Linux通用块设备层

http://www.ilinuxkernel.com

目 录

图目录

1 概述

在块设备上的操作，涉及内核中的多个组成部分，如图1所示。假设一个进程使用系统调用read（）读取磁盘上的文件。下面步骤是内核响应进程读请求的步骤；

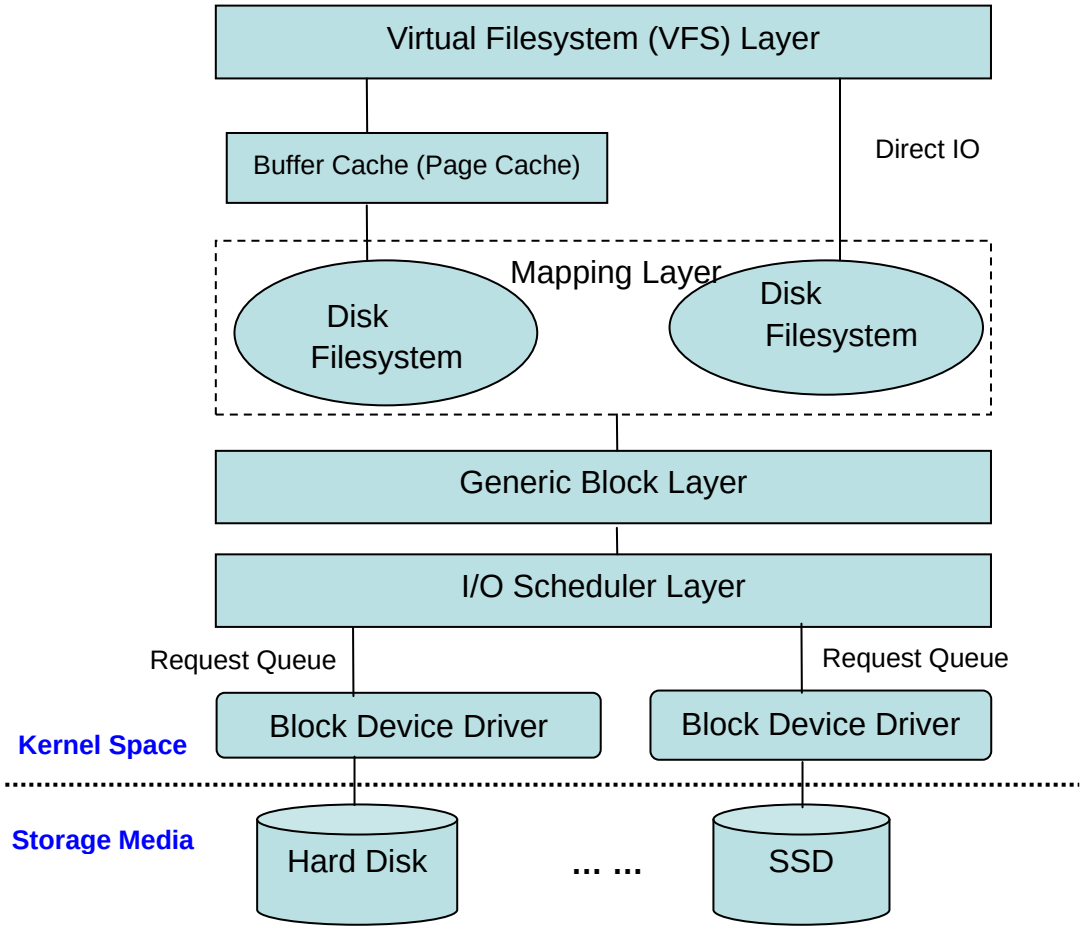


图1 Linux内核块设备I/O流程

（1）系统调用read（）会触发相应的VFS（Virtual Filesystem Switch）函数，传递的参数有文件描述符和文件偏移量。

（2）VFS确定请求的数据是否已经在内存缓冲区中；若数据不在内存中，确定如何执行读操作。

（3）假设内核必须从块设备上读取数据，这样内核就必须确定数据在物理设备上的位置。这由映射层（Mapping Layer）来完成。

（4）此时内核通过通用块设备层（Generic Block Layer）在块设备上执行读操作，启动I/O操作，传输请求的数据。

（5）在通用块设备层之下是I/O调度层（I/O Scheduler Layer），根据内核的调度策略，对等待的I/O等待队列排序。

（6）最后，块设备驱动（Block Device Driver）通过向磁盘控制器发送相应的命令，执行真正的数据传输。

对于（1）、（2）两个步骤，在Linux虚拟文件系统中，我们讨论了VFS（Virtual Filesystem Switch）主要数据结构和操作，结合相关系统调用（如sys_read（）、sys_write（）等）的源码，我们不难理解VFS层相关的操作和实现。而对于第（3）步中的Mapping Layer需要结合具体的文件系统解释，我们暂时不考虑。

本文主要分析（4）步，就是通用块设备层。

2 块设备相关的概念

系统中能够随机访问固定大小数据片（chunk）的设备称为块设备，这些数据片就称作块。最常见的块设备是硬盘，除此之外，还有CD-ROM驱动器和SSD等。它们通常安装文件系统的方式使用。

内核管理块设备要比管理字符设备复杂。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。所以内核不必提供一个专门的子系统来管理字符设备，但是对于块设备的管理却必须要有一个专门提供服务的子系统。不仅仅是因为块设备的复杂性远远高于字符设备，更重要的原因是块设备对执行性能要求很高；对硬盘每多一分利用都会对系统的整体性能带来提升，其效果要远远比键盘吞吐速度成倍的提高大得多。另外，块设备的复杂性会为这种优化留下很大的空间。

2.1 扇区（Sectors）

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，而最常见的大小是512字节。扇区的大小是设备的物理属性，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，不过许多块设备能够一次传输多个扇区。虽然大多数块设备的扇区大小都是512字节，不过其他大小的扇区也很常见（比如，很多CD-ROM盘的扇区都是2k大小）。若块设备使用的扇区大于512字节，则相应的底层驱动程序负责相应的转换工作。

数据在块设备上的位置由块索引和块内偏移确定。块索引（sector indices）在32或64位系统上的变量类型为sector_t。

2.2 块（Blocks）

对于硬件设备来说，扇区是基本的数据传输单元；而对于VFS（虚拟文件系统），块（block）是基本的数据传输单元。例如，当内核访问文件的数据时，它首先从磁盘上读取一个块，这个块有文件的inode，该块对应于磁盘上一个或多个扇区。

由于扇区是块设备的最小可寻址单元，所以块不能比扇区还小，只能整数倍于扇区大小。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，而且不能超过一页的长度。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、1K或4K。

至于块（block）大小，与具体的硬件设备无关。在块设备上创建文件系统时，管理员可以选择块大小；因此在同一个磁盘上，多个分区可能使用不同的块大小。

2.3 段（Segments）

每个磁盘I/O操作都是设备上相邻的扇区与内存之间传输数据。在大多数情况下，数据的传输通过DMA方式；块设备驱动向磁盘控制器发起命令，触发数据传输；当数据传输结束时，控制器给块设备驱动发送一个中断。

简单的DMA操作，只能传输磁盘上相邻的扇区。这是物理属性的限制：若磁盘控制器DMA传输在非连续的扇区上，则会导致性能下降；因为在磁盘表面上移动读/写磁头，是相当耗时的工作。

旧磁盘控制器，仅支持简单的DMA操作：每次数据传输，数据在内存中也是连续的。现在的磁盘控制器支持“分散/聚合”（scatter-gather）DMA操作，这种操作模式下，数据传输可以在多个非连续的内存区域中进行。

