# 背景

当内核开始一个读操作（比如进程发起一个read()系统调用），它首先会检查需要的数据是否在页高速缓存中。如果在，则放弃访问磁盘，而直接从内存中读取。这个行为称作缓存命中。如果数据没有在缓存中，称为缓存未命中，那么内核必须调度块I/O操作从磁盘去读取数据。然后内核将读来的数据放入页缓存中，于是任何后续相同的数据读取都可以命中缓存了。

注：这里说明了什么时候使用块I/O调度，即页高速缓存未命中的情况下，执行磁盘的块I/O操作。

# 概述

系统中能够随机（不需要按顺序）访问固定大小数据片（chunks）的硬件设备称作块设备，这些固定大小的数据片就称为块。最常见的块设备是硬盘，此外还有软盘驱动器、光驱和闪存等块设备。

另一种基本的设备类型是字符设备。字符设备按照字符流的方式被有序访问，像串口和键盘都属于字符设备。

块设备与字符设备的区别在于是否可以随机访问数据。比如，键盘是一个标准的字符设备，键盘驱动程序会按照和输入完全相同的顺序读取字符串。

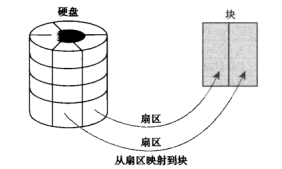
内核管理块设备要比管理字符设备细致得多，需要考虑的问题和完成的工作相对于字符设备来说要复杂很多。这是因为字符设备仅仅需要控制一个位置——当前位置，而块设备访问的位置必须能够在介质的不同区间前后移动。所以事实上内核不必提供一个专门的子系统来管理字符设备，但是对块设备的管理却必须要有一个专门的提供服务的子系统。

# 块设备

块设备中最小的可寻址单元是扇区。扇区大小一般是2的整数倍，而最常见的是512字节。扇区的大小是块设备的物理属性，扇区是所有块设备的基本单元——块设备无法对比它还小的单元进行寻址和操作，尽管许多块设备能够一次对多个扇区进行操作。

因为各种软件的用途不同，所以他们都会用到自己的最小逻辑可寻址单元——块。块是文件系统的一种抽象——只能基于块来访问文件系统（之所以引入块，是对扇区的抽象，管理扇区太小了比较繁琐，需要使用更大的块管理比较合理）。虽然**物理磁盘寻址是按照扇区级进行的，但是内核执行的所有磁盘操作都是按照块进行的**。由于扇区是设备的最小可寻址单元，所以块不能比扇区还小，只能整倍于扇区大小。另外，内核（对有扇区的硬件设备）还要求块大小是2的整数倍，而且不能超多一个页的长度。所以，对块大小的最终要求是，必须是扇区大小的2的整数倍，并且要小于页面大小。所以通常块大小是512字节、1KB或4KB。

扇区——**设备的最小寻址单元**，有时会称作“硬扇区”或“设备块”；同样，块——**文件系统的最小寻址单元**，有时会称作“文件块”或“I/O块”。



# 缓冲区/缓冲区头

## 概述

当一个块被调入内存时（也就是说，在读入后或等待写出时），它要存储在一个**缓冲区**中。每个缓冲区与一个块对应，它相当于是磁盘块在内存中的表示。块包含一个或多个扇区，但是大小不能超多一个页面，所以**一个页可以容纳一个或多个内存中的块**（页面是缓冲区的概念）。由于内核在处理数据时需要一些相关的控制信息（比如块属于哪一个块设备，块对应于哪个缓冲区等），所以每一个缓冲区都有一个对应的描述符。该描述符用buffer\_head结构体表示，称作**缓冲区头**。

