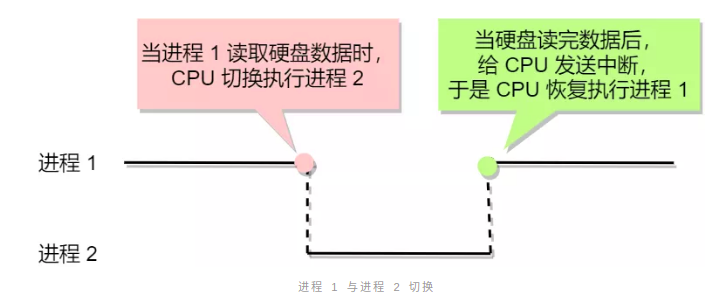


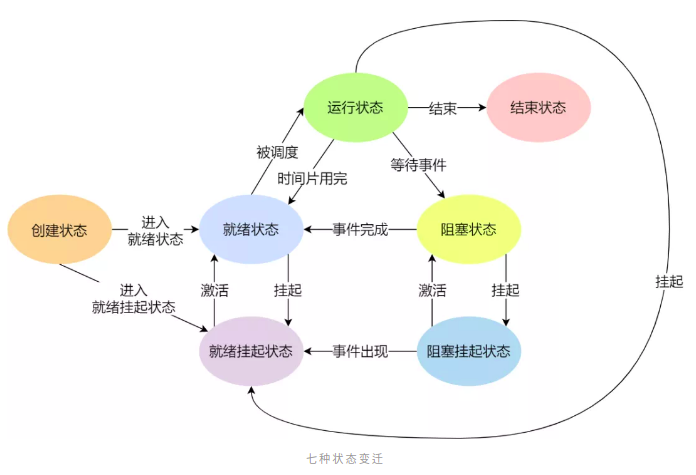
# 进程

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」。

当进程1中的某一条指令耗费的时间较长时, 如从磁盘中取数据, 那么cpu就会取执行另一进程2, 当进程1数据取完后, 再来执行进程1, 这就是进程间的上下文切换.



## 进程的状态



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。

## 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 具体包含什么信息呢？

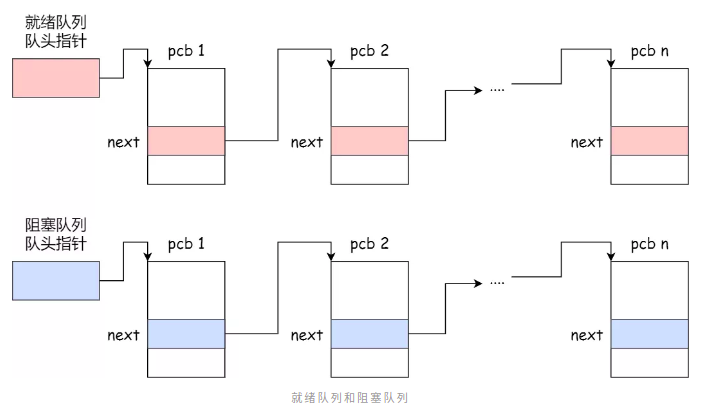
* 进程描述信息：
* 进程标识符：标识各个进程，每个进程都有一个并且唯一的标识符；
* 用户标识符：进程归属的用户，用户标识符主要为共享和保护服务；
* 进程控制和管理信息：
* 进程当前状态，如 new、ready、running、waiting 或 blocked 等；
* 进程优先级：进程抢占 CPU 时的优先级；
* 资源分配清单：
* 有关内存地址空间或虚拟地址空间的信息，所打开文件的列表和所使用的 I/O 设备信息。
* CPU 相关信息：
* CPU 中各个寄存器的值，当进程被切换时，CPU 的状态信息都会被保存在相应的 PCB 中，以便进程重新执行时，能从断点处继续执行。

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；
* 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。

## CPU上下文含义

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

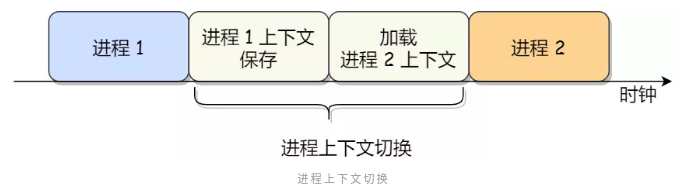
进程的上下文切换到底是切换什么呢？

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。

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？

* 公平调度算法时, 分配的cpu时间片用完了
* 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)

