# 虚拟化CPU

## 概述

虽然机器只有少量的物理CPU可用，但是操作系统可用提供几乎有无数个CPU可用的假象。操作系统通过虚拟化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，这样操作系统就提供了存在多个虚拟CPU的假象，这就是时分共享CPU技术，允许用户运行多个并发进程。这样做的潜在开销就是性能损失，因为如果CPU必须共享，那么每个进程都必须慢一点。

## 进程

操作系统为正在运行的程序提供的抽象，就是所谓的进程。一个进程就是一个正在运行的程序。

进程的机器状态包含两个部分：一个是内存，另一个是寄存器。

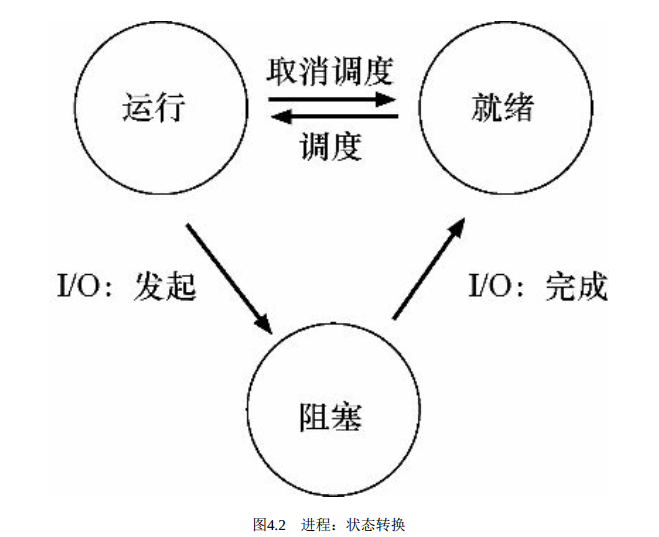
### 进程的创建

操作系统运行程序的第一件事就是将代码和所有静态数据加载到内存中，然后必须为程序的运行时栈分配一些内存，栈中可能会存放一些局部变量、函数参数和返回地址等等。然后操作系统也可能为程序的堆分配一些内存，C程序中，堆用于显示请求的动态分配数据，使用malloc()这样的函数请求内存，使用free()这样的函数释放内存。另外还会进行一些其他初始化任务，特别是输入/输出，默认情况下每个进程都有3个打开的文件描述符，用于标准输入、输出和错误。经过这些任务后，就可以启动程序，在入口处(main函数)开始运行。

### 进程的状态

进程可用处于以下3种状态：

1. 运行(running): 在运行状态下，进程正在CPU上运行，这意味着它正在执行指令；
2. 就绪(ready): 在就绪状态下，进程已准备好运行，但由于某种原因，操作系统不选择在此时运行；
3. 阻塞：在阻塞状态下，一个进程执行了某种操作，直到发生其他事件才会准备运行。例如发起IO请求时，它会被阻塞。

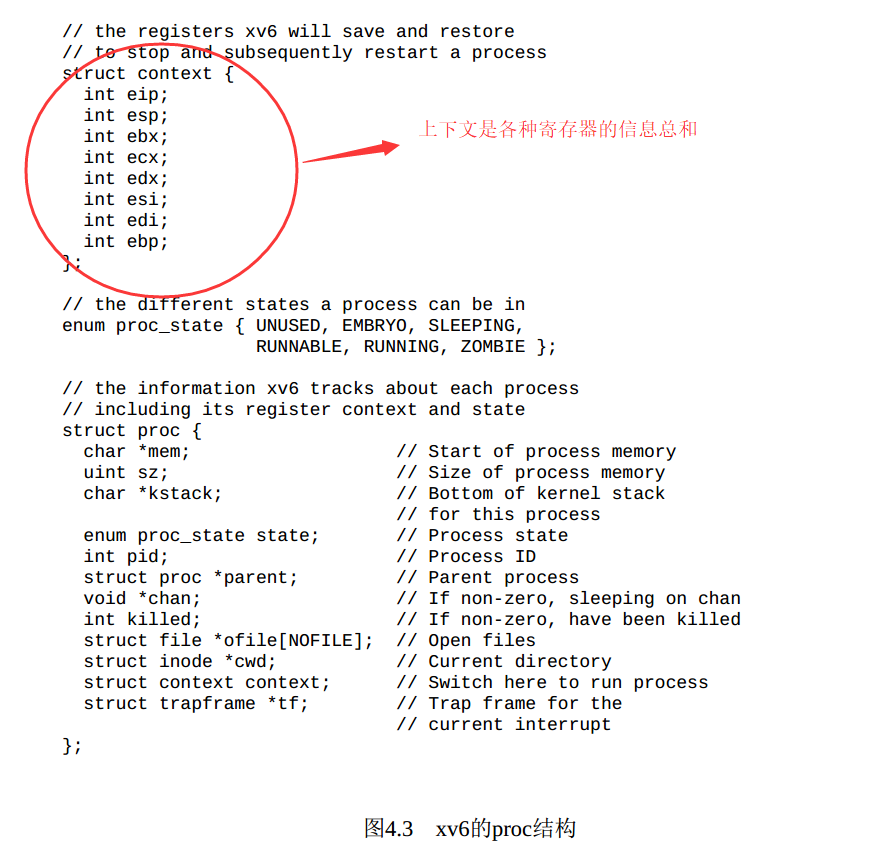


注意：只有就绪状态下的进程才会参与调度。

### 进程的数据结构

操作系统会使用一些关键的数据结构来跟踪进程。操作系统会追踪进程的一些重要信息，例如对停止的进程，寄存器上下文将保存其寄存器的内容。当需要恢复程序的运行时，操作系统又将这些保存的寄存器内容还原到寄存器中。当程序从阻塞状态恢复时，操作系统必须能够更改进程的状态等等。

下面是xv6内核的进程信息。



### 受限制执行

进程的执行并不是不受限制的，它在不同模式下访问硬件资源的权限不一样。操作系统提供了两种模式，如下。

用户模式：在这个模式下，应用程序不能完全访问硬件资源，例如不能发起IO操作；

内核模式：在这个模式下，操作系统可以访问机器的全部资源。

程序一般在用户模式下工作，如果程序中希望执行某种特权操作，它需要进行系统调用它允许内核向用户程序暴露某些关键功能，例如访问文件系统、创建和销毁进程、与其他进程通信，以及分配更多内存。

执行系统调用时，会发起一个内中断(在8086CPU中应该是执行了int或者into指令)，也叫软中断，中断过程会保存当前的寄存器的值到栈中，然后根据中断类型找到需要执行的中断代码的入口地址，然后执行，执行完后弹出栈中的寄存器的值。在执行中断代码过程中，会将程序的模式改为内核模式，这样程序可以执行特殊的操作指令，在执行操作完毕后，会将程序的模式再次修改为用户模式。

### 进程切换

操作系统通过“时钟中断”的方式管理进程，让每个进程都能够得到运行的机会。时钟设备可以编程为每隔几毫秒产生一次中断(应该是外中断)，产生中断后，操作系统会执行中断程序，这时操作系统就可以重新获得CPU的控制权来控制当前进程停止，并启动另一个进程，

停止当前进程，启动另一个进程就是上下文切换。操作系统会执行一些底层汇编代码，来保存通用寄存器(AX,BX,CX,DX),程序计数器(CS:IP),栈指针(SS:SP)等等各种寄存器的值(各个寄存器的值组成了程序的上下文)，然后恢复下一个程序的之前保存在内存中的各个寄存器的值到各个寄存器中。CPU根据恢复后的CS:IP的值(CS:IP保存着下一个指令的地址)自然就可以继续运行下一个程序的代码。

## 进程调度

### 调度指标一：周转时间

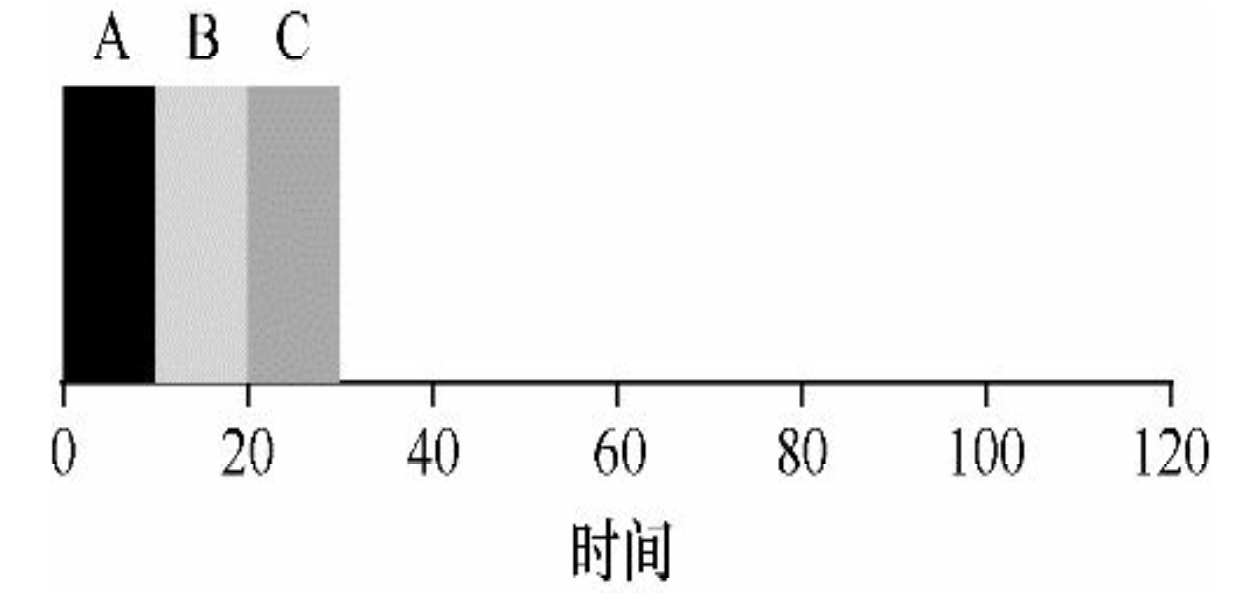
周转时间是用来衡量任务完成快慢的，平均周转时间越短的操作系统，任务完成得越快。

**一个任务的周转时间=任务的完成时间点-任务的到达时间点**

### FIFO: 先进先出调度

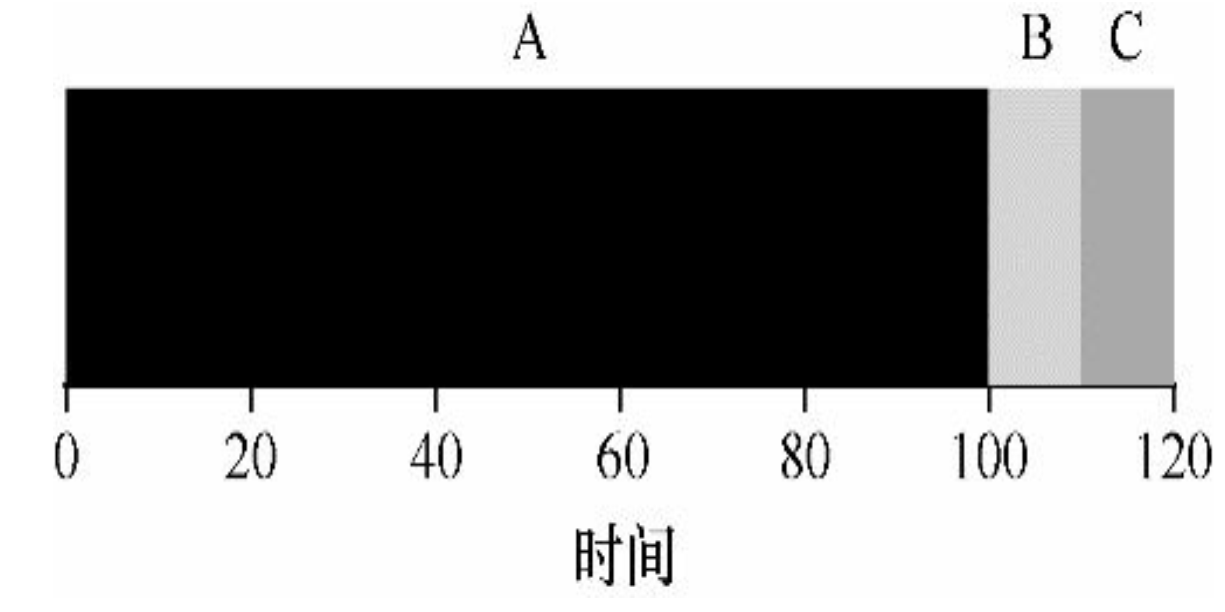
这时一个最基本，也最简单的调度算法，下面看使用这种调度算法的周转时间。

假设一：有三个工作A，B和C。它们都需要运行10s，A，B，C几乎同时都是0时间点到达，A比B快，B比C快。那么它们的完成时间点如图：



这时，三个任务的平均周转时间为(10+20+30)/3=20s。

假设二：有三个工作A,B和C。A需要运行100s，B和C需要运行10s，A，B，C几乎同时在0时间点到达，A比B快，B比C快。这时三个任务的平均周转时间为：(100+110+120)/3=110s。

