进程

为什么需要进程

早期计算机面临的问题

冯诺依曼计算机还有一个特点是按地址访问并**顺序执行指令**。早期的计算机严格遵循这个特点，使用的是**单道批处理**的操作系统，即一个只进行一个程序，由它支配系统所有资源，资源利用率和系统吞吐率极低。后来出现了**多道程序设计技术**，造就了**多道批处理**的操作系统。这种操作系统在内存中同时加载多道程序，它们在系统中并发执行，共享系统中的各种资源。当其中一道程序因为要等待IO或者访问存储器等而暂停执行后，CPU就去执行其他程序。多道批处理系统提高了资源利用率和系统吞吐率。但是多道批处理系统因为程序共享系统的资源，所以程序之间相互限制，经常发生冲突。

进程是怎样解决这些问题的

我们一次次的改进计算机的根本目的可以这么说，为了尽可能快地解决尽可能多的复杂问题，简化一下就是一个字——“**快**”。以上说了那么多问题，都严重阻碍了计算机的运行速度，进程就是为了解决这些问题而诞生的。作为一个程序员，抽象是必备的能力。我们重新抽象下进程这个概念。

想想上面碰到的问题，你如果是一个工程师你怎么解决。既然多道批处理系统的程序共享资源会互相限制而造成冲突。那么我们就给运行的程序套一层壳——进程。使得程序只有通过进程才能申请系统资源。**进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。**

既然CPU运行速度远高于存储器访问速度，那么由操作系统对每一个进程作上下文切换即进程调度。将进程加入到CPU的等待队列，按照一定的调度方式加载进程到CPU执行，如果这道进程要进行IO操作等非常耗时的举动，CPU就触发中断，让这个进程一边等着去，加紧处理下一个进程，CPU一刻都不能闲着。等IO操作完成，再把进程加载回来，继续执行。**在早期面向进程设计的计算机结构中，进程是程序的基本执行实体。** *（在当代面向线程设计的计算机结构中，进程是线程的容器，线程是程序的基本执行实体。线程是什么后面再说）*

以上述说视角都是以CPU为中心，好像满足了CPU一切就都解决了。但进程的出现对程序也是有好处的。操作系统通过**虚拟化（virtualizing）** CPU 来 提供这种假象。 通过让一个进程只运行一个时间片， 然后切换到其他进程， 操作系统 提供了存在多个虚拟 CPU 的假象。 这就是**时分共享（time sharing）** CPU 技术， 允许用户运行多个并发进程。**从概念上说，每个进程拥有它自己的虚拟CPU。**

程序是指令、数据及其组织形式的描述，进程是程序的实体。

进程的状态转移

死掉的程序只是存储器上的数据，活过来的程序就是进程。没错，进程是有生命的。

进程的诞生

- 系统初始化（init）：启动操作系统时，通常会创建若干个进程。
- 正在运行的程序执行了创建进程的系统调用（比如 fork）
- 用户请求创建一个新进程：在许多交互式系统中，输入一个命令或者双击图标就可以启动程序，以上任意一种操作都可以选择开启一个新的进程，在基本的 UNIX 系统中运行 X，新进程将接管启动它的窗口。
- 初始化一个批处理工作

