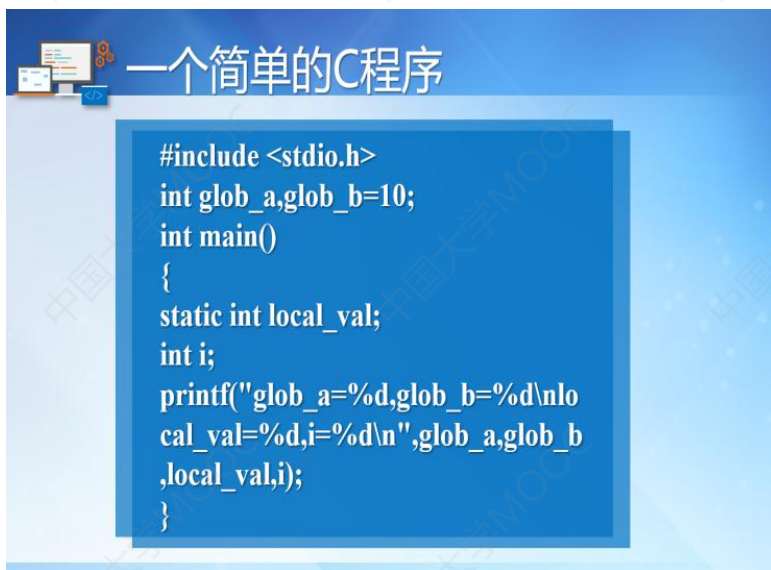


## 进程引入导读宝典

在前面的章节中我们多次提到过进程，这一讲，我们就来学习进程



首先，我们来看对于这样一个简单的 C 程序，要让它执行，需要经过以下步骤。



首先通过汇编器进行汇编，然后通过编译器进行编译，再通过链接器形成可执行文件，最后通过装载器，加载到内存交给操作系统来执行。

**问题 1**，从程序到到进程的距离有多远？在 DOS 下有进程这个概念么？

## 从程序到进程

在Linux环境下对应的步骤：

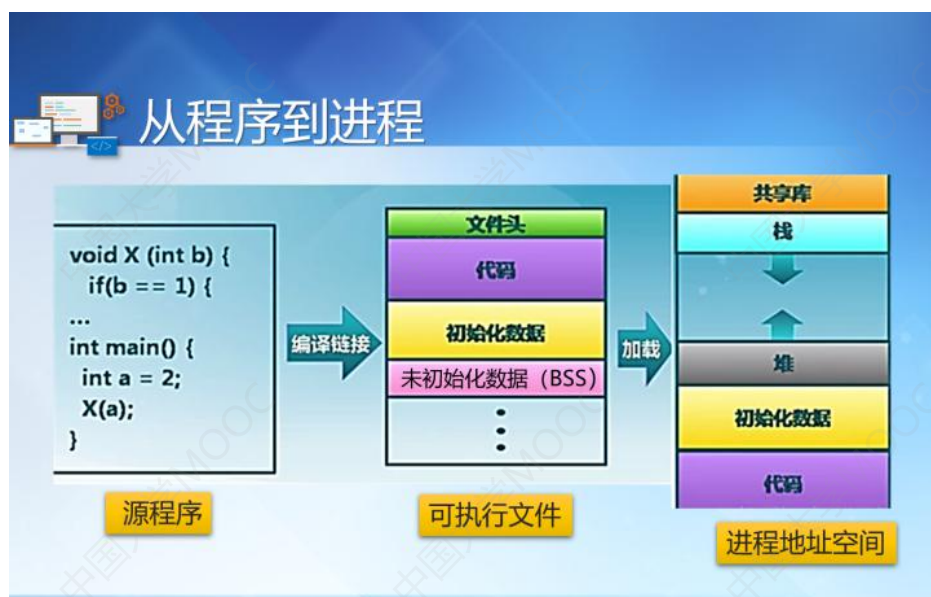
1. `gcc -S hello.c -o hello.s` // 汇编
2. `gcc -c hello.s -o hell.o` // 编译
3. `gcc hello.c -o hello` // 链接
4. `./hello` // 装载并执行