注：**扇区对应磁盘，块对应文件系统，页对应缓冲区**。

缓冲区头的目的在于描述磁盘块和物理内存缓冲区（在特定页面上的字节序列）之间的映射关系。这个结构体在内核中只扮演一个描述符的角色，说明从缓冲区到块的映射关系。

注：并不是所有的磁盘都会将块与缓冲区关联起来，也有时候不需要使用操作系统的缓冲区，比如MySQL就是自己定义管理缓冲，不使用操作系统的缓冲区。

## 弊端

在2.6内核版本之前，缓冲区头作用比现在更重要。因为缓冲区头作为内核中的I/O操作单元，不仅仅描述了从磁盘块到物理内存的映射，而且还是所有块I/O操作的容器。可是，将缓冲区头作为I/O操作单元有两个弊端：

首先，缓冲区头是一个很大且不易控制的数据结构体（现在是缩减过的了），而且缓冲区头对数据的操作既不方便也不清晰。对于内核来说，它更倾向于操作页面结构，因为页面操作起来更为简便，同时效率也高。使用一个巨大的缓冲区头表示每一个独立的缓冲区（可能比页面小）效率低，所以在2.6版本中，许多I/O操作都是通过内核直接对页面或者地址空间操作来实现的，不再使用缓冲区头了。

其次，它仅能描述单个缓冲区，当作为所有I/O的容器使用时，缓冲区头会促使内核必然会造成不必要的负担和空间浪费。所以2.5开发版内核的主要目标就是为块I/O操作引入一种新型、灵活并且轻量级的容器，即bio结构体。

# bio结构体

## 概述

## 对比

bio结构体代表的是I/O操作，它可以包括内存中的一个或多个页；而另一方面，buffer\_head结构体代表的是一个缓冲区，它描述的仅仅是磁盘中的一个块。因为**缓冲区头关联的是单独页中的单独磁盘块，所以它可能会引起不必要的分隔，将请求按块为单位划分，只能靠以后才能再重新组合**。由于bio结构体是轻量级的，它描述的块可以不需要连续存储区，并且不需要分割I/O操作。

利用bio结构体代替buffer\_head结构体还有以下的好处：

1. bio结构体很容易处理高端内存，因为它处理的是物理页而不是直接指针。
2. bio结构体既可以代表普通页I/O，同时也可以代表直接I/O（指那些不通过页高速缓存的I/O操作）。
3. bio结构体便于执行分散-集中（矢量化）块I/O操作，操作中的数据可取自多个物理页面。
4. bio结构体相比缓冲区头属于轻量级的结构体。因为它只需要包含块I/O操作所需要的信息就行了，不用包含与缓冲区本身相关的不必要信息。

但是，还是需要理解缓冲区头这个概念，它负责描述磁盘块到页面的映射。bio结构体不包含任何和缓冲区相关的状态信息——它仅仅是一个矢量数组，描述一个或多个单独块I/O操作的数据片段和相关信息。在当前设置中，当bio结构体描述当前正在使用的I/O操作时，buffer\_head结构体仍然需要包含缓冲区信息。内核通过这两种结构分别保存各自的信息，可以保证每种结构所含的信息量尽可能地少。

# 请求队列

块设备将它们挂起的I/O请求保持在请求队列中，该队列由request\_queue结构体表示，包含一个双向请求链表以及相关控制信息。通过内核中向文件系统这样高层的代码将请求加入到队列中。请求队列只要不为空，队列对应的块设备驱动程序就会从队列头获取请求，然后将其送入对应的块设备上去。请求队列中的每一项都是一个单独的请求，有request结构体表示。

# I/O调度程序

如果简单地以内核产生请求的次序直接将请求发现块设备的话，性能肯定让人难以接受。磁盘寻址是整个计算机中最慢的操作之一，每一次寻址（定位硬盘磁头到特定块上的某个地址）需要花费不少时间。所以尽量缩短寻址时间无疑是提高系统性能的关键。

为了优化寻址操作，内核既不会简单地按请求接收次序，也不会立即将其提交给磁盘。相反，它会在提交前，先执行名为合并与排序的预操作，这种预操作可以极大地提高系统的整体性能。在内核中负责提交I/O请求的子系统称为I/O调度程度。