对于每个“分散/聚合”DMA操作，块设备驱动向磁盘控制器发送：

- 初始磁盘扇区号和传输的总共扇区数；
- 内存区域的描述链表，每个描述都包含一个地址和长度。

磁盘控制器来管理整个数据传输；例如，在读操作中，控制器从磁盘相邻的扇区上读取

数据，然后将数据分散存储在内存的不同区域。

为了利用“分散/聚合”DMA操作，块设备驱动必须能处理被称为段（segments）的数据单元。一个段就是一个内存页面或一个页面的部分，它包含磁盘上相邻扇区的数据。这样一个“分散/聚合”DMA操作可能会涉及多个段。

这里说明一下，块设备驱动不需要知道块、块大小和块缓冲区。进而，即使 上层看到段包含几个块缓冲区，块设备驱动并不在意它。

我们会在后面的内容中看到，在通用块设备层（generice block layer）中，若多个页面在内存中连续，且相应的磁盘数据在磁盘上也相邻，那么可以将多个段合并。合并操作产生的大内存区域称为物理段（physical segment）。

在内核中有许多与块设备存储有关；每个部分管理磁盘数据时，使用不同的数据长度。图2是一个页面中典型的磁盘数据布局。

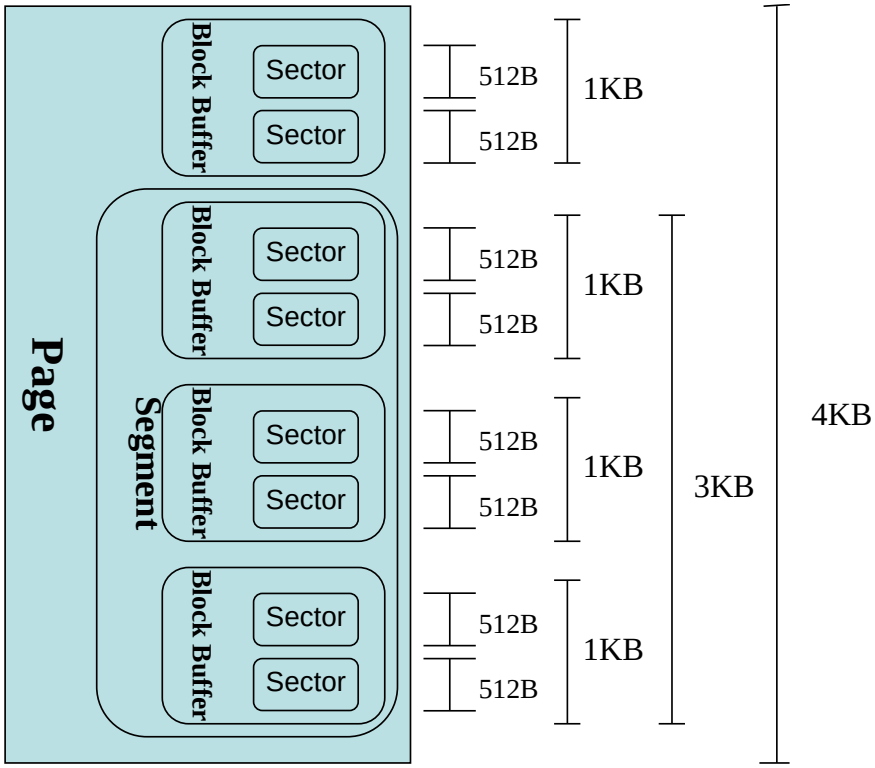


图2 页内磁盘数据布局

- 块设备的控制器传输的固定数据单元大小称为扇区（sector）。因此I/O调度器和块设备驱动必须以扇区为单位管理数据。
- 虚拟文件系统、映射层（mapping layer）管理磁盘数据的逻辑单元大小称为块（block）。对于文件系统来说，块是最小的磁盘数据存储单元。

- 前面在分散/聚合DMA中，我们提到块设备驱动应该能够处理称为“段”的数据单元；每个“段”是内存中的一页或页的一部分，“段”中的数据在磁盘上是连续的。
- 磁盘缓冲区处理的数据单元大小为“页”，每个对应一个页帧。（注1）
- 通用块设备层粘合所有上层和底层的部分，这样它就知道扇区、块、段和数据页。

注：页（page）和页帧（page frame）的区别。事实上，页和页帧大小都是一样的，一般情况下都是4KB。页是逻辑上的概念，页帧是物理上的4K连续RAM单元（4K对齐）。

3 通用块设备层

通用块设备层（Generic Block Layer）是内核的一个组成部分，它处理系统所有对块设备的请求。有通用块设备层后，内核可以方便地：

- 将数据存放在高端内存—当CPU访问高端内存的数据时，页就被临时映射到内核的线性地址空间，然后解除映射。
- 实现零拷贝（zero-copy），即磁盘数据直接拷贝到用户地址空间，而不需要先拷贝到内核地址空间。实际上就是，内核进行I/O数据传输使用的页面被映射到用户进程的地址空间中。
- 管理逻辑卷，如LVM（Logical Volume Manager）和RAID（Redundant Array of Inexpensive Disks）。
- 使用现在的磁盘控制器新特性，如大的磁盘缓存、增强的DMA功能、板上I/O请求调度等。

3.1 缓冲区和缓冲区头

当一个块被调入内存时（也就是，在读入或等待写出时），它要存储在一个缓冲区中。每个缓冲区都有一个块对应，它相当于是磁盘块在内存中的表示。前面提到过，块包含一个或多个扇区，但大小不能超过一个页面，所以一个页面可以容纳一个或多个内存中的块。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，块对应于哪一个缓冲区等），所以每个缓冲区都有一个对应的描述符。

每个块都要有自己的块缓冲区（block buffer），内核使用内存上的这块缓冲区来保存块数据。当内核从块设备上读取一个块时，就用从硬件上读取的数据填充块缓冲区；同理，当内核向块设备写数据时，就用块缓冲区中的数据写到块设备上。每个缓冲区都有一个缓冲区头（buffer head），描述类型为buffer_head，这个数据结构里包含了内核处理缓冲区所需要的信息；于是，在每个缓冲区上操作之前，内核都要先检查它的缓冲区头。buffer_head的定义在文件include/linux/buffer_head.h中。

```
00061: struct buffer_head {
00062:     unsigned long b_state;          /* buffer state bitmap (see above) */
00063:     struct buffer_head *b_this_page;/* circular list of page's buffers */
00064:     struct page *b_page;            /* the page this bh is mapped to */
00065:
00066:   sector_t b_blocknr;     /* start block number */
00067:     size_t b_size;            /* size of mapping */
00068:     char *b_data;             /* pointer to data within the page */
00069:
00070:     struct block_device *b_bdev;
00071:     bh_end_io_t *b_end_io;      /* I/O completion */
00072:     void *b_private;        /* reserved for b_end_io */
00073:     struct list_head b_assoc_buffers;   /* associated with another mapping */
00074:     struct address_space *b_assoc_map;        /* mapping this buffer is
00075:                                 associated with */
00076:     atomic_t b_count;           /* users using this buffer_head */
00077: };
```

