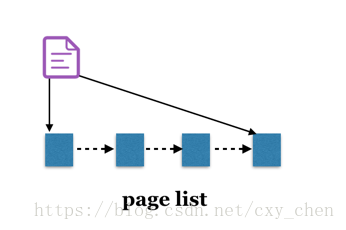
# Linux中page、buffer\_head、bio的联系

在Linux Block IO层，说到关键数据结构我想可能就只有标题中描述的三种了。我们今天就来详细描述这三种数据结构在文件系统和块设备层扮演的角色以及他们之间的联系。 

## PAGE

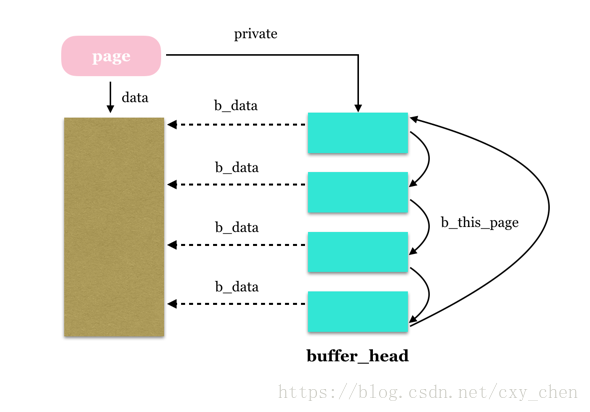
page在内核中被称为缓存页，在文件系统中扮演最核心的角色。Linux使用内存缓存文件数据，而所有的文件内容都被分割成page然后通过一定方式组织起来，便于查找。   
   
page大小固定，当前一般为4KB。一个大文件的缓存可能会占据很多page。 

## BUFFER\_HEAD

buffer\_head顾名思义，表示缓冲区头部。这个缓冲区缓冲的是磁盘等块设备数据，而buffer\_head则是描述缓冲区的元数据。

前面说过，内核以page为单位管理文件内容，page典型大小为4KB，而一般块设备的访问以扇区为单位，扇区的一般大小为512 byte，当然随着技术的进步，这个大小也在不断提升。而文件系统最小逻辑可寻址单元称为块。块的大小要比扇区大，但又比页小，典型大小为1K。 内核执行磁盘的所有操作是按照块来操作的。

在如此背景之下，便诞生了buffer\_head这样的数据结构，它是内核page与磁盘上物理数据块之间的桥梁。一方面，每个page包含多个buffer\_head（一般4个），另外一方面，buffer\_head中又记录了底层设备块号信息。这样，通过page->buffer\_head->block就能完成数据的读写。

page与buffer\_head数据结构之间关系如下图所示:假设page大小为4KB，而文件系统块大小为1KB。   


page通过private字段索引该page的第一个buffer\_head，而所有的buffer\_head通过b\_this\_page形成一个单循环链表；   
buffer\_head中的b\_data指向缓存文件的块数据；   
buffer\_head内还通过b\_page指向其所属的page（图中未画出）

由于buffer\_head描述的是文件系统块缓存，既然缓存，便存在数据一致性问题：缓存中的数据可能比磁盘数据落后，缓存中的数据也可能比磁盘数据新。因此，需要一种机制来描述这些状态。buffer\_head中需要一系列的状态位。

enum bh\_state\_bits {

BH\_Uptodate,

BH\_Dirty,

BH\_Lock,

BH\_Req,

BH\_Uptodate\_Lock,

BH\_Mapped,

BH\_New,

BH\_Async\_Read,

BH\_Async\_Write,

BH\_Delay,

BH\_Boundary,

BH\_Write\_EIO,

BH\_Unwritten,

BH\_Quiet,

BH\_Meta,

BH\_Prio,

BH\_Defer\_Completion,

BH\_PrivateStart,

}

其中：   
BH\_Uptodate: 表示缓存数据与磁盘数据一致   
BH\_Dirty: 表示缓存数据被更新，有待同步至磁盘上。   
这些状态何时会被更新呢？

以BH\_Uptodate为例，当文件系统向块设备层发起一次读请求时，会注册一个完成时的回调函数end\_buffer\_read\_sync，在IO层返回读取结果后，该函数内根据读取结果：

* 如果读取成功，则说明buffer\_head缓冲区中的数据与磁盘上一致，设置BH\_Update。
* 如果读取失败，说明buffer\_head缓冲区的数据处于一种不一致状态，此时，需要清除buffer\_head的BH\_Update。

static void \_\_end\_buffer\_read\_notouch(struct buffer\_head \*bh, int uptodate)

