输出helloword过程

- 将程序编译成二进制文件，加载到内存
  - 程序的入口src/runtime/rt0_darwin_amd64.s
    - JMP _rt0_amd64 函数，将传递的参数交给程序使用
      - JMP runtime.rt0_go函数开始初始化
        - 
          分配g0的函数栈帧在对内存上 64kb，初始化stackguard0、stackguard1、stack.lo和stack.hi字段设置初始值，这几个字段与栈的位置，以及栈扩容相关；
        - 
          设置tls本地线程，将m0和g0相互绑定
        - 调用check（）
        - 调用args（）
        - 调用osinit（）osinit主要是获取CPU的核数并保持在全局变量ncpu中
        - schedinit（）
          - 设置最大线程数m的数量为10000，m0结构的初始化
          - 调用mcommoninit初始化m0 g0与m0, m0与p的互相绑定 allg和allm全局链表始终指向表头，allp是一个slice
          - 根据cpu的核数或者设置的GOMAXPROCS完成对应数量p对象的创建和初始化
          - 调用procresize（）初始化全部p
            - 计算当前真正p的数量nprocs，初始化保存所有p的全局变量allp,allp为一个切片，它里面保存的对象为*p类型，利用make初始化allp.
            - 循环创建nprocs个p对象并初始化它，并把*p对象保存到全局变量allp切片中.
            - 将m0和allp[0]互相绑定，并将allp[0]状态设置为_Prunning
            - 将allp[1:nprocs]中的p放入到全局变量sched的pidle空闲队列中
          - 完成m0和p(allp[0])的互相绑定,至此，
      - runtime.main
        - 调用newproc（）创建main goroutine 来运行runtime.main的代码
          - 在调用main包中main函数前，会做一些其他工作，主要有：
          - 新建一个线程执行sysmon函数，sysmon的工作是系统后台监控，通过轮询的方式判断是否需要进行垃圾回收和调度抢占
          - 启动gc清扫的goroutine
          - 执行runtime包的初始化，main包以及main包import的所有包的初始化
          - runtime.main会调用我们的main.mian函数，这样我们的用户代码就被执行到
            - 可以看到runtime.main执行完main.main函数之后，此时函数调用链是schedule()-->execute()-->gogo()-->runtime.main()-->main.main()，直接调用exit系统调用结束进程了，读到这里有同学可能疑问？前面不是说在创建goroutine的时候伪造成g是被goexit函数调用的，按理说执行完g逻辑之后，要返回goexit函数继续执行。这是对main goroutine以外的其他goroutine准备的，对于非main goroutine执行完之后会返回到goexit函数继续执行，而main goroutine执行完直接结束整个进程了，因为main goroutine执行完意味着程序结束了，所以不用再做其他处理了，也就无需再回到goexit函数继续执行。
      - 开启mstart（）循环调度
        - mstart函数设置了g的堆栈保护字段stackguard0和stackguard1之后，直接调用了函数mstart1
        - 调用mstart1（）mstart1会检测当前的g是否是g0,设置工作线程m的备用信号堆栈和信号掩码，如果当前的m是m0，做一些特殊处理，设置系统信号量的处理函数，检测m是否有起始任务函数，如果有就执行它，例如sysmon函数，获取一个p并将它与m进行绑定，
        - 调用schedule()调度
          - 寻找一个可以运行的goroutine
            - 如果当前真正进行垃圾回收(GC)操作，需要暂停所有程序的运行（STW）,调用gcstopm休眠当前的工作线程m
            - 保证公平性，每进行61次调度时优先从全局队列中获取待运行的g
            - 从p的本地队列中获取，优先从p.runnext中获取，没有再从p.runq中获取
            - 调用findrunnable从其他p中偷取g,如果也没有偷取到的话，将当前的m休眠，等待被唤醒
          - 执行获取的g
            - execute切换到选择的要运行的g的栈执行，设置g为运行状态和是否被抢占等信息之后，调用gogo函数从当前g0栈切换到要运行g的栈，真正开始执行用户程序。
            - execute调用gogo从g0栈切换到用户g栈，在刚开始时，队列中只有一个main goroutine，所以第一个被运行的用户g就是main goroutine.执行完gogo后会跳转到runtime.main（main goroutine的function）运行。
            - 调用newproc1函数中会将创建的goroutine放入runq中，遵循一下规则：_p_为g将放入哪个p的本地队列,gp为待放入的g, next为true表示将gp放入到_p_的runnext中，如果_p_的runnext中已经有g,会进行替换，之前的g会放入到_p_的本地队尾。如果next为false,直接将gp放入到本地队列_p_的队尾.如果本地队列满了， 将当前P中前len(p)/2个G批量放入到全局队列中，但是当goroutine首次创建的时候，next的值为true，所以直接放入到runnext中；当p从全局搬运goroutine时，当获取的数量超过sched.runqsiez的一半时，会将剩余的goroutine放回到全局runq，此时的next值为false，因此调用runqput（）直接放入到lrunq的本地队列。
            - 

m获取g的方式
- 从全局队列中获取G
  - globrunqget从全局队列最多获取max个g,并将获取的max-1个g放入_p_的本地队列，并将剩下的1个作为返回值给调用方进行调度。该函数是一个通用函数，也就是一次可以从全局队列中搬多个g到本地队列，进行批量搬运。考虑到公平性，搬运的数量有一定限制，将全局队列中g的数量根据p的数量均分，每个p能从全局队列中最多获取到sched.runqsize/gomaxprocs + 1，证公平性每个p都有机会从全局队列中拿到一部分g，同时搬运的数量不能超过本地队列长度的一半（128）。如果取回来的g数量超过本地队列_p_可以容纳的怎么解决呢？可以看到下的runqput函数中如果本地队列满了又放回全局队列，所以g不会丢失。
- 从P的本地队列中获取G
  - runqget从_p_的本地队列中取走一个g返回。优先从p的runnext中获取，runnext只保存一个g,当runnext没有g时才会从runq中获取。当循环队列的队头与队尾相同时，表明队列中没有可运行的g。否则，从队头获取一个g返回。因为存在其他P偷取g造成同时访问的情况，所以写了两个for()死循环，通过cas手段保证并发安全。
- 从其他P的本地队列中偷取G
  - findrunnable从其他p中偷取可以运行的g,在从其他p偷取g前会当前是否有已经准备好运行的网络协程,如果有返回一个可运行的协程。并通过injectglist函数将其余协程放入全局运行队列等待被调度。如果没有可以运行的网络协程，全局运行队列中也没有可运行的g，当前也没有垃圾回收中进行标记的g需要运行，这时会调用runqsteal从其他p中的本地队列中偷取它里面一半数量的g到当前的本地队列