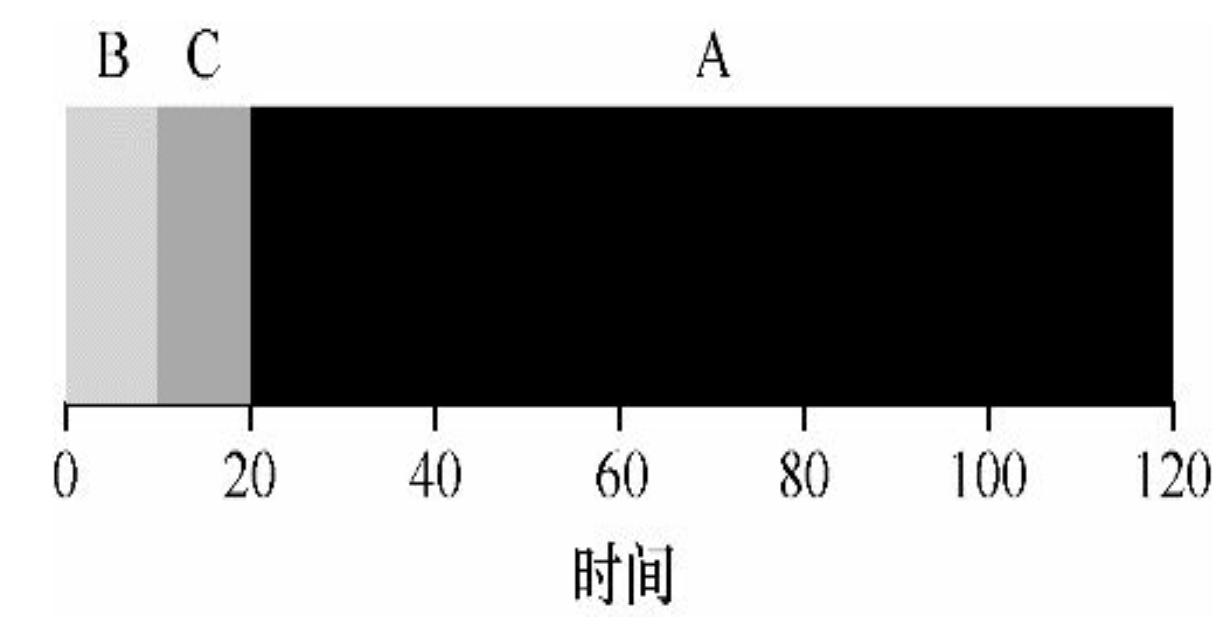


从这可以看出，一个长时间的任务会导致其他任务的等待时间变得很长了，从而系统整体周转时间变长。

### SJF：最短任务优先

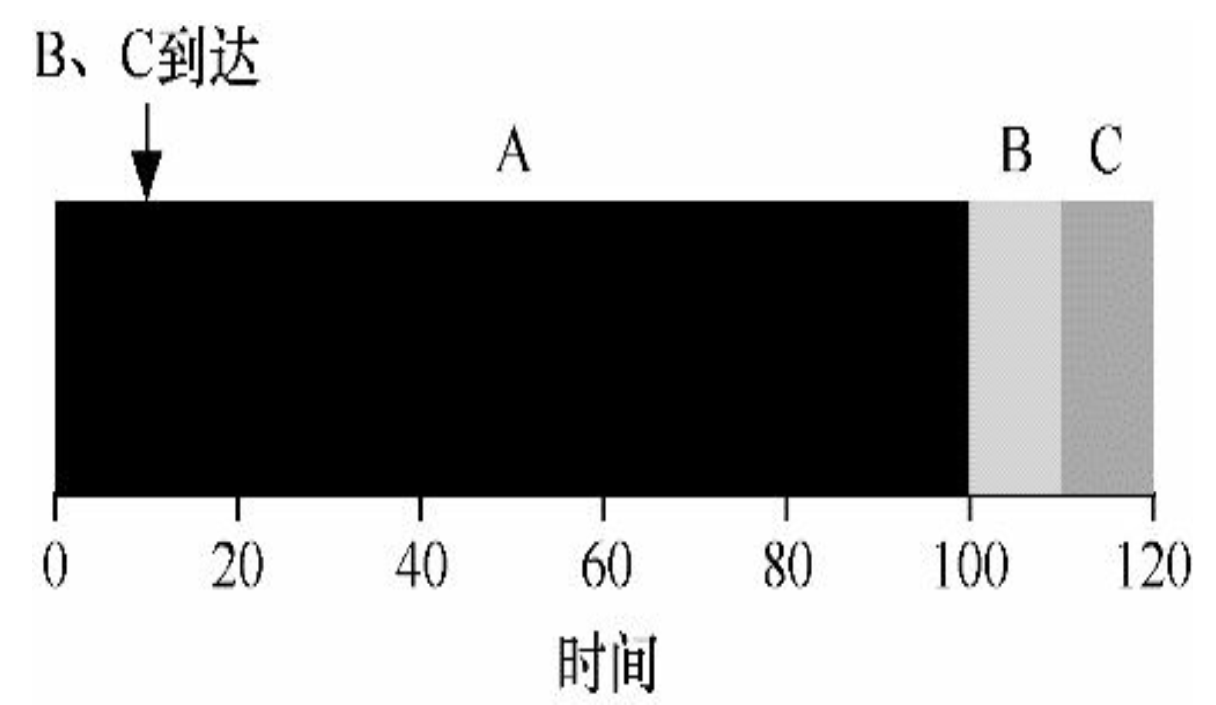
最短任务优先调度原则是：先运行最短的任务，然后是次短的任务，依次类推。

对于假设二的各个工作情况，如果采用SJF调度策略，那么运行情况就会如下：



三个工作的平均周转时间为：(10+20+120)/3=50s，比FIFO调度策略要快很多。

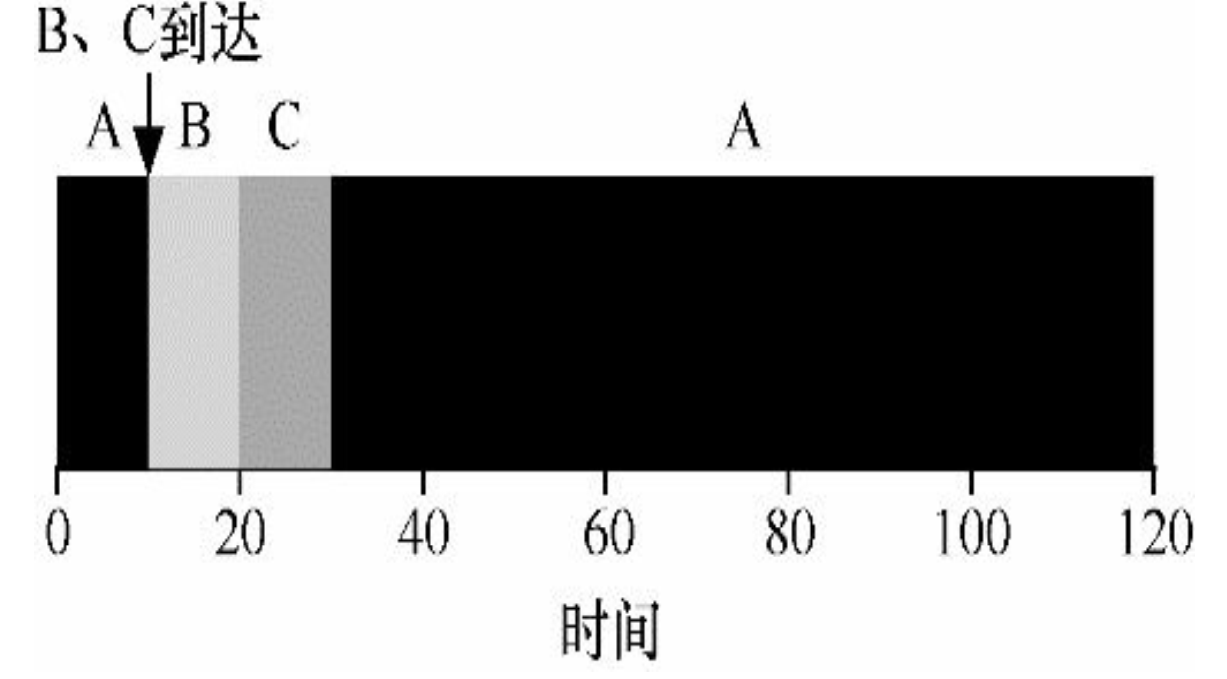
假设三：有三个工作任务A,B,C，A需要运行100s，B和C需要运行10s，A在时间点0到达，B和C在时间点10到达。那么采用SJF的策略后运行的情况如下:



可以看出，这时候SJF调度策略并不生效，B和C仍然要等待A执行完毕。

### STCF: 最短完成时间优先

STCF调度策略中郧西抢占其他工作，每当有新工作进入系统时，它就会确定剩余工作和新工作中，谁的剩余时间最少，然后调度该工作。，那么对于假设三的情况，使用STCF调度策略后，运行情况如下：



工作的平均周转时间为：(120+10+20)/3=50s。比采用SJF策略要好很多。

### 调度指标二：响应时间

对于只衡量周转时间来看，STCF策略是一个很好的策略。但是在实际使用中，我们还需要考虑另一个指标：响应时间。因为用户需要尽可能短时间内得到反馈。

**响应时间=任务首次运行的时间点-任务到达的时间点**

例如：如果我们有A,B,C三个工作任务都需要处理10s，A在0时间点到达，B和C在10时间点到达，那么它们的平均响应时间为：(0+0+10)/3=3.3s。

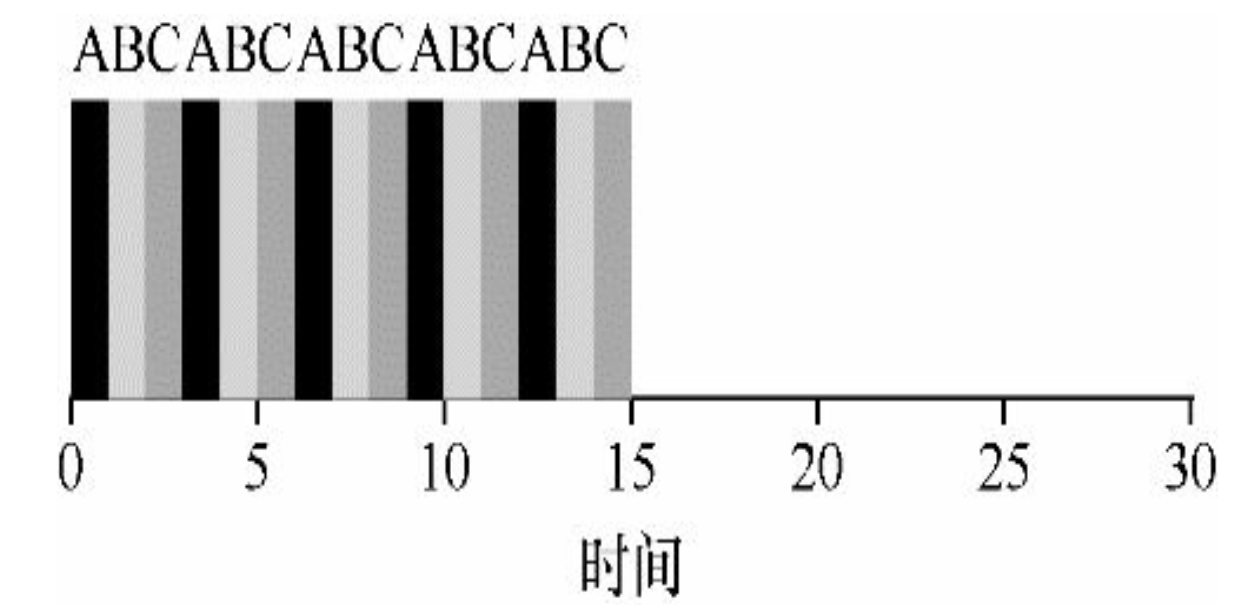
对于这个指标，STCF表现并不是很好，例如如果三个工作任务同时达到，那么第三个工作必须等待前两个工作全部运行后才能运行(第一种假设情况)，这会导致假设你在终端输入，不得不等待10s或者20s才能看到系统的响应。

### RR: 轮转

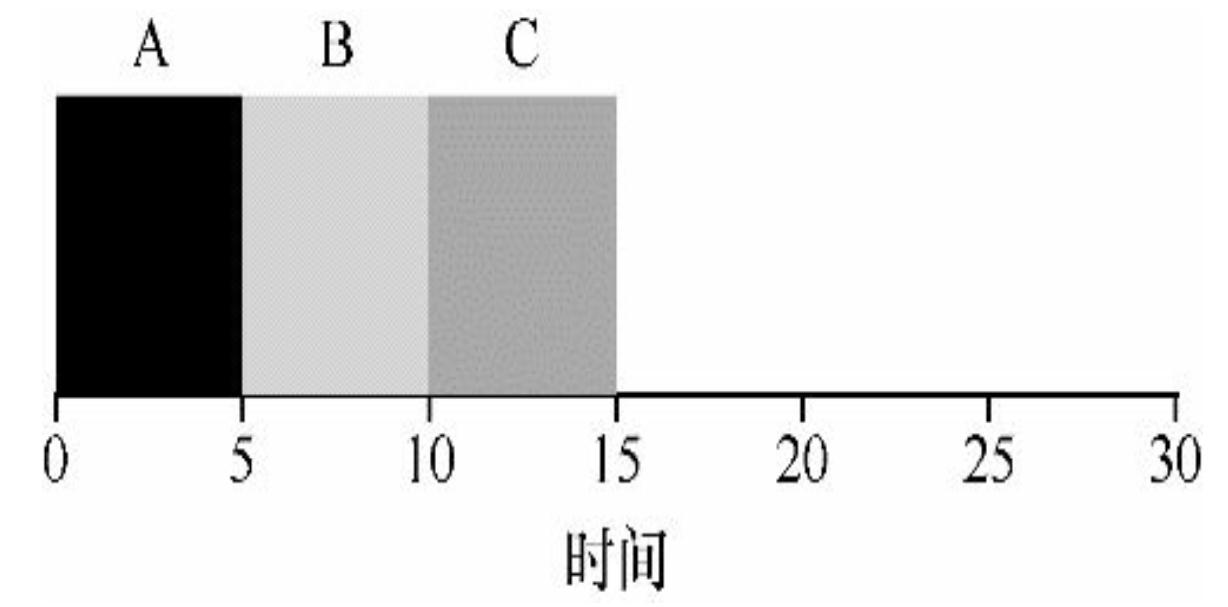
为了得到更好的响应时间，需要新的调度算法：RR

RR在一个时间片内运行一个工作，时间片结束会切换到队列中的下一个任务，而不是运行一个任务直到结束。注意：时间片长度必须是时钟中断周期的倍数，例如时间中断是每隔10ms中断一次，那么时间片可以是10ms、20ms和30ms等等。

假设四：有三个工作任务A、B、C，它们都需要运行5s，它们都是在0时间点到达。这里假设时间片为1s。那么它们的运行情况如下:



如果是SJF，那么它们的运行情况如下：



使用RR策略的平均响应时间是：(0+1+2)/3=1; 使用SJF策略的平均响应时间是:

(0+5+10)/3=5。

所以，RR这种调度策略可以使平均响应时间得到极大提高。

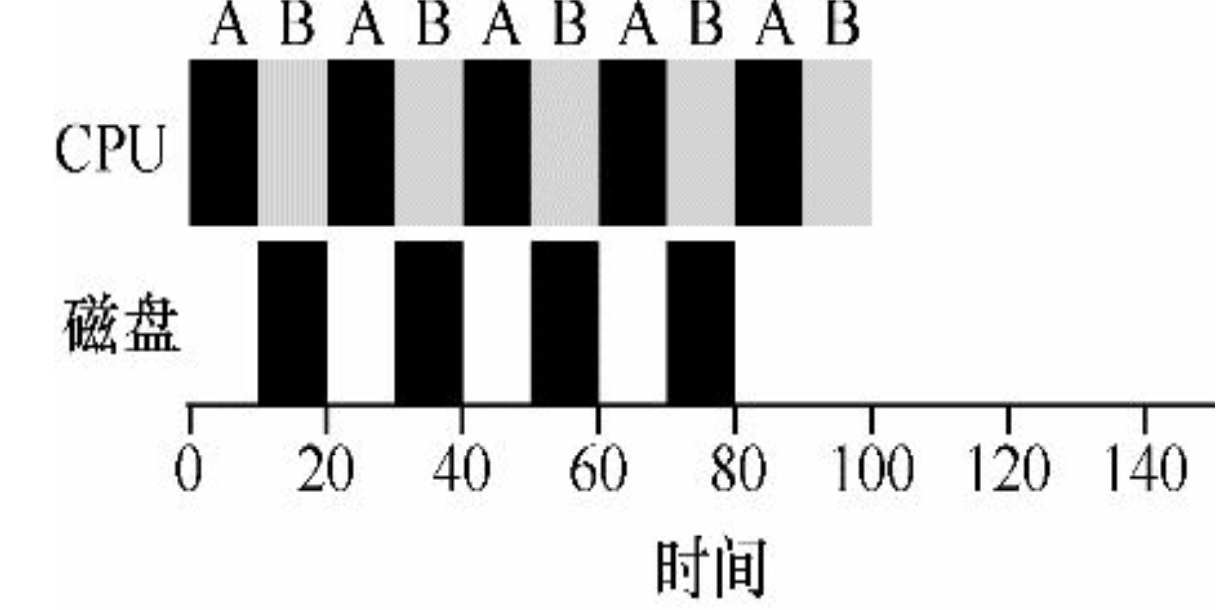
但是对于平均周转时间,使用RR策略的平均周转时间是: (13+14+15)/3=14s; 使用SJF策略的平均周转时间是: (5+10+15)/3=10s。

从平均周转时间来看,RR策略并不会比SJF这种策略好。这是可以理解的，需要根据实际情况进行权衡。

### 有IO的处理

假设有两个工作任务A和B，每项工作都需要50ms的工作时间。但是有个区别:A运行10ms，然后发出I/O请求(假设I/O每个都需要10ms),而B只是使用了CPU 50ms，不执行I/O。

在有I/O的情况下，调度策略不会把A工作任务的整体看作一个任务，而是把A任务中的每个10ms看作一个单独的任务。这样处理后，如果是使用STCF调度策略，运行情况如下:



A的第一个10ms作为一个任务和B比较，10ms更短，所以会优先执行A的第一个10ms任务，执行后，A会进行I/O。此时，运行中的任务就只剩下B，所以B得以运行。当A从I/O中恢复后，A的第二个10ms任务和B比较，10ms更短，所以运行A的第二个10ms任务，运行完后进入I/O……依次类推就得到如上运行状况图。

### 工作任务时长无法预知的处理

在实际情况中，调度程序是不知道每隔工作任务实际会运行多长时间的，所以还需要进一步的调度策略。

### MLFQ: 多级反馈队列

多级反馈队列需要解决两个问题：首先，它要优化周转时间；其次，MLFQ希望用户能够得到更好的交互体验，因此需要降低响应时间。前面提到的算法，比如SJF或者STCF算法是通过执行短工作来降低周转时间，但是实际上操作系统并不知道任务要运行多久，另外还有RR算法是通过不断切换工作任务来降低响应时间，但是周转时间却不好。多级反馈队列可以弥补这些算法的不足。