{

if (uptodate) {

set\_buffer\_uptodate(bh);

} else {

clear\_buffer\_uptodate(bh);

}

unlock\_buffer(bh);

}

void end\_buffer\_read\_sync(struct

buffer\_head \*bh, int uptodate)

{

\_\_end\_buffer\_read\_notouch(bh, uptodate);

put\_bh(bh);

}

# bio, bio\_vec, page, request, request\_queue关系

通用块层的核心数据结构称为bio描述符，它描述了块设备的io操作。每一个bio结构都包含一个磁盘存储区标识符（存储区中的起始扇区号和扇区数目）和一个或多个描述与IO操作相关的内存区段（bio\_vec数组）

bio结构中的字段

struct bio {

sector\_t bi\_sector; //要传输的第一个扇区。

struct bio \*bi\_next; //链接到请求队列的下一个bio

struct block\_device \*bi\_bdev;//指向块设备描述符的指针

unsigned long bi\_flags; //bio的状态标志

unsigned long bi\_rw;//IO操作标志 READ/WRITE

unsigned short bi\_vcnt; //bio中的bio\_vec数组中当前元素的个数

unsigned short bi\_idx; //bio的bio\_vec数组中段的当前索引值

unsigned int bi\_phys\_segments; //合并后bio中的物理段的数目

unsigned int bi\_size; //所需传输的数据字节数

unsigned int bi\_seg\_front\_size;

unsigned int bi\_seg\_back\_size;

unsigned int bi\_max\_vecs; //bio的bio\_vec数组中允许的最大段数

unsigned int bi\_comp\_cpu; /\* completion CPU\*/

atomic\_t bi\_cnt; //bio引用计数器

struct bio\_vec \*bi\_io\_vec; /\* the actual vec list\*/

bio\_end\_io\_t \*bi\_end\_io;//bio的IO操作结束时调用的方法

void \*bi\_private;//通用块层和块设备驱动程序的IO完成方法使用的指针

bio\_destructor\_t \*bi\_destructor; //释放bio时调用的析构方法

};