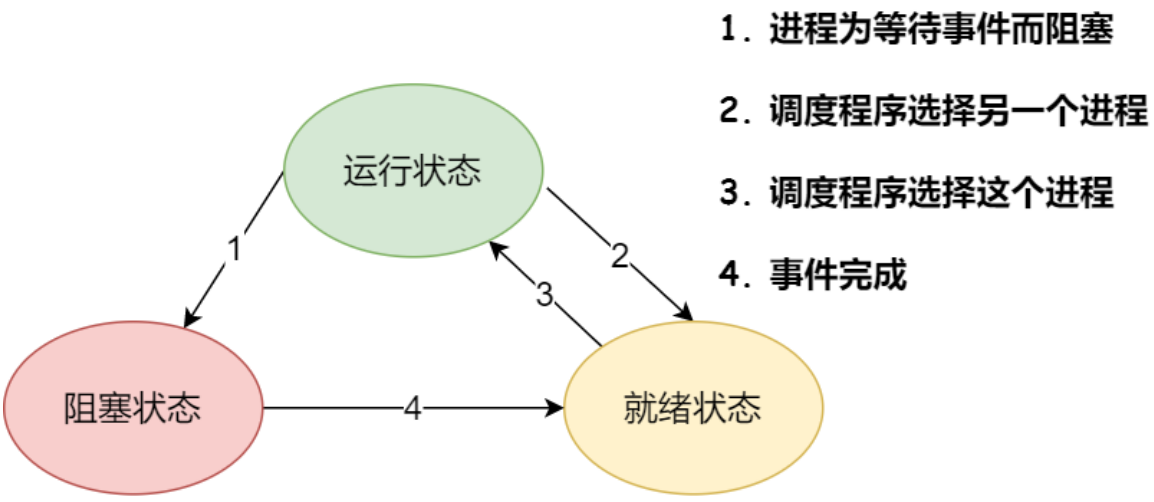
> 在 UNIX 中，仅有一个系统调用来创建一个新的进程，这个系统调用就是 `fork`。这个调用会创建一个与调用进程相关的副本。在 fork 后，一个父进程和子进程会有相同的`内存映像`，相同的环境字符串和相同的打开文件。
>
> 在 Windows 中，情况正相反，一个简单的 Win32 功能调用 `CreateProcess`，会处理流程创建并将正确的程序加载到新的进程中。这个调用会有 10 个参数，包括了需要执行的程序、输入给程序的命令行参数、各种安全属性、有关打开的文件是否继承控制位、优先级信息、进程所需要创建的窗口规格以及指向一个结构的指针，在该结构中新创建进程的信息被返回给调用者。「在 Windows 中，从一开始父进程的地址空间和子进程的地址空间就是不同的」。

进程的死亡

- `正常退出(自愿的)`：多数进程是由于完成了工作而终止。当编译器完成了所给定程序的编译之后，编译器会执行一个系统调用告诉操作系统它完成了工作。这个调用在 UNIX 中是 `exit`，在 Windows 中是 `ExitProcess`。
- `错误退出(自愿的)`：比如执行一条不存在的命令，于是编译器就会提醒并退出。
- `严重错误(非自愿的)`
- `被其他进程杀死(非自愿的)`：某个进程执行系统调用告诉操作系统杀死某个进程。在 UNIX 中，这个系统调用是 kill。在 Win32 中对应的函数是 `TerminateProcess`（注意不是系统调用）。