MLFQ中有许多独立的队列，每个队列有不同的优先级。任何时刻，一个工作只能存在于一个队列中。MLFQ总是优先执行较高优先级的工作。

MLFQ的关键工作在于如何设置优先级。MLFQ会根据工作的运行情况，不断调整它的优先级。例如，如果一个工作不断放弃CPU而等待键盘输入，这是交互型进程的可能行为，MLFQ因此会让它保持高优先级。相反，如果一个工作长时间占用CPU，MLFQ会降低其优先级。这种方式主要是为了优先执行短时间工作从而提高周转时间的目标。

通过上面的描述，得出MLFQ的几条基本规则:

规则1：如果A的优先级> B的优先级，运行A；

规则2：如果A的优先级=B的优先级，轮转运行A和B；

规则3：工作进入系统时，放在最高优先级；

规则4：工作用完整个时间片后，降低其优先级(移入下一个队列)；

规则5: 如果工作在其时间片以内主动释放CPU，则优先级不变。

通过以上几点规则，一个长工作时间的任务的运行情况如下：



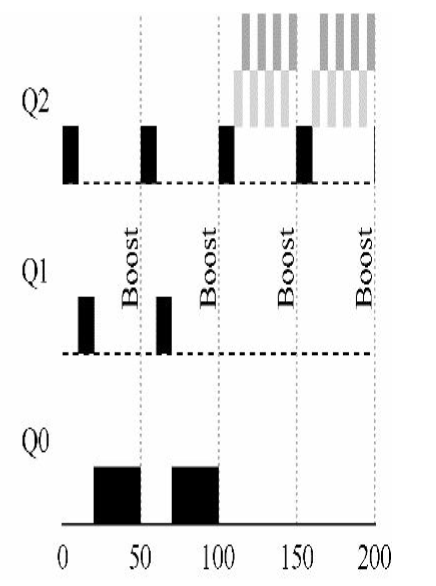
这个例子中，该工作首先被放入了最高优先级，执行完一个时间片之后，调度程序将工作的优先级减1，因此进入Q1，在Q1执行完一个时间片之后，最终进入最低优先级队列中，然后一直留在那里。

目前的规则中存在一个问题，有可能会有饥饿问题产生，如果系统中有太多交互型工作，那么就会不断占用CPU，导致长时间工作永远无法得到CPU。

所以需要添加一个规则：

规则6： 经过一段时间S，就将系统中所有工作重新加入到最高优先级队列。

通过这个规则，如果系统中有一个交互型的任务1(用灰色表示)，有一个需要长时间占用CPU的任务2(用黑色表示)。

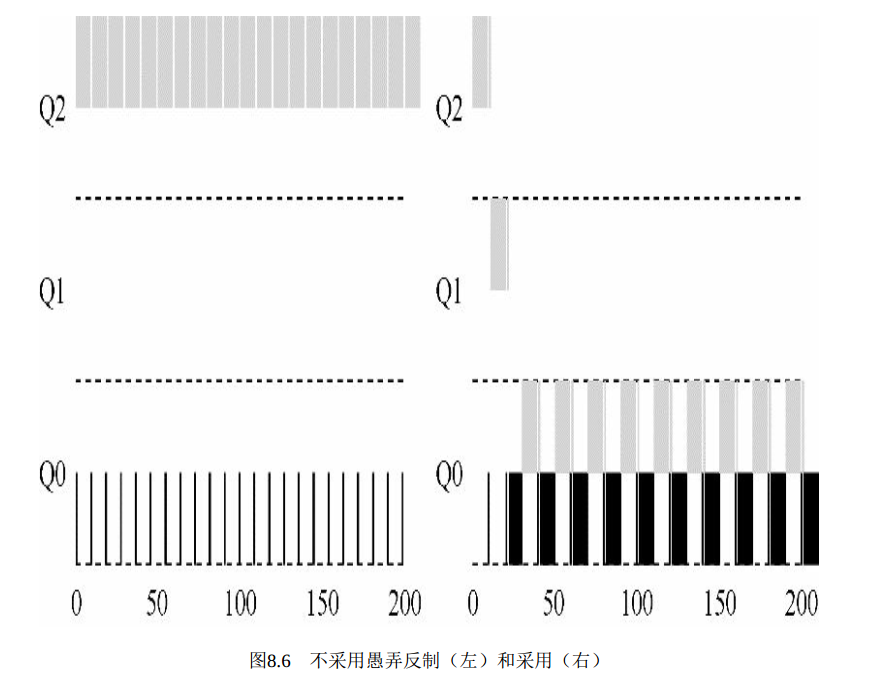


还有一个问题，如果编写的程序故意在时间片快用完之前放弃CPU，那么极端情况下，这个程序可能会得到大部分的CPU使用权。

为了解决这个问题，调度程序应该记录一个进程在某一层级消耗的总时间，而不是在调度时重新计时，只要进程用完了自己的配额，就将它降到低一优先级的队列中去。所以修改规则4和规则5，如下：

规则4: 一旦工作用完了其在某一层中的时间配额(无论中间主动放弃了多少次CPU)，就降低其优先级(移入低一级队列)。

下图对比了在原有规则下和在新规则下如果存在愚弄程序(在时间片快结束时主动放弃CPU以保留优先级)，它可以在时间片快结束时发起一个IO操作，这个工作任务用灰色表示，另外还有一个正常长时间工作程序，这个工作任务用黑色表示，它们的运行状况如下：



通过这些规则，基本能够保证调度程序正常使用。

**这些规则总结如下：**

规则1： 如果A的优先级>B的优先级，运行A；

规则2： 如果A的优先级=B的优先级，轮转运行B；

规则3： 工作进入系统时，放在最高优先级；

规则4： 一旦工作用完其在某一层中的时间配额(无论中间主动放弃了多少次CPU)，就降低其优先级(移入下一级队列);

规则5: 经过一段时间S，就将系统中所有工作重新加入最高优先级队列。

## 多处理器调度

### 缓存一致性问题

在单CPU系统中，存在多级的硬件缓存，一般来说会让处理器更快地执行程序。程序第一次读取数据时，处理器判断该数据很可能会被再次使用，因此将其放入CPU缓存中，之后再次需要使用同样的数据时，CPU会先查找缓存，这样取数据会快很多。

但是这在多CPU系统时会出现问题。CPU1上的程序从内存地址A读取了数据，将其放入到缓存中，然后接着又修改了这个数据，将它的缓存更新为新值D，不过并没有立即更新内存中的值，它会在一段时间后更新这个值，此时时间片用完，CPU1中断了这个程序的执行。等到下一次这个程序执行时，CPU2执行了这个程序，但是此时它从内存地址A中读取了数据，但是这时读取的数据是旧值，而不是正确的更新后的值。

硬件提供了解决这个问题的基本方案：通过监控内存的访问，硬件可以保证获得正确的数据。在基于总线的系统中，每个缓存都通过监听连接所有缓存和内存的总线，来发现内存访问。如果CPU发现对它放在缓存中的数据的更新，会作废本地副本，或更新它。

### 缓存亲和度

一个进程在某个CPU上运行时，会在该CPU的缓存中维护许多状态。下次该进程在相同CPU上运行时，由于缓存中的数据而执行得更快。相反，在不同的CPU上执行，会由于需要重新加载数据而很慢。因此多处理器调度应该考虑到这种缓存亲和性，并尽可能将进程保持在同一个CPU上。

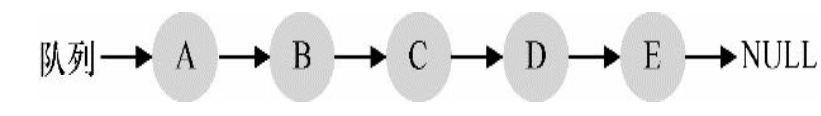
### 单队列调度

多CPU调度的一个最简单的方式就是将所有需要调度的工作都放入一个单独的队列，我们称之为单队列多处理器调度。

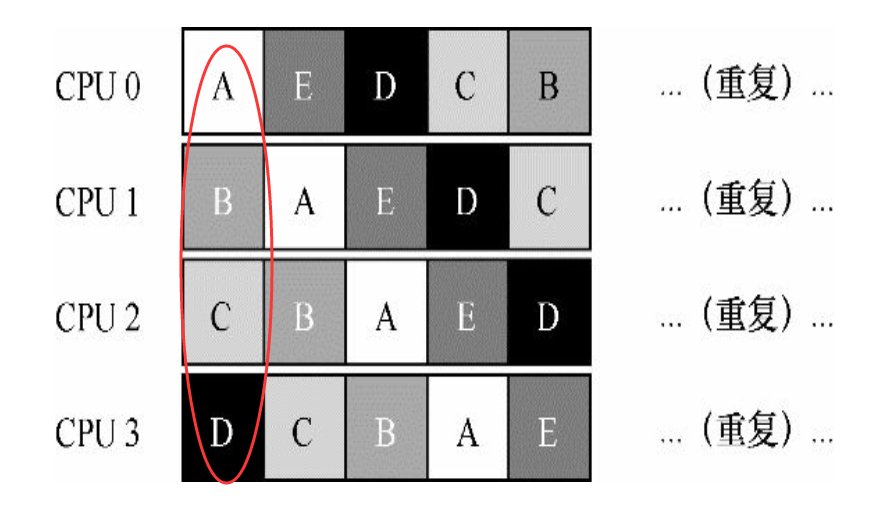
单队列调度最大的优势就是简单，但它有短板，第一个就是调度程序需要在代码中加锁来保证访问单个队列时不会冲突(例如寻找下一个运行的工作)。

第二个问题就是缓存亲和性问题，它不能保证相同的任务每次运行在相同的CPU上。

例如调度队列如下：



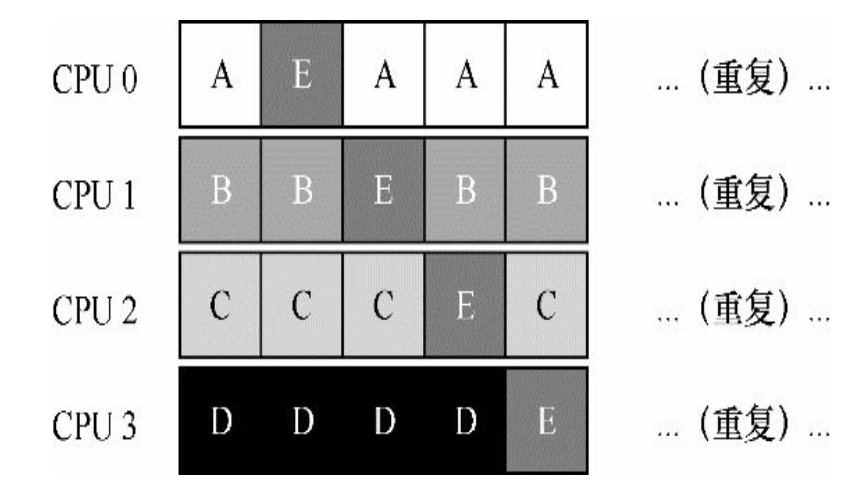
假设有4个CPU，那么它们运行这些任务的运行情况如下：



每个CPU都不断地从队列中拉取任务来执行，但是这导致了相同的任务每次都在不同的CPU上执行，缓存亲和性得不到保证。

为了解决这个问题，一般单调度队列的调度程序会添加额外的机制来保证缓存亲和性。

为了保证一些工作的亲和性，就必须牺牲另一些工作的亲和性。



例如上图的调度程序中，牺牲了E工作的缓存亲和性，换来了A，B，C和D的缓存亲和性。

### 多队列调度

在多队列调度策略中，会存在多个队列(比如每个CPU都有一个队列)，当工作进入系统后，系统会依照一些启发性规则将其放入调度队列中，这样一来，每隔CPU调度之间相互独立，就避免了单队列的方式由于数据共享及同步带来的问题。

例如，系统有两个CPU，有4个工作任务，它们存放在不同的CPU中，如下：

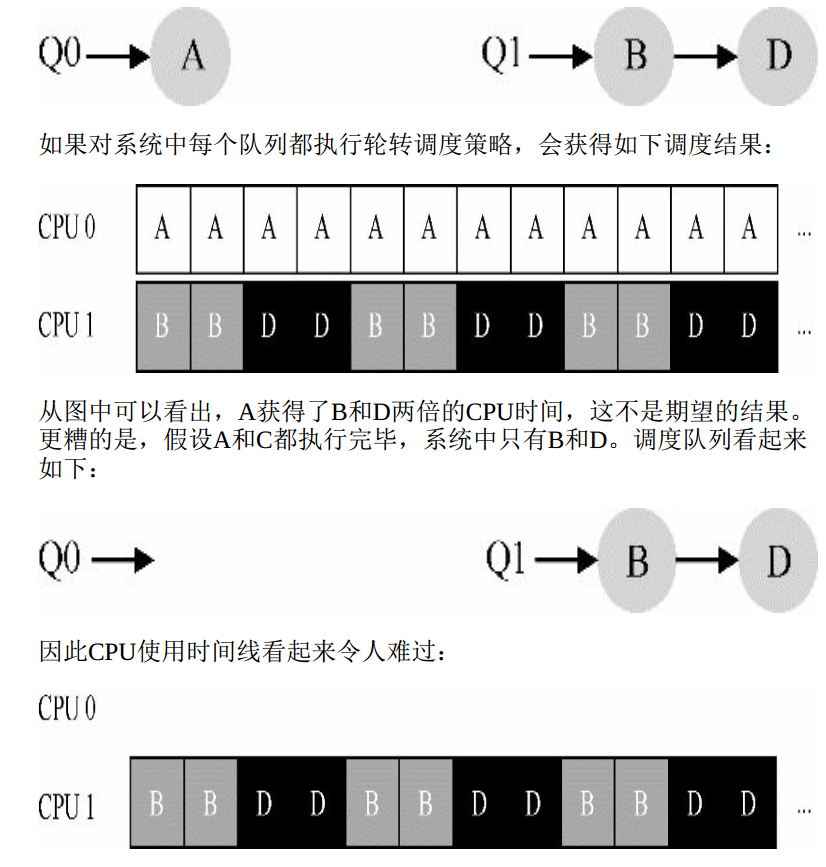


不同CPU运行自己的调度策略，运行状况如下：



这种策略天生就具有良好的缓存亲和度，同时还不会有锁的问题。

但是它也有缺点，假设某个CPU上它的任务执行完毕了，另一个CPU上的任务还需要很长的时间才能结束，那么在接下来的过程中，前一个CPU也不会帮着另一个CPU执行任务。



为了能够解决这个问题，那么最好的办法就是让工作能够移动。实现这个办法可以采用一个算法：**工作窃取算法**。在这个算法中，工作量较少的CPU会不定期查看其他的CPU的工作量，如果目标CPU的队列更满，那么就从目标CPU的队列中窃取一个或多个工作，实现负载均衡。

# 虚拟化内存

## 地址转换

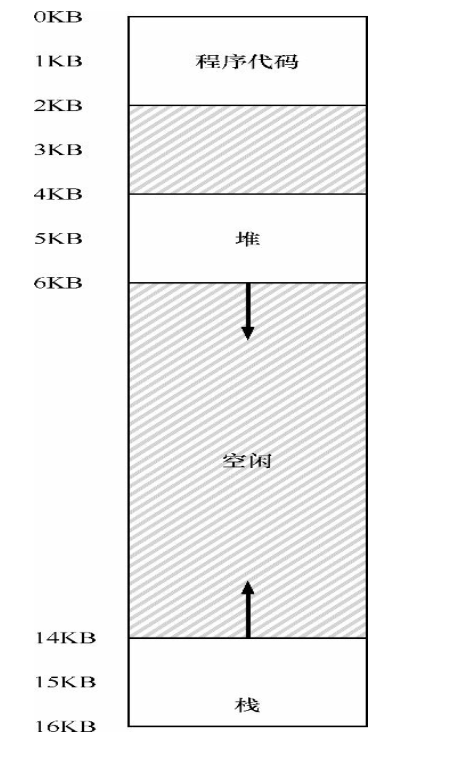
为了能够虚拟化内存，需要使用到地址转换技术。利用地址转换，硬件对每次内存访问进行处理，将指令中的虚拟地址转换为数据实际存储的物理地址。通过这个技术，每个程序都可以拥有私有的内存。