bio中的每个段是由一个bio\_vec数据结构描述的，bio\_vec数据结构如下

bio\_vec{

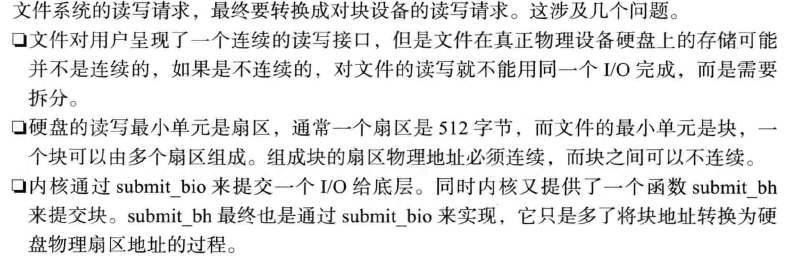
struct page\* bv\_page //指向段在页框描述符的指针

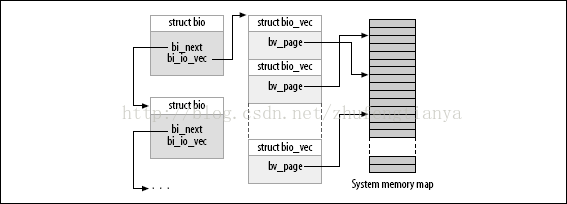
unsigned int bv\_len //段的字节长度

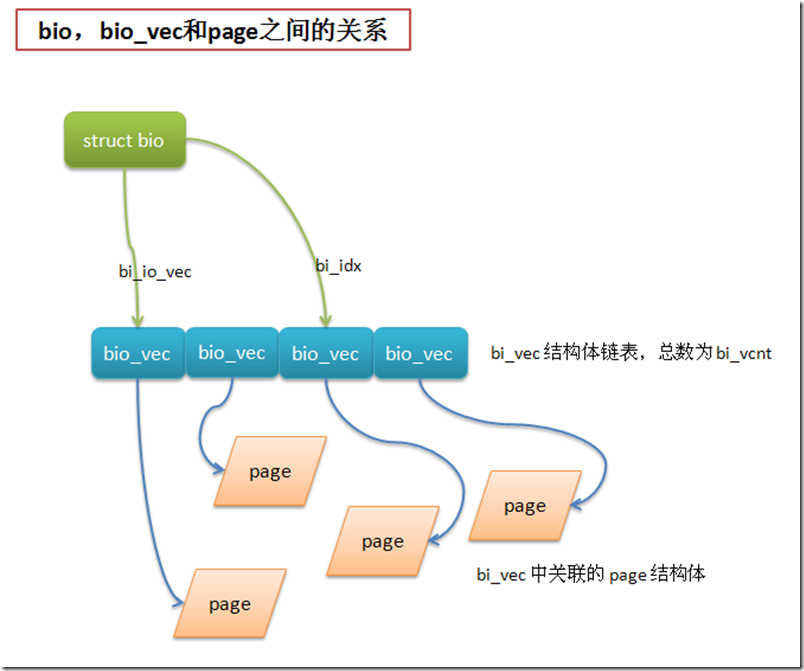
unsigned int bv\_offset //页框中数据的偏移量

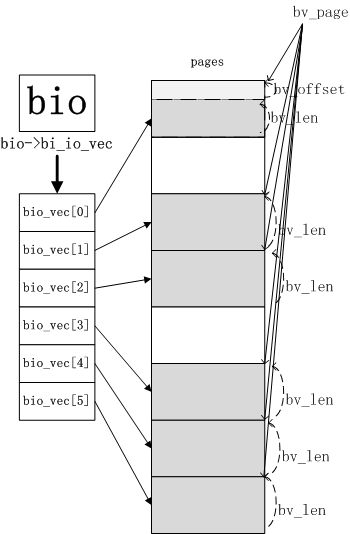
}

bio中的bi\_io\_vec字段指向bio\_vec数组的第一个元素，bi\_vcnt则说明了数组当前元素的个数，而bi\_max\_vecs则限定了数组的长度。









在通用块层启动一次新的IO操作时，会调用bio\_alloc函数分配一个新的bio结构，bio是由slab分配器分配的。内核同时也为bio\_vec结构分配内存池。

一个bio可能有很多个bio段（bio\_vec），这些bio段可能在内存上不连续（位于不同的页），但他们在**磁盘上对应的位置是连续**的。

1). 一个BIO所请求的数据在块设备中是连续的，对于不连续的数据块需要放到多个BIO中。

2). 一个BIO所携带的数据大小是有上限的，该上限值由bi\_max\_vecs间接指定，超过上限的数据块必须放到多个BIO中。

一般上层构建bio的时候都是只有一个bio段，可以参考\_submit\_bh函数。一个bio\_vec对应一个或多个文件系统块（1K，2K或4K，即buffer\_head对应的结构）

do\_mapge\_read()函数通过从get\_block()（对应于ext2文件系统是ext2\_getblock()得到的信息来判断页中的块是否连续如果连续则通过mpage\_alloc()申请一个具有多个段（包含多个io\_vec）的bio结构，然后调用bio\_add\_page()将该页添加到bio中。如果页不连续，则调用block\_read\_full\_page()以每次一块的方式来读取该页，对于页中的每块将调用submit\_bh()来生成并提交bio，对于submit\_bh() 生成的bio中的io\_vec数组只有一个元素。

bio：代表了一个io请求

request：一个request中包含了一个或多个bio，为什么要有request这个结构呢？它存在的目的就是为了进行io的调度。通过request这个辅助结构，我们来给bio进行某种调度方法的排序，从而最大化地提高磁盘访问速度。

每个块设备的待处理请求都是用一个请求描述符request来表示的，request结构如下(只列出了较为重要的项)：

struct request {

union {

struct list\_head queuelist; //请求队列链表的指针，将在同一请求队列的request链接在一起

struct llist\_node ll\_list;

};

struct request\_queue \*q; //request所在的请求队列

unsigned int \_\_data\_len; /\* request传输数据的字节总数 \*/

sector\_t \_\_sector; /\*要传送的起始扇区号\*/

struct bio \*bio; /\*请求中第一个没有完成传送操作的bio\*/

struct bio \*biotail; /\*请求链表中最后一个bio\*/

struct hlist\_node hash; /\* merge hash \*/

struct gendisk \*rq\_disk; /\*请求所对应的磁盘对象\*/

struct hd\_struct \*part;

unsigned short nr\_phys\_segments; /\*请求的物理段数(物理内存上不相邻的段数)\*/

void \*special;

char \*buffer; /\*当前数据传送的内存缓冲区的指针\*/

unsigned char \_\_cmd[BLK\_MAX\_CDB];

unsigned char \*cmd; //存放给磁盘适配器的命令的缓冲区

unsigned short cmd\_len; //cmd的长度

rq\_end\_io\_fn \*end\_io; //request完成的回调函数

void \*end\_io\_data; //回调函数的参数

/\* for bidi \*/

struct request \*next\_rq;