`objdump -d hello` //反汇编

这一系列的过程，在 Linux 环境下可以通过 `gcc` 命令，分别加上不同的参数，就可以形成汇编程序，目标文件和可执行文件，如果想看可执行文件的本来面目，可以通过 `objdump` 命令进行反汇编。在这里我们可以看到，交给操作系统执行的程序不是源程序了，而是二进制的可执行文件，在 Linux 下这种文件格式叫 ELF，在 Windows 下叫 `exe`。这种可执行文件交给操作系统执行后就变为进程了。

在这里我们说一个程序执行后就变为进程了，这个说法是不严谨的，实际上，操作系统引入进程这个概念绝不是为了程序的执行，比如 DOS 操作系统也能执行程序（你写的 C 代码就在 DOS 下执行），可是并不需要进程的概念，为什么？因为系统只有一个程序执行，就像你一个人占了一间屋子，资源都给你了，不需要太多的管理，可是现在屋子里有 10 个人了，问题就麻烦多了，比如，就需要有一个负责人，搞一张表把大家的出入，资源使用情况等都登记下来，给每个人也统一起一个名称叫房客，也编一个号，房客 1，房客 2 如此等等。

**问题 2：**交给操作系统执行的程序为什么不是源程序，而是可执行文件？



Linux 下可执行文件是什么格式呢？它由四部分组成，分别为可执行文件头，代码区，初始化数据区，未初始化数据区等。当可执行文件加载到内存后，程序就摇身一变成为进程了，

我们看进程的地址空间中，栈中存放函数的参数、返回值、局部变量等，而堆是用来为用户程序中的 `malloc()` 等分配内存的，共享库中则存放像 `printf` 这样的库函数。

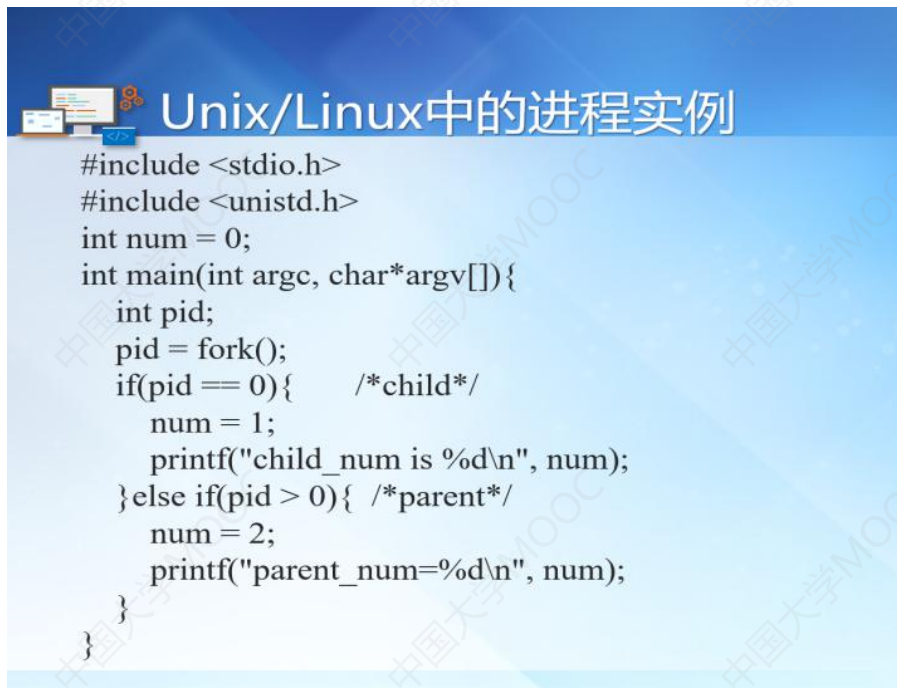
问题 3：不管是 Windows 还是 Linux 都有一种可执行文件格式，这个格式其实就是一种数据结构，也就是不管什么程序来了，是一行的“hello world”简单程序，还是几万行的数据库程序，都按可执行文件的格式往里套，就像服务员都穿上统一的衣服一样，问题是，为什么要这么做？这样做是否意味着操作系统对所有的程序都一视同仁了？

```
struct exec {  
    unsigned long a_midmag; /* 魔数和其它信息 */  
    unsigned long a_text; /* 文本段的长度 */  
    unsigned long a_data; /* 数据段的长度 */  
    unsigned long a_bss; /* BSS 段的长度 */  
    unsigned long a_syms; /* 符号表的长度 */  
    unsigned long a_entry; /* 程序进入点 */  
    unsigned long a_trsize; /* 文本重定位表的长度 */  
    unsigned long a_drsize; /* 数据重定位表的长度 */  
};
```

