

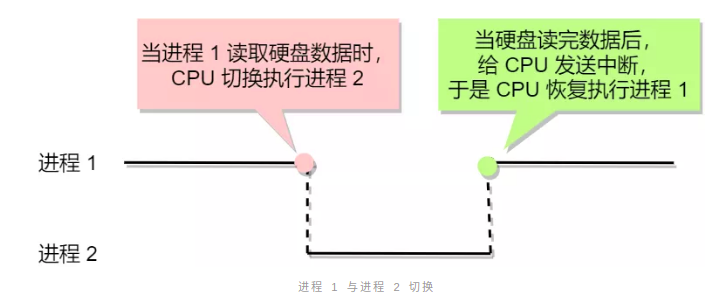
# 进程

我们编写的代码只是一个存储在硬盘的静态文件，通过编译后就会生成二进制可执行文件，当我们运行这个可执行文件后，它会被装载到内存中，接着 CPU 会执行程序中的每一条指令，那么这个运行中的程序，就被称为「进程」。

现在我们考虑有一个会读取硬盘文件数据的程序被执行了，那么当运行到读取文件的指令时，就会去从硬盘读取数据，但是硬盘的读写速度是非常慢的，那么在这个时候，如果 CPU 傻傻的等硬盘返回数据的话，那 CPU 的利用率是非常低的。

做个类比，你去煮开水时，你会傻傻的等水壶烧开吗？很明显，小孩也不会傻等。我们可以在水壶烧开之前去做其他事情。当水壶烧开了，我们自然就会听到“嘀嘀嘀”的声音，于是再把烧开的水倒入到水杯里就好了。

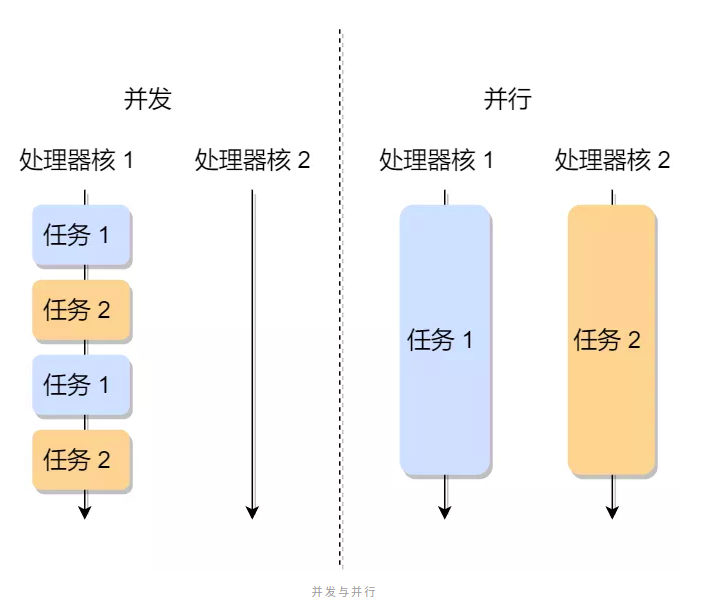
所以，当进程要从硬盘读取数据时，CPU 不需要阻塞等待数据的返回，而是去执行另外的进程。当硬盘数据返回时，CPU 会收到个中断，于是 CPU 再继续运行这个进程。



这种多个程序、交替执行的思想，就有 CPU 管理多个进程的初步想法。

对于一个支持多进程的系统，CPU 会从一个进程快速切换至另一个进程，其间每个进程各运行几十或几百个毫秒。

虽然单核的 CPU 在某一个瞬间，只能运行一个进程。但在 1 秒钟期间，它可能会运行多个进程，这样就产生并行的错觉，实际上这是并发。

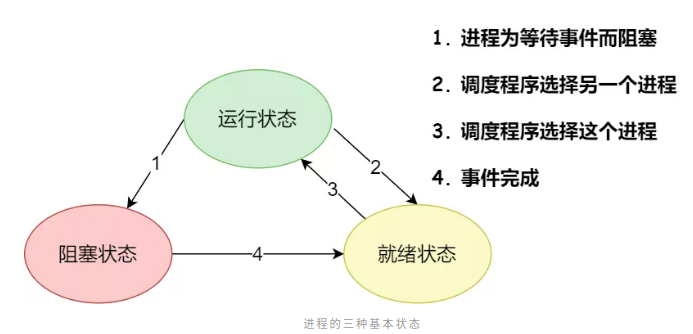


## 进程的状态

在上面，我们知道了进程有着「运行 - 暂停 - 运行」的活动规律。一般说来，一个进程并不是自始至终连续不停地运行的，它与并发执行中的其他进程的执行是相互制约的。

它有时处于运行状态，有时又由于某种原因而暂停运行处于等待状态，当使它暂停的原因消失后，它又进入准备运行状态。

所以，在一个进程的活动期间至少具备三种基本状态，即运行状态、就绪状态、阻塞状态。



上图中各个状态的意义：

运行状态（Runing）：该时刻进程占用 CPU；

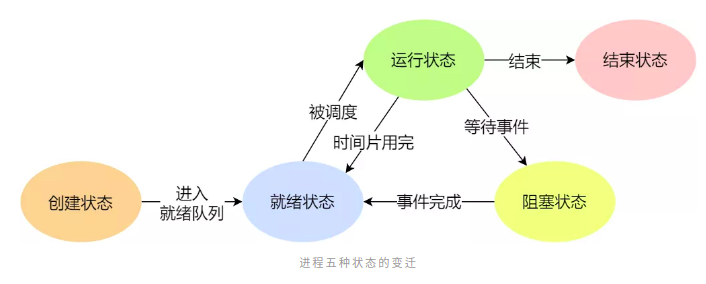
就绪状态（Ready）：可运行，但因为其他进程正在运行而暂停停止；

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；

当然，进程另外两个基本状态：

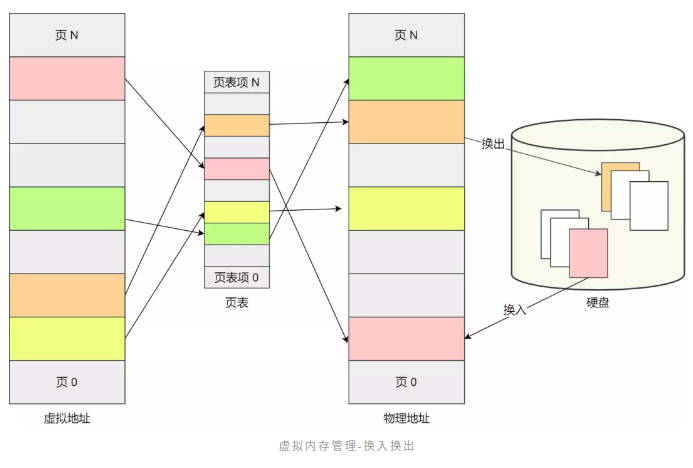
创建状态（new）：进程正在被创建时的状态；

结束状态（Exit）：进程正在从系统中消失时的状态；

* 
* 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；

另外，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

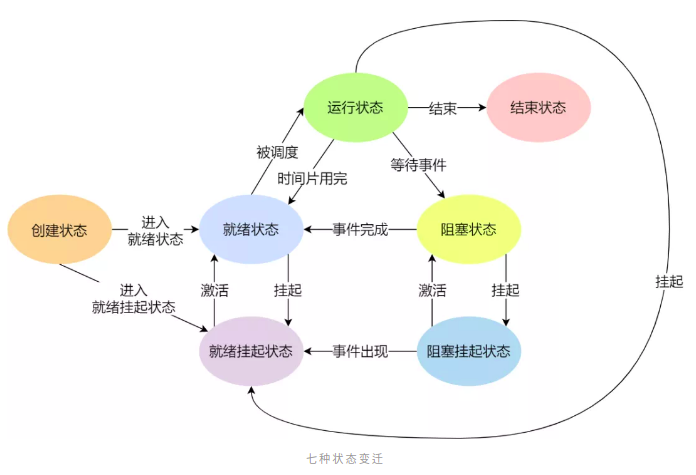


挂起状态可以分为两种：

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；

这两种挂起状态加上前面的五种状态，就变成了七种状态变迁（留给我的颜色不多了），见如下图：



## 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。

PCB 具体包含什么信息呢？

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；

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。

可见，PCB 包含信息还是比较多的。

每个 PCB 是如何组织的呢？

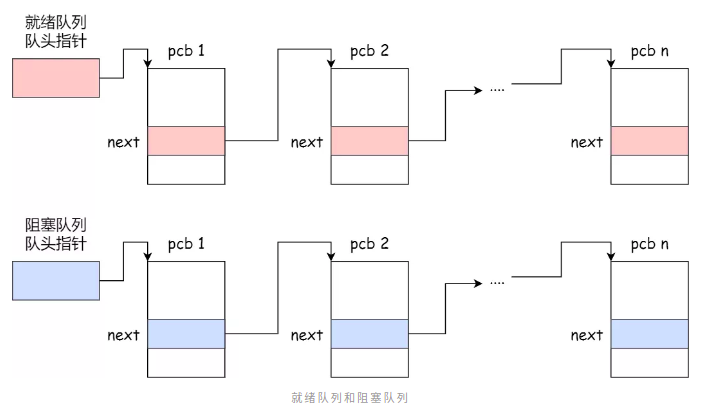
通常是通过链表的方式进行组织，把具有相同状态的进程链在一起，组成各种队列。比如：

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：



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。

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。

## 进程的控制

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。

## 进程的上下文切换

各个进程之间是共享 CPU 资源的，在不同的时候进程之间需要切换，让不同的进程可以在 CPU 执行，那么这个一个进程切换到另一个进程运行，称为进程的上下文切换。

在详细说进程上下文切换前，我们先来看看 CPU 上下文切换

大多数操作系统都是多任务，通常支持大于 CPU 数量的任务同时运行。实际上，这些任务并不是同时运行的，只是因为系统在很短的时间内，让各个任务分别在 CPU 运行，于是就造成同时运行的错觉。

任务是交给 CPU 运行的，那么在每个任务运行前，CPU 需要知道任务从哪里加载，又从哪里开始运行。

所以，操作系统需要事先帮 CPU 设置好 CPU 寄存器和程序计数器。

CPU 寄存器是 CPU 内部一个容量小，但是速度极快的内存（缓存）。我举个例子，寄存器像是你的口袋，内存像你的书包，硬盘则是你家里的柜子，如果你的东西存放到口袋，那肯定是比你从书包或家里柜子取出来要快的多。

再来，程序计数器则是用来存储 CPU 正在执行的指令位置、或者即将执行的下一条指令位置。

所以说，CPU 寄存器和程序计数是 CPU 在运行任何任务前，所必须依赖的环境，这些环境就叫做 CPU 上下文。

既然知道了什么是 CPU 上下文，那理解 CPU 上下文切换就不难了。

CPU 上下文切换就是先把前一个任务的 CPU 上下文（CPU 寄存器和程序计数器）保存起来，然后加载新任务的上下文到这些寄存器和程序计数器，最后再跳转到程序计数器所指的新位置，运行新任务。

系统内核会存储保持下来的上下文信息，当此任务再次被分配给 CPU 运行时，CPU 会重新加载这些上下文，这样就能保证任务原来的状态不受影响，让任务看起来还是连续运行。

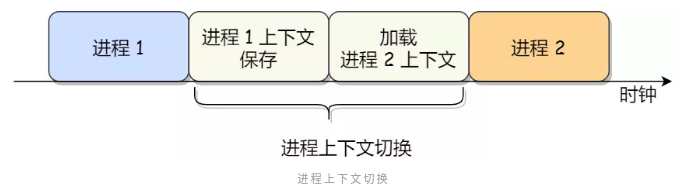
上面说到所谓的「任务」，主要包含进程、线程和中断。所以，可以根据任务的不同，把 CPU 上下文切换分成：进程上下文切换、线程上下文切换和中断上下文切换。