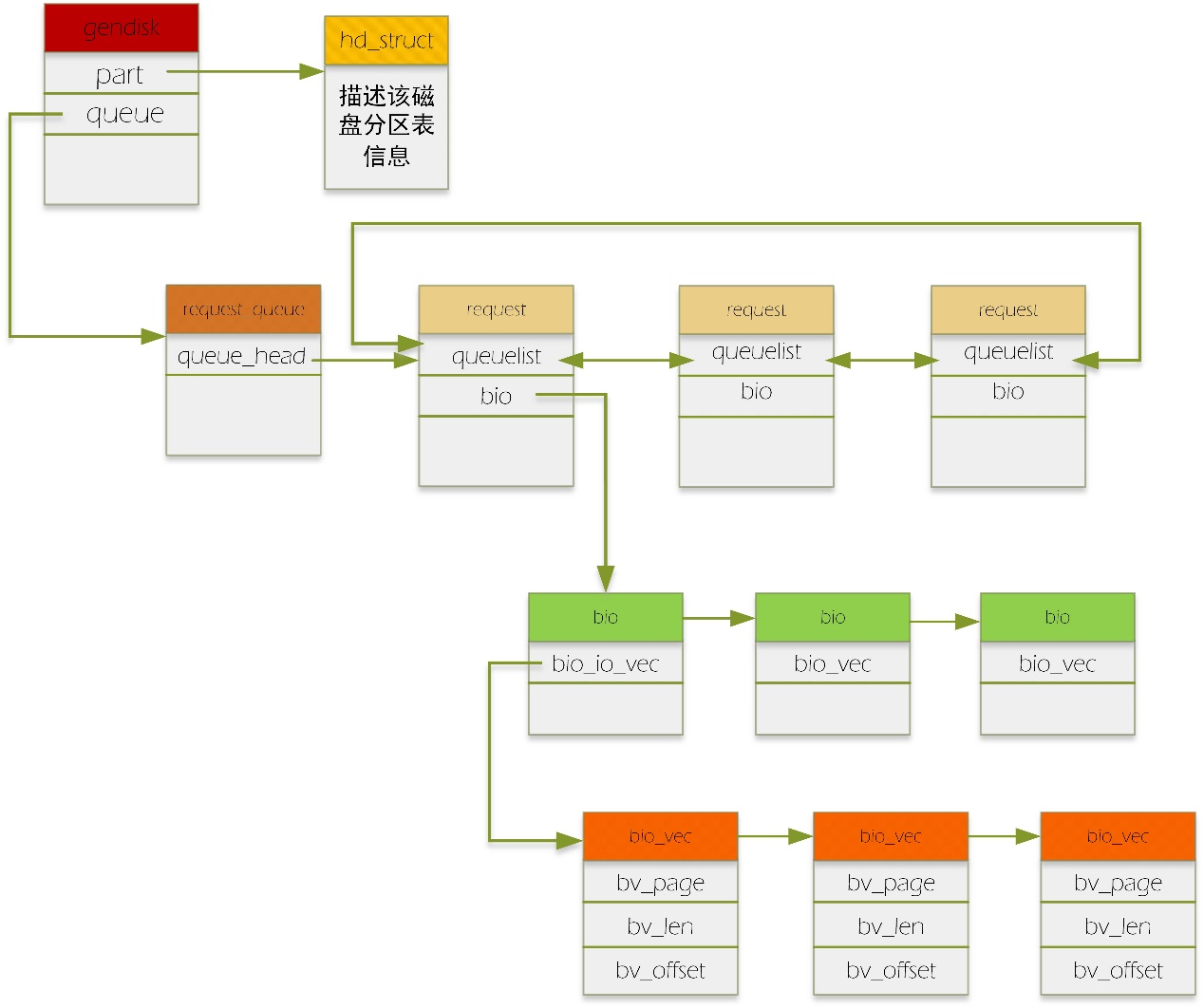
}

每个请求包含一个或多个bio结构，bio之间用有序链表连接起来，按bio起始扇区的位置从小到大，而且这些bio之间在磁盘扇区是相邻的，也就是说一个bio的结尾刚好是下一个bio的开头。