为了能够转换虚拟地址，每个CPU需要两个硬件寄存器：基址寄存器和界限寄存器(类似于CS:IP这种)。采用这种方式，在编写和编译程序时假设地址空间从零开始。但是，当程序真正执行时，操作系统会决定其在物理内存中的实际加载地址，并将起始地址记录在基址寄存器中。所以虚拟地址会通过以下方式转换为物理地址：

physical address = virtual address + base。

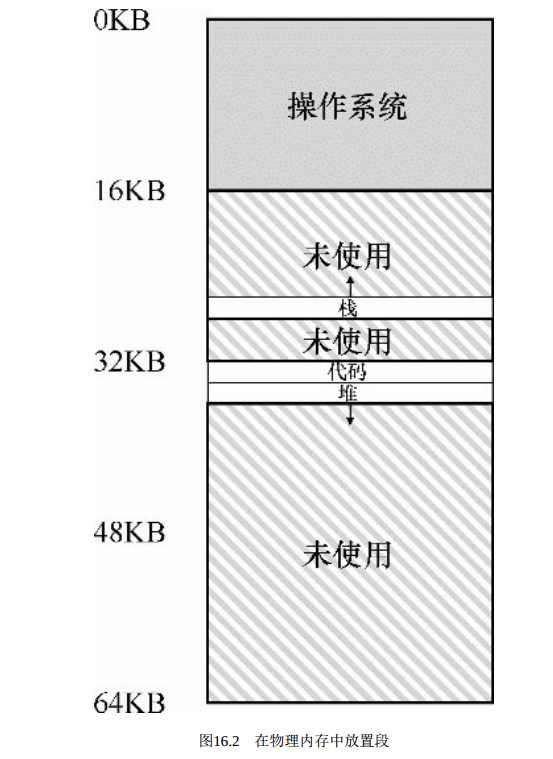
## 分段

利用地址转换，操作系统很容易将不同进程重定位到不同的物理内存区域。但是只是通过这样的方式并不够。



如果进程的所有地址空间都是统一放在一片连续的内存中，那么可能会出现有一大段空闲空间不会被使用。

为了解决这个问题，出现了分段的概念。硬件也支持了这种做法，出现了多个段寄存器:CS,DS,SS，可以分别作为代码，堆，栈的段寄存器。



通过分段，进程的各个不同部分的地址空间可以存放在不同的位置。

不过分段也有一些问题。问题一：由于段的大小不同，空闲内存被割裂成各种奇怪的大小，因此当系统运行一段时间之后，会产生很多碎片，满足内存分配请求可能会很难。问题二：分段还不足以支持稀疏地址空间，如果有一个很大但是稀疏的堆，都在一个逻辑段中，整个堆仍然必须完整地加载到内存中。

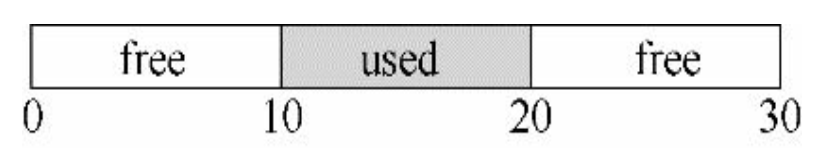
## 空闲空间管理

管理空闲空间的主要目的是需要让碎片最小化。

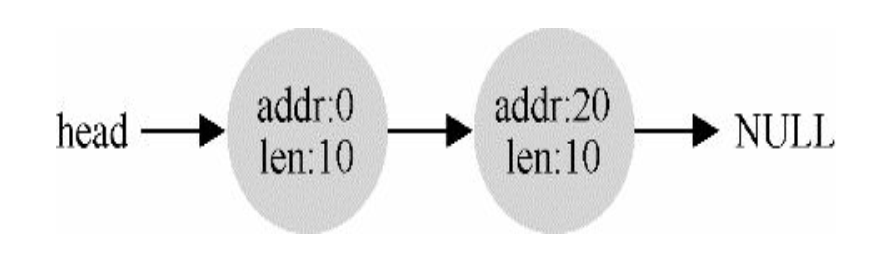
**简单的例子：**

管理空闲空间的数据结构通常称为空闲列表。该结构包含了管理内存区域中所有空闲块的引用。

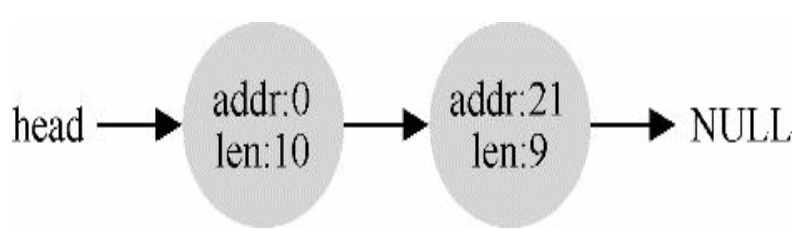
假设有以下空间：



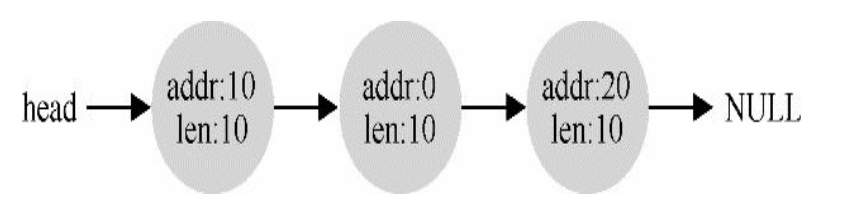
那么空闲列表示意图如图：



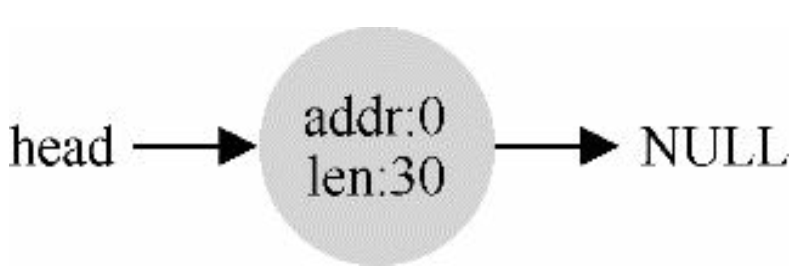
如果这时候申请一个字节的内存，那么分配程序会执行所谓的**分割**动作：它找到一块可以满足请求的空闲空间，将其分割，第一块返回给用户，第二块留在空闲列表中。这时候空闲列表示意图如图：



如果不是申请内存，而是归还了10字节的内存，那么可能得到如下的结果：



这时候，分配程序会堆空闲空间进行合并，合并的方式很简单：在归还一块空闲内存时，会仔细查看要归还内存块的地址以及邻近的空闲空间块。如果新归还的空间与一个原有空闲块相邻，就将它们合并为一个较大的空闲块。通过合并，最后空闲列表会变成如下：



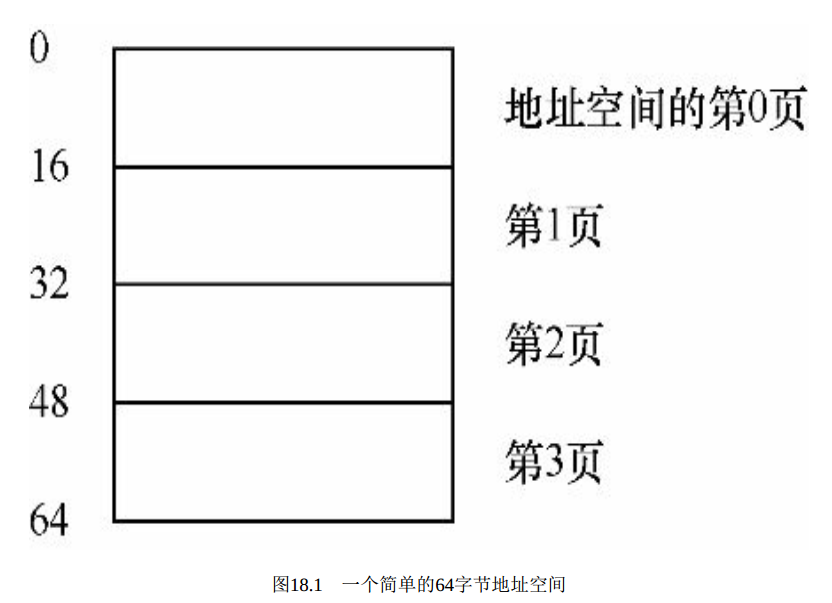
实际上，这时空闲列表最初的样子，通过合并和分割，分配程序可以更好地确保大块的空闲空间能提供给应用程序。

## 分页

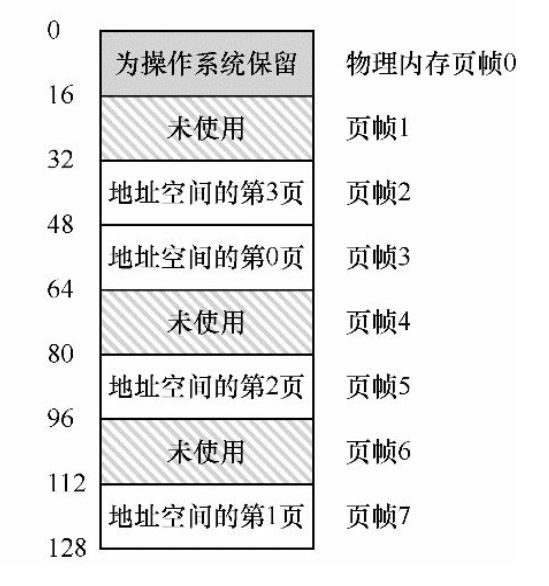
即使通过分割和合并，因为空间被分割成不同长度的分片，那么肯定会导致碎片化问题，系统在运行一段时间后，会出现很多小空闲空间散布在一些已分配的空间之间，这些空闲空间因为太小，所以很难被使用。

为了解决这个问题，可以考虑将空间分割成固定长度的分片。这样内存如果以固定长度大小的分片为单位来使用，那么所有的分片都能够随时被分配给应用程序使用，而无需担心出现碎片化的问题。这种管理方式叫做**分页**。

如图展示了一个只有64字节的小地址空间，有4个16字节的页。



另外，物理内存也由一组固定大小的槽快组成。如图：

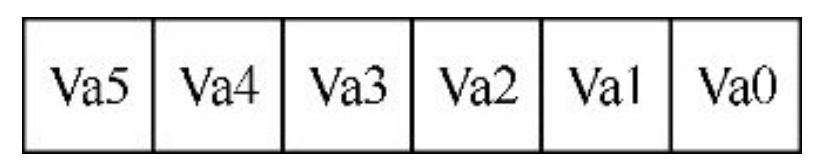


从图中可以看出，进程的地址空间可以存放在物理内存中的任何位置，虽然它们在物理上不连续，但是对于进程而言，它们在虚拟地址空间上是连续的，这是一个对内存很好的抽象。

### 页表

为了记录地址空间的每个虚拟页放在物理内存中的位置。操作系统通常为每个进程保存了一个数据结构，称为**页表**。页表的主要作用是为地址空间的每个虚拟页面保存地址转换，从而让我们知道每个页在物理内存中的位置。

为了转换虚拟地址，虚拟地址需要包含两个部分：**虚拟地址页面号(VPN)**和页面的偏移量。假设进程的虚拟地址空间是64字节，那么为了表示所有的地址，所以虚拟地址需要6位。如下：



因为页的大小是16字节，所以可以进一步划分：



地址的前2位是虚拟页号，告诉是哪一页，后4为是偏移量，指明在页中的偏移量。

例如下面的汇编命令：

movl 21,%eax

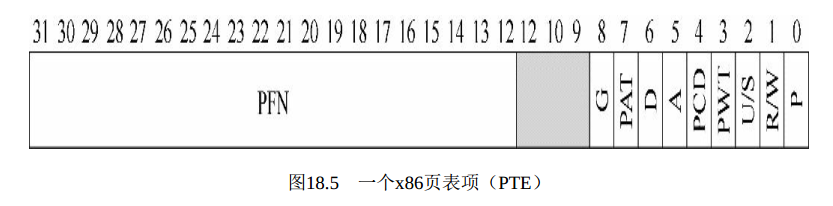
将”21”变成了二进制形式，是“010101”：



所以虚拟地址”21”是在虚拟页01的第5个字节处。

### 页表内容

页表就是一种数据结构，用于将虚拟地址映射到物理地址，因此，任何数据结构都可以采用，最简单的形式称为线性页表，就是一个数组。操作系统通过虚拟页号(VPN)检索该数组，并在该索引处查找页表项(PTE)，以便找到期望的物理帧号(PFN)。



每个PTE的内存除了PFN之外，还有一些其他位。

有效位：通常用于指示特定地址转换是否有效；

保护位：表明页是否可以读取、写入或执行；

存在位：表示该页在物理存储器还是在磁盘上；

参考位：也叫访问位，用于追踪页是否被访问，也用于确定哪些页很受欢迎，因此应该保留在内存中；

以及一些其他位…

### 页表的问题

虽然页表是对内存很好的抽象，但是它也有一个问题：增加了一次内存访问。因为有了页表，所有访问内存的操作都得先访问内存中的页表得到物理地址，然后才能访问真正的地址。

### 快速地址转换(TLB)

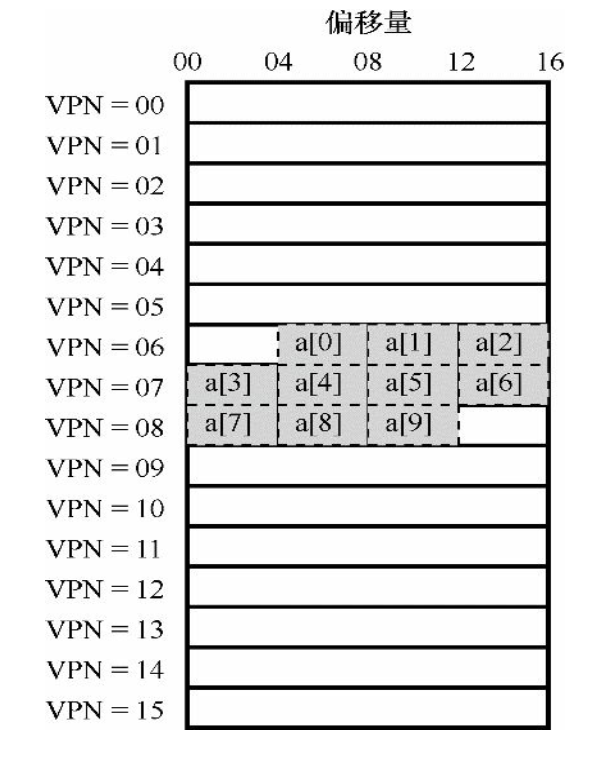
使用了分页策略虚拟化内存之后，访问内存的操作都额外增加了一次内存访问页表的操作，所以访问内存就变得慢了起来。为了能够加快内存访问，需要用到硬件缓存(TLB)。有了TLB，对于每次内存访问，硬件就先检查TLB，看看其中是否有期望的转换映射，如果有，就完成转换，不用访问页表。

TLB算法的大致流程如下：首先从虚拟地址中提取页号(VPN)，然后检查TLB是否有该VPN的转换映射。如果有，那么TLB就命中，接着就从相关的TLB中取出页帧号(PFN)，与原来虚拟地址中的偏移量组合形成期望的物理地址(PA)，并访问内存。如果没有在TLB中找到映射关系，那么就需要访问页表找到映射关系，并更新TLB，更新成功后，系统重新尝试指令的运行，然后就会在TLB中找到映射关系。

如果TLB没有命中，那么引起的开销更大，所以应该尽量避免TLB未命中。

示例：

假设有一个8位的小虚拟地址空间，页大小位16字节，所以虚拟地址可以分为4位的VPN和4位的偏移量。另外假设有一个10个4字节整形数组成的数组，起始的虚拟地址是100，那么它在虚拟地址空间的位置如下：



现在考虑一个简单的循环操作：



那么当访问第一个数组元素a[0]时，CPU会先根据虚拟地址的VPN检查TLB，发现未命中，所以这次会访问内存中的页表更新TLB。

当访问第二个数组元素a[1]时，这时虚拟地址的VPN和a[0]一样，查找TLB时，命中。

依次类推，访问数组中的所有元素，会出现3次未命中，7次命中，命中率为70%。

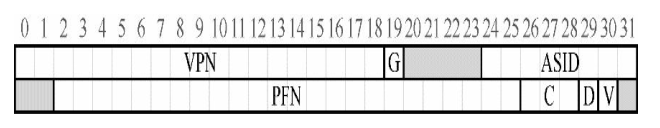
通过这个例子能看出来，TLB提高了此程序的内存访问速度。另外注意，此例子中页大小很小，如果页大小变大的话，命中率可以进一步提高。