进程的上下文切换到底是切换什么呢？

进程是由内核管理和调度的，所以进程的切换只能发生在内核态。

所以，进程的上下文切换不仅包含了虚拟内存、栈、全局变量等用户空间的资源，还包括了内核堆栈、寄存器等内核空间的资源。

通常，会把交换的信息保存在进程的 PCB，当要运行另外一个进程的时候，我们需要从这个进程的 PCB 取出上下文，然后恢复到 CPU 中，这使得这个进程可以继续执行，如下图所示：



大家需要注意，进程的上下文开销是很关键的，我们希望它的开销越小越好，这样可以使得进程可以把更多时间花费在执行程序上，而不是耗费在上下文切换。

发生进程上下文切换有哪些场景？

为了保证所有进程可以得到公平调度，CPU 时间被划分为一段段的时间片，这些时间片再被轮流分配给各个进程。这样，当某个进程的时间片耗尽了，就会被系统挂起，切换到其它正在等待 CPU 的进程运行；

进程在系统资源不足（比如内存不足）时，要等到资源满足后才可以运行，这个时候进程也会被挂起，并由系统调度其他进程运行；

当进程通过睡眠函数 sleep 这样的方法将自己主动挂起时，自然也会重新调度；

当有优先级更高的进程运行时，为了保证高优先级进程的运行，当前进程会被挂起，由高优先级进程来运行；

发生硬件中断时，CPU 上的进程会被中断挂起，转而执行内核中的中断服务程序；

以上，就是发生进程上下文切换的常见场景了。

## 多进程

多进程模型是启动多个服务进程。原来由一个进程做的事，当一个进程忙不过来，创建几个功能一样的进程来帮它一起干活，人多力量大。

由于多进程地址空间不同，数据不能共享，一个进程内创建的变量在另一个进程是无法访问。操作系统看不下去了，凭什么同在一台机器，彼此相爱的两个进程不能说说话呢？

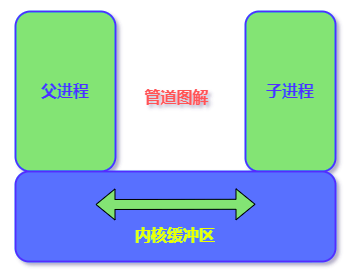
于是操作系统提供了各种系统调用，搭建起各个进程间通信的桥梁，这些方法统称为进程间通信 IPC (IPC InterProcess Communication)

### **常见进程间通信方式**

管道的实质是一个内核缓冲区，进程以先进先出 FIFO 的方式从缓冲区存取数据。是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系（父子进程间）的进程间通信。

**管道工作原理**

* 管道一端的进程顺序的将数据写入缓冲区，另一端的进程则顺序的读出数据。
* 缓冲区可以看做是一个循环队列，一个数据只能被读一次，读出来后在缓冲区就不复存在了。
* 当缓冲区为读空或写满，读数据的进程或写数据进程进入等待队列。
* 空的缓冲区有新数据写入，或者满的缓冲区有数据读出时，唤醒等待队列中的进程继续读写。



#### **命名管道 FIFO**

上面介绍的管道也称为匿名管道，只能用于亲缘关系的进程间通信。为了克服这个缺点，出现了有名管道 FIFO 。有名管道提供了一个路径名与之关联，以文件形式存在于文件系统中，这样即使不存在亲缘关系的进程，只要可以访问该路径也能相互通信。

命名管道支持同一台计算机的不同进程之间，可靠的、单向或双向的数据通信。



#### **信号 Signal**

信号是Linux系统中用于进程间互相通信或者操作的一种机制，信号可以在任何时候发给某一进程，无需知道该进程的状态。如果该进程当前不是执行态，内核会暂时保存信号，当进程恢复执行后传递给它。

如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程。

信号在用户空间进程和内核之间直接交互，内核可以利用信号来通知用户空间的进程发生了哪些系统事件，信号事件主要有两个来源：

* 硬件来源：用户按键输入Ctrl+C退出、硬件异常如无效的存储访问等。
* 软件终止：终止进程信号、其他进程调用 kill 函数、软件异常产生信号。

#### **消息队列  Message Queue**

消息队列是存放在内核中的消息链表，每个消息队列由消息队列标识符表示， 只有在内核重启或主动删除时，该消息队列才会被删除。

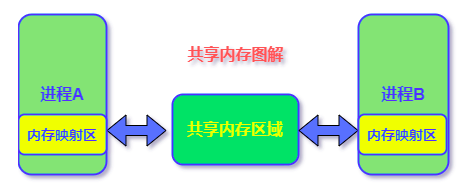
消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。另外，某个进程往一个消息队列写入消息之前，并不需要另外读进程在该队列上等待消息的到达。



#### **共享内存 Shared memory**

共享内存是一个进程把地址空间的一段，映射到能被其他进程所访问的内存，一个进程创建、多个进程可访问，进程就可以直接读写这一块内存而不需要进行数据的拷贝，从而大大提高效率。

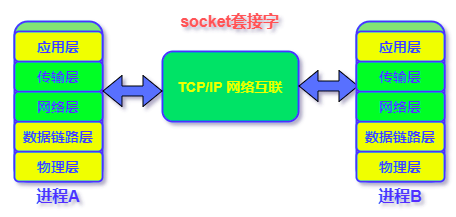
共享内存使得多个进程可以可以直接读写同一块内存空间，是最快的可用 IPC 形式，是针对其他通信机制运行效率较低而设计的。共享内存往往与其他通信机制，如信号量配合使用，来实现进程间的同步和互斥通信。



#### **套接字 Socket**

套接字你可能没听过这个名字，但绝对是接触的最多的一种进程间通信方式。因为我们熟悉的 TCP/IP 协议栈，也是建立在 socket 通信之上，TCP/IP 构建起了当前的互联网通信网络。

它是一种通信机制，凭借这种机制，既可以在本机进程间通信，也可以跨网络通过，因为，套接字通过网络接口将数据发送到本机的不同进程或远程计算机的进程。



# 线程

## 为什么使用线程？

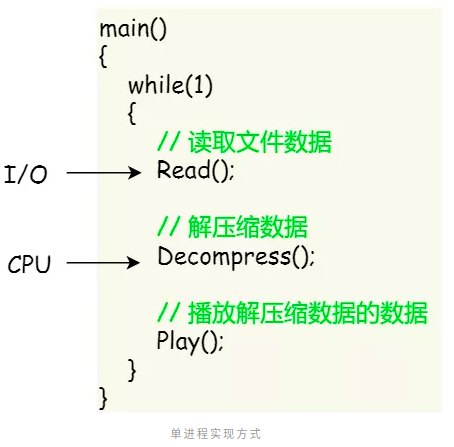
我们举个例子，假设你要编写一个视频播放器软件，那么该软件功能的核心模块有三个：

从视频文件当中读取数据；

对读取的数据进行解压缩；

把解压缩后的视频数据播放出来；

对于单进程的实现方式，我想大家都会是以下这个方式：

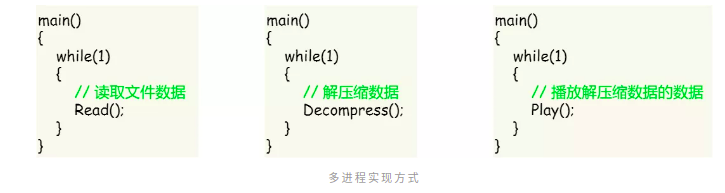


对于单进程的这种方式，存在以下问题：

播放出来的画面和声音会不连贯，因为当 CPU 能力不够强的时候，Read  的时候可能进程就等在这了，这样就会导致等半天才进行数据解压和播放；

各个函数之间不是并发执行，影响资源的使用效率；

那改进成多进程的方式：



对于多进程的这种方式，依然会存在问题：

进程之间如何通信，共享数据？

维护进程的系统开销较大，如创建进程时，分配资源、建立 PCB；终止进程时，回收资源、撤销 PCB；进程切换时，保存当前进程的状态信息；

那到底如何解决呢？需要有一种新的实体，满足以下特性：

实体之间可以并发运行；

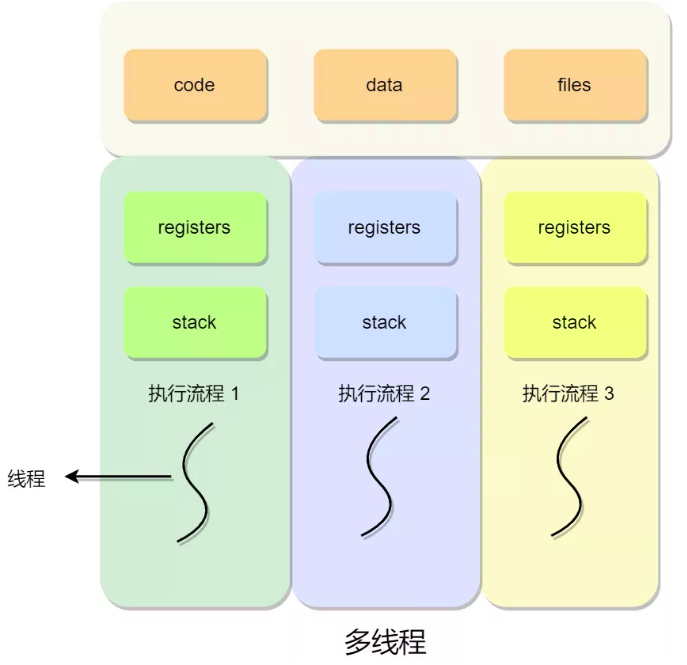
实体之间共享相同的地址空间；

这个新的实体，就是线程( Thread )，线程之间可以并发运行且共享相同的地址空间。

## 什么是线程？

线程是进程当中的一条执行流程。

同一个进程内多个线程之间可以共享代码段、数据段、打开的文件等资源，但每个线程都有独立一套的寄存器和栈，这样可以确保线程的控制流是相对独立的。



线程的优缺点？

线程的优点：

一个进程中可以同时存在多个线程；

各个线程之间可以并发执行；

各个线程之间可以共享地址空间和文件等资源；

线程的缺点：

当进程中的一个线程奔溃时，会导致其所属进程的所有线程奔溃。

举个例子，对于游戏的用户设计，则不应该使用多线程的方式，否则一个用户挂了，会影响其他同个进程的线程。

## 线程与进程的比较

**线程与进程的比较如下：**

* 进程是资源（包括内存、打开的文件等）分配的单位，线程是 CPU 调度的单位；
* 进程拥有一个完整的资源平台，而线程只独享必不可少的资源，如寄存器和栈；
* 线程同样具有就绪、阻塞、执行三种基本状态，同样具有状态之间的转换关系；
* 线程能减少并发执行的时间和空间开销；

**对于，线程相比进程能减少开销，体现在：**

* 线程的创建时间比进程快，因为进程在创建的过程中，还需要资源管理信息，比如内存管理信息、文件管理信息，而线程在创建的过程中，不会涉及这些资源管理信息，而是共享它们；
* 线程的终止时间比进程快，因为线程释放的资源相比进程少很多；
* 同一个进程内的线程切换比进程切换快，因为线程具有相同的地址空间（虚拟内存共享），这意味着同一个进程的线程都具有同一个页表，那么在切换的时候不需要切换页表。而对于进程之间的切换，切换的时候要把页表给切换掉，而页表的切换过程开销是比较大的；
* 由于同一进程的各线程间共享内存和文件资源，那么在线程之间数据传递的时候，就不需要经过内核了，这就使得线程之间的数据交互效率更高了；