活着的进程

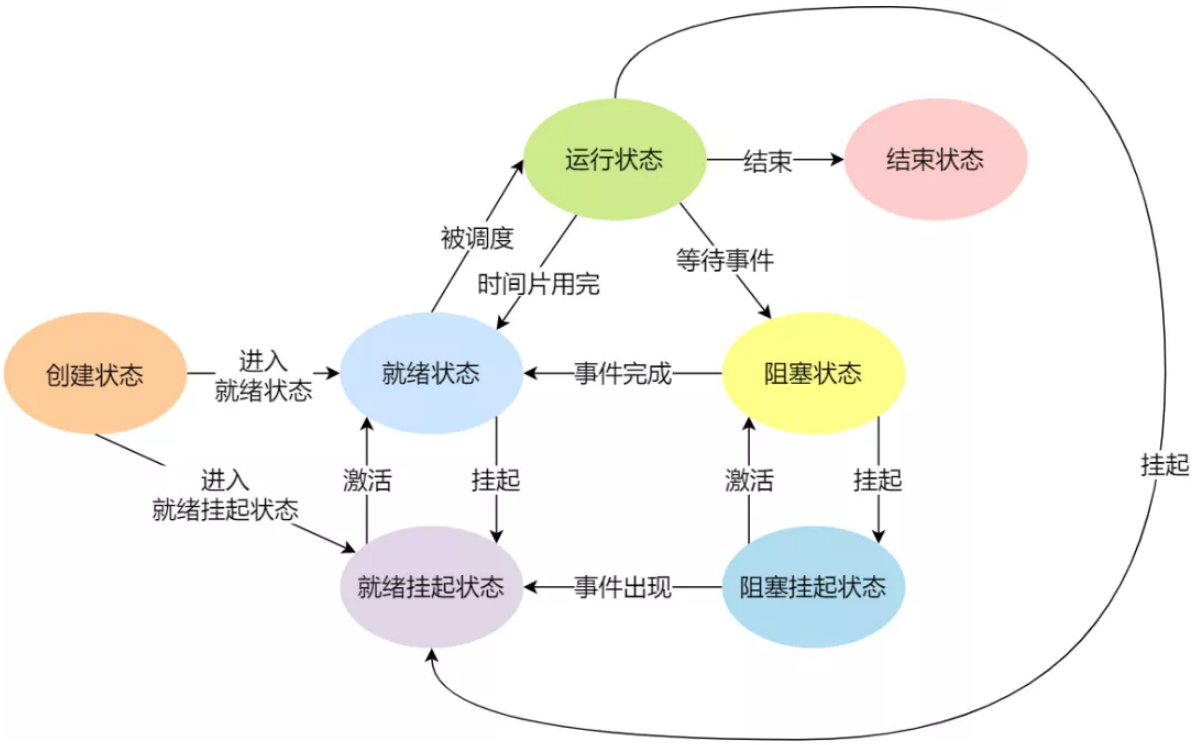
进程其实很苦逼，并不是或者就能一直运行到结束，毕竟资源不够多、CPU算得不够快啊。所以活着的进程除了真正在上CPU上运行还要做一件事——等。作为一个进程，你既要等CPU处理其他进程（就绪状态），也要等资源访问（阻塞状态）。他们之间的状态转移图如图所示。



还有一个状态叫**挂起状态**，它表示进程没有占有物理内存空间。这跟阻塞状态是不一样，阻塞状态是等待某个事件的返回。由于虚拟内存管理原因，进程的所使用的空间可能并没有映射到物理内存，而是在硬盘上，这时进程就会出现挂起状态，另外调用 sleep 也会被挂起。挂起分为阻塞挂起和就绪挂起。挂起状态分为阻塞挂起状态和就绪挂起状态。

进程的状态转移

这两种挂起状态加上前面的三种状态和创建结束两种状态，就变成了七种状态变迁，见如下图：



- 运行状态（*Runing*）：该时刻进程占用 CPU；
- 就绪状态（*Ready*）：可运行，但因为其他进程正在运行而暂停停止；
- 阻塞状态（*Blocked*）：该进程正在等待某一事件发生（如等待输入/输出操作的完成）而暂时停止运行，这时，即使给它CPU控制权，它也无法运行；
- 创建状态（*new*）：进程正在被创建时的状态；
- 结束状态（*Exit*）：进程正在从系统中消失时的状态；
- 阻塞挂起状态：进程在外存（硬盘）并等待某个事件的出现；
- 就绪挂起状态：进程在外存（硬盘），但只要进入内存，即刻立刻运行；

状态转移如下：

- *NULL -> 创建状态*：一个新进程被创建时的第一个状态；
- *创建状态 -> 就绪状态*：当进程被创建完成并初始化后，一切就绪准备运行时，变为就绪状态，这个过程是很快的；
- *就绪态 -> 运行状态*：处于就绪状态的进程被操作系统的进程调度器选中后，就分配给 CPU 正式运行该进程；
- *运行状态 -> 结束状态*：当进程已经运行完成或出错时，会被操作系统作结束状态处理；
- *运行状态 -> 就绪状态*：处于运行状态的进程在运行过程中，由于分配给它的运行时间片用完，操作系统会把该进程变为就绪态，接着从就绪态选中另外一个进程运行；
- *运行状态 -> 阻塞状态*：当进程请求某个事件且必须等待时，例如请求 I/O 事件；
- *阻塞状态 -> 就绪状态*：当进程要等待的事件完成时，它从阻塞状态变到就绪状态；