### TLB的内容

TLB是由很多条数据组成的，一条TLB项的内容示意图如下：



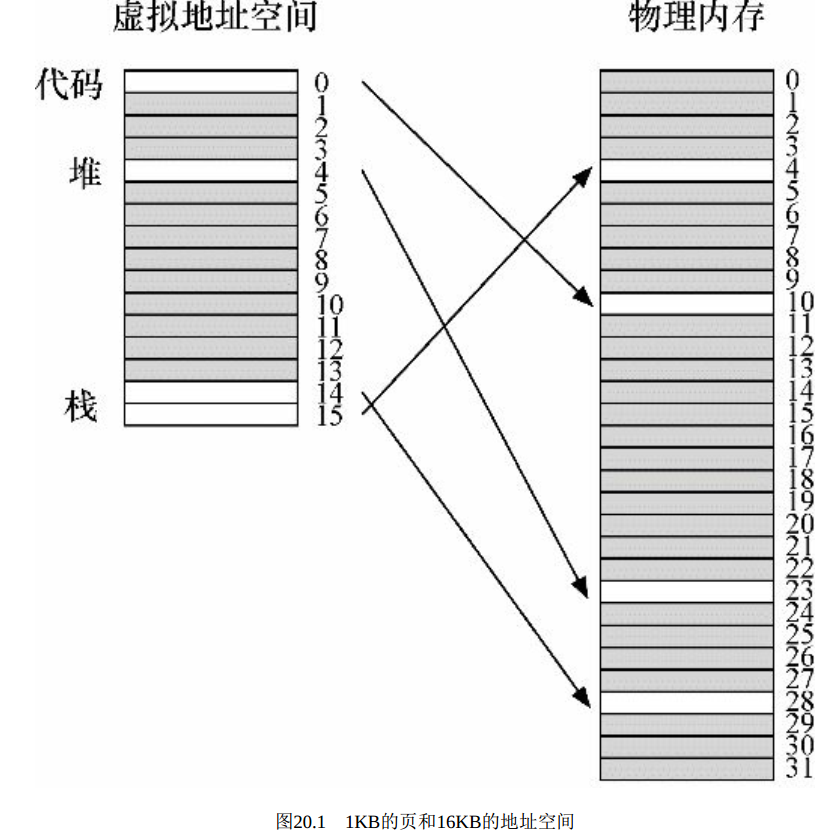
一个实际的TLB表项如下：



### 多级页表

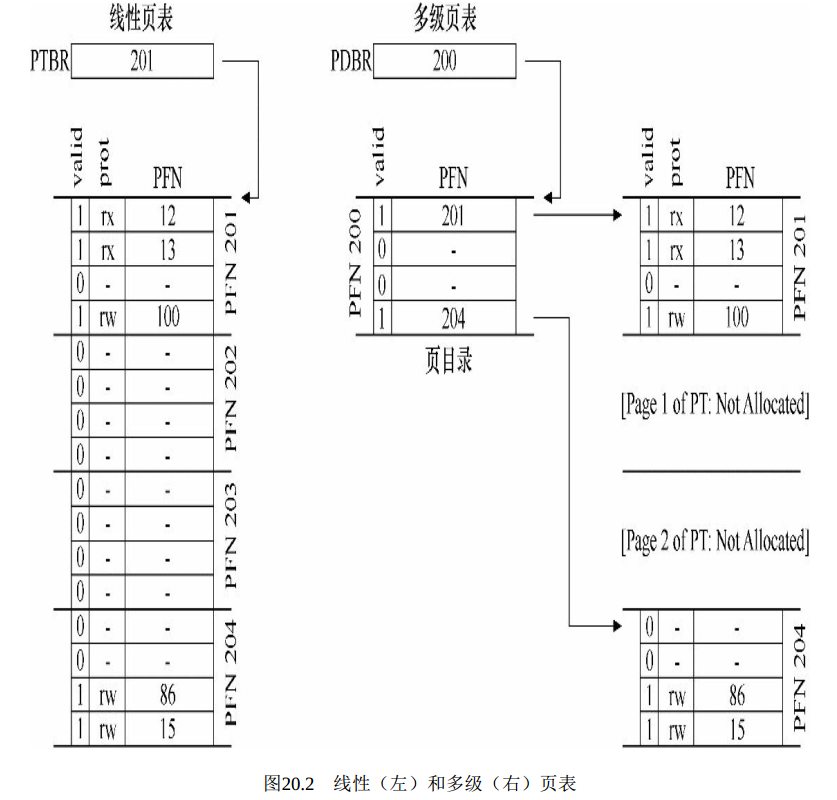
引入分页的第二个问题就是：页表太大。假设一个32位的地址空间，一页的大小时4KB，那么为了表示所有的地址就需要2^20次方张页，所以一个页表就需要存储1百多万的页表项，假设一个页表项为4字节，那么一个页表的大小为4MB。因为一个进程需要有一个页表，所以如果有1百个进程，那么光是页表，就需要占掉400MB的内存。

首先来看下页表的实际使用情况是否存在浪费.假设我们有一个地址空间,其中堆和栈的使用部分很小.



我们可以看到,其实页表中大部分的页表项都没有被使用，这存在了巨大的浪费，而多级页表就可以解决这个问题。

多级页表的思想很简单：将页表以一定的长度再次分页(一般以页大小作为分隔的长度)；然后使用一个页目录结构记录这些页表上的页在哪个位置，另外如果页表上整页的页表项都没有被使用，那么就完全不分配该页的页表。

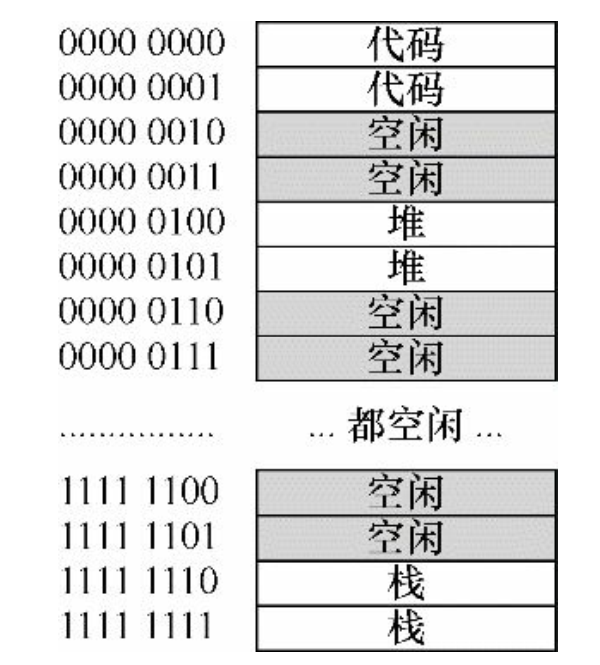


通过图可以看到，202和203页上的页表项都没有被使用，页目录仅将页表的两页标记为有效(第一个和最后一个);因此，页表的这两页就驻留在内存中。通过页目录对页表再次分页的方式可以让原始的线性页表的一部分不用驻留在内存中。

页目录的使用例子：

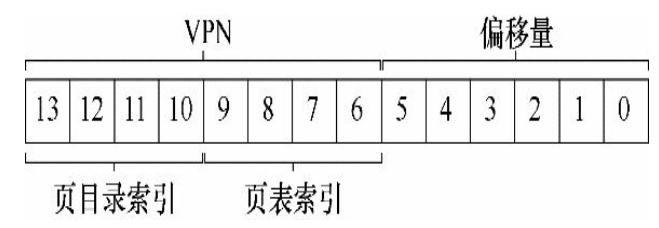
假设一个大小为16KB的小地址空间，其中包含64字节的页。因此，我们有一个14位的虚拟地址，VPN为8位，偏移量为6位。

如果是原始的线性页表，那么它会有256项页表项，假设虚拟页0和1用于代码，虚拟页4和5用于堆，虚拟页254和255用于栈，那么页表如下：



如果使用多级页表，假设每个页表项为4个字节，那么页表的大小为1024B，我们按页的大小对页表项再次分页，所以1024/64=16，我们对页表再次以16个页表项为单位再次分页，又分成了16个页，组成了页目录。

如果使用原始线性页表，那么虚拟地址是8位VPN，6位偏移量。如果使用多级页表，因为页目录有16个，所以需要将VPN分出4位来索引目录：

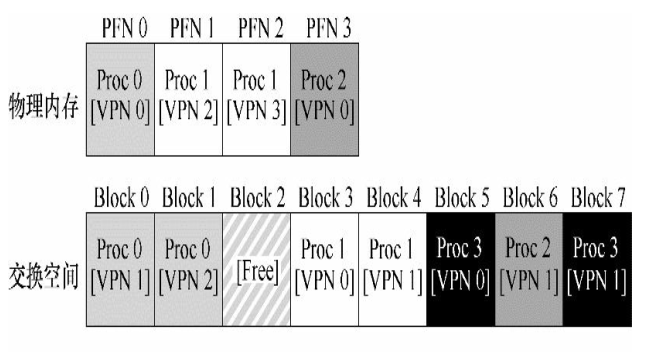


所以对于一个具体的虚拟地址，可以先通过前4位在页目录中查找是否有对应的页目录项，如果查找到了，再根据后面4位到页目录项对应的页表中查找具体的页表项，再根据页表项查找物理页帧号，最后根据偏移量找到具体的物理内存地址。这个过程对比原来的物理内存地址查找方式多了一步查找页目录项，不过利用TLB可以缓存对应关系，所以只要命中率高，就不会有过多损耗。

## 超越物理内存

### 交换空间

操作系统不会把所有的页都存放在物理内存中(注意这里的页是物理页，不是虚拟页)，一般会把一些暂时不用的页存放在硬盘上。硬盘上常常会开辟一部分空间用于物理页的移入和移出，这样的空间叫做交换空间。



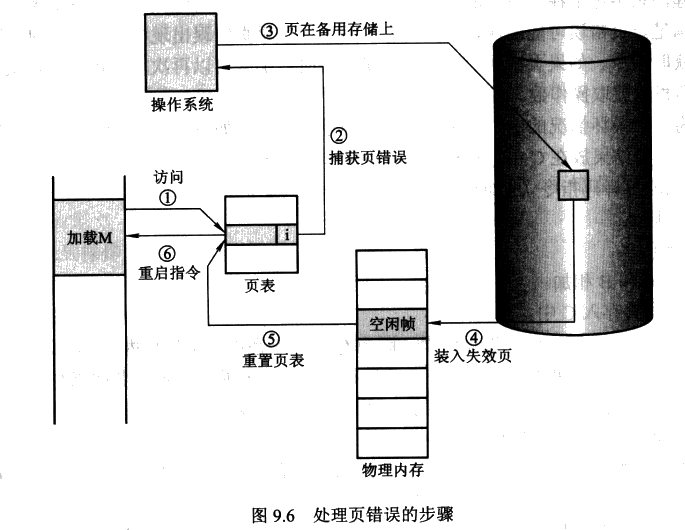
如上图，有4个进程，前三个进程的物理页在物理内存和交换空间中都存在，而第4个进程的物理页全部存在于交换空间中。

### 存在位

根据虚拟地址查找物理内存地址时，会现在TLB中查找，如果没找到，会查找页表，根据虚拟地址中的VPN查找页表项(PTE)，如果从PTE中找到物理页(PFN)后，将其插入TLB中，然后重试该指令。在PTE中有一个位是存在位，如果存在位值为1，那么该页存在于物理内存中，如果值为0，那么该页存在于磁盘中。根据VPN查找PTE时，可能会发现存在位值为0，该页并不在物理内存中，这个时候就会触发页错误操作。

### 页错误

PTE中还有某些位是用来存储此页的硬盘地址的，以在发生页错误时，从硬盘中将此页重新加载到内存中，当加载完成后，会更新页表，将此页标记为存在，然后再重试指令，此时根据VPN查找PTE时就可以找到对应的PFN地址了。



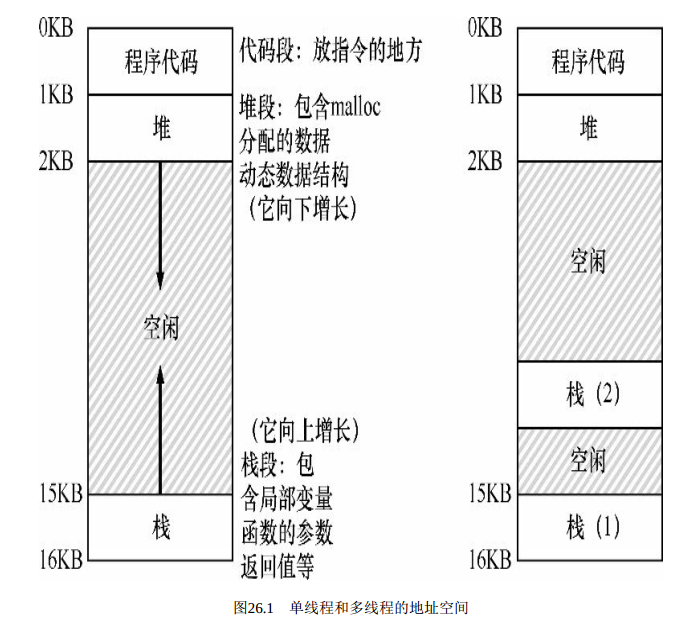
### 换出

除了有将页从硬盘加载到内存中的操作外，操作系统还支持将一些不常用的页换出到交换空间中的操作。一般来说，操作系统会设置高水位线(HW)和低水位线(LW)来帮助决定何时从内存中清除页。当操作系统发现有少于LW个页可用时，后台负责释放内存的线程会开始运行，直到有HW个可用的物理页。

# 并发

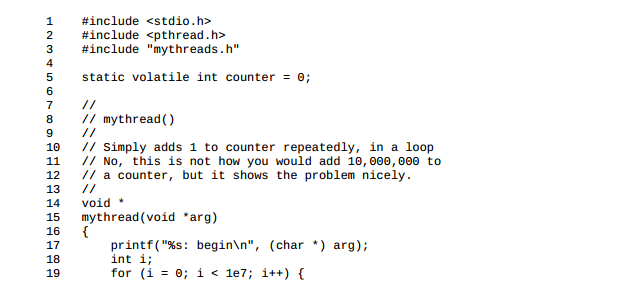
## 线程

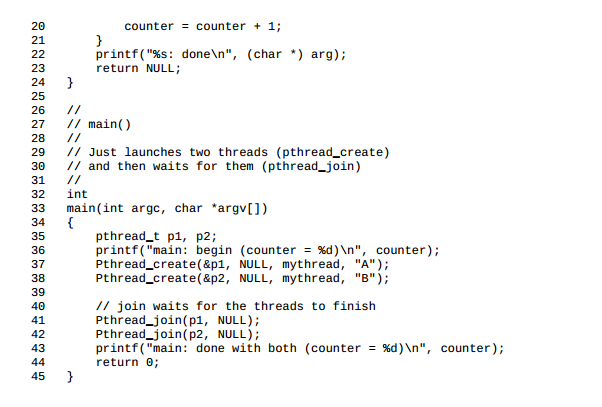
现在的进程都是多个线程组成的。每个线程类似于独立的进程，但是同一个进程中的线程，它们共享地址空间。线程的上下文切换和进程的上下文切换也差不多，不过线程切换不会切换地址空间(不需要切换使用的页表)。另外，每个线程都有单独的栈。



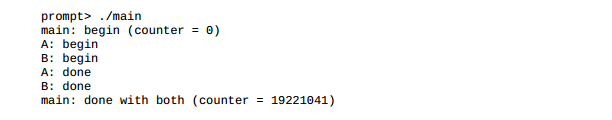
## 多线程共享数据的问题

有下面的程序：有两个线程同时会对共享数据count进行递增1000万次，预期的结果是2000万。





不过实际的结果可以如下：



问题原因：counter = counter + 1这段代码翻译成汇编如下：



它是由三条指令组成的，先从内存中拉取数据到寄存器中，再将寄存器的值加1，然后再将寄存器的值放入内存中。

因为这三个指令并不是原子性操作，所以这时候由于操作系统对于线程的调度，可能在第一个线程刚做完第1步，或者第二步时发生了线程切换，切换到了另一个线程，另一个线程此时从内存中拉取的值还是原来的值，所以就出现了问题。