各个成员变量含义：

b_state：缓冲区状态标志；

b_this_page：页面中的缓冲区；

b_page：存储缓冲区的页面；

b_blocknr：逻辑块号；

b_size：块大小（以字节为单位）；

b_data：页面中的缓冲区；

b_dev：块设备；

b_end_io：I/O完成方法；

b_private：I/O完成方法使用的数据；

b_assoc_buffers：相关的映射链表；

b_assoc_map：当前buffer所属的address_space；

b_count：缓冲区使用计数；

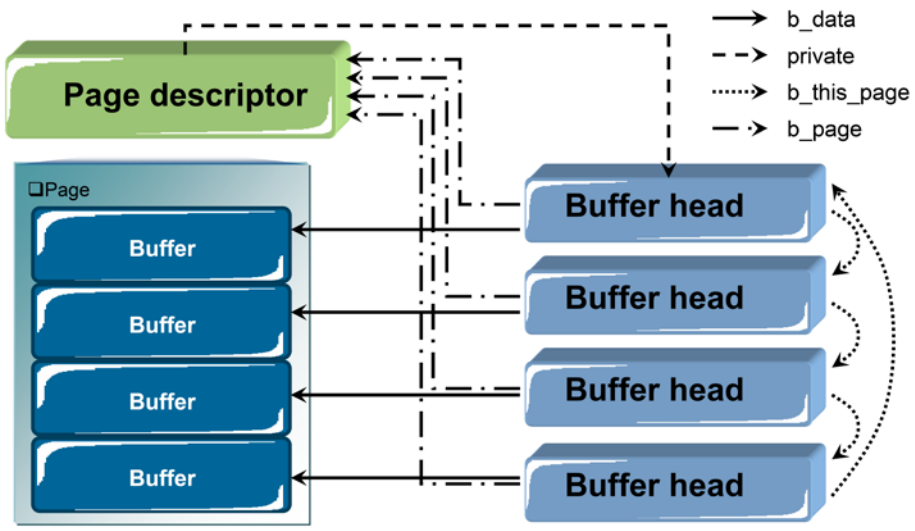


图3 缓冲页与缓冲区头的关系

b_state域表示缓冲区的状态，可以是表1中一种标志或多种标志的组合。合法的标志存放在bh_state_bits枚举中，其定义也在文件include/linux/buffer_head.h中。

```
00019: enum bh_state_bits {
00020:     BH_Uptodate,    /* Contains valid data */
00021:     BH_Dirty,       /* Is dirty */
00022:     BH_Lock,/* Is locked */
00023:     BH_Req,         /* Has been submitted for I/O */
00024:     BH_Uptodate_Lock,/* Used by the first bh in a page, to serialise
00025:                 * IO completion of other buffers in the page
00026:                 */
00027:
00028:     BH_Mapped, /* Has a disk mapping */
00029:     BH_New,         /* Disk mapping was newly created by get_block */
00030:     BH_Async_Read,      /* Is under end_buffer_async_read I/O */
00031:     BH_Async_Write,         /* Is under end_buffer_async_write I/O */
00032:     BH_Delay,       /* Buffer is not yet allocated on disk */
00033:     BH_Boundary,        /* Block is followed by a discontiguity */
00034:     BH_Write_EIO,   /* I/O error on write */
00035:     BH_Ordered,     /* DEPRECATED: ordered write */
00036:     BH_Eopnotsupp,      /* DEPRECATED: operation not supported (barrier) */
00037:     BH_Unwritten,       /* Buffer is allocated on disk but not written */
00038:     BH_Quiet,       /* Buffer Error Prinks to be quiet */
00039:
00040:     BH_PrivateStart,/*    not a state bit, but the first bit available
00041:                 * for private allocation by other entities
00042:                 */
00043: };
```

表1 bh_state标志含义

标志	含义
BH_Uptodate	buffer里面的数据是有效的
BH_Dirty	buffer里面的数据是脏的，即当前数据比磁盘上的数据更新，需要回写到磁盘上
BH_Lock	buffer里的数据正在进行磁盘I/O，被锁住，防止并发访问

BH_Req	buffer正在处理I/O请求状态
BH_Mapped	buffer里的数据有效，且与磁盘上的数据块建立映射
BH_New	buffer已经通过get_block（）新建立映射，但还未访问
BH_Async_Read	buffer正在通过end_buffer_async_read（）进行异步读
BH_Async_Write	buffer正在通过end_buffer_async_write（）进行异步写
BH_Delay	磁盘在还没有为buffer分配数据块
BH_Boundary	当前块之后，为不连续的块
BH_Write_EIO	buffer出现写错误
BH_Ordered	ordered写
BH_Eopnosupp	不支持的操作
BH_Unwritten	在磁盘上已经为buffer分配空间，但还未回写
BH_Quiet	取消buffer错误打印

bh_state_bits列表还包含了一个特殊标志—BH_PrivateStart，该标志不是可用状态标志，使用它是为了指明可被其他代码使用的起始位。块I/O层不会使用BH_PrivateStart或更高的位，那么某个驱动程序希望通过b_state域存储信息时就可以安全地使用这些位。驱动程序可以在这些位中定义自己的状态标志，只要保证自定义的状态标志不与块I/0层的专用位发生冲突就可以了。

b_count域表示缓冲区的使用计数，可以通过内联函数get_bh（）和put_bh（）来此域增减，其定义也在文件include/linux/buffer_head.h中。

```
00279: static inline void get_bh(struct buffer_head *bh)
00280: {
00281:     atomic_inc(&bh->b_count);
00282: }
00283:
00284: static inline void put_bh(struct buffer_head *bh)
00285: {
00286:     smp_mb__before_atomic_dec();
00287:     atomic_dec(&bh->b_count);
00288: }
00289:
```

在操作缓冲区头之前，应该先使用get_bh（）函数增加缓冲区头的引用计数，确保该缓冲区头不会再被释放；当完成对缓冲区头的操作之后，还必须使用put_bh（）函数减少引用计数。

与缓冲区对应的磁盘物理块由b_blocknr域索引，该值是b_bdev域指明的块设备中的逻辑块号。

与缓冲区所对应的内存物理页由b_page域表示，另外，b_data域直接指向相应的块（它

位于b_page域所指明的页面中某个位置上），块的大小由b_size域表示，所以块在内存中的起始位置在b_data处，结束位置在（b_data+b_size）处。

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。

3.2 bio结构体

I/O操作的基本容器由bio结构体表示，它定义在文件include/linux/bio.h中。该结构体代表了正在活动的以段（segment）链表形式组织的块I/O操作。一个段是一小块连续的内存缓冲区。这样，单个缓冲区就不一定要连续。所以使用段来描述缓冲区，即使一个缓冲区分散在内存的多个位置上，bio结构体也能对内核保证I/O操作的执行。这样的向量I/O称为分散-聚合I/O。

bio结构体的定义如下：

```
00060: /*
00061:  * main unit of I/O for the block layer and lower layers (ie drivers and
00062:  * stacking drivers)
00063:  */
00064: struct bio {
00065:     sector_t        bi_sector;      /* device address in 512 byte
00066:
00067:     struct bio      *bi_next;       /* request queue link */
00068:     struct block_device     *bi_bdev;
00069:     unsigned long       bi_flags;       /* status, command, etc */
00070:     unsigned long       bi_rw;      /* bottom bits READ/WRITE,
00071:                          * top bits priority
00072:                          */
00073:
00074:     unsigned short      bi_vcnt;        /* how many bio_vec's */
00075:     unsigned short      bi_idx;         /* current index into bvl_vec */
00076:
00077:     /* Number of segments in this BIO after
00078:      * physical address coalescing is performed.
00079:      */
00080:     unsigned int        bi_phys_segments;
00081:
00082:     unsigned int        bi_size;    /* residual I/O count */
00083:
00084:     /*
00085:      * To keep track of the max segment size, we account for the
00086:      * sizes of the first and last mergeable segments in this bio.
00087:      */
00088:     unsigned int        bi_seg_front_size;
00089:     unsigned int        bi_seg_back_size;
00090:
00091:     unsigned int        bi_max_vecs;    /* max bvl_vecs we can hold */
00092:
00093:     unsigned int        bi_comp_cpu;    /* completion CPU */
00094:
00095:     atomic_t        bi_cnt;         /* pin count */
00096:
00097:     struct bio_vec      *bi_io_vec;     /* the actual vec list */
00098:
```

```
00099:     bio_end_io_t       *bi_end_io;
00100:
00101:     void               *bi_private;
00102: #if defined(CONFIG_BLK_DEV_INTEGRITY)
00103:     struct bio_integrity_payload *bi_integrity;    /* data integrity */
00104: #endif
00105:
00106:     bio_destructor_t       *bi_destructor;     /* destructor */
00107:
00108:     /*
00109:      * We can inline a number of vecs at the end of the bio, to avoid
00110:      * double allocations for a small number of bio_vecs. This member
00111:      * MUST obviously be kept at the very end of the bio.
00112:      */
00113:     struct bio_vec      bi_inline_vecs[0];
00114: } « end bio » ;
```