I/O调度程序将磁盘I/O资源分配给系统中所有挂起的块I/O请求。具体地说，这种资源分配是通过将请求队列中挂起的请求合并和排序来完成的。注意不要将I/O调度程序和进程调度程序混淆，进程调度程序的作用是将处理器资源分配给系统中的运行进程。进程调度程序和I/O调度程序都是将一个资源虚拟给多个对象，对进程调度程序来说，处理器被虚拟并被系统中的运行进程共享。这种虚拟提供给用户的就是多任务和分时操作系统，像Unix系统。相反，I/O调度程序虚拟块设备给多个磁盘请求，以便降低磁盘寻址时间，确保磁盘性能的最优化。

IO调度器的总体目标是希望让磁头能够总是往一个方向移动，移动到底了再往反方向走，这恰恰就是现实生活中的电梯模型，所以IO调度器也被叫做电梯。而相应的算法也就被叫做电梯算法。而Linux中IO调度的电梯算法有好几种，一个叫做as，一个叫做cfq，一个叫做deadline，还有一个叫做noop。具体使用哪种算法我们可以在启动的时候通过内核参数elevator来指定。

## 作用

I/O调度程序的工作是管理块设备的请求队列。它决定队列中的请求队列排序以及在什么时刻派发请求到块设备。这样做有利于减少磁盘寻址时间，从而提高全局吞吐量。注意，全局这个定语很重要，坦率地讲，一个I/O调度器可能为了提高系统整体性能，而对某些请求不公。

I/O调度程序通过两种方法减少磁盘寻址时间：合并与排序。合并指将两个或多个请求结合成一个新请求。

考虑以下这种情况，文件系统提交请求到请求队列——从文件中读取一个数据区（当然，最终所有的操作都是针对扇区和块进行的，而不是文件，还假定请求的块都是来自文件块），如果这时队列中已经存在一个请求，它访问的磁盘扇区和当前请求访问的磁盘扇区相邻（比如，同一文件中早些时候被读取的数据区），那么这两个请求就可以合并为一个对单个和多个相邻磁盘扇区操作的新请求。通过合并这些请求，I/O调度程序将多次请求的开销压缩成一次请求的开销。更重要的是，请求合并后只需要传递给磁盘一条寻址命令，就可以访问到请求合并前必须多次寻址才能访问完的磁盘区域了，因此合并请求显然能减少系统开销和磁盘寻址次数。

现在，假设在读请求被提交给请求队列的时候，队列中并不需要操作相邻扇区的其他请求，此时就无法将当前请求与其他请求合并，当然，可以将其插入请求队列的尾部。但是如果有其他请求需要操作磁盘上类似的位置呢？如果存在一个请求，它要操作的磁盘扇区位置与当前请求比较接近，那么是不是该让这两个请求在请求队列上也相邻呢？事实上，I/O调度程序的确是这样处理上述情况的，整个请求队列将按扇区增长方向有序排序。使所有请求按照磁盘上扇区的排列顺序有序排列（尽可能的）的目的不仅是为了缩短单独一次请求的寻址时间，更重要的优化在于，通过保持磁盘头以直线方向移动，缩短了所有请求的磁盘寻址空间。该排序算法类似于电梯调度——电梯不能随意地从一层跳到另一层，它应该向一个方向移动，当抵达了同一方向上的最后一层后，再掉头向另一个方向移动。处于这种相似性，所以I/O调度程序（或这种排序算法）称作电梯调度。

## 算法

### Linus电梯

Linus电梯能执行合并与排序预处理。当有新的请求加入队列时，它首先会检查其他每一个挂起的请求是否可以和新请求合并。Linus电梯I/O调度程序可以执行向前和向后合并，合并类型描述的是请求向前面还是向后面，这一点和已有请求相连。如果新请求正好连在一个现存的请求前，就是向前合并；相反如果新请求直接连载一个现存的请求后，就是向后合并。鉴于文件的分布（通常以扇区号的增长表现）特点和I/O操作执行方式具有典型性（一般都是从头读向尾，很少从反方向读），所以向前合并相比向后合并要少得多，但是Linus电梯还是会对两种合并类型都进行检查。