这是可执行文件 `a.out` 的标准格式的结构。

如何切实的感知进程呢？在 window 下可以通过任务管理器查看进程的动态变化过程，可以看到，一个程序，每执行一次就是一个进程，也就是说程序与进程不是一一对应关系，一个程序可以对应多个进程。在 Linux 下，可以通过 `top` 命令查看进程。我们可以看到进程的 `pid`，占用 CPU 的时间，优先级，进程的状态，占用的内存等等，这些就是进程的属性信息。

问题 4：不管是 Linux 的 `top` 命令还是 Windows 下的任务管理器，系统中多进程的特征是如何体现的，是进程 `PID`，占用 CPU 的时间，内存的大小么？还有哪些属性么，请运行 `TOP` 命令，并仔细分析。



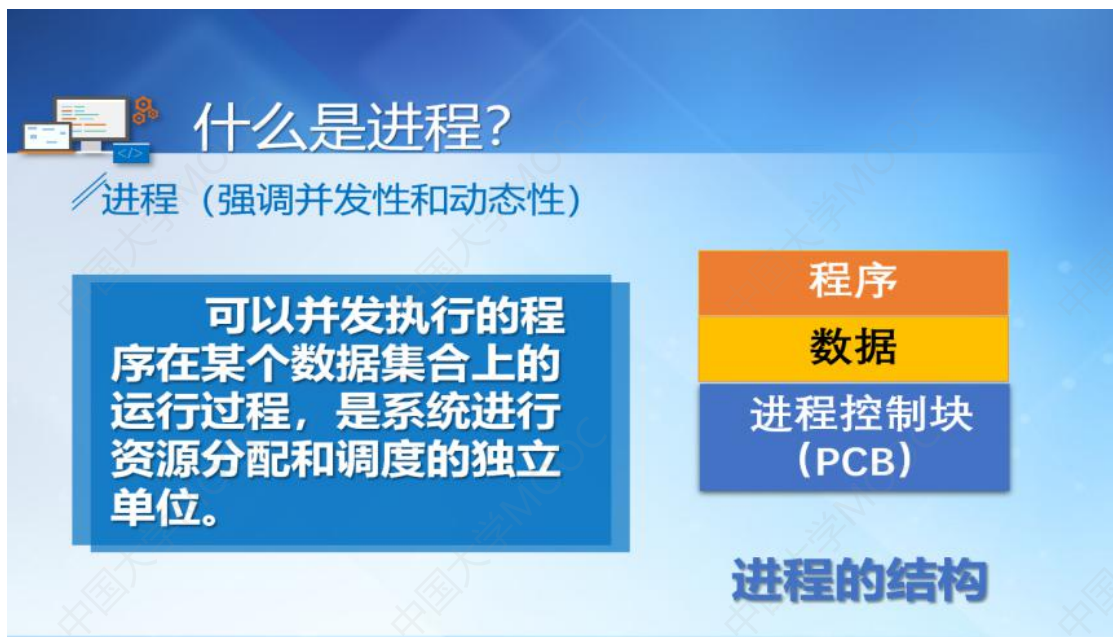
```
#include <stdio.h>  
#include <unistd.h>  
int num = 0;  
int main(int argc, char*argv[]){  
    int pid;  
    pid = fork();  
    if(pid == 0){ /*child*/  
        num = 1;  
        printf("child_num is %d\n", num);  
    }else if(pid > 0){ /*parent*/  
        num = 2;  
        printf("parent_num=%d\n", num);  
    }  
}
```

在这里，我们调用 `fork()` 创建一个新进程，`fork` 本身就是分叉的意思，当执行 `fork()` 以后，一个新的进程就诞生了，也就是说父子进程同时存在了，执行流一分为二，，在父进程