**竞态条件：**如果一个系统或进程的输出依赖于不受控制的事件出现顺序或者出现时机。例如如果两个线程竞争同一个资源，如果对资源的访问顺序敏感，就称存在竞态条件。

**临界区：**导致竞态条件的代码就叫做临界区。临界区是访问共享变量的代码片段，一定不能由多个线程同时执行。

**原子性期望：** 为了解决共享数据的问题，我们期望如果能够单步做完临界区的操作而不发生中断，那么就不会存在问题，这就是原子性。这需要操作系统和硬件提供支持。

## 多线程问题二：等待另一个线程

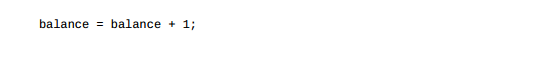
多线程除了在访问共享变量是会出现问题，还有一种场景会出现问题。一个线程经常会需要等待另一个线程完成某些操作后才继续往下执行。例如需要主线程需要等待多个子线程对数据加载处理完成后才开始继续下一步操作。

这个问题的解决也需要操作系统和硬件提供支持。

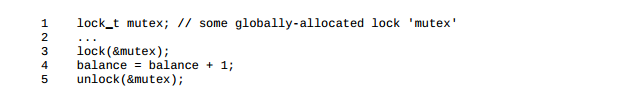
## 锁

### 概述

锁是为了解决访问共享数据问题而出现的解决方案。假设有下面的临界区代码：



这是对共享变量balance操作的代码，在多线程并发执行时会出错。下面使用锁对它加一些代码：



锁就是一个变量，这个锁变量保存了锁在某个时刻的状态。它如果是可用的，那么表示没有线程持有锁，它如果是被占用的，那么表示有一个线程持有锁。调用lock()会尝试获取锁，如果没有其他线程持有锁，该线程会获得锁，进入临界区，这时如果另外一个线程对相同的锁变量调用lock()时，因为锁被另一个线程占用 ，该调用不会返回。这样就保证了临界区代码只能由一个线程执行了。

**锁为程序员提供了最小程度的线程调度控制**。

### 锁的评价维度

如何实现一个锁的前提是确定标准判别锁是否真的可用。

第一： 锁是否能完成它的基本任务，互斥。完成了互斥，就能够阻止多个线程进入临界区；

第二： 公平性。每个线程是否有公平的机会抢到锁，即是否有竞争的线程会被饿死；

第三： 性能使用锁之后增加的时间开销。

### 锁的实现

#### 控制中断

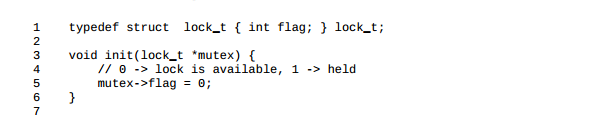
最早提供的互斥解决方案之一就是在临界区关闭中断。这个解决方案是为单处理器系统开发的。通过在进入临界区之前关闭中断，可以保证临界区的代码不会被中断，从而原子性地执行，结束之后，重新打开中断，程序就能正常执行。

这个方案的优点就是简单。但是缺点也很多，这个方案必须信任程序能够合理利用这一机制，如果一个程序在代码一开始就调用lock()关闭中断，那么它只要不调用unlock()重新打开中断，那么CPU的资源就会一致被它霸占。第二个缺点就是它不支持多处理器，如果多线程运行在不同的CPU上，关闭中断也没有意义，线程可以运行在其他处理器上。第三缺点就是，关闭中断可能导致中断丢失，如果磁盘设备完成了读取请求，发出中断让CPU处理，但是CPU无法被中断而错失了这个事实，那么之后操作系统就无法唤醒等待读取的进程了。

#### 测试并设置指令

这是硬件提供支持锁的指令。

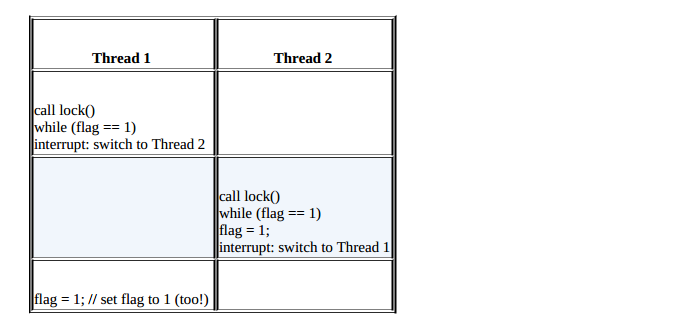
为了理解这个指令，可以先简单实现一个锁，查看下为什么硬件要提供这个指令。实现锁的想法很简单：用一个变量标志锁是否被某些线程占用。第一个线程进入临界区时，调用lock(),检查标志是否为1，如果不是1，那么锁没有被占用，可以进入临界区，然后将标志设置为1，表明线程持有锁。结束临界区时，线程调用unlock()，清除标志，表明锁未被持有。





当使用这个实现后，当第一个线程正处于临界区代码时，如果另一个线程调用lock()，它会在while循环中自旋等待，直到第一个线程调用unlock()清空标志。

但是这个实现会有问题，假设两个线程按照下面的顺序执行，那么会出现两个线程都能够进入临界区的情况。

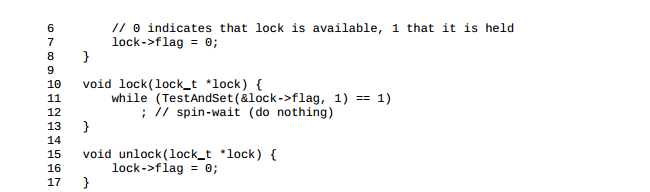


接下来使用TestAndSet指令实现一个锁。TestAndSet指令执行的内容用代码表示如下：



它会替换旧的值为新的值，然后返回旧的值，这是一个原子性的操作。现在使用这个指令更改前面的实现锁的代码如下：





当两个线程同时调用lock时，由于TestAndSet的原子性，就不会出现两个线程拿到的flag值都是0的情况，这样就只有一个线程能够进入到临界区中。

这样就利用硬件的支持实现了一个锁，下面来评价这个锁：

互斥： 能够满足互斥；

公平： 这个锁实现没有提供任何机制保证公平，所以可能出现某个线程饿死的情况；

性能： 这是锁是一个自旋锁，不会放弃CPU，所以会浪费CPU。

#### 比较并交换指令

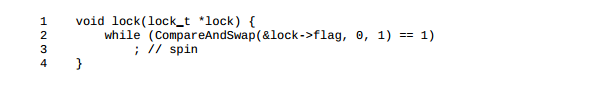
比较并交换指令用代码表示如下：





比较并交换的基本思路是：检测ptr的值是否和expected相等，如果相等，那么更新ptr的值，如果不相等，那么就什么也不做。这些操作后，返回ptr的旧值。

使用这个指令，lock()的实现如下：



由于CompareAndSwap指令的原子性，同样只会有一个线程进入临界区。

#### 获取并增加指令

FetchAndAdd指令：它能原子性地返回旧值，并让该值自增1。使用这个指令实现锁的代码如下：



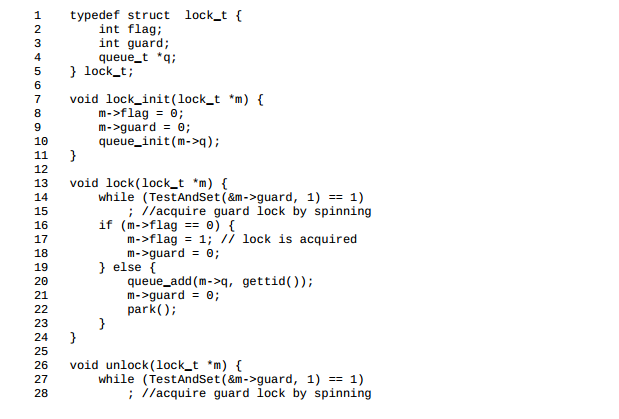
ticket可以让当前线程持有一个值，turn表示目前锁持有的值，如果线程持有的值和当前线程持有的值一样，那么就可以进入临界区。

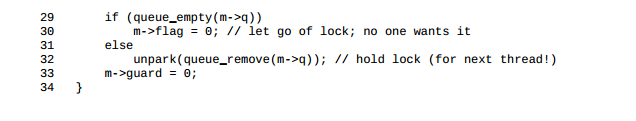
这个锁实现相比前两个锁实现保证了公平性：先等待的线程会先获得执行的机会。每有一个线程想要竞争锁时，FetchAndAdd指令就会让ticket的值分配给线程，然后让ticket的值自增1，每当线程执行完毕后，锁的turn值会自增1，如果线程的ticket值和turn值相等，那么线程就可以进入临界区。这样就相当于给等待的线程进行了排队操作。

#### 非自旋锁

利用硬件提供的三个原子性操作可以实现锁，但是它们都是自旋锁，如果锁不是很快就释放，那么会浪费锁。

操作系统提供了一些调用，能够让线程休眠和唤醒，例如Solaris系统，它提供了两个调用: park()和unpark()， pack能让调用线程休眠，unpack(threadID)则会唤醒threadID标识的线程。使用者两个调用结合硬件提供的原子性操作实现锁的代码如下：





这里有个问题，如果在即将调用pack()之前，发生了线程切换，然后切换到了刚好持有锁的线程，线程此时调用了unpack()，这时候再切换到之前的线程，那么这时候这个线程调用park()之前就会永久地睡眠。这种问题有时称为唤醒/等待竞争。

为了解决这个问题，操作系统提供了另外一个原语,separk()，通过separk()，一个线程表明自己马上要park，如果另一个线程刚好被调度，并且调用了unpark,那么后续的park就会直接返回。所以lock方法可以做一点修改：



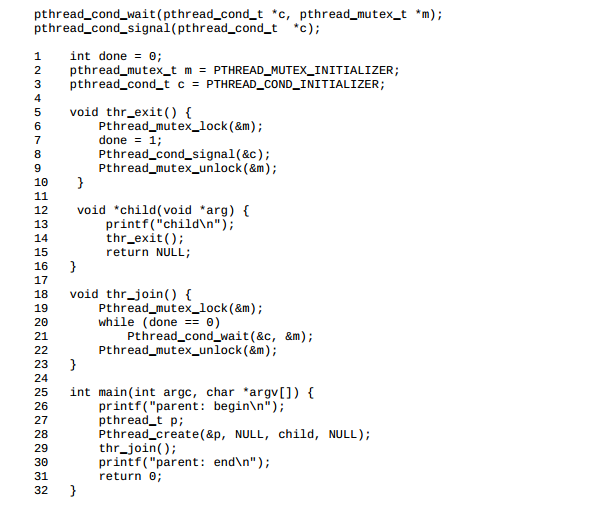
## 条件变量

锁并不是并发程序设计所需要的唯一原语。在很多情况下，线程需要检查某一条件满足之后，才会继续运行。

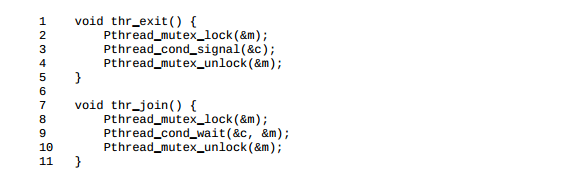
线程可以使用条件变量，来等待一个条件变成真。条件变量是一个显式队列，当某些执行状态不满足时，线程可以把自己加入到队列，等待该条件。另外某个线程，当它改变了上述状态时，就可以唤醒一个或者多个等待的线程，让它们继续执行。

C语言中要声明条件变量可以这样写：pthread\_cond\_t c;条件变量有两种相关的操作: wait()和signal()。当线程要睡眠的时候，调用wait()；当线程想唤醒等待在某个条件变量上的睡眠线程时，调用signal()。

例子： 假设有两个线程，一个主线程，一个子线程，主线程需要等待子线程执行完毕后才能开始下一步的操作。那么代码如下：

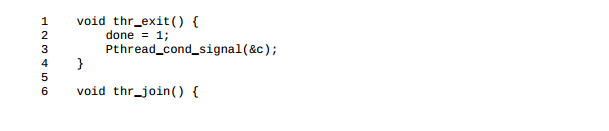


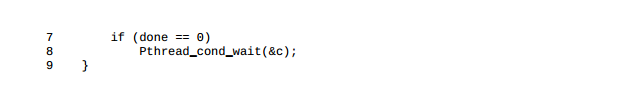
这里使用了一个条件变量:doen，如果不使用条件变量，查看如下代码：



假设子线程立即执行，并且调用了thr\_exit，那么在这种情况下，子线程发送了信号，但是此时却没有在条件变量上休眠的线程，当父线程运行时，就会调用wait()并卡在那里。

另外，这里调用wait和signal时都加上了锁，如果说不加上锁，查看如下代码：





这里会出现一个竞态条件，如果说父线程调用thr\_join(),然后检查完done的值为0，正准备wait()时，父线程被中断，切换了线程到子线程，子线程修改变量done为1，发出信号，这时候父线程调用wait()后就会长眠不醒。

所以，在调用wait()和signal()时一定要加上锁。

## 信号量

### 定义

信号量是一个整数值的对象，可以用两个函数来操作它。在POSIX标准中，是sem\_wait()和sem\_post()。因为信号量的初始值能够决定其行为，所以首先需要初始化它，才能调用其他函数与之交互。



第二个值设置为0，表示信号量在同一进程中的多个线程共享的，第三个参数的值是信号量的整数值。

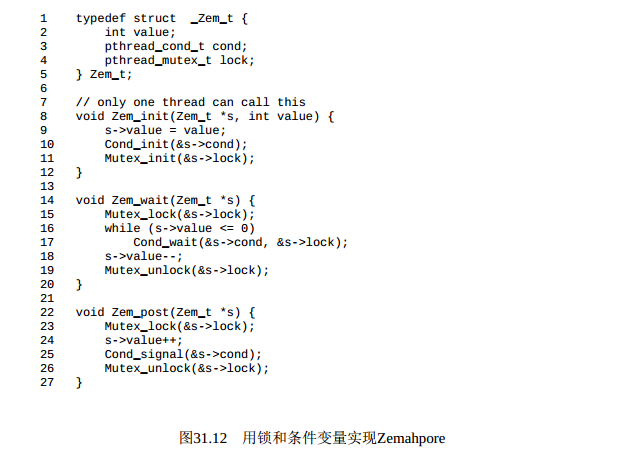
调用sem\_wait()要么立刻返回(当信号量的值大于等于1时)，要么会让调用线程挂起，直到之后的一个post操作。当然，也可能多个调用线程都调用sem\_wait()，因此都在队列中等待被唤醒。

sem\_post()并没有等待某些条件满足，它直接增加信号量的值，如果有等待线程，唤醒其中一个。

最后，当信号量的值为负数时，这个值就是等待线程的个数。

### 实现

使用锁和条件变量可以实现信号量。



这个实现和定义中的信号量略有不同，信号量的值不会是负数，所以通过这个信号量的值也看不到等待的线程数有多少。不过虽然看不到等待的线程数，但是也能够满足多个线程在信号量上进行等待的要求。

# 文件系统接口

## 文件概念

文件是记录在外存上的相关信息的具有名称的集合。通常，文件表示程序(源形式和目标形式)和数据。文件根据其类型具有一定结构，文本文件是由行(或页)组成，而行或页是由字符组成的，源文件是由子程序和函数组成，而它们又是由声明和执行语句组成的。目标文件是一系列字节序列，它们按目标系统链接器所能理解的方式组成。