最初，通用块层创建一个仅包含一个bio结构的请求。可能存在新bio与请求中已存在的数据物理相邻的情况，就把bio加入该请求，否则用该bio初始化一个新的请求。

request\_queue：每个磁盘对应一个request\_queue.该队列挂的就是request请求。

具体如下图：(有颜色方框头表示数据结构的名字)



请求到达block层后，通过generic\_make\_request这个入口函数，在通过调用一系列相关的函数把bio变成了request。具体的做法如下：如果几个bio要读写的区域是连续的，即积攒成一个request（一个request上挂多个连续的bio，就是我们通常说的“合并bio请求”），如果一个bio跟其他的bio都连不上，那它就自己创建一个新的request，把自己挂在这个request下。当然，合并bio的个数也是有限的，这个可以通过配置文件配置。上层的一次请求可能跨越了多个扇区，形成不连续的扇区段，那么该请求构造的每个bio对应着一个连续的扇区段。故一个请求可以构造出多个bio。

合并后的request放入每个设备对应的request\_queue中。之后设备驱动调用peek\_request从request\_queue中取出request，进行下一步处理。

这里要注意的是，在实现设备驱动时，厂家可以直接从request\_queue中拿出排队好的request，也可以实现自己的bio排队方法，即实现自己的make\_request\_fn方法，即直接拿文件系统传来的bio来自己进行排队，按需设计。

我们完全可以绕过文件系统，绕过页高速缓存，直接构造我们的bio请求并交给通用块层，以此完成我们对磁盘（块设备）上数据的读取。

以下贴出我自己构造bio请求读磁盘的过程,亲测有效：

1. struct bio \*bio = bio\_alloc(GFP\_NOIO, 1); 首先通过bio\_alloc分配一个bio结构。
2. struct block\_device \*dev= lookup\_bdev(“/dev/sdb”); 通过设备文件路径获得我们想读写的块设备的描述符。/dev/sdb是我的第二块磁盘，没有进行过格式化。
3. 初始化bio，其中我们对bio中page调用了alloc\_pages\_current(GFP\_KERNEL,0)来获得一个页框，bio->bi\_end\_io初始化为我们自己写的函数，bio请求完成时会触发该函数。
4. blkdev\_get(dev,FMODE\_READ,0) 此步骤非常重要，其实bd\_acquire（lookup\_bdev会在内部调用它）确实可以获取一个block\_device，根据inode的dev\_t来确定。这里的block\_device完全是个空壳，唯一的最重要信息就是dev\_t的bdev->bd\_dev,具体怎么和设备关联起来都是在blkdev\_get做的。
5. submit\_bio(READ, bio);提交，上层通过它把构建好的bio交给通用块层进行处理，如果没有第四步，会出错，因为bio->bi\_bdev->bd\_disk会为NULL,正如4所说，而如果是/dev/sda则无影响，没有4依然可以，因为sda被文件系统格式化过，系统中有其他程序已经调用过blkdev\_get将bdev和具体的设备关联在一起了。
6. 最后bio请求（我们这里是读请求）完成后，调用我们自己定义的end\_io函数，注意我们要在完成时记得将bio结构还给内核，通过调用bio\_put完成归还

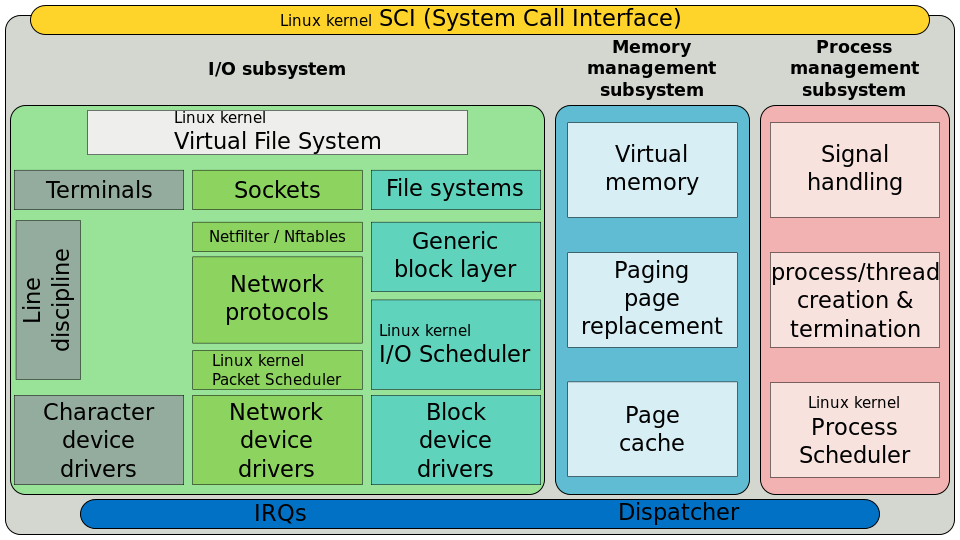
# [Linux IO Scheduler（Linux IO 调度器）](https://www.cnblogs.com/cobbliu/p/5389556.html)