中 `fork()` 返回子进程的 `pid`，在子进程中 `fork()` 返回 0。

问题 5：这个程序的执行结果会是什么样呢？父子进程都得到了执行么？请运行该程序，并切实感受一个新进程的诞生



那么，什么是进程，简单的说，进程是程序的一次执行，稍微严格的说，进程是可以并发执行的程序在某个数据集合上的运行过程，是系统进行资源分配和调度的基本单位。进程由三部分组成，程序，数据和进程控制块（简称 PCB），什么是进程控制块？PCB 是操作系统对进程进行描述的一种数据结构，存放了进程方方面面的信息，下一讲会进行详细介绍。

（在这里，进程的引入显得唐突，于是观看哈工大李老师视频 <https://www.bilibili.com/video/av17036347?p=15>）

问题 6：李老师为什么说（7'58''）创建一个进程，启动 shell 你就可以使用计算机了？



进程有哪些特征，首先是它的结构性，它由三部分组成，其次是它的动态性，由创建到死亡是一个过程，像人的生命一样，你可以把自己想象成进程，然后是它的并发性，就是说多个进程可以并发执行，而独立性，表现为它可以单独的获得资源，也可以单独的被调度，最后是它的异步性，就是说各个进程独立运行，在系统中“走走停停”，运行顺序是不确定的，

并不是谁先运行谁就先结束，与很多因素有关。

问题 7：如果把你与进程类比的话，有哪些类似的地方？

程序与进程的区别	
程序	进程
静态的	动态的
可长期保存	有生命周期
一个程序对应多个进程	一个进程可包含多个程序
代码+数据	代码+数据+进程控制块

程序和进程有何区别？可以类比为歌谱和唱歌，一个程序执行以后就成为进程，前者为静态的，后者为动态的。有执行就有结束，因此它有生命周期，同一个程序，每执行一次就是一个新的进程，这是一对多的关系，而一个进程中，可以包含多个程序。一个程序包含代码以及输入输出的数据，而进程是操作系统要管理的对象，除了程序和数据，OS 还需要通过进程控制块对进程进行管理。

问题 8：程序与进程的区别，你还可以想出什么例子来？



P11:如何描述进程的动态变化过程，我们可以根据运动员参加比赛的过程来理解。首先运动员要在起跑线上做好预备动作。裁判员一声枪响，运动员开跑。这是一场 110 米栏比赛，一个运动员不小心摔倒了，无法继续进行比赛，先暂停下来。这三个场景分别对应了进程的三种基本状态。预备动作对应就绪态，万事俱备，只欠发令枪声。开跑对应运行态，拼命往前冲。摔倒对应阻塞态（或睡眠态），暂时跑不了了。这就是进程的三种基本状态。



当一个进程刚创建时，它处于就绪态，图中大家在就绪队列中排队。就绪态的进程一切具备，只差 CPU，一旦被调度程序选中，就从就绪态转换为运行态，到 CPU 上运行去了。

执行过程中分给它的时间用完了，就从运行态转换为就绪态。

在它运行的过程中，如果遇到输入输出请求，比如从键盘输入数据，或是要向显示器输出数据，它就从运行态转换为阻塞态，在等待队列中排队。所以，请注意，阻塞态的上一个状态一定是运行态，阻塞态的进程都是在运行的过程中被阻塞的。