bio结构体中各成员变量的含义：

bi_sector：磁盘上相关的扇区

bi_next：请求链表

bi_bdev：相关的块设备

bi_flags：状态和命令标志

bi_rw：读还是写

bi_vcnt：bio_vecs偏移量

bi_idx：bi_io_vec的当前索引

bi_phys_segments：结合后的段数目

bi_hw_segments：重新映射後的段数目

bi_size：I/O计数

bi_hw_front_size：第一个可合并的段大小

bi_hw_back_size：最后一个可合并的段大小

bi_max_vecs：bio_vecs数目上限

bi_io_vec：bio_vec链表

bi_end_io：I/O完成方法

bi_cnt：使用计数

bi_private：拥有者的私有方法

bi_desctructor：销毁方法

使用bio结构体的目的主要是代表正在现场执行的I/O操作，所以该结构体中的主要成员变量都是用来管理相关信息的，其中最重要的几个成员变量是bi_io_vecs、bi_vcnt和

bi_idx。图4显示了bio结构体及相关结构体之间的关系。

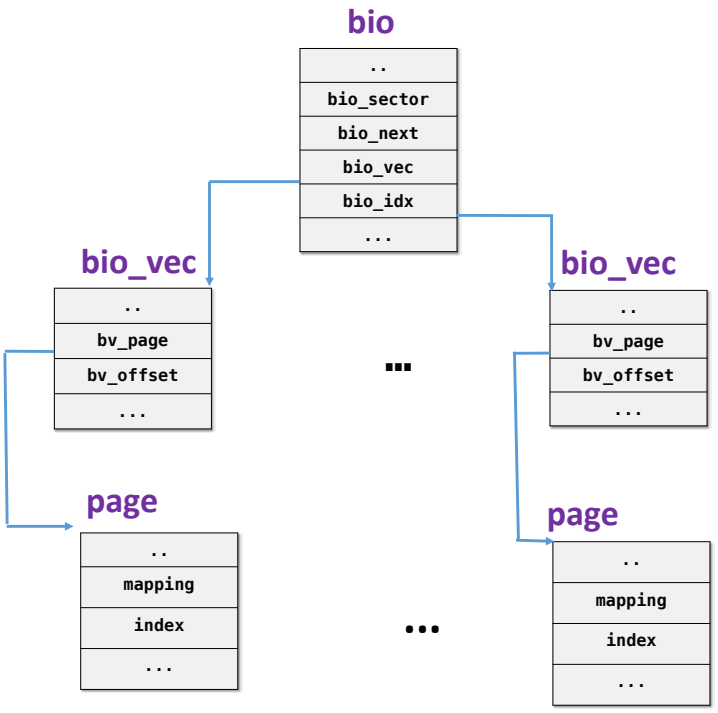


图4 bio结构体、bio_vec结构体和page之间的关系

bi_io_vecs指向一个bio_vec结构体数组，该结构体链表包含了一个特定I/O操作所需要使用到的所有段（segment）。每个bio_vec结构都是一个形式为<page, offset, len>的向量，它描述的是一个特定的段：段所在的物理页、块在物理页中的偏移量、从给定偏移量开始的块长度。整个bio_io_vec结构体数组表示了一个完整的缓冲区。bio_vec结构体定义在文件include/linux/bio.h中。

```
00045: /*
00046:  * was unsigned short, but we might as well be ready for > 64kB I/O pages
00047:  */
00048: struct bio_vec {
00049:     struct page   *bv_page;
00050:     unsigned int   bv_len;
00051:     unsigned int   bv_offset;
00052: };
00053:
```

bv_page：指向这个缓冲区所驻留的物理页面；

bv_len：这个缓冲区以字节为单位的大小；

bv_offset：缓冲区所驻留的页中以字节为单位的偏移量。

在每个给定的块I/O操作中，bi_vcnt域用来描述bi_io_vec所指向的bio_vec数组中的向量数目。当块I/O操作执行完毕后，bi_idx指向数组的当前索引。

总之，每个块I/O请求都通过一个bio结构体表示。每个请求包含一个或多个块，这些块存储在bio_vec结构体数组中。这些结构体描述了每个片段在物理页中的实际位置，并且像向量一样被组织在一起。I/O操作的第一个段（segment）由bi_io_vec结构体所指向，其他的段在其后依次放置，共有bi_vcnt个段。当块I/O层开始执行请求，需要使用各个片段时，bi_idx就会不断更新，从而总指向当前片段。

bi_idx指向数组的当前bio_vec段，块I/O层通过它可以跟踪块I/O操作的完成进度。但该成员变量更重要的作用在于分割bio结构体。像RAID这样的驱动器可以把单独的bio结构体（原本是为单个设备使用准备的）分割到RAID阵列中的各个硬盘上去。RAID设备驱动只需拷贝这个bio结构体，再把bi_idx域设置为每个独立硬盘操作时需要的位置就可以了。

bi_cnt域记录bio结构体的使用计数，如果该域值减为0，就应该销毁该结构体，并释放它所占的内存。通过下面两个函数可以管理使用计数。

```
void bio_get(struct bio *bio)
void bio_put(struct bio *bio)
```

前者增加使用计数，后者减少使用计数（如果减到0，则销毁bio结构体）。在操作正在活动的bio结构体时，一定要首先增加它的使用计数，以免在操作过程中bio结构体被释放；相反，在操作完毕后，要减少使用计数。

最后要说明的是bi_private域，这是一个拥有者（也就是创建者）的私有域，只有创建了bio结构体的拥有者可以读写该域。

3.3 缓冲区头和bio结构体比较

缓冲区头（buffer head）和bio结构体之间存在明显差别。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割，将请求按块为单位划分，只能靠以后才能重新组合。由于bio结构是轻量级的，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结构体有以下好处：

- 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。
- 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。
- bio结构体便于执行分散-聚合I/O操作，操作中的数据可以来自多个物理页面。

- bio结构体相比缓存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。

但还是需要缓冲区头这个概念，因为它还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—它仅仅是一个向量数组，描述一个或多个块I/O操作的数据段和相关信息。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结构体仍然需要保护缓冲区信息。内核通过两种结构分别保存各自的信息，可以保证每种结构所包含的信息量尽可能地少。

4 通用块设备层对请求的处理

前面介绍了块设备的相关概念、buffer_head和bio结构体。接下来主要分析内核中通用块设备层的源码。在分析源码之前，首先介绍通用块设备层中的磁盘（disk）概念。

4.1 磁盘和磁盘分区的表示

磁盘（disk）（注1）是一个逻辑块设备，物理设备在内核中的表示，它由通用块设备层来处理。通常一个磁盘对应一个硬件块设备，如硬盘、软盘和光盘等。然而，一个磁盘也可以是多个物理磁盘分区组成的虚拟设备，也可以是内存上某些页组成。通过通用块设备层，内核中的上层部分可以以相同方式访问物理磁盘设备。