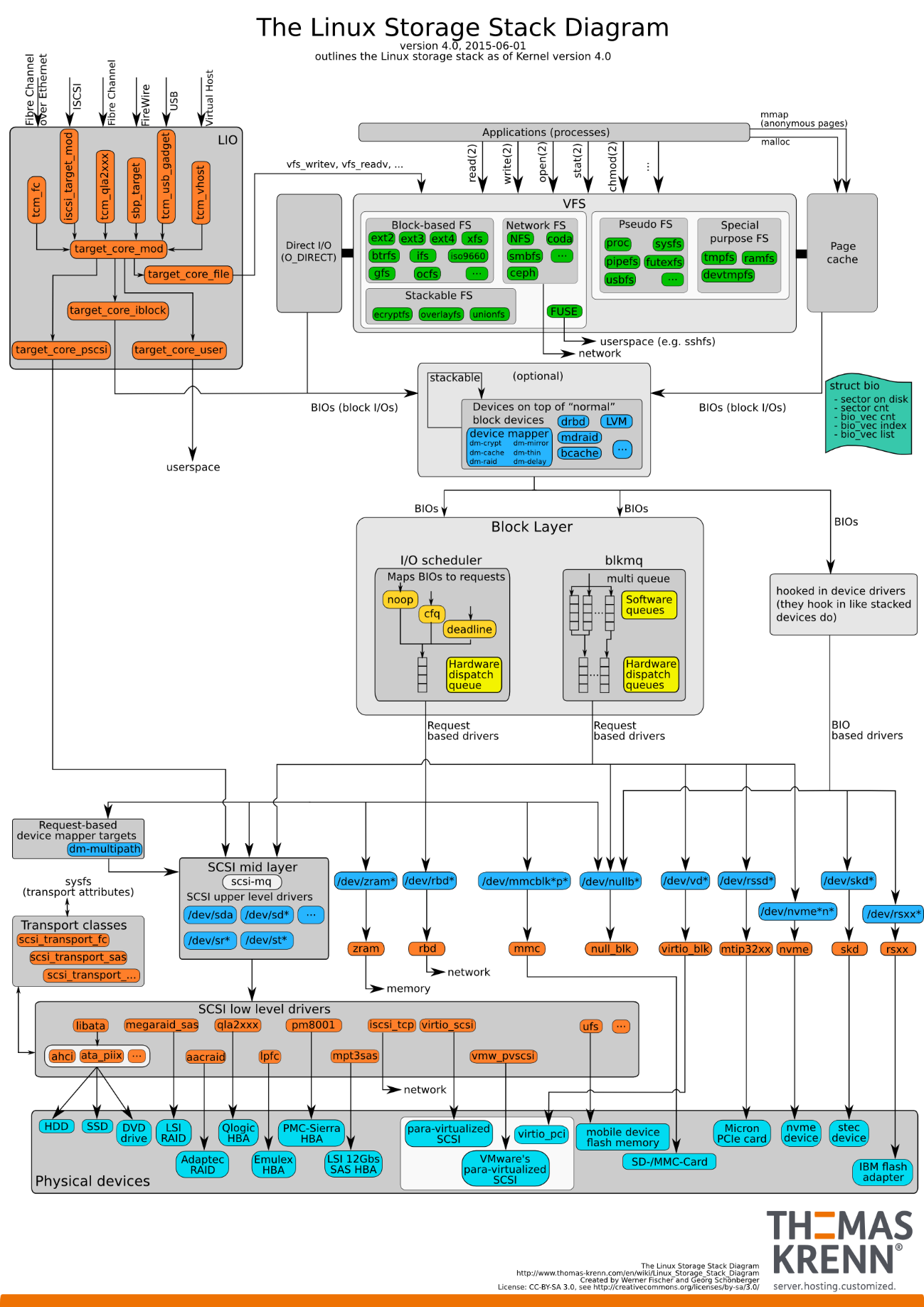
每个块设备或者块设备的分区，都对应有自身的请求队列(request\_queue),而每个请求队列都可以选择一个I/O调度器来协调所递交的request。I/O调度器的基本目的是将请求按照它们对应在块设备上的扇区号进行排列，以减少磁头的移动，提高效率。每个设备的请求队列里的请求将按顺序被响应。实际上，除了这个队列，每个调度器自身都维护有不同数量的队列，用来对递交上来的request进行处理，而排在队列最前面的request将适时被移动到请求队列中等待响应。