如何控制进程

进程的控制结构

在操作系统中，是用**进程控制块**（*process control block，PCB*）数据结构来描述进程的。

PCB 是进程存在的唯一标识，这意味着一个进程的存在，必然会有一个 PCB，如果进程消失了，那么 PCB 也会随之消失。

进程管理	存储管理	文件管理
寄存器	text segment 的指针	根目录
程序计数器	data segment 的指针	工作目录
程序状态字	stack segment 的指针	文件描述符
堆栈指针		用户ID
进程状态		组 ID
优先级		
调度参数		
进程ID		
父进程		
进程组		
信号		
进程开始时间		
使用的 CPU 时间		
子进程的 CPU 时间		
下次定时器时间		

- **进程描述信息：**
  - 进程标识符：标识各个进程，每个进程都有一个并且唯一的标识符；
  - 用户标识符：进程归属的用户，用户标识符主要为共享和保护服务；
  - 工作目录等等
- **进程控制和管理信息：**
  - 进程当前状态，如 new、ready、running、waiting 或 blocked 等；
  - 进程优先级：进程抢占 CPU 时的优先级；
- **资源分配清单：**
  - 有关内存地址空间或虚拟地址空间的信息，所打开文件的列表和所使用的 I/O 设备信息。
- **CPU 相关信息：**
  - CPU 中各个寄存器的值，当进程被切换时，CPU 的状态信息都会被保存在相应的 PCB 中，以便进程重新执行时，能从断点处继续执行。

进程的完整结构

从静态角度看，进程实体 = 进程控制块（PCB）+ 程序段 + 数据段。从动态角度看，进程是可并发运行程序在其数据集合上的运行过程。

Linux进程结构：可由四部分组成：PCB、代码段、数据段、堆栈段。

- 进程控制块是进程存在的惟一标识，系统通过PCB的存在而感知进程的存在。
- 代码段存放程序的可执行代码段存放程序的全局变量、常量、静态变量。
- 堆栈段中的堆用于存放动态分配的内存变量段中的栈用于函数调用，它存放着函数的参数、函数内部定义的局部变量。
- 系统通过PCB对进程进行管理和调度。PCB包括创建进程、执行程序、退出进程以及改变进程的优先级等。而进程中的PCB用一个名为 task struct的结构体来表示，定义在include/ linux/sched. h中，每当创建新进程时，便在内存中审请个空的 task struct组构，填入所需信息，同时，指向该结构的指针也被加入到task数组中，所有进程控制块都存储在task数组中。