当进程接收到了输入数据或输出完毕，就从阻塞态转换为就绪态，进入就绪队列。通过以上几种状态以及转换，可以感受进程的动态特性。

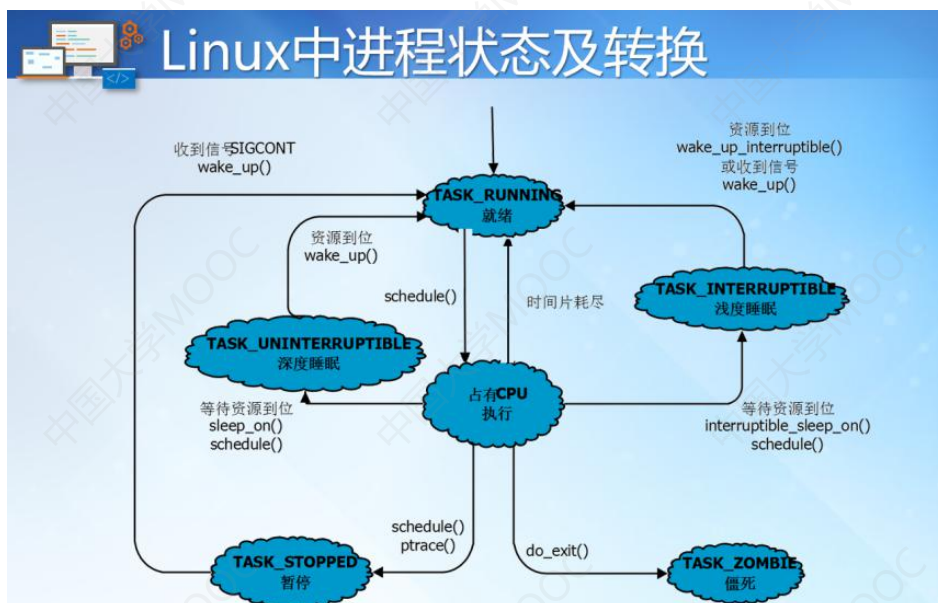
**问题 9：**为什么要用状态来描述进程？



上一张图，如果更清晰的表达出来，就是这张图，一个进程从创建到就绪，再到运行，中间可能经过等待，最后结束退出，整个是一个生命周期。

**问题:10：**当你失恋了就会从高兴转换为忧伤状态，进程状态之间的转换都是什么原因？





前面介绍的三种基本状态，是任何一个操作系统中都存在的，但落实到具体操作系统的时候，就不止这三种了，如图 Linux 中睡眠态就有两种，一种叫浅度睡眠，就是收到信号就可以唤醒它（相当于闹钟能把你叫醒），一种是深度睡眠，需要调用唤醒函数才能唤醒它（相当于妈妈才能把你叫醒），当你调试一个程序的时候，进程就进入暂停状态，当进程退出，资源尚未回收时就进入僵死状态，状态之间转换所调用的函数是内核函数。

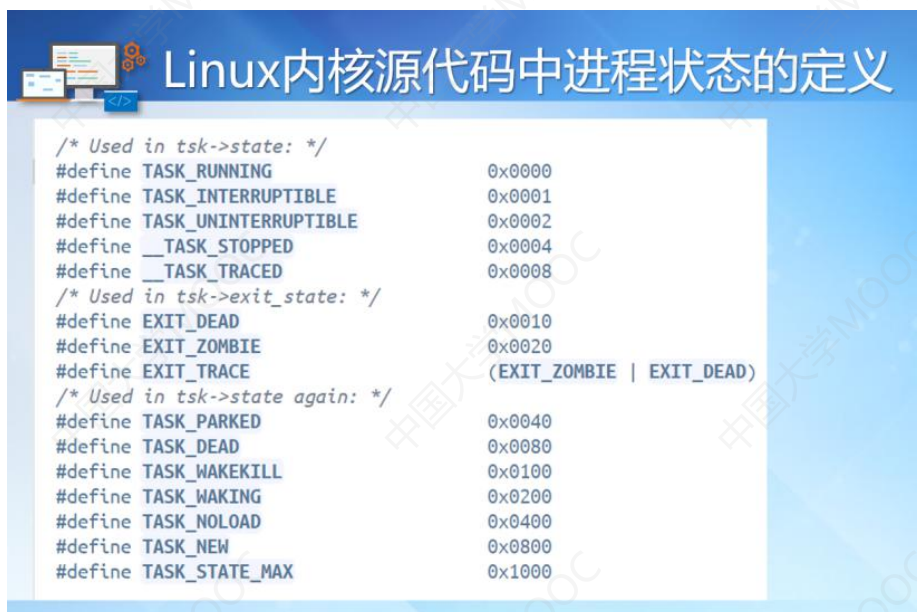
**问题:11:** 操作系统原理讲的进程三种基本状态相当于基类，比如，学生，而具体操作系统的状态相当于子类（比如本科生），Linux 内核中目前有十多种状态了，问题是可以查看进程的状态吗？请用 **top** 命令查看。