所以，线程比进程不管是时间效率，还是空间效率都要高。

## 线程的上下文切换

在前面我们知道了，线程与进程最大的区别在于：线程是调度的基本单位，而进程则是资源拥有的基本单位。

所以，所谓操作系统的任务调度，实际上的调度对象是线程，而进程只是给线程提供了虚拟内存、全局变量等资源。

对于线程和进程，我们可以这么理解：

* 当进程只有一个线程时，可以认为进程就等于线程；
* 当进程拥有多个线程时，这些线程会共享相同的虚拟内存和全局变量等资源，这些资源在上下文切换时是不需要修改的；

另外，线程也有自己的私有数据，比如栈和寄存器等，这些在上下文切换时也是需要保存的。

线程上下文切换的是什么？

这还得看线程是不是属于同一个进程：

当两个线程不是属于同一个进程，则切换的过程就跟进程上下文切换一样；

当两个线程是属于同一个进程，因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据；

所以，线程的上下文切换相比进程，开销要小很多。

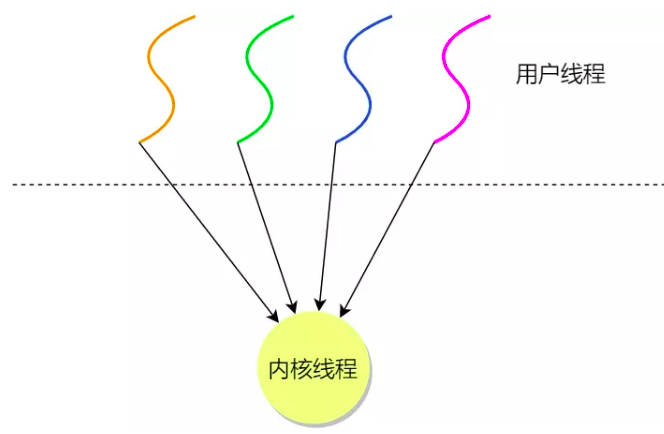
## 线程的实现

主要有三种线程的实现方式：

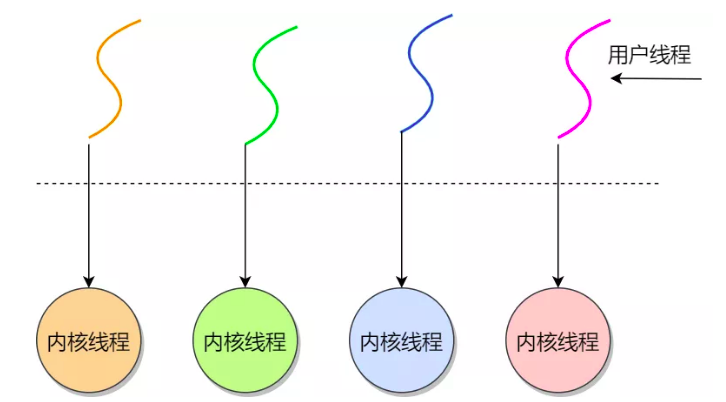
* 用户线程（User Thread）：在用户空间实现的线程，不是由内核管理的线程，是由用户态的线程库来完成线程的管理；
* 内核线程（Kernel Thread）：在内核中实现的线程，是由内核管理的线程；
* 轻量级进程（LightWeight Process）：在内核中来支持用户线程；

那么，这还需要考虑一个问题，用户线程和内核线程的对应关系。

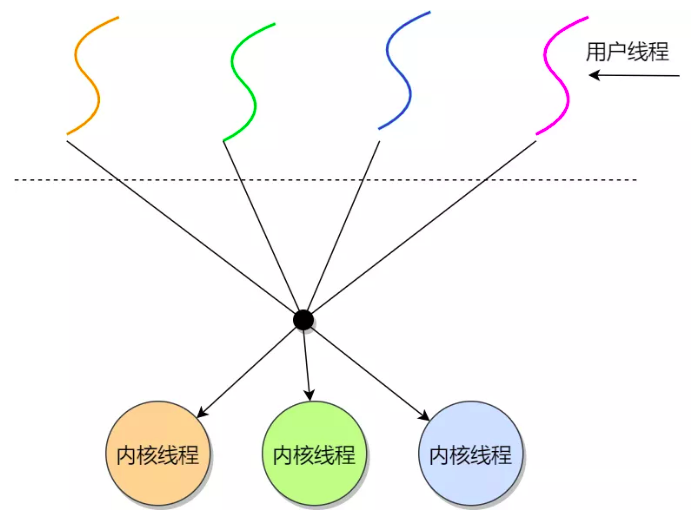
首先，第一种关系是多对一的关系，也就是多个用户线程对应同一个内核线程：



第二种是一对一的关系，也就是一个用户线程对应一个内核线程：



第三种是多对多的关系，也就是多个用户线程对应到多个内核线程：

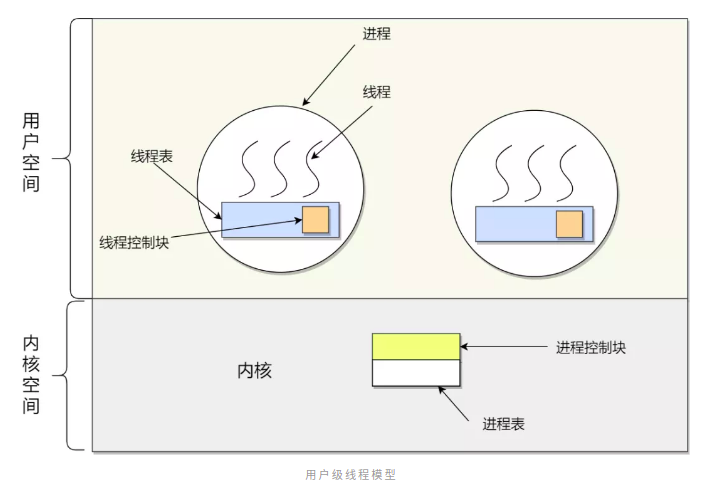


用户线程如何理解？存在什么优势和缺陷？

用户线程是基于用户态的线程管理库来实现的，那么线程控制块（Thread Control Block, TCB） 也是在库里面来实现的，对于操作系统而言是看不到这个 TCB 的，它只能看到整个进程的 PCB。

所以，用户线程的整个线程管理和调度，操作系统是不直接参与的，而是由用户级线程库函数来完成线程的管理，包括线程的创建、终止、同步和调度等。

用户级线程的模型，也就类似前面提到的多对一的关系，即多个用户线程对应同一个内核线程，如下图所示：



用户线程的优点：

* 每个进程都需要有它私有的线程控制块（TCB）列表，用来跟踪记录它各个线程状态信息（PC、栈指针、寄存器），TCB 由用户级线程库函数来维护，可用于不支持线程技术的操作系统；
* 用户线程的切换也是由线程库函数来完成的，无需用户态与内核态的切换，所以速度特别快；

用户线程的缺点：

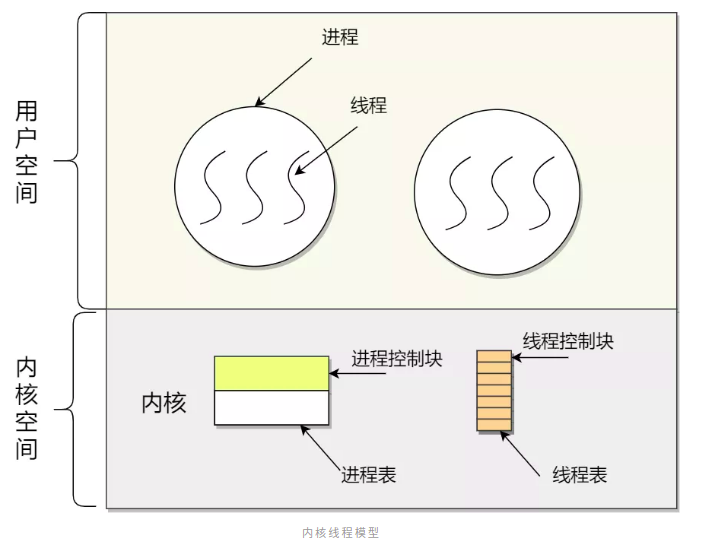
* 由于操作系统不参与线程的调度，如果一个线程发起了系统调用而阻塞，那进程所包含的用户线程都不能执行了。
* 当一个线程开始运行后，除非它主动地交出 CPU 的使用权，否则它所在的进程当中的其他线程无法运行，因为用户态的线程没法打断当前运行中的线程，它没有这个特权，只有操作系统才有，但是用户线程不是由操作系统管理的。
* 由于时间片分配给进程，故与其他进程比，在多线程执行时，每个线程得到的时间片较少，执行会比较慢；

以上，就是用户线程的优缺点了。

那内核线程如何理解？存在什么优势和缺陷？

内核线程是由操作系统管理的，线程对应的 TCB 自然是放在操作系统里的，这样线程的创建、终止和管理都是由操作系统负责。

内核线程的模型，也就类似前面提到的一对一的关系，即一个用户线程对应一个内核线程，如下图所示：



内核线程的优点：

* 在一个进程当中，如果某个内核线程发起系统调用而被阻塞，并不会影响其他内核线程的运行；
* 分配给线程，多线程的进程获得更多的 CPU 运行时间；

内核线程的缺点：

* 在支持内核线程的操作系统中，由内核来维护进程和线程的上下问信息，如 PCB 和 TCB；
* 线程的创建、终止和切换都是通过系统调用的方式来进行，因此对于系统来说，系统开销比较大；

以上，就是内核线的优缺点了。

最后的轻量级进程如何理解？

轻量级进程（Light-weight process，LWP）是内核支持的用户线程，一个进程可有一个或多个 LWP，每个 LWP 是跟内核线程一对一映射的，也就是 LWP 都是由一个内核线程支持。

另外，LWP 只能由内核管理并像普通进程一样被调度，Linux 内核是支持 LWP 的典型例子。

在大多数系统中，LWP与普通进程的区别也在于它只有一个最小的执行上下文和调度程序所需的统计信息。一般来说，一个进程代表程序的一个实例，而 LWP 代表程序的执行线程，因为一个执行线程不像进程那样需要那么多状态信息，所以 LWP 也不带有这样的信息。

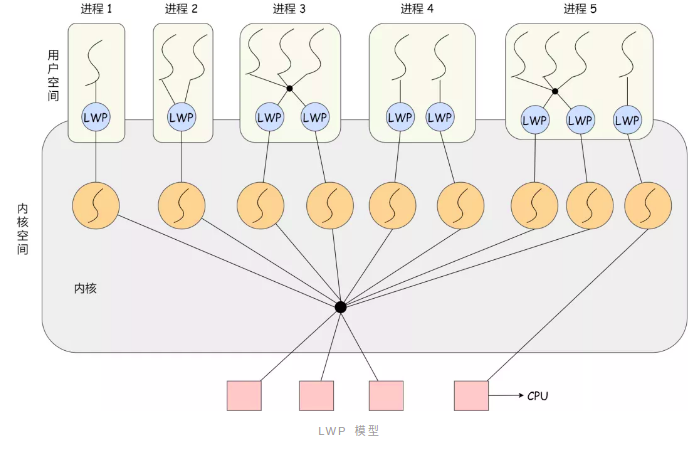
在 LWP 之上也是可以使用用户线程的，那么 LWP 与用户线程的对应关系就有三种：