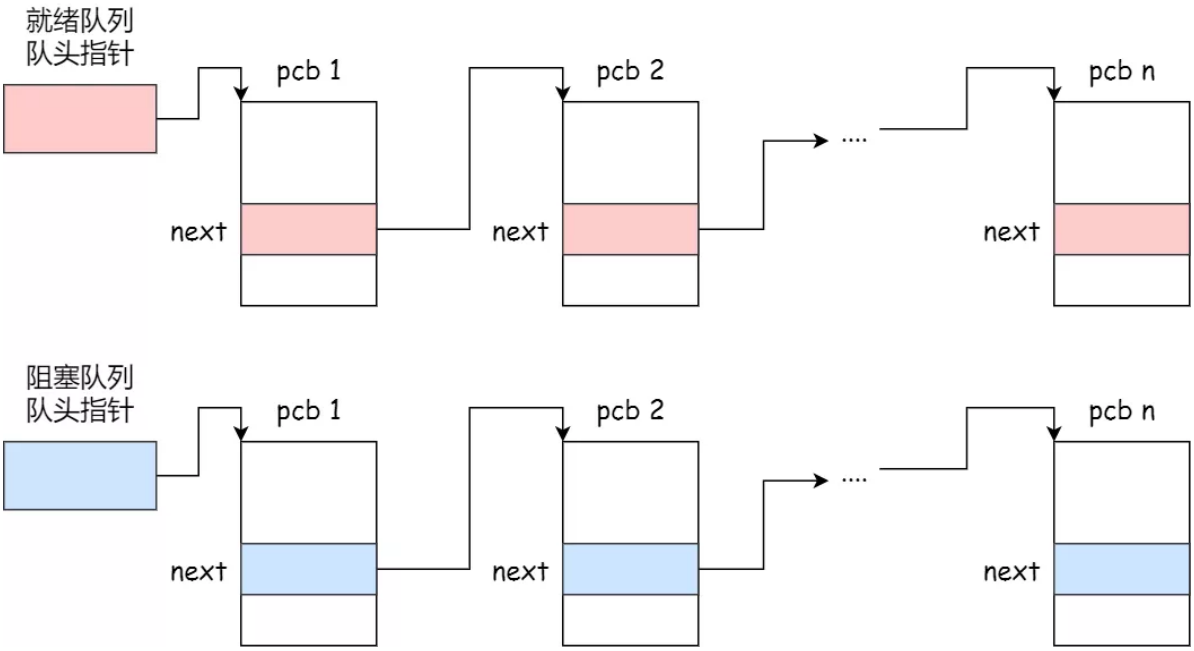
如何组织进程

链表->就绪/运行/阻塞队列

通常是通过**链表**的方式进行组织，把具有**相同状态的进程链在一起，组成各种队列。**比如：

- 将所有处于就绪状态的进程链在一起，称为**就绪队列**；
- 把所有因等待某事件而处于等待状态的进程链在一起就组成各种**阻塞队列**；
- 另外，对于运行队列在单核 CPU 系统中则只有一个运行指针了，因为单核 CPU 在某个时间，只能运行一个程序。