IO调度发生在Linux内核的IO调度层。这个层次是针对Linux的整体IO层次体系来说的。从read()或者write()系统调用的角度来说，Linux整体IO体系可以分为七层，它们分别是：

1. VFS层：虚拟文件系统层。由于内核要跟多种文件系统打交道，而每一种文件系统所实现的数据结构和相关方法都可能不尽相同，所以，内核抽象了这一层，专门用来适配各种文件系统，并对外提供统一操作接口。
2. 页缓存层：负责针对page的缓存。
3. 文件系统层：不同的文件系统实现自己的操作过程，提供自己特有的特征，具体不多说了，大家愿意的话自己去看代码即可。
4. 通用块层：由于绝大多数情况的io操作是跟块设备打交道，所以Linux在此提供了一个类似vfs层的块设备操作抽象层。下层对接各种不同属性的块设备，对上提供统一的Block IO请求标准。
5. IO调度层：因为绝大多数的块设备都是类似磁盘这样的设备，所以有必要根据这类设备的特点以及应用的不同特点来设置一些不同的调度算法和队列。以便在不同的应用环境下有针对性的提高磁盘的读写效率，这里就是大名鼎鼎的Linux电梯所起作用的地方。针对机械硬盘的各种调度方法就是在这实现的。
6. 块设备驱动层：驱动层对外提供相对比较高级的设备操作接口，往往是C语言的，而下层对接设备本身的操作方法和规范。
7. 块设备层：这层就是具体的物理设备了，定义了各种真对设备操作方法和规范

     IO调度器在内核栈中所处位置如下：





内核中实现的IO调度器主要有四种--Noop,Deadline,CFG, Anticipatory。

### 1，Noop算法

[Noop调度算法](https://en.wikipedia.org/wiki/Noop_scheduler)是内核中最简单的IO调度算法。Noop调度算法也叫作电梯调度算法，它将IO请求放入到一个FIFO队列中，然后逐个执行这些IO请求，当然对于一些在磁盘上连续的IO请求，Noop算法会适当做一些合并。这个调度算法特别适合那些不希望调度器重新组织IO请求顺序的应用。

     这种调度算法在以下场景中优势比较明显：

     1）在IO调度器下方有更加智能的IO调度设备。如果您的Block Device Drivers是Raid，或者SAN，NAS等存储设备，这些设备会更好地组织IO请求，不用IO调度器去做额外的调度工作；

     2）上层的应用程序比IO调度器更懂底层设备。或者说上层应用程序到达IO调度器的IO请求已经是它经过精心优化的，那么IO调度器就不需要画蛇添足，只需要按序执行上层传达下来的IO请求即可。

     3）对于一些非旋转磁头氏的存储设备，使用Noop的效果更好。因为对于旋转磁头式的磁盘来说，IO调度器的请求重组要花费一定的CPU时间，但是对于SSD磁盘来说，这些重组IO请求的CPU时间可以节省下来，因为SSD提供了更智能的请求调度算法，不需要内核去画蛇添足。[这篇文章](http://www.nuodb.com/techblog/tuning-linux-io-scheduler-ssds)提及了SSD中使用Noop效果会更好。

### 2，Deadline算法

     Deadline算法的核心在于保证每个IO请求在一定的时间内一定要被服务到，以此来避免某个请求饥饿。

     Deadline算法中引入了四个队列，这四个队列可以分为两类，每一类都由读和写两类队列组成，一类队列用来对请求按起始扇区序号进行排序，通过红黑树来组织，称为sort\_list；另一类对请求按它们的生成时间进行排序，由链表来组织，称为fifo\_list。每当确定了一个传输方向(读或写)，那么将会从相应的sort\_list中将一批连续请求dispatch到requst\_queue的请求队列里，具体的数目由fifo\_batch来确定。只有下面三种情况才会导致一次批量传输的结束：