1 : 1，即一个 LWP 对应 一个用户线程；

N : 1，即一个 LWP 对应多个用户线程；

N : N，即多个 LMP 对应多个用户线程；

接下来针对上面这三种对应关系说明它们优缺点。先下图的 LWP 模型：



**1 : 1 模式**

一个线程对应到一个 LWP 再对应到一个内核线程，如上图的进程 4，属于此模型。

优点：实现并行，当一个 LWP 阻塞，不会影响其他 LWP；

缺点：每一个用户线程，就产生一个内核线程，创建线程的开销较大。

**N : 1 模式**

多个用户线程对应一个 LWP 再对应一个内核线程，如上图的进程 2，线程管理是在用户空间完成的，此模式中用户的线程对操作系统不可见。

优点：用户线程要开几个都没问题，且上下文切换发生用户空间，切换的效率较高；

缺点：一个用户线程如果阻塞了，则整个进程都将会阻塞，另外在多核 CPU  中，是没办法充分利用 CPU 的。

**M : N 模式**

根据前面的两个模型混搭一起，就形成 M:N 模型，该模型提供了两级控制，首先多个用户线程对应到多个 LWP，LWP 再一一对应到内核线程，如上图的进程 3。

优点：综合了前两种优点，大部分的线程上下文发生在用户空间，且多个线程又可以充分利用多核 CPU 的资源。

## 多线程模型

启动多个相同功能的进程能提高服务处理能力，但由于各个进程的地址空间相互隔离，通信不便。

于是，多线程服务模型出场。通过前面的学习我们知道，一个进程内的多个线程可以共享进程的全部系统资源。进程内创建的多个线程都可以访问进程内的全局变量。

当然没有免费的午餐，线程虽然能方便的访问进程资源，但也带来了额外的问题。比如多线程访公共资源带来的同步与互斥问题，不同线程访问资源的先后顺序会相互影响，如果不做好同步和互斥会产生预期之外的结果，甚至死锁。

### **什么是多线程同步**

多线程同步是线程之间的一种直接制约关系，一个线程的执行依赖另一个线程的通知，当它没有得到另一个线程的通知时必须等待，直到消息到达时才被唤醒，即有很强的执行先后关系。

比如你搭建了一个商城服务。这个服务的下单流程是这样的：第一步必须要先挑选商品加入购物车，第二步才能结账计算订单金额，假设这两个步骤的操作分别由两个线程去完成，则这两个线程的操作顺序很重要，必须是先下单再结账，这就是线程同步。

### **什么是多线程互斥**

多线程互斥指的是多线程对资源访问的排他性。所谓排他性，就是当有多个线程都要使用某一共享资源时，任何时刻最多只允许一个线程获得对这个共享资源的使用权，当共享资源被其中一个线程占有时，其他未获得资源的线程必须等待，直到占用资源的线程释放资源。

打个比方，你们班只有一台投影仪，当一个同学在上面放电影的时候，如果老师进来上课要用这个投影仪，那就只能由这个同学放弃投影仪的使用权，交给老师上课投影使用，对，教室里唯一的投影仪是共享资源，具有排他性，老师和学生比作是两个线程的话，那这两个线程是互斥的访问共享资源（投影仪）

### **多线程同步和互斥方法**

Linux 系统提供以下几种方法来解决多线程的同步和互斥问题，分别是：互斥锁、条件变量、读写锁、自旋锁。

#### **条件变量**

条件变量是用来等待而不是用来上锁的。条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。适合多个线程等待某个条件的发生，不使用条件变量，那么每个线程就不断尝试互斥锁并检测条件是否发生，浪费系统资源。

通常条件变量和互斥锁同时使用。条件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件，可以用来实现线程间的同步。

#### **读写锁**

互斥量要么是加锁状态，要么是不加锁状态，而且一次只有一个线程对其进行加锁。读写锁可以有3种状态：读加锁状态、写加锁状态和不加锁状态。

一次只有一个线程可以占有写模式读写锁，但是可以有多个线程同时占有读模式的读写锁。因此，读写锁适合于对数据结构的读次数比写次数多得多的情况，且读写锁比互斥量具有更高的并行性。

读写锁加锁规则

1：如果某线程申请了读锁，其它线程可以再申请读锁，但不能申请写锁；

2：如果某线程申请了写锁，其它线程不能申请读锁，也不能申请写锁。

#### **自旋锁**

互斥锁得不到锁时，线程会进入休眠，引发任务上下文切换，任务切换涉及一系列耗时的操作，因此用互斥锁一旦遇到阻塞切换代价是十分昂贵的。

而自旋锁阻塞后不会引发上下文切换，当锁被其他线程占有时，获取锁的线程便会进入自旋，不断检测自旋锁的状态，直到得到锁，所谓的自旋就是循环等待的意思， 也就是保持在内核态，不会向互斥锁那样回到用户态。

自旋锁在用户态使用的比较少，在内核使用的比较多。自旋锁适用于临界区代码比较短，锁的持有时间比较短的场景，否则会让其他线程一直等待造成饥饿现象。

#### **信号量**

信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。

信号量是一个特殊类型的变量，它可以被增加或者减少。可根据操作信号量值的结果判断是否对公共资源具有访问的权限，当信号量值大于 0 时，则可以访问，否则将阻塞。但对其的访问被保证是原子操作，即使在一个多线程程序中也是如此。

信号量类型：

二进制信号量，它只有0和1两种取值。适用于临界代码每次只能被一个执行线程运行，就要用到二进制信号量。

计数信号量。它可以有更大的取值范围，适用于临界代码允许有限数目的线程执行，就需要用到计数信号量

# 3、调度

进程都希望自己能够占用 CPU 进行工作，那么这涉及到前面说过的进程上下文切换。

一旦操作系统把进程切换到运行状态，也就意味着该进程占用着 CPU 在执行，但是当操作系统把进程切换到其他状态时，那就不能在 CPU 中执行了，于是操作系统会选择下一个要运行的进程。

选择一个进程运行这一功能是在操作系统中完成的，通常称为调度程序（scheduler）。

那到底什么时候调度进程，或以什么原则来调度进程呢？

## 调度时机

在进程的生命周期中，当进程从一个运行状态到另外一状态变化的时候，其实会触发一次调度。

比如，以下状态的变化都会触发操作系统的调度：

* 从就绪态 -> 运行态：当进程被创建时，会进入到就绪队列，操作系统会从就绪队列选择一个进程运行；
* 从运行态 -> 阻塞态：当进程发生 I/O 事件而阻塞时，操作系统必须另外一个进程运行；
* 从运行态 -> 结束态：当进程退出结束后，操作系统得从就绪队列选择另外一个进程运行；

因为，这些状态变化的时候，操作系统需要考虑是否要让新的进程给 CPU 运行，或者是否让当前进程从 CPU 上退出来而换另一个进程运行。

另外，如果硬件时钟提供某个频率的周期性中断，那么可以根据如何处理时钟中断  
，把调度算法分为两类：

* 非抢占式调度算法挑选一个进程，然后让该进程运行直到被阻塞，或者直到该进程退出，才会调用另外一个进程，也就是说不会理时钟中断这个事情。
* 抢占式调度算法挑选一个进程，然后让该进程只运行某段时间，如果在该时段结束时，该进程仍然在运行时，则会把它挂起，接着调度程序从就绪队列挑选另外一个进程。这种抢占式调度处理，需要在时间间隔的末端发生时钟中断，以便把 CPU 控制返回给调度程序进行调度，也就是常说的时间片机制。

## 调度原则

原则一：如果运行的程序，发生了 I/O 事件的请求，那 CPU 使用率必然会很低，因为此时进程在阻塞等待硬盘的数据返回。这样的过程，势必会造成 CPU 突然的空闲。所以，为了提高 CPU 利用率，在这种发送 I/O 事件致使 CPU 空闲的情况下，调度程序需要从就绪队列中选择一个进程来运行。

原则二：有的程序执行某个任务花费的时间会比较长，如果这个程序一直占用着 CPU，会造成系统吞吐量（CPU 在单位时间内完成的进程数量）的降低。所以，要提高系统的吞吐率，调度程序要权衡长任务和短任务进程的运行完成数量。

原则三：从进程开始到结束的过程中，实际上是包含两个时间，分别是进程运行时间和进程等待时间，这两个时间总和就称为周转时间。进程的周转时间越小越好，如果进程的等待时间很长而运行时间很短，那周转时间就很长，这不是我们所期望的，调度程序应该避免这种情况发生。

原则四：处于就绪队列的进程，也不能等太久，当然希望这个等待的时间越短越好，这样可以使得进程更快的在 CPU 中执行。所以，就绪队列中进程的等待时间也是调度程序所需要考虑的原则。

原则五：对于鼠标、键盘这种交互式比较强的应用，我们当然希望它的响应时间越快越好，否则就会影响用户体验了。所以，对于交互式比较强的应用，响应时间也是调度程序需要考虑的原则。



针对上面的五种调度原则，总结成如下：

CPU 利用率：调度程序应确保 CPU 是始终匆忙的状态，这可提高 CPU 的利用率；

系统吞吐量：吞吐量表示的是单位时间内 CPU 完成进程的数量，长作业的进程会占用较长的 CPU 资源，因此会降低吞吐量，相反，短作业的进程会提升系统吞吐量；

周转时间：周转时间是进程运行和阻塞时间总和，一个进程的周转时间越小越好；

等待时间：这个等待时间不是阻塞状态的时间，而是进程处于就绪队列的时间，等待的时间越长，用户越不满意；

响应时间：用户提交请求到系统第一次产生响应所花费的时间，在交互式系统中，响应时间是衡量调度算法好坏的主要标准。

说白了，这么多调度原则，目的就是要使得进程要「快」。

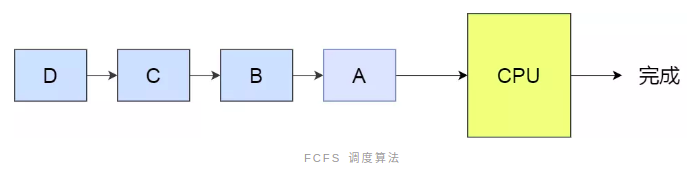
## 调度算法

不同的调度算法适用的场景也是不同的。

接下来，说说在单核 CPU 系统中常见的调度算法。

01 先来先服务调度算法

最简单的一个调度算法，就是非抢占式的先来先服务（First Come First Severd, FCFS）算法了。



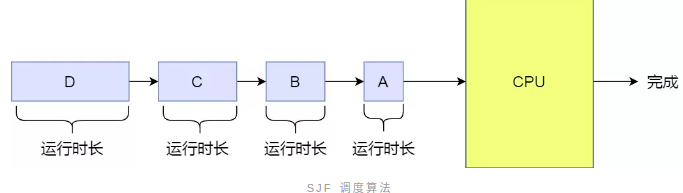
顾名思义，先来后到，每次从就绪队列选择最先进入队列的进程，然后一直运行，直到进程退出或被阻塞，才会继续从队列中选择第一个进程接着运行。

这似乎很公平，但是当一个长作业先运行了，那么后面的短作业等待的时间就会很长，不利于短作业。

FCFS 对长作业有利，适用于 CPU 繁忙型作业的系统，而不适用于 I/O 繁忙型作业的系统。

02 最短作业优先调度算法

最短作业优先（Shortest Job First, SJF）调度算法同样也是顾名思义，它会优先选择运行时间最短的进程来运行，这有助于提高系统的吞吐量。



这显然对长作业不利，很容易造成一种极端现象。

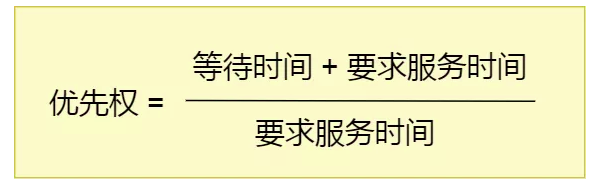
比如，一个长作业在就绪队列等待运行，而这个就绪队列有非常多的短作业，那么就会使得长作业不断的往后推，周转时间变长，致使长作业长期不会被运行。