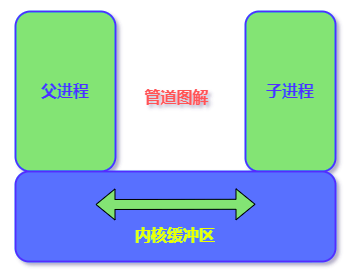
### **常见进程间通信方式**

#### 管道

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。



#### **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。



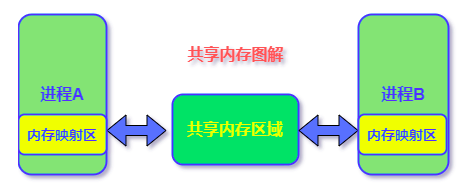
#### **信号 Signal**

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。

#### **共享内存 Shared memory**

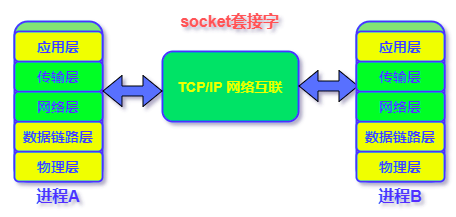
共享内存使得多个进程可以可以直接读写同一块内存空间，是最快的可用 IPC 形式，是针对其他通信机制运行效率较低而设计的。共享内存往往与其他通信机制，如信号量配合使用，来实现进程间的同步和互斥通信。



#### **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？

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，需要额外的共享内存或者套接字资源
* 维护进程的系统开销较大，

如创建进程时，建立 PCB, 分配资源；

终止进程时，回收资源、撤销 PCB；

进程切换时，保存当前进程的上下文；

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；

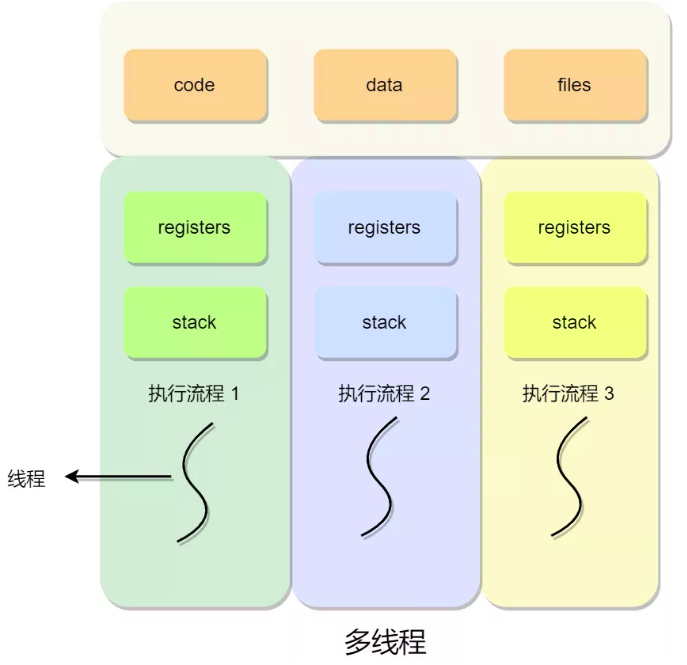
实体之间共享相同的地址空间；

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(共享变量)、数据段、打开的文件(减少额外的进程上下文切换)等资源，但每个线程都有独立一套的寄存器和栈，方便线程上下文切换。



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

**对于，线程相比进程能减少开销，体现在：**

* 线程的所涉及的资源较少, 所以创建和释放较快
* 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），切换的是同一个页表中的消息。而对于进程之间的切换，需要切换页表
* 由于同一进程的各线程间共享内存和文件资源，那么就不需要共享内存作为数据的中转站了

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

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，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；

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；
* 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, 而且加入没有内核线程的参与, 用户线程无法被中断.
* 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；

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；

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；

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。

## 多线程模型

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**

#### **读写锁**

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, 读锁也称共享锁

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, 写锁也称自旋锁。

#### **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、多协程模型

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。

### **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。

# 4、调度

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

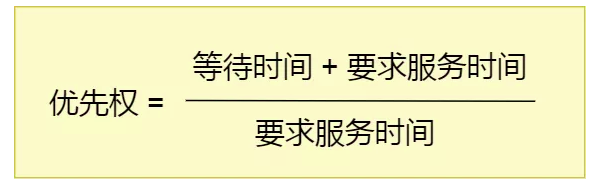
FCFS 对长作业有利, cpu利用率高

02 最短作业优先调度算法

最短作业优先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。

03 高响应比优先调度算法

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。



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。

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 利用率；
* 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

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。