注1：这一节中，磁盘是逻辑上的设备；若是写成物理磁盘，则表示物理上的设备。这两个概念容易引起混淆。

内核使用gendisk结构，定义在include/linux/genhd.h中，来表示一个独立的磁盘设备。实际上内核还使用gendisk表示分区，但是驱动程序不需要了解这些。在gendisk结构中的许多成员必须由驱动程序进行初始化。

```
00137: struct gendisk {
00138:      /* major, first_minor and minors are input parameters only,
00139:       * don't use directly.    Use disk_devt() and disk_max_parts().
00140:       */
00141:      int major;              /* major number of driver */
00142:      int first_minor;
```

```
00143:      int minors;                 /* maximum number of minors, =1 for
00144:                                   * disks that can't be partitioned. */
00145:
00146:      char disk_name[DISK_NAME_LEN];    /* name of major driver */
00147:      char *(*devnode)(struct gendisk *gd, mode_t *mode);
00148:      /* Array of pointers to partitions indexed by partno.
00149:       * Protected with matching bdev lock but stat and other
00150:       * non-critical accesses use RCU.  Always access through
00151:       * helpers.
00152:       */
00153:      struct disk_part_tbl *part_tbl;
00154:      struct hd_struct part0;
00155:
00156:      const struct block_device_operations *fops;
00157:      struct request_queue *queue;
00158:      void *private_data;
00159:
00160:      int flags;
00161:      struct device *driverfs_dev;    // FIXME: remove
00162:      struct kobject *slave_dir;
00163:
00164:      struct timer_rand_state   *random;
00165:
00166:      atomic_t sync_io;      /* RAID */
00167:      struct work_struct async_notify;
00168: #ifdef    CONFIG_BLK_DEV_INTEGRITY
00169:      struct blk_integrity *integrity;
00170: #endif
00171:      int node_id;
00172: } « end gendisk » ;
```

major;

first_minor;

minors：磁盘使用这些成员描述设备号。一个驱动器至少使用一个设备号。如果驱动器是可被分区的（大多数情况下），用户将要为每个可能的分区都分配一个次设备号。minors的值常取16，可以包含15个分区。但是，某些磁盘驱动程序设置每个设备可使用64个次设备号；

disk_name[32]：设置磁盘设备的名字。该名字将显示在/proc/partitions和sysfs中；

fops：设置前面描述的各种设备操作；

queue：内核使用该结构为设备管理I/O请求；

flags：用来描述驱动器状态的标志（很少使用）。如果用户设备包含了可移动介质，其将设置为GENHD_FL_REMOVABLE。CD-ROM设备被设置为GENHD_FL_CD。如果处于某些原因，不希望在/proc/partitions中显示分区信息，则可将该标志设为GENHD_FL_SUPPRESS_PARTITION_INFO。

private_data：块设备驱动程序可能使用该成员保存指向其内部数据结构的指针。

driverfs_dev：指向磁盘物理设备的device目标。

slave_dir：嵌入在kobject中。

random：指向记录磁盘中断时间的数据结构；由内核随机数产生器使用。

sync_io：仅由RAID使用，记录写到磁盘的扇区数。

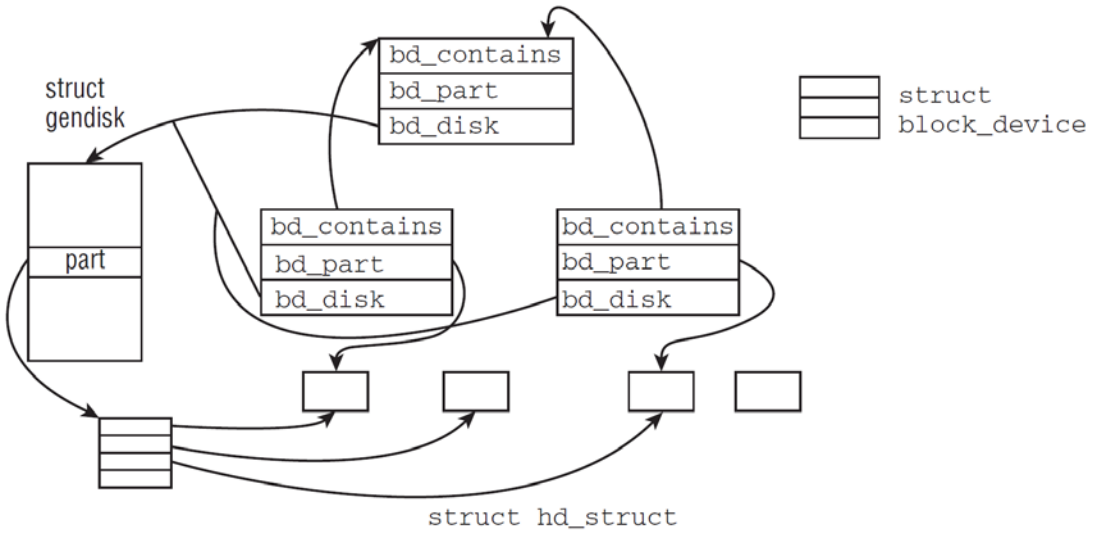
gendisk中的fops成员的数据结构类型为struct block_device_operations，其定义在include/linux/fs.h中，结构体中包括了对块设备操作的几个关键方法。

```
01335: struct block_device_operations {
01336:     int (*open) (struct block_device *, fmode_t);
01337:     int (*release) (struct gendisk *, fmode_t);
01338:     int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
01339:     int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
01340:     int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
01341:     int (*direct_access) (struct block_device *, sector_t,
01342:                           void **, unsigned long *);
01343:     int (*media_changed)   (struct gendisk *);
01344:     unsigned long long (*set_capacity) (struct gendisk *,
01345:                           unsigned long long);
01346:     int (*revalidate_disk)   (struct gendisk *);
01347:     int (*getgeo)(struct block_device *, struct hd_geometry *);
01348:     /* this callback is with swap_lock and sometimes page table lock held */
01349:     void (*swap_slot_free_notify) (struct block_device *, unsigned long);
01350:     struct module *owner;
01351: };
01352:
```

物理磁盘通常被分成多个逻辑分区。每个块设备文件可以表示一个整个物理磁盘或者其中的一个分区。如/dev/sda、/dev/sda1、/dev/sda2等。若一个物理磁盘有多个分区，则磁盘的布局保存在hd_struct数据结构数组中，数组的地址由gendisk结构体中的part成员保存。hd_struct数据结构的定义在文件include/linux/genhd.h中。

```
00090: struct hd_struct {
00091:     sector_t start_sect;
00092:     sector_t nr_sects;
00093:     sector_t alignment_offset;
00094:     unsigned int discard_alignment;
00095:     struct device __dev;
00096:     struct kobject *holder_dir;
00097:     int policy, partno;
00098: #ifdef CONFIG_FAIL_MAKE_REQUEST
00099:     int make_it_fail;
00100: #endif
00101:     unsigned long stamp;
00102:     int in_flight[2];
00103: #ifdef CONFIG_SMP
00104:     struct disk_stats *dkstats;
00105: #else
00106:     struct disk_stats dkstats;
00107: #endif
00108:     struct rcu_head rcu_head;
00109: } « end hd_struct » ;
```

hd_struct结构体中的成员变量含义很直接。start_sect是该分区在物理磁盘上的起始扇区，分区大小为nr_sects个扇区。reads,read_sectors,writes,write_sectors都是用来统计读写信息的。policy的含义为1时，表示分区只读。partno是物理磁盘上的分区索引号。