03 高响应比优先调度算法

前面的「先来先服务调度算法」和「最短作业优先调度算法」都没有很好的权衡短作业和长作业。

那么，高响应比优先 （Highest Response Ratio Next, HRRN）调度算法主要是权衡了短作业和长作业。

每次进行进程调度时，先计算「响应比优先级」，然后把「响应比优先级」最高的进程投入运行，「响应比优先级」的计算公式：

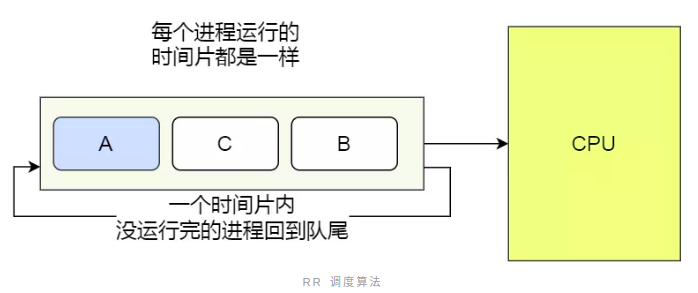


从上面的公式，可以发现：

如果两个进程的「等待时间」相同时，「要求的服务时间」越短，「响应比」就越高，这样短作业的进程容易被选中运行；

如果两个进程「要求的服务时间」相同时，「等待时间」越长，「响应比」就越高，这就兼顾到了长作业进程，因为进程的响应比可以随时间等待的增加而提高，当其等待时间足够长时，其响应比便可以升到很高，从而获得运行的机会；

04 时间片轮转调度算法

最古老、最简单、最公平且使用最广的算法就是时间片轮转（Round Robin, RR）调度算法。  


每个进程被分配一个时间段，称为时间片（Quantum），即允许该进程在该时间段中运行。

* 如果时间片用完，进程还在运行，那么将会把此进程从 CPU 释放出来，并把 CPU 分配另外一个进程；
* 如果该进程在时间片结束前阻塞或结束，则 CPU 立即进行切换；

另外，时间片的长度就是一个很关键的点：

* 如果时间片设得太短会导致过多的进程上下文切换，降低了 CPU 效率；
* 如果设得太长又可能引起对短作业进程的响应时间变长。将

通常时间片设为 20ms~50ms 通常是一个比较合理的折中值。

05 最高优先级调度算法

前面的「时间片轮转算法」做了个假设，即让所有的进程同等重要，也不偏袒谁，大家的运行时间都一样。

但是，对于多用户计算机系统就有不同的看法了，它们希望调度是有优先级的，即希望调度程序能从就绪队列中选择最高优先级的进程进行运行，这称为最高优先级（Highest Priority First，HPF）调度算法。

进程的优先级可以分为，静态优先级或动态优先级：

* 静态优先级：创建进程时候，就已经确定了优先级了，然后整个运行时间优先级都不会变化；
* 动态优先级：根据进程的动态变化调整优先级，比如如果进程运行时间增加，则降低其优先级，如果进程等待时间（就绪队列的等待时间）增加，则升高其优先级，也就是随着时间的推移增加等待进程的优先级。

该算法也有两种处理优先级高的方法，非抢占式和抢占式：

* 非抢占式：当就绪队列中出现优先级高的进程，运行完当前进程，再选择优先级高的进程。
* 抢占式：当就绪队列中出现优先级高的进程，当前进程挂起，调度优先级高的进程运行。

但是依然有缺点，可能会导致低优先级的进程永远不会运行。

06 多级反馈队列调度算法

多级反馈队列（Multilevel Feedback Queue）调度算法是「时间片轮转算法」和「最高优先级算法」的综合和发展。

顾名思义：

* 「多级」表示有多个队列，每个队列优先级从高到低，同时优先级越高时间片越短。
* 「反馈」表示如果有新的进程加入优先级高的队列时，立刻停止当前正在运行的进程，转而去运行优先级高的队列；

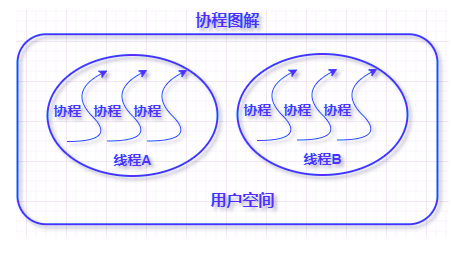
来看看，它是如何工作的：

* 设置了多个队列，赋予每个队列不同的优先级，每个队列优先级从高到低，同时优先级越高时间片越短；
* 新的进程会被放入到第一级队列的末尾，按先来先服务的原则排队等待被调度，如果在第一级队列规定的时间片没运行完成，则将其转入到第二级队列的末尾，以此类推，直至完成；
* 当较高优先级的队列为空，才调度较低优先级的队列中的进程运行。如果进程运行时，有新进程进入较高优先级的队列，则停止当前运行的进程并将其移入到原队列末尾，接着让较高优先级的进程运行；

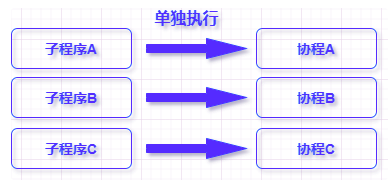
可以发现，对于短作业可能可以在第一级队列很快被处理完。对于长作业，如果在第一级队列处理不完，可以移入下次队列等待被执行，虽然等待的时间变长了，但是运行时间也会更长了，所以该算法很好的兼顾了长短作业，同时有较好的响应时间。

# 4、多协程模型

什么是协程呢？协程 Coroutines 是一种比线程更加轻量级的微线程。类比一个进程可以拥有多个线程，一个线程也可以拥有多个协程，因此协程又称微线程和纤程。



可以粗略的把协程理解成子程序调用，每个子程序都可以在一个单独的协程内执行。



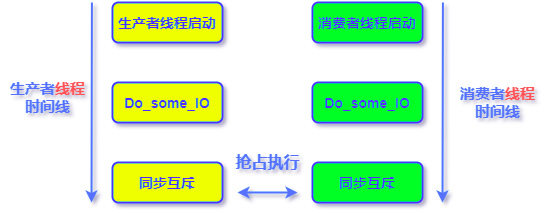
### **协程服务模型**

为了说明什么是协程模型，先用多线程下的生产者消费者模型举个栗子。

启动两个线程分别执行两个函数  Do\_some\_IO 和 Do\_some\_process ，第一个做耗时的 IO 处理操作，第二个对 IO 操作结果做快速的处理计算工作。伪代码如下：

多线程执行过程是这样的：

1. 生产者线程先调用函数 Do\_some\_IO 做比较耗时的 IO 操作，比如从网络套接字中读取数据这类操作。
2. 在生产者线程执行 Do\_some\_IO 完成数据读取之前，消费者线程要阻塞等待。
3. 在消费者线程执行 Do\_some\_process 完成数据处理完成之前，生产者线程要阻塞等待。
4. 在消费者线程执行 Do\_some\_process 完成数据处理完成之后，要通知生成者线程继续 Do\_some\_IO



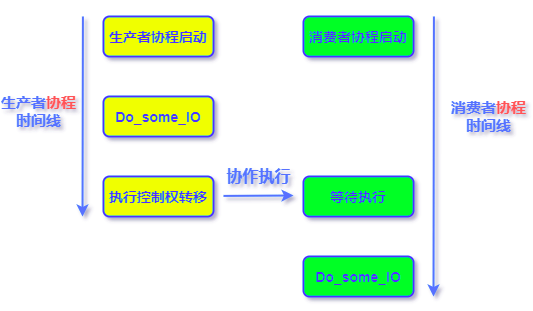
可以看到，多线程模型为了保证各个线程并行工作，需要额外做很多线程间的同步和通知工作，而且线程频繁的在阻塞和唤醒间切换，我们知道 Linux 下线程是轻量级线程 LWP ，每次线程切换涉及用户态和内核态的切换，还是很消耗性能的。

同样的场景在协程模型里是怎么处理的呢？还是用前面的例子，说明协程模型的执行流程。

· 分配生产者协程执行 Do\_some\_IO 做 IO 处理操作，分配消费者协程执行 Do\_some\_process 计算处理操作。

· 在生产者协程工作期间，消费者协程保持等待。

· 当生产者协程完成 IO 处理，返回处理结果给消费者，并把程序执行权限交给消费者协程向下执行。



### **协程优势**

由于协程在线程内实现，因此始终都是一个线程操作共享资源，所以不存在多线程抢占资源和资源同步问题。

生产者协程和消费者协程，互相配合协作完成工作，而不是相互抢占，而且协程创建和切换的开销比线程小得多。

# 5、Linux进程必知必会

Linux 是一个多道程序设计系统，因此系统中存在彼此相互独立的进程同时运行。此外，每个用户都会同时有几个活动的进程。因为如果是一个大型系统，可能有数百上千的进程在同时运行。

在某些用户空间中，即使用户退出登录，仍然会有一些后台进程在运行，这些进程被称为 守护进程(daemon)。

Linux 中有一种特殊的守护进程被称为 计划守护进程(Cron daemon) ，计划守护进程可以每分钟醒来一次检查是否有工作要做，做完会继续回到睡眠状态等待下一次唤醒。

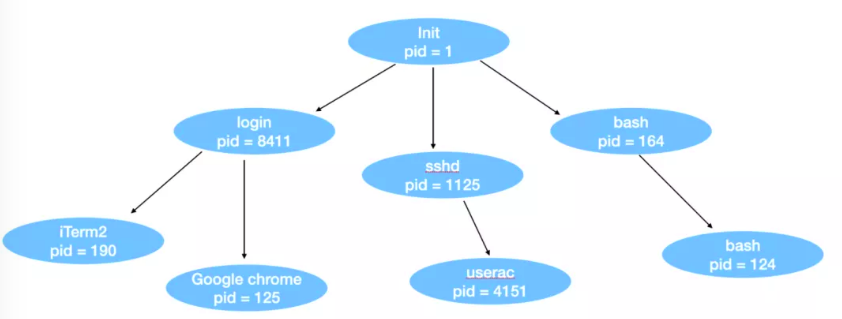
在 Linux 系统中，进程通过非常简单的方式来创建，fork 系统调用会创建一个源进程的拷贝(副本)。调用 fork 函数的进程被称为 父进程(parent process)，使用 fork 函数创建出来的进程被称为 子进程(child process)。父进程和子进程都有自己的内存映像。如果在子进程创建出来后，父进程修改了一些变量等，那么子进程是看不到这些变化的，也就是 fork 后，父进程和子进程相互独立。

虽然父进程和子进程保持相互独立，但是它们却能够共享相同的文件，如果在 fork 之前，父进程已经打开了某个文件，那么 fork 后，父进程和子进程仍然共享这个打开的文件。对共享文件的修改会对父进程和子进程同时可见。

那么该如何区分父进程和子进程呢？子进程只是父进程的拷贝，所以它们几乎所有的情况都一样，包括内存映像、变量、寄存器等。区分的关键在于 fork 函数调用后的返回值，如果 fork 后返回一个非零值，这个非零值即是子进程的 进程标识符(Process Identiier, PID)，而会给子进程返回一个零值，可以用下面代码来进行表示