## Linux中的进程状态

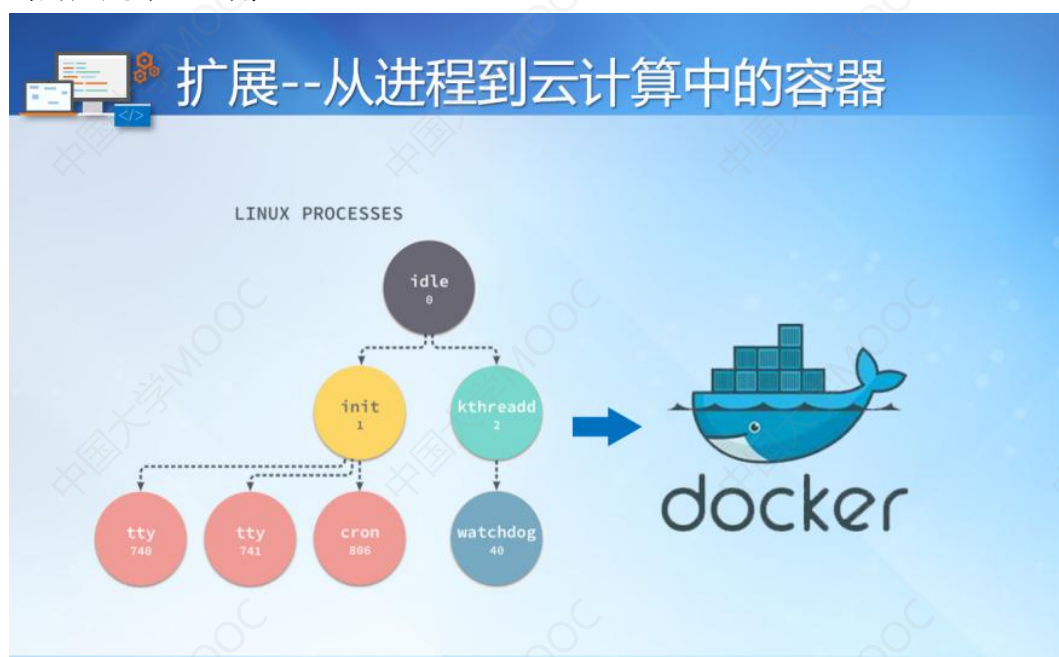
进程状态 (在ps或top命令中看到的状态)	状态编码 (在内核代码中定义的值)	状态的含义
R (running)	0	运行或将要运行
S (sleeping)	1	被中断而等待一个事件，可能会被一个信号激活
D (deep sleep)	2	被中断而等待一个事件，不会被信号激活
T (stopped)	4	由于任务的控制或者外部追踪而被终止
t (tracing stop)	8	
Z (zombie stop)	16	僵死，但是它的父进程尚未调用wait函数
X (dead)	32	死亡状态，这个状态永远也看不见

通过 **top** 命令我们会看到，在单 CPU 的机器上，只有一个进程处于 R 运行状态，一般情况下，进程列表中的绝大多数进程都处于浅度睡眠 **TASK\_INTERRUPTIBLE** 状态，也就是 S 状态。僵死状态 Z，就是子进程已经结束，但父进程还没有回收它的资源。

**问题 12,** 类比一下我们人从出生到死亡过程的状态，你有何启发？



P16:这里给出 Linux 内核 5.3 版本中状态的定义，可以看出，有十多个状态，每个状态的值是 2 的 n 次方，这种定义方式巧妙之处何在？一是，进程状态的个数是可扩充的，当一个新的状态增加时，只需要增加一个宏定义就可以，二是通过逻辑与运算，就可以快速算出一个进程的状态，如何在源代码中查看，请进入 Linux 源代码 <https://elixir.bootlin.com/> 网站，输入“TASK\_RUNNING”这个标识符，就可以在 sched.h 的第 76 行~92 行看到这些定义，务必亲自查看，否则，你在这里看到的只是冰山一角。



容器作为目前云技术的核心技术，与进程到底有多大关系？

对于进程来说，它的静态表现就是程序，平常都安安静静地待在磁盘上；而一旦运行起来，它就变成了计算机里的数据和状态的总和，这就是它的动态表现。

而容器技术的核心功能，就是通过约束和修改进程的动态表现，从而为其创造出一个“边界”。对于 Docker 等大多数 Linux 容器来说，Cgroups 技术是用来制造约束的主要手段，什么意思，就是对进程进行分组管理，而 Namespace 技术则是用来修改进程视图的主要方法，



什么意思，就是资源隔离方案。在此我们抛砖引玉，感兴趣的同学可以进一步查阅资料探讨下去。