当内核在系统中发现一个新磁盘时，调用alloc_disk（）分配相关数据结构，如gendisk，hd_struct等，然后调用add_disk（）将磁盘添加到系统中。注意：一旦调用了add_disk，磁盘设备就被“激活“，表示可以使用，并随时会调用它们提供的方法。

4.2 向通用块设备层发送请求

为了描述方便，我们以ext4文件系统为例，并且假设上层请求的数据在磁盘上是连续的。以向设备上写数据为例，即以write（）系统调用为描述对象。

有时内核一次触发几个数据块的传输，而数据块物理上是不相邻的。ll_rw_block（）函数产生块设备请求；内核和设备驱动程序的很多地方都会调用这个函数。

4.2.1 读写类型

内核支持以下类型读写，定义在include/linux/fs.h中。

```
00162: #define RW_MASK          REQ_WRITE
00163: #define RWA_MASK     (1 << BIO_RW_AHEAD)
00164:
00165: #define READ         0
00166: #define WRITE            1
00167: #define READA            RWA_MASK
00168: #define SWRITE           (WRITE | READA)
00169:
00170: #define READ_SYNC    (READ    |    (1    <<    BIO_RW_SYNCIO)    |    (1    <<
BIO_RW_UNPLUG))
```

```
00171: #define READ_META     (READ | (1 << BIO_RW_META))
00172:  #define  WRITE_SYNC_PLUG      (WRITE  |  (1  <<  BIO_RW_SYNCIO)  |  (1  <<
BIO_RW_NOIDLE))
00173: #define WRITE_SYNC      (WRITE_SYNC_PLUG | (1 << BIO_RW_UNPLUG))
00174: #define WRITE_ODIRECT_PLUG   (WRITE | (1 << BIO_RW_SYNCIO))
00175: #define WRITE_META     (WRITE | (1 << BIO_RW_META))
00176: #define SWRITE_SYNC_PLUG                        \
00177:              (SWRITE | (1 << BIO_RW_SYNCIO) | (1 << BIO_RW_NOIDLE))
00178: #define SWRITE_SYNC (SWRITE_SYNC_PLUG | (1 << BIO_RW_UNPLUG))
00179: #define WRITE_BARRIER    (WRITE_SYNC | (1 << BIO_RW_BARRIER))
00180:
00181: #define WRITE_FLUSH (WRITE_SYNC | (1 << BIO_RW_FLUSH))
00182: #define WRITE_FUA     (WRITE_SYNC | (1 << BIO_RW_FUA))
00183: #define WRITE_FLUSH_FUA     (WRITE_FLUSH | WRITE_FUA)
```

READ、WRITE：正常的读写。

READ_SYNC：同步读。设备没有plug，调用者可以立即等待此次读，而不用关心设备unplug。

READA：用于预读，优先级较低。

SWRITE：使用write_dirty_buffer（）。让ll_rw_lock（）先锁住buffer。

WRITE_SYNC_PLUG：同步写，等同于WRITE。但向下传递提示有人将等待这个IO，该设备必须一直处于unplugged状态。

WRITE_ODIRECT_PLUG：用O_DIRECT写。

WRITE_SYNC：和WRITE_SYNC_PLUG类似，但提交请求后立即unplug设备。操作等同于READ_SYNC。

SWRITE_SYNC/SWRITE_SYNC_PLUG：和WRITE_SYNC/WRITE_SYNC_PLUG类似，但首先锁住buffer。参考SWRITE的含义。

WRITE_FLUSH：类似WRITE_SYNC，但提前进行cache刷新。

WRITE_FUA：类似WRITE_SYNC，但会保证在写数据结束后，数据真正写到非易失性介质上。

WRITE_FLUSH_FUA：WRITE_FLUSH和WRITE_FUA的结合。提前进行Cache刷新，且保证数据真正写到非易失性存储介质上。

4.2.2 ll_rw_lock（）

函数源码在fs/buffer.c中，参数含义：

- rw：其值可以是READ、READ_SYNC、WRITE、READA或者SWRITE等。

• nr：请求的数据块数量。

• bhs：buffer_head指针数组，有nr个指针，指向说明块的缓冲区首部（每个缓存冲区块的大小必须相同，而且必须处于同一个块设备）。

```
03126: void ll_rw_block(int rw, int nr, struct buffer_head *bhs[])
03127: {
03128:     int i;
03129:
03130:     for (i = 0; i < nr; i++) {
03131:         struct buffer_head *bh = bhs[i];
03132:
03133:         if (rw == SWRITE || rw == SWRITE_SYNC || rw == SWRITE_SYNC_PLUG)
03134:             lock_buffer(bh);
03135:         else if (!trylock_buffer(bh))
03136:             continue;
03137:
03138:         if (rw == WRITE || rw == SWRITE || rw == SWRITE_SYNC ||
03139:           rw == SWRITE_SYNC_PLUG) {
03140:             if (test_clear_buffer_dirty(bh)) {
03141:                 bh->b_end_io = end_buffer_write_sync;
03142:                 get_bh(bh);
03143:                 if (rw == SWRITE_SYNC)
03144:                     submit_bh(WRITE_SYNC, bh);
03145:                 else
03146:                     submit_bh(WRITE,    bh);
03147:                 continue;
03148:             }
03149:         } else {
03150:             if (!buffer_uptodate(bh)) {
03151:                 bh->b_end_io = end_buffer_read_sync;
03152:                 get_bh(bh);
03153:                 submit_bh(rw, bh);
03154:                 continue;
03155:             }
03156:         }
03157:         unlock_buffer(bh);
03158:     } « end for i=0;i<nr;i++ »
03159: } « end ll_rw_block »
03160: EXPORT_SYMBOL(ll_rw_block);
```

ll_rw_block（）重复所有的缓冲区头，对每个缓冲区头，执行以下动作：

（1）若操作类型为SWRITE或SWRITE_SYNC或SWRITE_SYNC_PLUG，就要先把buffer锁住。否则测试并设置缓冲区头额度BH_Lock标志；若缓冲区已经上锁，则说明在其他内核控制路径上已经触发数据传输；若数组中有其他的缓冲区头待处理，则选择下一个缓冲区头。

（2）若操作类型为WRITE/SWRITE/SWRITE_SYNC/SWRITE_SYNC_PLUG之一，表明为写操作。首先则设置并清除缓冲区头的BH_Dirty标志。若该标志并未设置，也就没有必要把数据块写到磁盘上。接下来设置缓冲区头的b_end_io方法为end_buffer_write_sync

（），递增缓冲区头的计数b_count，调用submit_bh（）提交请求。若写类型为SWRITE_SYNC，就在submit_bh（）里传递WRITE_SYNC参数。

（3）非写数据，即为读数据。则检查缓冲区头的BH_Update标志是否设置；若该标志设置，则没有必要再从磁盘上读数据。否则，设置b_end_io的方法为end_buffer_read_sync（），递增缓冲区头的计数b_count，调用submit_bh（）提交请求。

（4）执行到3157行时，说明必须读或写数据，且已通过调用submit_bh（）提交请求。此时清除缓冲区头的BH_Lock标志，解锁；并唤醒所有等待该数据块的进程。

函数中，大家只看到缓冲区头计数增加，但没看到递减计数。递减缓冲区头的操作在end_buffer_write_sync（）或在end_buffer_read_sync（）中进行。