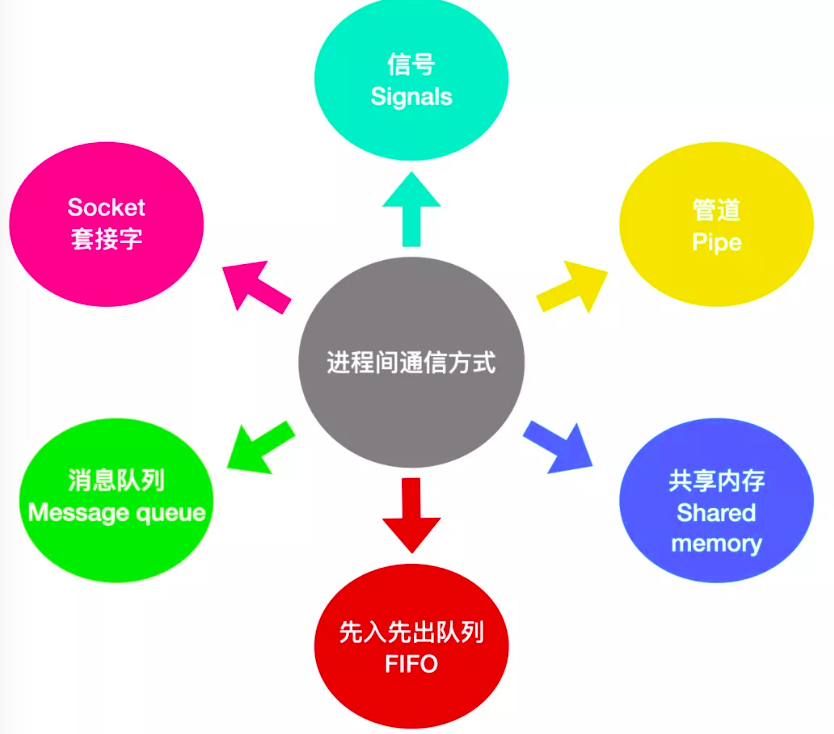


父进程在 fork 后会得到子进程的 PID，这个 PID 即能代表这个子进程的唯一标识符也就是 PID。如果子进程想要知道自己的 PID，可以调用 getpid 方法。当子进程结束运行时，父进程会得到子进程的 PID，因为一个进程会 fork 很多子进程，子进程也会 fork 子进程，所以 PID 是非常重要的。我们把第一次调用 fork 后的进程称为 原始进程，一个原始进程可以生成一颗继承树



## **Linux 进程间通信**

Linux 进程间的通信机制通常被称为 Internel-Process communication,IPC下面我们来说一说 Linux 进程间通信的机制，大致来说，Linux 进程间的通信机制可以分为 6 种



信号是 UNIX 系统最先开始使用的进程间通信机制，因为 Linux 是继承于 UNIX 的，所以 Linux 也支持信号机制，通过向一个或多个进程发送异步事件信号来实现，信号可以从键盘或者访问不存在的位置等地方产生；信号通过 shell 将任务发送给子进程。

你可以在 Linux 系统上输入 kill -l 来列出系统使用的信号，下面是我提供的一些信号



进程可以选择忽略发送过来的信号，但是有两个是不能忽略的：SIGSTOP 和 SIGKILL 信号。SIGSTOP 信号会通知当前正在运行的进程执行关闭操作，SIGKILL 信号会通知当前进程应该被杀死。除此之外，进程可以选择它想要处理的信号，进程也可以选择阻止信号，如果不阻止，可以选择自行处理，也可以选择进行内核处理。如果选择交给内核进行处理，那么就执行默认处理。

### **管道 pipe**

在两个进程之间，可以建立一个通道，一个进程向这个通道里写入字节流，另一个进程从这个管道中读取字节流。管道是同步的，当进程尝试从空管道读取数据时，该进程会被阻塞，直到有可用数据为止。shell 中的管线 pipelines 就是用管道实现的，当 shell 发现输出

管道实际上就是 |，两个应用程序不知道有管道的存在，一切都是由 shell 管理和控制的。

### **先入先出队列 FIFO**

先入先出队列 FIFO 通常被称为 命名管道(Named Pipes)，命名管道的工作方式与常规管道非常相似，但是确实有一些明显的区别。未命名的管道没有备份文件：操作系统负责维护内存中的缓冲区，用来将字节从写入器传输到读取器。一旦写入或者输出终止的话，缓冲区将被回收，传输的数据会丢失。相比之下，命名管道具有支持文件和独特 API ，命名管道在文件系统中作为设备的专用文件存在。当所有的进程通信完成后，命名管道将保留在文件系统中以备后用。命名管道具有严格的 FIFO 行为



### **消息队列 Message Queue**

一听到消息队列这个名词你可能不知道是什么意思，消息队列是用来描述内核寻址空间内的内部链接列表。可以按几种不同的方式将消息按顺序发送到队列并从队列中检索消息。每个消息队列由 IPC 标识符唯一标识。消息队列有两种模式，一种是严格模式， 严格模式就像是 FIFO 先入先出队列似的，消息顺序发送，顺序读取。还有一种模式是 非严格模式，消息的顺序性不是非常重要。

### **套接字 Socket**

还有一种管理两个进程间通信的是使用 socket，socket 提供端到端的双相通信。一个套接字可以与一个或多个进程关联。就像管道有命令管道和未命名管道一样，套接字也有两种模式，套接字一般用于两个进程之间的网络通信，网络套接字需要来自诸如TCP（传输控制协议）或较低级别UDP（用户数据报协议）等基础协议的支持。

套接字有以下几种分类

顺序包套接字(Sequential Packet Socket)：此类套接字为最大长度固定的数据报提供可靠的连接。此连接是双向的并且是顺序的。

数据报套接字(Datagram Socket)：数据包套接字支持双向数据流。数据包套接字接受消息的顺序与发送者可能不同。

流式套接字(Stream Socket)：流套接字的工作方式类似于电话对话，提供双向可靠的数据流。

原始套接字(Raw Socket)：可以使用原始套接字访问基础通信协议。

### **Linux 中进程管理系统调用**

现在关注一下 Linux 系统中与进程管理相关的系统调用。在了解之前你需要先知道一下什么是系统调用。

操作系统为我们屏蔽了硬件和软件的差异，它的最主要功能就是为用户提供一种抽象，隐藏内部实现，让用户只关心在 GUI 图形界面下如何使用即可。操作系统可以分为两种模式

内核态：操作系统内核使用的模式

用户态：用户应用程序所使用的模式

我们常说的上下文切换 指的就是内核态模式和用户态模式的频繁切换。而系统调用指的就是引起内核态和用户态切换的一种方式，系统调用通常在后台静默运行，表示计算机程序向其操作系统内核请求服务。

系统调用指令有很多，下面是一些与进程管理相关的最主要的系统调用

#### **fork**

fork 调用用于创建一个与父进程相同的子进程，创建完进程后的子进程拥有和父进程一样的程序计数器、相同的 CPU 寄存器、相同的打开文件

#### **exec**

exec 系统调用用于执行驻留在活动进程中的文件，调用 exec 后，新的可执行文件会替换先前的可执行文件并获得执行。也就是说，调用 exec 后，会将旧文件或程序替换为新文件或执行，然后执行文件或程序。新的执行程序被加载到相同的执行空间中，因此进程的 PID不会修改，因为我们没有创建新进程，只是替换旧进程。但是进程的数据、代码、堆栈都已经被修改。如果当前要被替换的进程包含多个线程，那么所有的线程将被终止，新的进程映像被加载执行。

这里需要解释一下进程映像(Process image) 的概念

什么是进程映像呢？进程映像是执行程序时所需要的可执行文件，通常会包括下面这些东西

**代码段（codesegment/textsegment）**

又称文本段，用来存放指令，运行代码的一块内存空间

此空间大小在代码运行前就已经确定

内存空间一般属于只读，某些架构的代码也允许可写

在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

**数据段（datasegment）**

可读可写

存储初始化的全局变量和初始化的 static 变量

数据段中数据的生存期是随程序持续性（随进程持续性） 随进程持续性：进程创建就存在，进程死亡就消失

**bss 段（bsssegment）：**

可读可写

存储未初始化的全局变量和未初始化的 static 变量

bss 段中的数据一般默认为 0

**Data 段**

是可读写的，因为变量的值可以在运行时更改。此段的大小也固定。

**栈（stack）：**

可读可写

存储的是函数或代码中的局部变量(非 static 变量)

栈的生存期随代码块持续性，代码块运行就给你分配空间，代码块结束，就自动回收空间

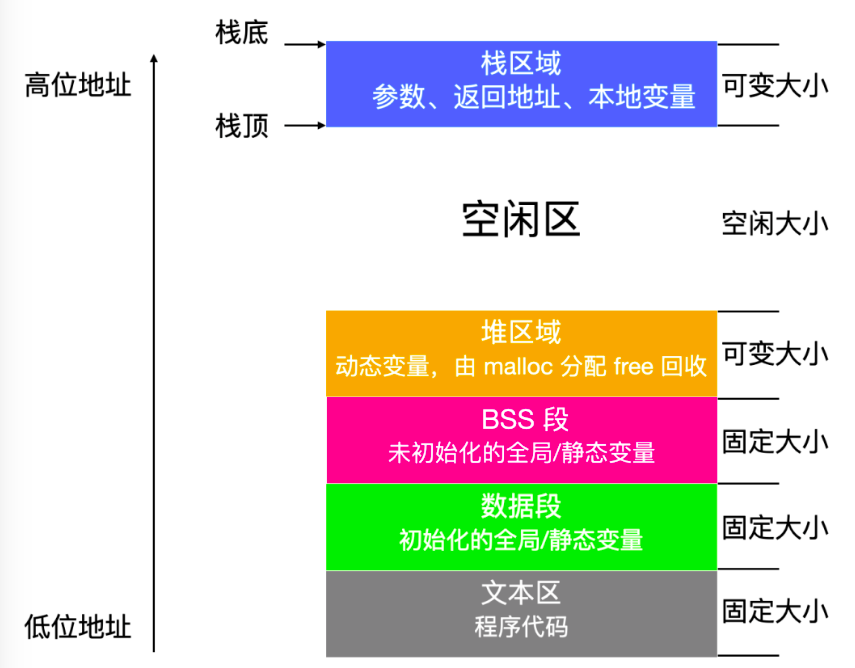
**堆（heap）：**

可读可写

存储的是程序运行期间动态分配的 malloc/realloc 的空间

堆的生存期随进程持续性，从 malloc/realloc 到 free 一直存在

下面是这些区域的构成图



exec 系统调用是一些函数的集合，这些函数是

execl

execle

execlp

execv

execve

execvp

下面来看一下 exec 的工作原理

当前进程映像被替换为新的进程映像

新的进程映像是你做为 exec 传递的参数

结束当前正在运行的进程

新的进程映像有 PID，相同的环境和一些文件描述符(因为未替换进程，只是替换了进程映像)

CPU 状态和虚拟内存受到影响，当前进程映像的虚拟内存映射被新进程映像的虚拟内存代替。

#### **waitpid**

等待子进程结束或终止

**exit**

在许多计算机操作系统上，计算机进程的终止是通过执行 exit 系统调用命令执行的。0 表示进程能够正常结束，其他值表示进程以非正常的行为结束。

其他一些常见的系统调用如下



## **Linux 进程和线程的实现**

Linux 进程就像一座冰山，你看到的只是冰山一角。

在 Linux 内核结构中，进程会被表示为 任务，通过结构体 structure 来创建。不像其他的操作系统会区分进程、轻量级进程和线程，Linux 统一使用任务结构来代表执行上下文。因此，对于每个单线程进程来说，单线程进程将用一个任务结构表示，对于多线程进程来说，将为每一个用户级线程分配一个任务结构。Linux 内核是多线程的，并且内核级线程不与任何用户级线程相关联。

对于每个进程来说，在内存中都会有一个 task\_struct 进程描述符与之对应。进程描述符包含了内核管理进程所有有用的信息，包括 调度参数、打开文件描述符等等。进程描述符从进程创建开始就一直存在于内核堆栈中。