## 文件属性

**名称**：文件符号名称是唯一的

**标识符**：表示文件系统内文件唯一的标签，通常为数字，对人而言这是不可读的文件名称

**类型**：被支持不同类型的文件系统所使用

**位置**：该信息为指向设备和设备上文件位置的指针

**大小**：文件当前大小，该属性也可包括文件允许的最大容量值

**保护**：决定谁能读、写、执行等的访问控制信息

**时间、日期和用户标识**： 文件创建、上次修改和上次访问的相关信息。

## 文件操作

**创建文件**：有两个必要步骤，第一：必须在文件系统中为文件找到空间，第二：在目录中为新文件创建一个条目

**写文件**：执行一个系统调用，其指明文件名称和要写入文件的内容。系统必须为该文件维护一个写位置的指针。

**读文件**：执行一个系统调用，并指明文件名称和要读入文件块的内存位置，系统要为该文件维护一个读位置的指针。每当发生读操作，必须更新读指针。

**在文件内重定位**：设置当前文件位置指针为给定值，在文件内重定位不需要包含真正的I/O。该文件操作也称为文件寻址。

**删除文件**：在目录中搜索给定名称的文件，找到目录后，释放所有的文件空间以便其他文件使用，并删除相应的目录条目。

**截短文件**：用户可能只需要删除文件内容而保留其属性，而不是强制删除文件再创建文件。

以上的绝大多数文件操作都涉及为给定文件搜索相关目录条目。为了避免这种不断的搜索操作，许多系统要求在首次使用文件时，需要使用系统调用open(),操作系统维护一个包含所有打开文件的信息表，可通过该表的一个索引指定文件，而不需要搜索。当文件不再使用时，进程可关闭它，操作系统从打开文件表中删除这一条目。

## 文件类型

如果操作系统识别了文件类型，那么它就能按合理方式对文件进行操作。

实现文件类型的常用技术是在文件名称内包含类型。名称可分为两部分：名称和扩展名

例如：只有具有扩展名.com,.exe,.bat的文件才可执行。另外，应用程序也可使用扩展名表示其感兴趣的文件类型，例如Word字处理器认为其文件具有.doc扩展名，这些扩展名不是由操作系统支持的，所以它们只作为给操作它们的应用程序的提示。

## 文件结构

源文件和目标文件具有一定结构，以适应相应处理程序的要求。例如，操作系统可能要求可执行文件具有特定结构，以便它能确定将文件装入到哪里以及第一个指令的位置是什么。

## 内部文件结构

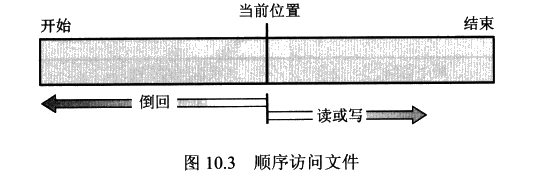
磁盘系统通常具有明确定义的块大小，这是由扇区大小决定的，所有磁盘I/O是按块来执行的，且所有块都是同样大小。另外，逻辑记录的长度是可变的，与物理记录大小不一样，需要进行转换。不管如何，文件都可以当做一系列块的组合，所有基本I/O操作都是按块来进行的。另外，由于磁盘空间总是按块来分配的，所以文件的最后一块的部分空间通常会被浪费，按块分配所浪费的字节称为内部碎片。

## 访问方式

**顺序访问**

文件信息按顺序，一个记录接着一个记录地加以处理。读操作读取下一文件部分，并自动前移指针，以跟踪I/O位置，写操作会向文件尾部增加内容，相应的文件指针移动到新增数据之后。

文件也可重新设置到开始位置，或者允许向前或向后跳过n个记录。



**直接访问**

直接访问方式基于文件的磁盘模型，以允许程序按任意顺序进行快速读和写。例如，可先读取块14，再读块53，最后再写7。

**其他访问方式**

其他访问方式建立在直接访问方式，这些访问通常设计创建文件索引，索引包含各块的指针，为了查找文件中的记录，首先搜索索引，再根据指针直接访问文件，以查找所需要的记录。

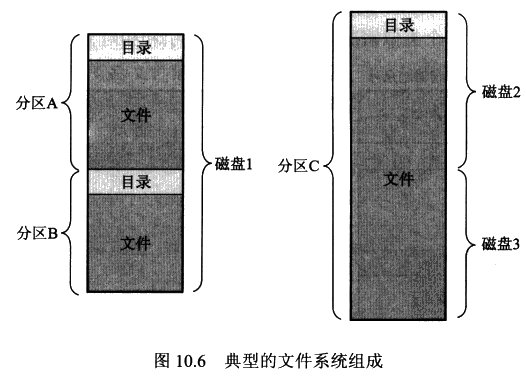
对于大文件，索引本身可能太大以至于不能保存在内存中，解决方法之一就是为索引文件再创建索引。

## 目录结构

系统可能有若干个文件系统，且文件系统可能有不同类型。为了管理这些数据，需要组织它们，这种管理涉及目录。

**存储结构**

一个磁盘可以装多个文件系统，或一部分用于文件系统而另一部分用于其他地方，如交换空间或非格式化的磁盘空间，这些部分称为**分区**或**片**或**小型磁盘**；

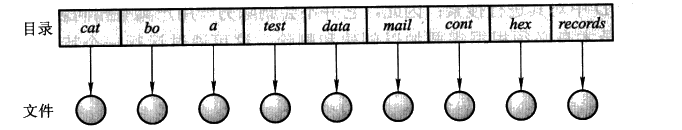


**目录概述**

目录可看作符号表，它能将文件名称转换成目录条目。它由以下相关操作：

**搜索文件**、**创建文件、删除文件、遍历文件、重命名文件、跟踪文件系统**

**单层目录结构**

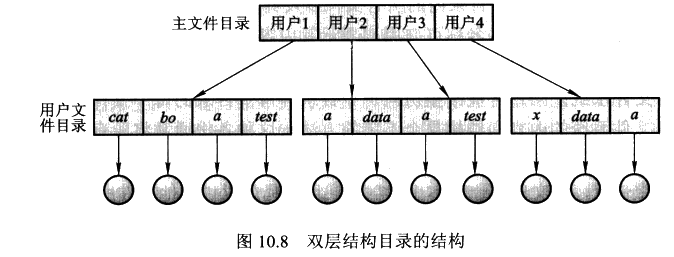


当有多个用户，或者文件过多时，单层目录结构有严格限制

**双层目录结构**

为每个用户创建独立目录。对于双层结构目录，每个用户都有自己的用户文件目录**UFD**，

每个UFD都有相似的结构，但只列出了单个用户的文件，当一个用户作业开始执行或一个用户注册时，就搜索系统的主文件目录**MFD**，通过用户名或账号可索引MFD，每个条目指向用户的UFD。



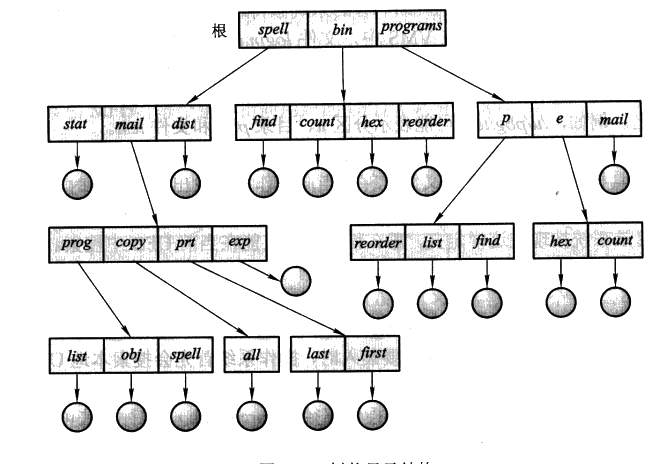
当用户需要在某个任务上进行合作和访问其他文件时，按用户分目录就是个缺点，

当访问其他用户的文件时，需要能够制定另一用户目录内的文件

当访问系统文件时，这些文件为所有用户共享，解决这一问题第一种方法可以在每个用户中复制一份这些系统文件；第二种方法是定义一个特殊的用户目录，它包含所有系统文件，当给定名称文件需要装入时，操作系统首选会搜索本地UFD，如果查询不到，那些自动搜索特殊用户目录。

**树状结构目录**

目录(或子目录)包含一组文件和子目录，一个目录只不过是一个需要按照特定方式访问的文件，所有目录具有相同的内部格式，每个目录条目都用一位来定义其为文件(0)或子目录(1)。



路径名有两种形式：**绝对路径名**和**相对路径名**。

绝对路径名从根开始并给出路径上的目录名直到所指定的文件；

相对路径名从当前目录开始定义路径。

**无环图目录**

树状结构禁止共享文件和目录。无环图允许目录含有共享子目录和文件，同一文件或子目录可出现在两个不同目录中。

实现共享文件和目录可以使用**链接**，链接实际上是另一文件或目录的指针。在遍历目录树时，操作系统忽略这些链接以维护系统的无环结构。

## 文件共享

**多用户情形**

为了实现共享和保护，多用户系统必须要比单用户系统维护更多的文件和目录属性。

绝大多数系统都采用**文件拥有者**和**组**的概念。拥有者是目录最高控制权的用户，可以改变属性和授权访问。组属性定义对文件拥有相同权限的用户子集。

一个文件或目录的拥有者ID和组ID与其他文件属性一起保存。当用户请求文件操作时，用户ID可与拥有者属性相比较，以确定请求者是不是文件拥有者，同样，可比较组ID，比较结果表示可使用哪些权限。

**远程文件系统情形**

随着网络和文本技术的发展，远程文件共享方式也不断改变。采用的第一种实现方式为，用户通过程序可实现在机器之间进行文件的人工传输。采用的第二种实现方式为分布式文件系统，远程目录可从本机上直接访问。 采用的第三种方式万维网，可用浏览器对远程文件的访问。

## 保护

第一种保护：可根据用户身份进行控制，不同用户可能对同一文件或目录需要有不同类型的访问。

其实现的第一种形式为：为每个文件和目录增加一个**访问控制列表**，以给定每个用户名及其所允许的访问类型。这种方法的优点是可以使用复杂的访问方法。问题在于其长度。

其实现的第二种形式为：为每个文件采用了三种用户类型：

**拥有者、组、其他**

例如UNIX系统定义了三个域以分别用于文件拥有者、组合其他用户，每个域为三位：RWX，其中r控制读访问，w控制写访问，而x控制执行。

第二种保护：

为每个文件加上密码。

## 文件系统实现

### 文件系统结构

为了改善I/O效率，内存与磁盘之间的I/O转移是以块为单位而不是以字节为单位来进行的。每块为一个或多个扇区，根据磁盘驱动器的不同，扇区从32~4096B不等，通常为512B。文件系统本身通常由许多不同的层组成的。



**I/O控制**

有设备驱动程序和中断程序组成，实现内存和磁盘之间的信息传输。设备驱动程序可以作为翻译器。其输入由高层命令组成，其输出由底层、硬件特性的命令组成

**基本文件系统**

只需要向合适的设备驱动程序发送一般命令就可对磁盘上的物理块进行读写。每个块由其数值磁盘地址来标识。(例如：驱动器1，柱面72，磁道3，扇区10)

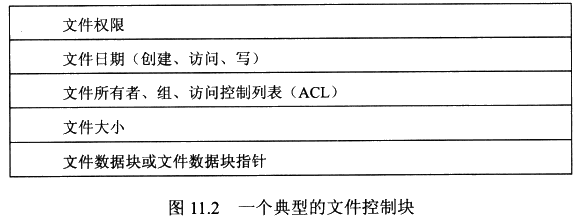
**文件组织模块**

知道文件及其逻辑块和物理块，由于知道所使用的文件分配类型和文件的位置，文件组织模块可以将逻辑地址转换成基本文件系统所用的物理地址。

**逻辑文件系统**

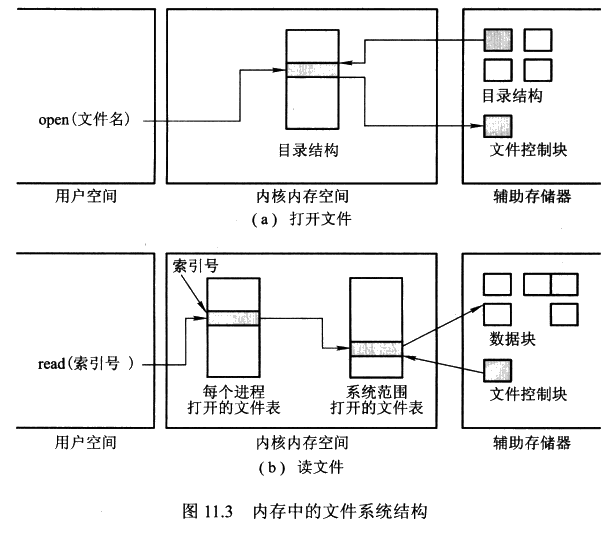
管理元数据，元数据包括文件系统的所有结构数据，而不包括实际数据。逻辑文件系统根据给定符号文件名来管理目录结构，并提供给模块所需要的信息。

逻辑文件系统通过文件控制块来维护文件结构。**文件控制块(FCB)**包含文件的信息，如拥有者、权限、文件内容的位置。逻辑文件系统也负责保护和安全。



内存内信息用于文件系统管理并通过缓存来提高性能。这些数据在文件系统安装的时候被加载，卸载的时候被丢弃。这些结构可能包括

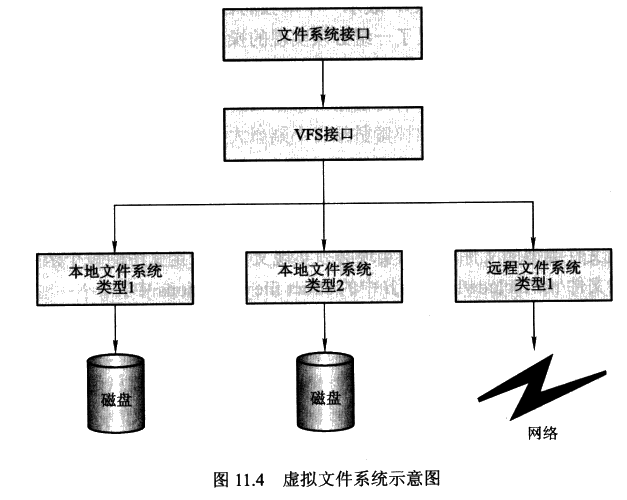
1. 一个内存中的安装表，包括所有安装卷的信息
2. 一个内存中的目录结构缓存，用来保存进来访问过的目录信息
3. **系统范围内的打开文件表**，包括每个打开文件的FCB副本和其他信息
4. **单个进程的打开文件表**，包括一个指向系统范围内已打开文件表中合适条目的指针和其他信息



### 虚拟文件系统

现代操作系统必须同时支持多个文件系统类型。但是操作系统需要把多个文件系统整合成一个目录结构，绝大多数操作系统包括UNIX都使用面向对象技术简化、组织和模块化实现过程。

采用数据结构和子程序，可以分开基本系统调用的功能和实现细节。因此，文件系统实现包括三个主要层次。



第一层为文件系统接口，包括open(), read(), write()和close()调用以及文件描述符。

第二层称为**虚拟文件系统(VFS)**层,它有两个目的：

1. VFS层通过定义一个清晰 VFS接口，以将文件系统的通用操作和具体实现分开。多个VFS接口的实现可以共存在同一台机器上，它允许访问已装在本地的多个类型的文件系统
2. VFS提供了在网络上唯一标识一个文件的机制。

第三层： VFS根据文件系统类型调用特定文件类型操作以处理本地请求，通过调用NFS协议程序来处理远程请求。第三层实现文件系统类型或远程文件协议。

**Linux中的VFS结构**

Linux的VFS定义的4种主要对象类型是：

**索引节点对象**：表示一个单独的文件

**文件对象**：表示一个打开的文件

**超级块对象**：表示整个文件系统

**目录条目对象**：表示一个单独的目录条目

VFS对每种类型的对象都定义了一组必须实现的操作。

### 目录实现

**线性列表**

最为简单的目录实现方法是使用存储文件名和数据指针的线性列表。优点是简单，缺点是查找文件需要线性搜索。

**哈希表**

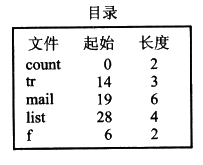
哈希表根据文件名得到一个值，并返回一个值向线性列表中元素的指针。因此，它大大地减少目录搜索时间。插入和删除也较简单，不过需要一些预备措施来避免冲突。

### 分配方法

磁盘的直接访问特点使大家能够灵活地实现文件，为了能有效地使用磁盘空间和快速访问文件，常用的主要磁盘空间分配方法有三个：连续、链接和索引。

**连续分配**

要求每个文件在磁盘上占有一组连续的块。文件的连续分配可以用第一块的磁盘和连续块的数量来定义。如果文件有n块长并从位置b开始，那么文件将占有b,b+1,b\_2,…,b+n-1。一个文件的目录条目包括开始块和该文件所分配区域的长度。