4.2.3 submit_bh（）

我们现在再来看一下submit_bh（）的实现，源码在fs/buffer.c中。submit_bh函数的主要任务是为buffer head分配一个bio对象，并且对其进行初始化，然后将bio提交给对应的块设备对象。提交给块设备的行为其实就是让对应的块设备驱动程序对其进行处理。

```
03047: int submit_bh(int rw, struct buffer_head * bh)
03048: {
03049:     struct bio *bio;
03050:     int ret = 0;
03051:
03052:     BUG_ON(! buffer_locked(bh));
03053:     BUG_ON(! buffer_mapped(bh));
03054:     BUG_ON(! bh->b_end_io);
03055:     BUG_ON(buffer_delay(bh));
03056:     BUG_ON(buffer_unwritten(bh));
03057:
03058:     /*
03059:      * Mask in barrier bit for a write (could be either a WRITE or a
03060:      * WRITE_SYNC
03061:      */
03062:     if (buffer_ordered(bh) && (rw & WRITE))
03063:         rw |= WRITE_BARRIER;
03064:
03065:     /*
03066:      * Only clear out a write error when rewriting
03067:      */
03068:     if (test_set_buffer_req(bh)   && (rw & WRITE))
03069:         clear_buffer_write_io_error(bh);
03070:
03071:     /*
03072:      * from here on down, it's all bio -- do the initial mapping,
03073:      * submit_bio -> generic_make_request may further map this bio around
03074:      */
03075:     bio = bio_alloc(GFP_NOIO,   1);
03076:
```

```
03077:      bio->bi_sector = bh->b_blocknr * (bh->b_size >> 9);
03078:      bio->bi_bdev = bh->b_bdev;
03079:      bio->bi_io_vec[0].bv_page = bh->b_page;
03080:      bio->bi_io_vec[0].bv_len    = bh->b_size;
03081:      bio->bi_io_vec[0].bv_offset    = bh_offset(bh);
03082:
03083:      bio->bi_vcnt = 1;
03084:      bio->bi_idx = 0;
03085:      bio->bi_size = bh->b_size;
03086:
03087:      bio->bi_end_io = end_bio_bh_io_sync;
03088:      bio->bi_private = bh;
03089:
03090:      bio_get(bio);
03091:      submit_bio(rw, bio);
03092:
03093:      if (bio_flagged(bio,   BIO_EOPNOTSUPP))
03094:          ret = -EOPNOTSUPP;
03095:
03096:      bio_put(bio);
03097:      return ret;
03098: } « end submit_bh »
03099: EXPORT_SYMBOL(submit_bh);
```

submit_bh（）函数假定缓冲区头已完全初始化，即b_bdev，b_blocknr和b_size成员变量已正确设置，能够确定请求数据在磁盘上的具体块位置。它从缓冲区头中的内容来创建一个bio请求，然后调用submit_bio（），即调用generic_make_request（）。

在Linux中，每个块设备在内核中都会采用bdev（block_device）对象进行描述。通过bdev对象可以获取块设备的所有所需资源，包括如何处理发送到该设备的IO方法。因此，在初始化bio的时候，需要设备目标block device，在Linux的请求转发层需要用到block device对象对bio进行转发处理。

submit_bh（）函数主要步骤如下：

（1）test_set_buffer_req（）将缓冲区头的标志设置为BH_Req，这样表明该块至少提交了一次。除此之外，若数据传输的方向是WRITE，则清除BH_Write_EIO标志。

（2）调用bio_alloc（）分配一个bio描述符。

（3）根据缓冲区头的内容来初始化bio描述符。

（4）增加bio的计数。

（5）调用submit_bio（）。

（6）减少bio的计数。

（7）成功返回0。

submit_bio（）函数在文件driver/block/ll_rw_blk.c中。

```
01808: void submit_bio(int rw, struct bio *bio)
01809: {
01810:     int count = bio_sectors(bio);
01811:
01812:     bio->bi_rw |= rw;
01813:
01814:     /*
01815:      * If it's a regular read/write or a barrier with data attached,
01816:      * go through the normal accounting stuff before submission.
01817:      */
01818:     if (bio_has_data(bio) && !(rw & (1 << BIO_RW_DISCARD))) {
01819:         if (rw & WRITE) {
01820:             count_vm_events(PGPGOUT, count);
01821:         } else {
01822:             task_io_account_read(bio->bi_size);
01823:             count_vm_events(PGPGIN, count);
01824:         }
01825:
01826:         if (unlikely(block_dump)) {
01827:             char b[BDEVNAME_SIZE];
01828:             printk(KERN_DEBUG "%s(%d): %s block %Lu on %s\n",
01829:             current->comm, task_pid_nr(current),
01830:                 (rw & WRITE) ? "WRITE" : "READ",
01831:                 (unsigned long long)bio->bi_sector,
01832:                 bdevname(bio->bi_bdev, b));
01833:         }
01834:     }
01835:
01836:     generic_make_request(bio);
01837: } « end submit_bio »
01838: EXPORT_SYMBOL(submit_bio);
01839:
```

submit_bio（）的功能比较简单，主要就是设置bio->bi_rw的值，即确定数据传输方向。若属于正常的读写数据，就在提交任务之前，更新I/O统计信息。然后调用generic_make_request（）。

这里可以关注一下较新的Linux内核可以统计每个进程的I/O信息，对故障或性能定位非常有帮助。

4.2.4 generic_make_request（）

通用块设备层的主要入口是generic_make_request（），源码也在在文件driver/block/ll_rw_blk.c中。

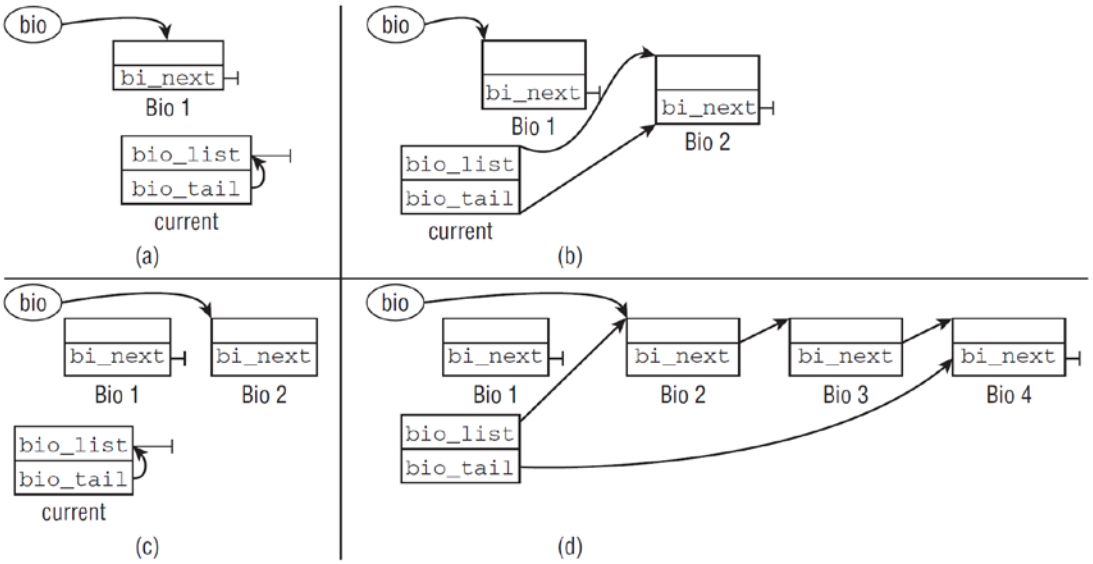
函数主要功能为维护进程bio链表和对__generic_make_request（）的直接封装调用。

```
01757: void generic_make_request(struct bio *bio)
01758: {
01759:     if (current->bio_tail) {
01760:         /* make_request is active */
01761:         *(current->bio_tail) = bio;
01762:         bio->bi_next = NULL;
01763:         current->bio_tail = &bio->bi_next;
01764:         return;
01765:     }
01766:     /* following loop may be a bit non-obvious, and so deserves some
01767:      * explanation.
01768:      * Before entering the loop, bio->bi_next is NULL (as all callers
01769:      * ensure that) so we have a list with a single bio.
01770:      * We pretend that we have just taken it off a longer list, so
01771:      * we assign bio_list to the next (which is NULL) and bio_tail
01772:      * to &bio_list, thus initialising the bio_list of new bios to be
01773:      * added.  ___generic_make_request may indeed add some more bios
01774:      * through a recursive call to generic_make_request.  If it
01775:      * did, we find a non-NULL value in bio_list and re-enter the loop
01776:      * from the top.  In this case we really did just take the bio
01777:      * of the top of the list (no pretending) and so fixup bio_list and
01778:      * bio_tail or bi_next, and call into ___generic_make_request again.
01779:      *
01780:      * The loop was structured like this to make only one call to
01781:      * ___generic_make_request (which is important as it is large and
01782:      * inlined) and to keep the structure simple.
01783:      */
01784:     BUG_ON(bio->bi_next);
01785:     do {
01786:         current->bio_list = bio->bi_next;
01787:         if (bio->bi_next == NULL)
01788:             current->bio_tail = &current->bio_list;
01789:         else
01790:             bio->bi_next = NULL;
01791:         __generic_make_request(bio);
01792:         bio = current->bio_list;
01793:     } while (bio);
01794:     current->bio_tail = NULL; /* deactivate */
01795: } « end generic_make_request »
01796: EXPORT_SYMBOL(generic_make_request);
```