Linux 和 Unix 一样，都是通过 PID 来区分不同的进程，内核会将所有进程的任务结构组成为一个双向链表。PID 能够直接被映射称为进程的任务结构所在的地址，从而不需要遍历双向链表直接访问。

我们上面提到了进程描述符，这是一个非常重要的概念，我们上面还提到了进程描述符是位于内存中的，这里我们省略了一句话，那就是进程描述符是存在用户的任务结构中，当进程位于内存并开始运行时，进程描述符才会被调入内存。

进程描述符可以归为下面这几类

* 调度参数(scheduling parameters)：进程优先级、最近消耗 CPU 的时间、最近睡眠时间一起决定了下一个需要运行的进程
* 内存映像(memory image)：我们上面说到，进程映像是执行程序时所需要的可执行文件，它由数据和代码组成。
* 信号(signals)：显示哪些信号被捕获、哪些信号被执行
* 寄存器：当发生内核陷入 (trap) 时，寄存器的内容会被保存下来。
* 系统调用状态(system call state)：当前系统调用的信息，包括参数和结果
* 文件描述符表(file descriptor table)：有关文件描述符的系统被调用时，文件描述符作为索引在文件描述符表中定位相关文件的 i-node 数据结构
* 统计数据(accounting)：记录用户、进程占用系统 CPU 时间表的指针，一些操作系统还保存进程最多占用的 CPU 时间、进程拥有的最大堆栈空间、进程可以消耗的页面数等。
* 内核堆栈(kernel stack)：进程的内核部分可以使用的固定堆栈
* 其他：当前进程状态、事件等待时间、距离警报的超时时间、PID、父进程的 PID 以及用户标识符等

有了上面这些信息，现在就很容易描述在 Linux 中是如何创建这些进程的了，创建新流程实际上非常简单。**为子进程开辟一块新的用户空间的进程描述符，然后从父进程复制大量的内容。为这个子进程分配一个 PID，设置其内存映射，赋予它访问父进程文件的权限，注册并启动**。

当执行 fork 系统调用时，调用进程会陷入内核并创建一些和任务相关的数据结构，比如内核堆栈(kernel stack) 和 thread\_info 结构。

**“**

关于 thread\_info 结构可以参考

https://docs.huihoo.com/doxygen/linux/kernel/3.7/arch\_2avr32\_2include\_2asm\_2thread\_\_info\_8h\_source.html

这个结构中包含进程描述符，进程描述符位于固定的位置，使得 Linux 系统只需要很小的开销就可以定位到一个运行中进程的数据结构。

进程描述符的主要内容是根据父进程的描述符来填充。Linux 操作系统会寻找一个可用的 PID，并且此 PID 没有被任何进程使用，更新进程标示符使其指向一个新的数据结构即可。为了减少 hash table 的碰撞，进程描述符会形成链表。它还将 task\_struct 的字段设置为指向任务数组上相应的上一个/下一个进程。

**“**

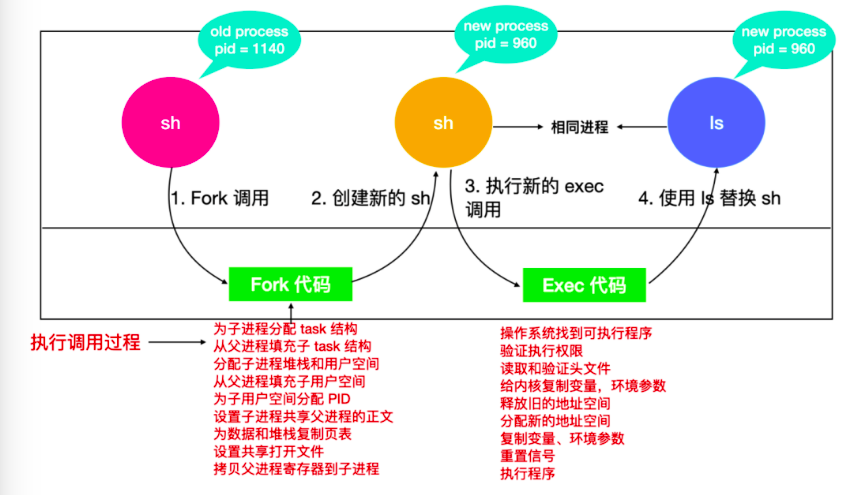
task\_struct ：Linux 进程描述符，内部涉及到众多 C++ 源码，我们会在后面进行讲解。

从原则上来说，为子进程开辟内存区域并为子进程分配数据段、堆栈段，并且对父进程的内容进行复制，但是实际上 fork 完成后，子进程和父进程没有共享内存，所以需要复制技术来实现同步，但是复制开销比较大，因此 Linux 操作系统使用了一种 欺骗 方式。即为子进程分配页表，然后新分配的页表指向父进程的页面，同时这些页面是只读的。当进程向这些页面进行写入的时候，会开启保护错误。内核发现写入操作后，会为进程分配一个副本，使得写入时把数据复制到这个副本上，这个副本是共享的，这种方式称为 写入时复制(copy on write)，这种方式避免了在同一块内存区域维护两个副本的必要，节省内存空间。

在子进程开始运行后，操作系统会调用 exec 系统调用，内核会进行查找验证可执行文件，把参数和环境变量复制到内核，释放旧的地址空间。

现在新的地址空间需要被创建和填充。如果系统支持映射文件，就像 Unix 系统一样，那么新的页表就会创建，表明内存中没有任何页，除非所使用的页面是堆栈页，其地址空间由磁盘上的可执行文件支持。新进程开始运行时，立刻会收到一个缺页异常(page fault)，这会使具有代码的页面加载进入内存。最后，参数和环境变量被复制到新的堆栈中，重置信号，寄存器全部清零。新的命令开始运行。

下面是一个示例，用户输出 ls，shell 会调用 fork 函数复制一个新进程，shell 进程会调用 exec 函数用可执行文件 ls 的内容覆盖它的内存。

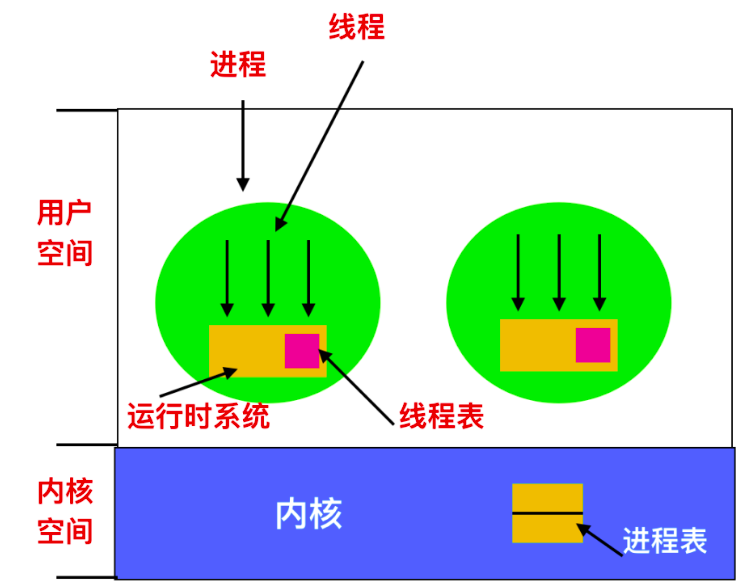


### **Linux 线程**

#### **用户级线程**

现在我们来讨论一下 Linux 中的线程，线程是轻量级的进程，想必这句话你已经听过很多次了，轻量级体现在所有的进程切换都需要清除所有的表、进程间的共享信息也比较麻烦，一般来说通过管道或者共享内存，如果是 fork 函数后的父子进程则使用共享文件，然而线程切换不需要像进程一样具有昂贵的开销，而且线程通信起来也更方便。线程分为两种：用户级线程和内核级线程

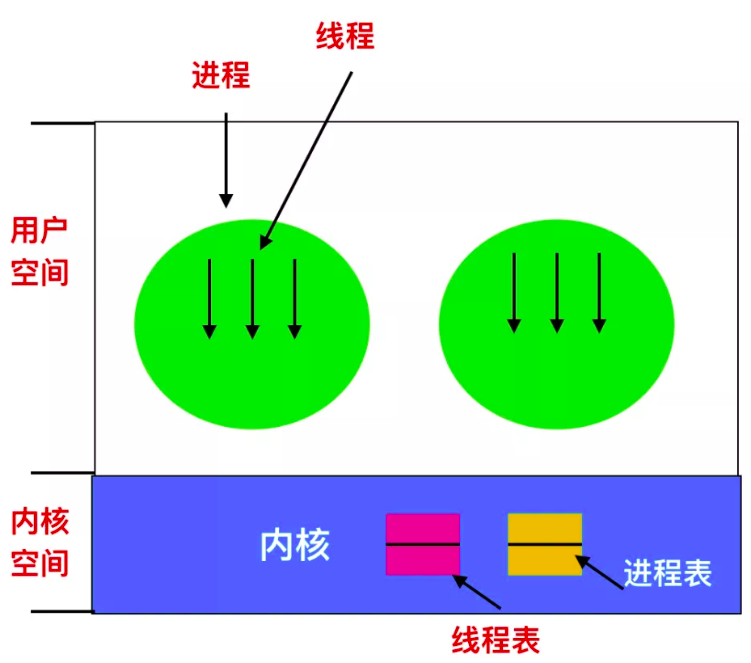
用户级线程避免使用内核，通常，每个线程会显示调用开关，发送信号或者执行某种切换操作来放弃 CPU，同样，计时器可以强制进行开关，用户线程的切换速度通常比内核线程快很多。在用户级别实现线程会有一个问题，即单个线程可能会垄断 CPU 时间片，导致其他线程无法执行从而 饿死。如果执行一个 I/O 操作，那么 I/O 会阻塞，其他线程也无法运行。



一种解决方案是，一些用户级的线程包解决了这个问题。可以使用时钟周期的监视器来控制第一时间时间片独占。然后，一些库通过特殊的包装来解决系统调用的 I/O 阻塞问题，或者可以为非阻塞 I/O 编写任务。

#### **内核级线程**

内核级线程通常使用几个进程表在内核中实现，每个任务都会对应一个进程表。在这种情况下，内核会在每个进程的时间片内调度每个线程。

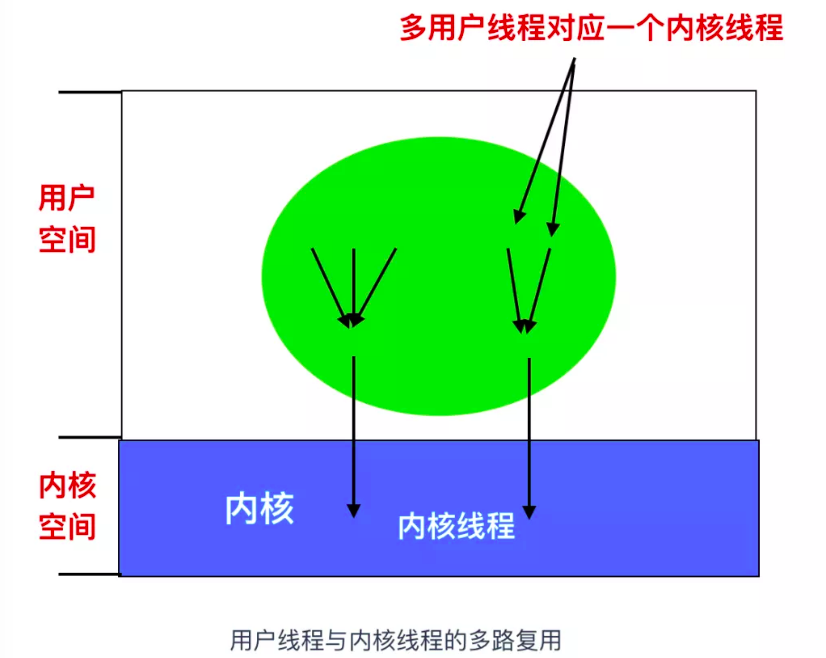


所有能够阻塞的调用都会通过系统调用的方式来实现，当一个线程阻塞时，内核可以进行选择，是运行在同一个进程中的另一个线程（如果有就绪线程的话）还是运行一个另一个进程中的线程。