如果合并尝试失败，那么就需要寻找可能的插入点（新请求在队列中的位置必须符合请求以扇区方向有序排序的原则）。如果找到，新请求将被插入到该点；如果没有合适的位置，那么新请求就被加入到队列尾部。另外，如果发现队列中有驻留时间过长的请求，那么新请求也将被加入到队列尾部，即使插入后还要排序。这样做是为了避免由于访问相近磁盘位置的请求太多，从而造成访问磁盘其他位置的请求难以得到执行机会这一问题。不幸的是，这种“年龄”检测方法并不是很有效，因为它并非是给等待了一段时间的请求提供实质性服务，它仅仅是在经过了一定时间后停止插入-排序请求，这改善了等待时间但最终还是会导致请求饥饿现象的发生，所以这是一个2.4内核I/O调度程序中必须要修改的缺陷。

### NOOP

空操作的I/O调度程序

NOOP算法的全写为No Operation。该算法实现了最最简单的FIFO队列，所有IO请求大致按照先来后到的顺序进行操作。之所以说“大致”，原因是NOOP在FIFO的基础上还做了相邻IO请求的合并，并不是完完全全按照先进先出的规则满足IO请求。NOOP假定I/O请求由驱动程序或者设备做了优化或者重排了顺序(就像一个智能控制器完成的工作那样)。

在有些SAN环境下，这个选择可能是最好选择。NOOP对于IO不那么操心，对所有的IO请求都用FIFO队列形式处理，默认认为IO不会存在性能问题。这也使得CPU也不用那么操心。当然，对于复杂一点的应用类型，使用这个调度器，用户自己就会非常操心。

### Deadline scheduler

DEADLINE在CFQ的基础上，解决了IO请求饿死的极端情况。除了CFQ本身具有的IO排序队列之外，DEADLINE额外分别为读IO和写IO提供了FIFO队列。读FIFO队列的最大等待时间为500ms，写FIFO队列的最大等待时间为5s。FIFO队列内的IO请求优先级要比CFQ队列中的高，而读FIFO队列的优先级又比写FIFO队列的优先级高。优先级可以表示如下：  
 FIFO(Read) > FIFO(Write) > CFQ

deadline算法保证对于既定的IO请求以最小的延迟时间，从这一点理解，对于DSS应用应该会是很适合的。

最终期限（deadline）I/O调度程序是为了解决Linus电梯所带来的**饥饿问题**而提出的。由于减少磁盘寻址时间的考虑，对某个磁盘区域上的繁重操作，无疑会使得磁盘其他位置上的操作请求得不到运行机会。实际上，一个对磁盘同一位置操作的请求流可以造成较远位置的其他请求永远得不到运行机会，这是一种很不公平的饥饿现象。

在最后期限I/O调度程序中，每个请求都有一个超时时间。默认情况下，读请求的超时时间是500ms，写请求的超时时间是5s。最后期限I/O调度请求类似于Linus电梯，也以磁盘物理位置为次序维护请求队列，这个队列称为排序队列。

### Anticipatory scheduler

预测I/O调度程序

虽然最后期限I/O调度程序为降低读操作响应时间做了许多工作，但是它同时也降低了系统吞吐量。假设一个系统处于很繁重的写操作期间，每次提交读请求，I/O调度程序都会迅速处理读请求，所以磁盘首先为读操作进行寻址，执行读操作，然后返回再寻址进行写操作，并且对每个读请求都重复这个过程。这种做法对读请求来说是件好事，但是两次寻址操作（一次对读操作定位，一次返回来进行写操作定位）却损害了系统全局吞吐量。预测I/O调度程序的目的就是保持良好的读响应的同时也能够提供良好的全局吞吐量。