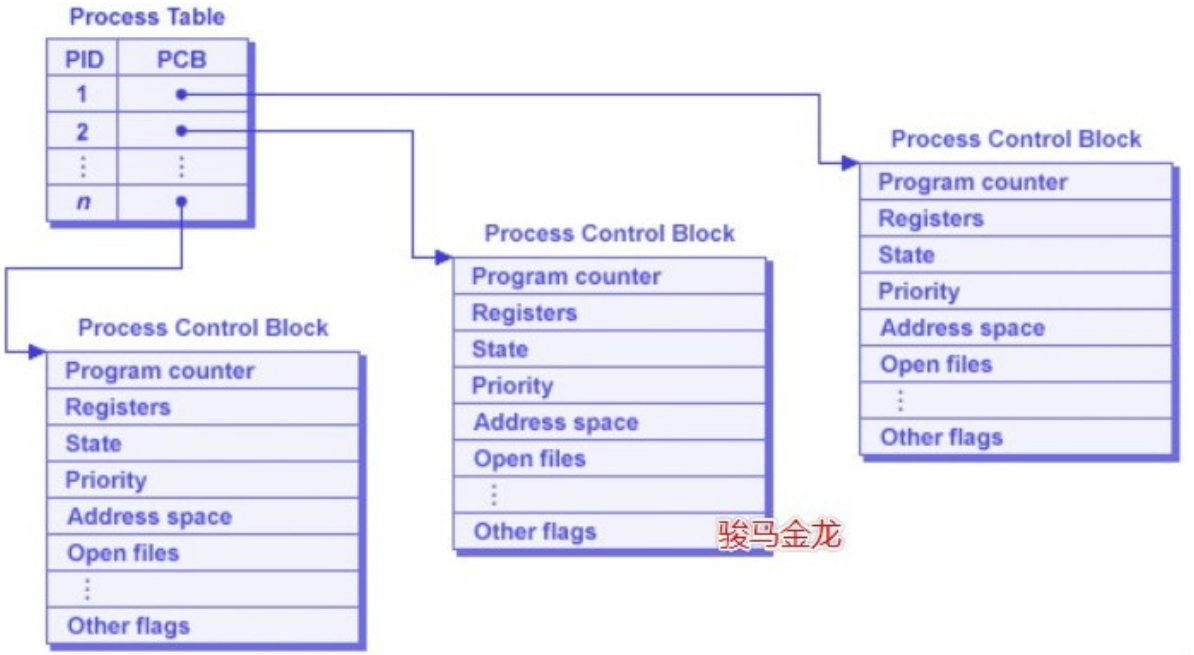
那么，就绪队列和阻塞队列链表的组织形式如下图：



进程表(process table)

除了链接的组织方式，还有索引方式，它的工作原理：将同一状态的进程组织在一个索引表中，索引表项指向相应的 PCB，不同状态对应不同的索引表。

内核负责管理维护所有进程，为了管理进程，内核在内核空间维护了一个称为进程表（Process Table）的数据结构，这个数据结构中记录了所有进程，每个进程在数据结构中都称为一个进程表项（Process Table Entry），如图。



从图中可知，进程表中除了记录了所有进程的PID，还使用一个字段记录了所有进程的指针，指向每个进程的进程控制块（Process Control Block，PCB）。

一般会选择链表，因为可能面临进程创建，销毁等调度导致进程状态发生变化，所以链表能够更加灵活的插入和删除。

PCB与进程的状态转移

我们熟知了进程的状态变迁和进程的数据结构 PCB 后，再来看看进程的**创建、终止、阻塞、唤醒**的过程，这些过程也就是进程的控制。

01 创建进程

操作系统允许一个进程创建另一个进程，而且允许子进程继承父进程所拥有的资源，当子进程被终止时，其在父进程处继承的资源应当还给父进程。同时，终止父进程时同时也会终止其所有的子进程。

创建进程的过程如下：

- 为新进程分配一个唯一的进程标识号，并申请一个空白的 PCB，PCB 是有限的，若申请失败则创建失败；
- 为进程分配资源，此处如果资源不足，进程就会进入等待状态，以等待资源；
- 初始化 PCB；
- 如果进程的调度队列能够接纳新进程，那就将进程插入到就绪队列，等待被调度运行；

02 终止进程

进程可以有 3 种终止方式：正常结束、异常结束以及外界干预（信号 `kill` 掉）。

终止进程的过程如下：

- 查找需要终止的进程的 PCB；
- 如果处于执行状态，则立即终止该进程的执行，然后将 CPU 资源分配给其他进程；
- 如果其还有子进程，则应将其所有子进程终止；
- 将该进程所拥有的全部资源都归还给父进程或操作系统；
- 将其从 PCB 所在队列中删除；

03 阻塞进程

当进程需要等待某一事件完成时，它可以调用阻塞语句把自己阻塞等待。而一旦被阻塞等待，它只能由另一个进程唤醒。

阻塞进程的过程如下：

- 找到将要被阻塞进程标识号对应的 PCB；
- 如果该进程为运行状态，则保护其现场，将其状态转为阻塞状态，停止运行；
- 将该 PCB 插入的阻塞队列中去；

04 唤醒进程

进程由「运行」转变为「阻塞」状态是由于进程必须等待某一事件的完成，所以处于阻塞状态的进程是绝对不可能叫醒自己的。

如果某进程正在等待 I/O 事件，需由别的进程发消息给它，则只有当该进程所期待的事件出现时，才由发现者进程用唤醒语句叫醒它。

唤醒进程的过程如下：

- 在该事件的阻塞队列中找到相应进程的 PCB；
- 将其从阻塞队列中移出，并置其状态为就绪状态；
- 把该 PCB 插入到就绪队列中，等待调度程序调度；

进程的阻塞和唤醒是一对功能相反的语句，如果某个进程调用了阻塞语句，则必有一个与之对应的唤醒语句。