从用户空间 -> 内核空间 -> 用户空间的开销比较大，但是线程初始化的时间损耗可以忽略不计。这种实现的好处是由时钟决定线程切换时间，因此不太可能将时间片与任务中的其他线程占用时间绑定到一起。同样，I/O 阻塞也不是问题。

#### **混合实现**

结合用户空间和内核空间的优点，设计人员采用了一种内核级线程的方式，然后将用户级线程与某些或者全部内核线程多路复用起来



### **Linux 调度**

下面我们来关注一下 Linux 系统的调度算法，首先需要认识到，Linux 系统的线程是内核线程，所以 Linux 系统是基于线程的，而不是基于进程的。

为了进行调度，Linux 系统将线程分为三类

实时先入先出

实时轮询

分时

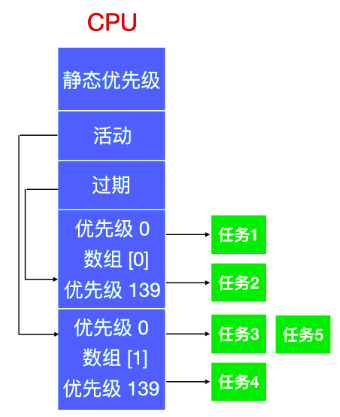
实时先入先出线程具有最高优先级，它不会被其他线程所抢占，除非那是一个刚刚准备好的，拥有更高优先级的线程进入。实时轮转线程与实时先入先出线程基本相同，只是每个实时轮转线程都有一个时间量，时间到了之后就可以被抢占。如果多个实时线程准备完毕，那么每个线程运行它时间量所规定的时间，然后插入到实时轮转线程末尾。

注意这个实时只是相对的，无法做到绝对的实时，因为线程的运行时间无法确定。它们相对分时系统来说，更加具有实时性

Linux 系统会给每个线程分配一个 nice 值，这个值代表了优先级的概念。nice 值默认值是 0 ，但是可以通过系统调用 nice 值来修改。修改值的范围从 -20 - +19。nice 值决定了线程的静态优先级。一般系统管理员的 nice 值会比一般线程的优先级高，它的范围是 -20 - -1。

下面我们更详细的讨论一下 Linux 系统的两个调度算法，它们的内部与调度队列(runqueue) 的设计很相似。运行队列有一个数据结构用来监视系统中所有可运行的任务并选择下一个可以运行的任务。每个运行队列和系统中的每个 CPU 有关。

Linux O(1) 调度器是历史上很流行的一个调度器。这个名字的由来是因为它能够在常数时间内执行任务调度。在 O(1) 调度器里，调度队列被组织成两个数组，一个是任务正在活动的数组，一个是任务过期失效的数组。如下图所示，每个数组都包含了 140 个链表头，每个链表头具有不同的优先级。



大致流程如下：

调度器从正在活动数组中选择一个优先级最高的任务。如果这个任务的时间片过期失效了，就把它移动到过期失效数组中。如果这个任务阻塞了，比如说正在等待 I/O 事件，那么在它的时间片过期失效之前，一旦 I/O 操作完成，那么这个任务将会继续运行，它将被放回到之前正在活动的数组中，因为这个任务之前已经消耗一部分 CPU 时间片，所以它将运行剩下的时间片。当这个任务运行完它的时间片后，它就会被放到过期失效数组中。一旦正在活动的任务数组中没有其他任务后，调度器将会交换指针，使得正在活动的数组变为过期失效数组，过期失效数组变为正在活动的数组。使用这种方式可以保证每个优先级的任务都能够得到执行，不会导致线程饥饿。

在这种调度方式中，不同优先级的任务所得到 CPU 分配的时间片也是不同的，高优先级进程往往能得到较长的时间片，低优先级的任务得到较少的时间片。

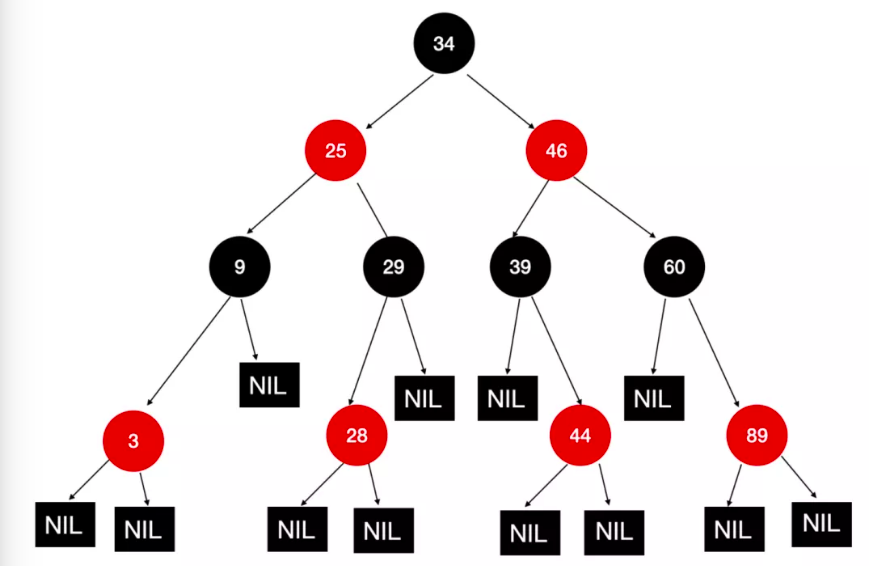
这种方式为了保证能够更好的提供服务，通常会为 交互式进程 赋予较高的优先级，交互式进程就是用户进程。

Linux 系统不知道一个任务究竟是 I/O 密集型的还是 CPU 密集型的，它只是依赖于交互式的方式，Linux 系统会区分是静态优先级 还是 动态优先级。动态优先级是采用一种奖励机制来实现的。奖励机制有两种方式：奖励交互式线程、惩罚占用 CPU 的线程。在 Linux O(1) 调度器中，最高的优先级奖励是 -5，注意这个优先级越低越容易被线程调度器接受，所以最高惩罚的优先级是 +5。具体体现就是操作系统维护一个名为 sleep\_avg 的变量，任务唤醒会增加 sleep\_avg 变量的值，当任务被抢占或者时间量过期会减少这个变量的值，反映在奖励机制上。

O(1) 使用启发式的这种方式，会使任务的优先级变得复杂并且不完善，从而导致在处理交互任务时性能很糟糕。

为了改进这个缺点，O(1) 调度器的开发者又提出了一个新的方案，即 公平调度器(Completely Fair Scheduler, CFS)。CFS 的主要思想是使用一颗红黑树作为调度队列。

CFS 会根据任务在 CPU 上的运行时间长短而将其有序地排列在树中，时间精确到纳秒级。下面是 CFS 的构造模型



CFS 的调度过程如下：

CFS 算法总是优先调度哪些使用 CPU 时间最少的任务。最小的任务一般都是在最左边的位置。当有一个新的任务需要运行时，CFS 会把这个任务和最左边的数值进行对比，如果此任务具有最小时间值，那么它将进行运行，否则它会进行比较，找到合适的位置进行插入。然后 CPU 运行红黑树上当前比较的最左边的任务。

在红黑树中选择一个节点来运行的时间可以是常数时间，但是插入一个任务的时间是 O(loog(N))，其中 N 是系统中的任务数。考虑到当前系统的负载水平，这是可以接受的。

调度器只需要考虑可运行的任务即可。这些任务被放在适当的调度队列中。不可运行的任务和正在等待的各种 I/O 操作或内核事件的任务被放入一个等待队列中。等待队列头包含一个指向任务链表的指针和一个自旋锁。自旋锁对于并发处理场景下用处很大。

#### **Linux 系统中的同步**

下面来聊一下 Linux 中的同步机制。早期的 Linux 内核只有一个 大内核锁(Big Kernel Lock,BKL) 。它阻止了不同处理器并发处理的能力。因此，需要引入一些粒度更细的锁机制。

Linux 提供了若干不同类型的同步变量，这些变量既能够在内核中使用，也能够在用户应用程序中使用。在地层中，Linux 通过使用 atomic\_set 和 atomic\_read 这样的操作为硬件支持的原子指令提供封装。硬件提供内存重排序，这是 Linux 屏障的机制。

具有高级别的同步像是自旋锁的描述是这样的，当两个进程同时对资源进行访问，在一个进程获得资源后，另一个进程不想被阻塞，所以它就会自旋，等待一会儿再对资源进行访问。Linux 也提供互斥量或信号量这样的机制，也支持像是 mutex\_tryLock 和 mutex\_tryWait 这样的非阻塞调用。也支持中断处理事务，也可以通过动态禁用和启用相应的中断来实现。

### **Linux 启动**

下面来聊一聊 Linux 是如何启动的。

当计算机电源通电后，BIOS会进行开机自检(Power-On-Self-Test, POST)，对硬件进行检测和初始化。因为操作系统的启动会使用到磁盘、屏幕、键盘、鼠标等设备。下一步，磁盘中的第一个分区，也被称为 MBR(Master Boot Record) 主引导记录，被读入到一个固定的内存区域并执行。这个分区中有一个非常小的，只有 512 字节的程序。程序从磁盘中调入 boot 独立程序，boot 程序将自身复制到高位地址的内存从而为操作系统释放低位地址的内存。

复制完成后，boot 程序读取启动设备的根目录。boot 程序要理解文件系统和目录格式。然后 boot 程序被调入内核，把控制权移交给内核。直到这里，boot 完成了它的工作。系统内核开始运行。

内核启动代码是使用汇编语言完成的，主要包括创建内核堆栈、识别 CPU 类型、计算内存、禁用中断、启动内存管理单元等，然后调用 C 语言的 main 函数执行操作系统部分。

这部分也会做很多事情，首先会分配一个消息缓冲区来存放调试出现的问题，调试信息会写入缓冲区。如果调试出现错误，这些信息可以通过诊断程序调出来。

然后操作系统会进行自动配置，检测设备，加载配置文件，被检测设备如果做出响应，就会被添加到已链接的设备表中，如果没有相应，就归为未连接直接忽略。

配置完所有硬件后，接下来要做的就是仔细手工处理进程0，设置其堆栈，然后运行它，执行初始化、配置时钟、挂载文件系统。创建 init 进程(进程 1 ) 和 守护进程(进程 2)。

init 进程会检测它的标志以确定它是否为单用户还是多用户服务。在前一种情况中，它会调用 fork 函数创建一个 shell 进程，并且等待这个进程结束。后一种情况调用 fork 函数创建一个运行系统初始化的 shell 脚本（即 /etc/rc）的进程，这个进程可以进行文件系统一致性检测、挂载文件系统、开启守护进程等。

然后 /etc/rc 这个进程会从 /etc/ttys 中读取数据，/etc/ttys 列出了所有的终端和属性。对于每一个启用的终端，这个进程调用 fork 函数创建一个自身的副本，进行内部处理并运行一个名为 getty 的程序。

getty 程序会在终端上输入

login:

等待用户输入用户名，在输入用户名后，getty 程序结束，登陆程序 /bin/login 开始运行。login 程序需要输入密码，并与保存在 /etc/passwd 中的密码进行对比，如果输入正确，login 程序以用户 shell 程序替换自身，等待第一个命令。如果不正确，login 程序要求输入另一个用户名。

整个系统启动过程如下