CFQ和DEADLINE考虑的焦点在于满足零散IO请求上。对于连续的IO请求，比如顺序读，并没有做优化。为了满足随机IO和顺序IO混合的场景，Linux还支持ANTICIPATORY调度算法。ANTICIPATORY的在DEADLINE的基础上，为每个读IO都设置了6ms的等待时间窗口。如果在这6ms内OS收到了相邻位置的读IO请求，就可以立即满足。  
   Anticipatory scheduler（as)曾经一度是Linux 2.6 Kernel的IO scheduler。Anticipatory的中文含义是”预料的, 预想的”, 这个词的确揭示了这个算法的特点，简单的说，有个IO发生的时候，如果又有进程请求IO操作，则将产生一个默认的6毫秒猜测时间，猜测下一个 进程请求IO是要干什么的。这对于随即读取会造成比较大的延时，对数据库应用很糟糕，而对于Web Server等则会表现的不错。这个算法也可以简单理解为面向低速磁盘的，因为那个”猜测”实际上的目的是为了减少磁头移动时间。

虽然最后期限I/O调度程序为降低读操作响应时间做了许多工作，但是它同时也降低了系统吞吐量。预测I/O调度程序的目标就是在保持良好的读操作的同时也能提供良好的全局吞吐量。

预测I/O调度的基础仍然是最后期限I/O调度程序，所以它们有很多相同之处。预测I/O调度程序也实现了三个队列（加上一个派发队列），并为每个请求设置了超时时间，这点与最后期限I/O调度程序一样。预测I/O调度程序主要的改进是它增加了预测启发（anticipation-heuristic）能力。

### CFQ

Completely Fair Queuing（CFQ，完全公平队列) 在2.6.18取代了Anticipatory scheduler成为Linux Kernel默认的IO scheduler。CFQ对每个进程维护一个IO队列，各个进程发来的IO请求会被CFQ以轮循方式处理。也就是对每一个IO请求都是公平的。这使得CFQ很适合离散读的应用(eg: OLTP DB)。我所知道的企业级Linux发行版中，SuSE Linux好像是最先默认用CFQ的。

完全公正的排队I/O调度程序（Complete Fair Queuing，CFQ）是为专有工作负荷设计的，不过，在实际中，也为多种工作负荷提供了良好的性能。

CFQ I/O调度程序把进入的I/O请求放入特定的队列中，这种队列是根据引起I/O请求的进程组织的。

CFQ I/O调度程序以时间片轮询调度队列，从每个队列中选取请求数，然后进行下一轮调度。这就在进程级提供了公平，确保每个进程接收公平的磁盘带宽片断。

在传统的SAS盘上，磁盘寻道花去了绝大多数的IO响应时间。CFQ的出发点是对IO地址进行排序，以尽量少的磁盘旋转次数来满足尽可能多的IO请求。在CFQ算法下，SAS盘的吞吐量大大提高了。但是相比于NOOP的缺点是，先来的IO请求并不一定能被满足，可能会出现饿死的情况。

## 查看

查看和修改IO调度器的算法非常简单。假设我们要对sda进行操作，如下所示：  
 cat /sys/block/sda/queue/scheduler  
 echo “cfq” > /sys/block/sda/queue/scheduler

## 选择

  1、CFQ和DEADLINE考虑的焦点在于满足零散IO请求上。对于连续的IO请求，比如顺序读，并没有做优化。为了满足随机IO和顺序IO混合的场景，Linux还支持ANTICIPATORY调度算法。ANTICIPATORY的在DEADLINE的基础上，为每个读IO都设置了6ms的等待时间窗口。如果在这6ms内OS收到了相邻位置的读IO请求，就可以立即满足。  
 IO调度器算法的选择，既取决于硬件特征，也取决于应用场景。  
 在传统的SAS盘上，CFQ、DEADLINE、ANTICIPATORY都是不错的选择；对于专属的数据库服务器，DEADLINE的吞吐量和响应时间都表现良好。然而在新兴的固态硬盘比如SSD、Fusion IO上，最简单的NOOP反而可能是最好的算法，因为其他三个算法的优化是基于缩短寻道时间的，而固态硬盘没有所谓的寻道时间且IO响应时间非常短。  
 2、对于数据库应用，Anticipatory Scheduler的表现是最差的。Deadline在DSS环境表现比CFQ更好一点，而CFQ综合来看表现更好一些。这也难怪RHEL 4默认的IO调度器设置为CFQ。而 RHEL 4比RHEL 3，整体IO改进还是不小的。