连续分配的问题在于会有外部碎片，另外一个问题是不知道文件需要多大的空间，如果为一个文件分配太小的空间，那么可能会发现文件不能扩展。

**链接分配**

采用链接分配，每个文件是磁盘文件的链表，磁盘块分布在磁盘的任何地方，目录包括文件第一块的指针和最后一块的指针。



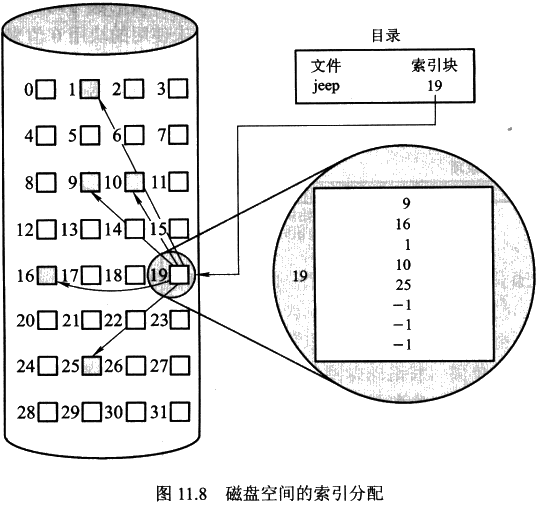
链接分配不会产生外部碎片，同时也不需要指定文件大小，可以无限扩展。

缺点在于：它只能顺序访问，另外指针还需要空间，为了解决指针的空间问题，常用的解决办法是将多个块组成簇。

**索引分配**

为了解决链表分配的不能直接访问问题，可以使用索引分配，通过将所有指针放在一起，形成一个索引块来解决问题

每个文件都有索引块，这个是磁盘块地址的数组，索引块的第i个条目指向文件的第i个块。目录条目包括索引块的地址。要读第i块，通过索引块的第i个条目的指针来查找和读入所需的块。



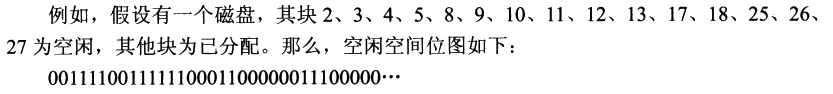
索引分配支持直接访问，同时没有外部碎片，但是索引块指正的快消通常比链接分配指针的开销大，同时，需要把握好索引块的大小。

### 空闲空间管理

因为磁盘空间有限，所以如果可能需要将删除文件的空间用于新文件，为了记录空闲磁盘空间，系统需要维护一个**空闲空间链表**。空闲空间链表记录了所有空闲磁盘空间。空闲空间链表虽然称为链表，但不一定表现为链表。

**位向量**

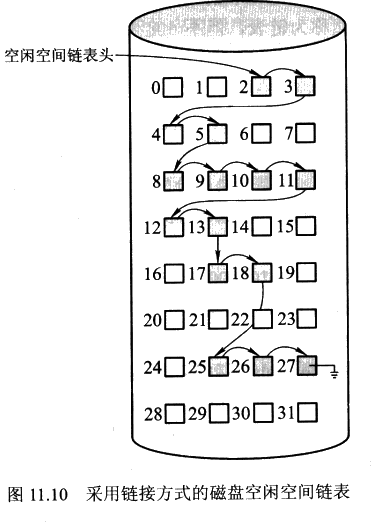
空闲空间表实现为位图或位向量，每块用一位表示。如果一块为空闲，那么其位为1，如果一块已分配，那么其位为0；



这种方法的主要优点在于查找第一个空闲块和n个连续空闲块时相对简单和高效。

**链表**

将所有空闲磁盘块用链表连接起来，并将指向第一空闲块的指针保存在磁盘的特殊位置，同时也缓存到内存中。



这种做法的缺点在于整个表时比较麻烦，但是这不是一个通常的操作，操作系统只不过简单地需要一个空闲块以分配给一个文件。

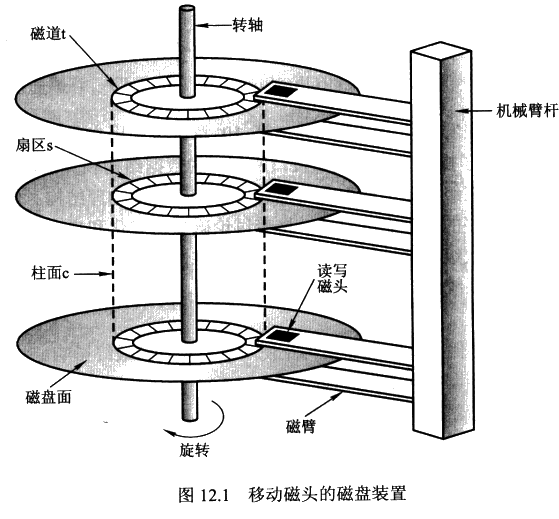
**组**

对空闲链表的一个改进是将n个空闲块的地址存在第一个空闲块中。这些块中的前n-1个确实为空，而最后一块包含另外n个空闲块的地址，如此继续。

## 大容量存储器结构

### 磁盘

它是由一个个盘片组成的，每个磁盘片为扁平圆盘，常用磁盘片的直径为1.8~5.25英寸。每个磁盘片的两面都涂着磁质材料。通过在磁片上进行磁记录可以保存信息。





磁盘片被划分为一条条**磁道**，，磁道再进一步划分为**扇区**。在同一磁道数的磁道集合形成了**柱面**。每个磁盘驱动器有数千个同心柱面，每个磁道可能包括数百个扇区。

磁盘速度分为两部分：

一是**传输速率**：是在驱动器和计算机之间的数据传输速率；

二是**随机访问时间(定位时间)**: 是由**寻道时间**(移动磁臂到所要的柱面所需时间)和**旋转等待时间**(等待所要的扇区旋转到磁臂下所需时间)

典型磁盘能以每秒数兆字节的速度传输，寻道时间和旋转等待时间为数毫秒。

**磁盘I/O总线**

磁盘驱动器通过一组称为**I/O总线**的线与计算机相连。有多种可用总线，EIDE，USB，FC等等。被称为**控制器**的特殊处理器执行总线上的数据传输。**主机控制器**是计算机上位于总线末端的控制器。**磁盘控制器**位于磁盘驱动器内。

为了执行磁盘I/O操作，计算机常常通过内存映射端口，在主机控制器上发送一个命令，主机控制器接着通过消息将该命令传送给磁盘控制器，磁盘控制器操纵磁盘驱动器硬件以执行命令。

磁盘驱动器有内置缓存，主机控制器也有内置缓存。

### 磁带

能存储数据相比较长久，且能存储大量数据，但与内存和磁盘相比器访问速度太慢。移动磁带的正确位置可能需要数分钟，所以随机访问要比磁盘随机访问慢千倍。

### 磁盘结构

现代磁盘驱动器可以看做一个一维的逻辑块的数组，逻辑块是最小的传输单位，逻辑块的大小通常为512B。一维逻辑块数组按顺序映射到磁盘的扇区，扇区0是最外面柱面的第一个磁道的第一个扇区。该映射是先按磁道内扇区顺序，再按柱面内磁道顺序，最后按从外到内的柱面顺序来排序的。

**常量线性速度**

每个磁道的位密度俊宇，磁道离磁盘中心越远，其长度越长，因而也能容纳更多的扇区，从外到内，每个磁道的扇区数也会减少，随着刺头由外到内，驱动器会增加速度以保持磁头续写的数据速率恒定。

**恒定圆角速度**

保持磁盘转动速度恒定，内磁道到外磁道的位密度要不断降低以保持数据率不变。

### 磁盘调度

**寻道时间：**磁臂将磁头移动到包含目标扇区的柱面的时间；

**旋转延迟：**磁盘需要将目标扇区转动到磁头下的时间；

**磁盘带宽：**所传递的总的字节除以从服务请求开始到最后传递结束的总时间；

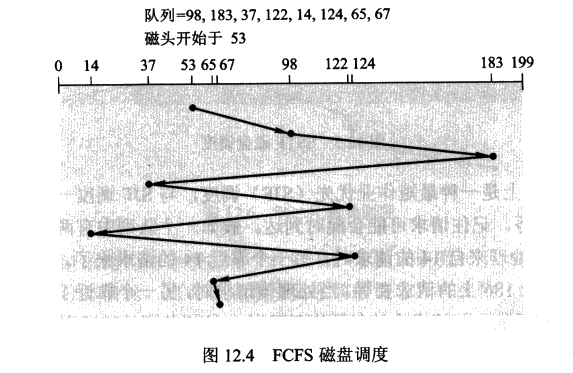
每当一个进程需要对磁盘进行I/O操作，它就向操作系统发出一个系统调用，该调用请求指定了一些信息：

1. 操作是输入还是输出
2. 所传输的磁盘地址是什么
3. 所传输的内存地址是什么
4. 所传输的扇区数是多少

如果所需的磁盘驱动器和控制器空闲，那么该请求会马上处理，如果磁盘驱动器或者控制器忙，那么任何新的服务请求都会加到该磁盘驱动器的待处理请求队列上。对于多个进程的多道程序设计系统，操作系统可以选择处理哪个待处理请求取决于调度策略

**FCFS调度**

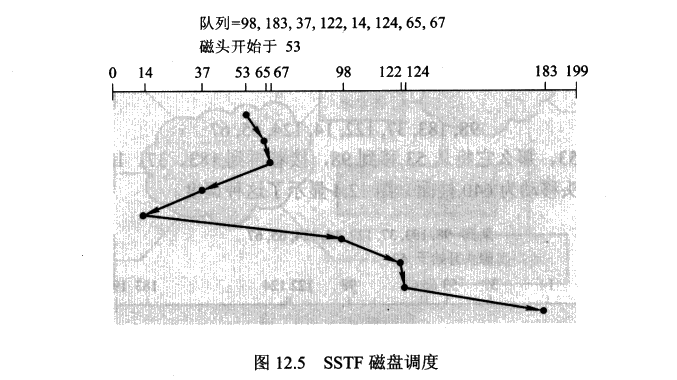
先来先服务算法，公平，但不能提供最快的服务。例如：有一个磁盘队列，其I/O对各个柱面上块的请求顺序如下：98，183，37,112,14,124,6,5,67。如果磁头开始位于53，那么它将从53移动到98，再依次移动到对应位置



可以修改顺序已减少磁头的移动

**SSTF调度**

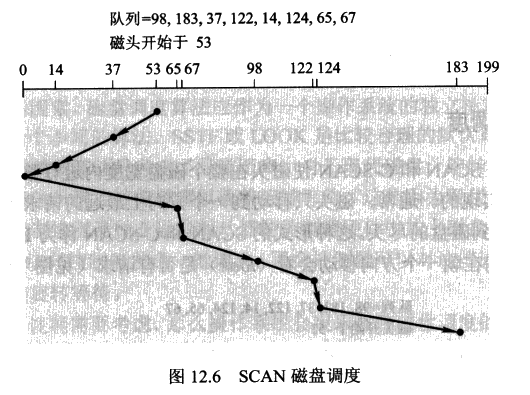
选择与当前磁头位置最近的待处理请求



这种磁头移动轨迹比上一种调度的轨迹小很多，但可能出现饥饿的情况。

**SCAN调度**

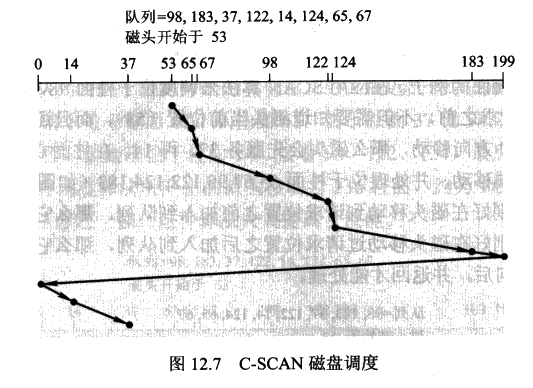
磁臂从磁盘的一端向另一端移动，同时当磁头移过每个柱面时，处理位于该柱面的服务请求，当到达另一端时，磁头改变移动方向，重复之前的逻辑



当调转方向时，紧靠磁头的请求会是少数，因为刚刚处理过，而另一端却聚集的请求最大，所以这存在优化。

**C-SCAN调度**

基于SCAN调度的缺陷，磁头从磁盘一端移到磁盘的另一端，随着移动不断地处理请求。不过，当磁头移动到另一端，它会马上返回到磁盘开始，返回时并不会处理请求



**LOOK调度**

磁头只移动到一个方向上最远的请求为止，接着，它马上回头，而不是继续到磁盘的尽头。



**以上所有算法只考虑了寻道距离因素**

### 磁盘管理

**低级格式化**

一个新的磁盘是一个空白板，它只是一些含有磁性记录材料的盘子。在磁盘能存储数据之前，它必须分成扇区以便磁盘控制器能读和写，这个过程称为低级格式化。

每个扇区的数据结构通常由头，数据区域，尾部组成

**分区**

为了使用磁盘存储文件，操作系统还需要将自己的数据结构记录在磁盘上。将磁盘分为一个或多个柱面组成的分区。操作系统将每个分区作为一个独立的磁盘。例如，一个分区可以用来存储操作系统的执行代码，而其他分区用来存储用户数据

**逻辑格式化（创建文件系统）**

操作系统将初始化的文件系统数据结构存储到磁盘上，这些数据结构包括空闲和已分配的空间和一个初始为空的目录。

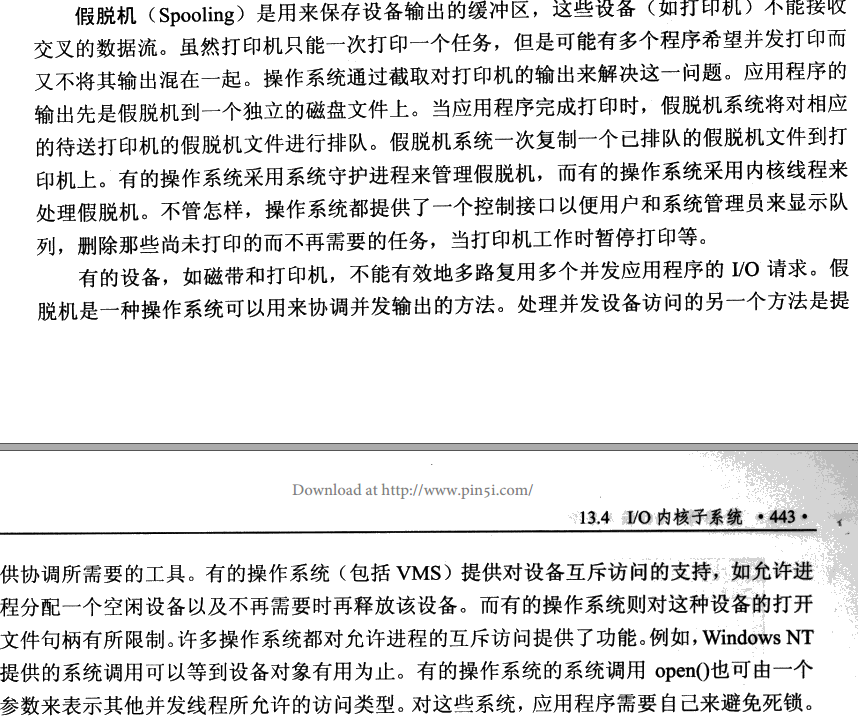
**簇**

为了提高效率，大多数操作系统将块集中到一大块，通常称作簇。磁盘I/O通过块完成，但是文件系统I/O通过簇完成，这样有效确保了I/O可以进行更多的顺序存取和更少的随机存储。

**缓冲**

缓冲区是用来保存两个设备之间或在设备和应用程序之间所传输数据的内存区域。

**假脱机与设备预留**



**错误处理**

作为一个规则，I/O系统调用通常返回一个位来表示调用状态信息，以表示成功或失败。对UNIX操作系统，一个名为errno的额外整数变量用来表示出错代码。

**I/O保护**

为了防止用户执行非法I/O，定义所有I/O指令为特权指令，因此，用户不能直接发出I/O指令，它们必须通过操作系统来进行。

另外，所有的内存映射和I/O端口内存位置都受到内存保护系统的保护，以阻止用户访问。

### I/O请求的生命周期