    1）对应的sort\_list中已经没有请求了

    2）下一个请求的扇区不满足递增的要求

    3）上一个请求已经是批量传输的最后一个请求了。

     所有的请求在生成时都会被赋上一个期限值(根据jiffies)，并按期限值排序在fifo\_list中，读请求的期限时长默认为为500ms，写请求的期限时长默认为5s，可以看出内核对读请求是十分偏心的，其实不仅如此，在deadline调度器中，还定义了一个starved和writes\_starved，writes\_starved默认为2，可以理解为写请求的饥饿线，内核总是优先处理读请求，starved表明当前处理的读请求批数，只有starved超过了writes\_starved后，才会去考虑写请求。因此，假如一个写请求的期限已经超过，该请求也不一定会被立刻响应，因为读请求的batch还没处理完，即使处理完，也必须等到starved超过writes\_starved才有机会被响应。为什么内核会偏袒读请求？这是从整体性能上进行考虑的。读请求和应用程序的关系是同步的，因为应用程序要等待读取的内容完毕，才能进行下一步工作，因此读请求会阻塞进程，而写请求则不一样，应用程序发出写请求后，内存的内容何时写入块设备对程序的影响并不大，所以调度器会优先处理读请求。

     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。

     在一些多线程应用下，Deadline算法比CFQ算法好。在一些数据库应用下，Deadline算法比CFQ算法好。

### 3，Anticipatory算法

    Anticipatory算法的核心是局部性原理，它期望一个进程昨晚一次IO请求后还会继续在此处做IO请求。在IO操作中，有一种现象叫“假空闲”（Deceptive idleness），它的意思是一个进程在刚刚做完一波读操作后，看似是空闲了，不读了，但是实际上它是在处理这些数据，处理完这些数据之后，它还会接着读，这个时候如果IO调度器去处理另外一个进程的请求，那么当原来的假空闲进程的下一个请求来的时候，磁头又得seek到刚才的位置，这样大大增加了寻道时间和磁头旋转时间。所以，Anticipatory算法会在一个读请求做完后，再等待一定时间t（通常是6ms），如果6ms内，这个进程上还有读请求过来，那么我继续服务，否则，处理下一个进程的读写请求。

     在一些场景下，Antocipatory算法会有非常有效的性能提升。值得一提的是，Anticipatory算法从Linux 2.6.33版本后，就被移除了，因为CFQ通过配置也能达到Anticipatory算法的效果。

### 4，CFQ算法

    CFQ（**Completely Fair Queuing**）算法，顾名思义，绝对公平算法。它试图为竞争块设备使用权的所有进程分配一个请求队列和一个时间片，在调度器分配给进程的时间片内，进程可以将其读写请求发送给底层块设备，当进程的时间片消耗完，进程的请求队列将被挂起，等待调度。 每个进程的时间片和每个进程的队列长度取决于进程的IO优先级，每个进程都会有一个IO优先级，CFQ调度器将会将其作为考虑的因素之一，来确定该进程的请求队列何时可以获取块设备的使用权。IO优先级从高到低可以分为三大类:RT(real time),BE(best try),IDLE(idle),其中RT和BE又可以再划分为8个子优先级。实际上，我们已经知道CFQ调度器的公平是针对于进程而言的，而只有同步请求(read或syn write)才是针对进程而存在的，他们会放入进程自身的请求队列，而所有同优先级的异步请求，无论来自于哪个进程，都会被放入公共的队列，异步请求的队列总共有8(RT)+8(BE)+1(IDLE)=17个。

    从Linux 2.6.18起，CFQ作为默认的IO调度算法。 对于通用的服务器来说，CFQ是较好的选择。

### 5，更改IO调度算法

在RHEL5/OEL5以及之后的版本中（比如RHEL6和RHEL7），可以针对每块磁盘制定I/O Scheduler，修改完毕立刻生效，比如：

[复制代码](javascript:void(0);)

$ cat /sys/block/sda1/queue/scheduler

[noop] anticipatory deadline cfq

#修改为cfq

$ echo 'cfq'>/sys/block/sda1/queue/scheduler

#立刻生效

$ cat /sys/block/sda1/queue/scheduler

noop anticipatory deadline [cfq]

调度算法源文件在内核代码 /block文件夹下