在每个进程的task_struct中，都包含有两个变量struct bio *bio_list, **bio_tail，generic_make_request（）的主要工作就是用这两个变量，来维护当前待添加的bio链表。

bio_tail是一个二级指针，这个值最初是NULL，当有bio添加进来，bio_tail将会指向bio->bi_next(如果bio全都递交上去了，则bio_tail将会指向bio_list)，也就是说除了第一次调用外，其他每次递归调用generic_make_request（）函数都会出现bio_tail不为NULL的情形，因此当bio_tail不为NULL时，则只将bio添加到由bio_list和bio_tail维护的链表中，然后直接返回，加入到链表的结点generic_make_request会在后面处理,而不再次调用__generic_make_request()，这样便防止了多重递归的产生。

__generic_make_request（）会执行__make_request（），直到最后__make_request函数返回0，才算是告诉generic_make_request无需再转发bio了，bio转发结束。



__generic_make_request（）函数会调用块设备的make_request_fn方法。普通设备make_request_fn实现为内核__make_request函数，一些特殊设备会使用自己的make_request_fn，例如md和dm设备，可以统称此类设备为 virtual disk。每个进程的内核堆都很小，一般是4K或者8K，因此generic_make_request的实现中，经过处理使得递归的深度不会超过一层。

4.2.5 __generic_make_request（）

__generic_make_request（）功能：调用blk_partition_remap（）函数检查该块设备是否是一个磁盘分区，如果是个分区，则需要做一些转化。bio->bio_sector的值是相对于分区的起始扇区号，这里需要转换成相对于整个磁盘的扇区号。然后再调用q->make_request_fn（）把bio请求插入到请求队列中。

q->make_request_fn（）把通用块层和I/O调度层连接起来，make_request_fn的值是在块设备驱动在加载时里调用blk_init_queue（）设置。

```
01640: static inline void __generic_make_request(struct bio *bio)
01641: {
01642:     struct request_queue *q;
01643:     sector_t old_sector;
01644:     int ret, nr_sectors = bio_sectors(bio);
01645:     dev_t old_dev;
01646:     int err = -EIO;
01647:
01648:     might_sleep();
01649:
01650:     if (bio_check_eod(bio, nr_sectors))
01651:         goto ↓end_io;
```

```
01652:
01653:      /*
01654:       * Resolve the mapping until finished. (drivers are
01655:       * still free to implement/resolve their own stacking
01656:       * by explicitly returning 0)
01657:       *
01658:       * NOTE: we don't repeat the blk_size check for each new device.
01659:       * Stacking drivers are expected to know what they are doing.
01660:       */
01661:      old_sector = -1;
01662:      old_dev = 0;
01663:      do {
01664:          char b[BDEVNAME_SIZE];
01665:
01666:          q = bdev_get_queue(bio->bi_bdev);
01667:          if (unlikely(! q)) {
01668:              printk(KERN_ERR
01669:                  "generic_make_request: Trying to access "
01670:                  "nonexistent  block-device %s (%Lu)\n",
01671:                  bdevname(bio->bi_bdev, b),
01672:                  (long long) bio->bi_sector);
01673:              goto ↓end_io;
01674:              }
01675:
01676:            if (unlikely(! bio_rw_flagged(bio, BIO_RW_DISCARD)    &&
01677:                nr_sectors > queue_max_hw_sectors(q))) {
01678:              printk(KERN_ERR   "bio too big device %s (%u > %u)\n",
01679:                  bdevname(bio->bi_bdev, b),
01680:                  bio_sectors(bio),
01681:                  queue_max_hw_sectors(q));
01682:              goto ↓end_io;
01683:          }
01684:
01685:          if (should_fail_request(bio))
01686:              goto ↓end_io;
01687:
01688:          /*
01689:           * If this device has partitions, remap block n
01690:           * of partition p to block n+start(p) of the disk.
01691:           */
01692:          blk_partition_remap(bio);
01693:
01694:          if (bio_integrity_enabled(bio) && bio_integrity_prep(bio))
01695:              goto ↓end_io;
01696:
01697:          if (old_sector != -1)
01698:              trace_block_remap(q,   bio, old_dev, old_sector);
01699:
01700:          old_sector = bio->bi_sector;
01701:          old_dev = bio->bi_bdev->bd_dev;
01702:
01703:          if (bio_check_eod(bio, nr_sectors))
01704:              goto ↓end_io;
01705:
01706:          /*
01707:           * Filter flush bio's early so that make_request based
01708:           * drivers without flush support don't have to worry
01709:           * about them.
01710:           */
01711:          if ((bio->bi_rw & (BIO_FLUSH | BIO_FUA)) && !q->flush_flags) {
01712:              bio->bi_rw &= ~(BIO_FLUSH | BIO_FUA);
01713:              if (!nr_sectors) {
01714:                  err = 0;
01715:                  goto ↓end_io;
01716:              }
```

```
01717:          }
01718:
01719:          if (bio_rw_flagged(bio, BIO_RW_DISCARD)   &&
01720:            !blk_queue_discard(q)) {
01721:              err = -EOPNOTSUPP;
01722:              goto ↓end_io;
01723:          }
01724:
01725:          if (blk_throtl_bio(q,   bio))
01726:              break;    /* throttled, will be resubmitted later */
01727:
01728:          /*
01729:           * If bio = NULL, bio has been throttled and will be submitted
01730:           * later.
01731:           */
01732:          if (!bio)
01733:              break;
01734:
01735:          trace_block_bio_queue(q,   bio);
01736:
01737:          ret = q->make_request_fn(q,   bio);
01738:      } « end do »  while (ret);
01739:
01740:      return;
01741:
01742: end_io:
01743:    bio_endio(bio, err);
01744: } « end ___generic_make_request »
01745:
```

该函数主要完成以下工作：

（1）bio_check_eod（）检查bio->bio_sector没有超过设备的最大扇区数。若bio->bio_sector超过设备的最大扇区数，则设置bio->bi_flags为BIO_EOF，打印一条内核错误信息，然后调用bio_endio（）（fs/bio.c）结束（1650行）。

（2）获取块设备的请求队列q（1666行），获取方法很简单，返回bdev->bd_disk->queue即可。

（3）调用blk_partion_remap（）（/drivers/block/ll_rw_blk.c），检查该块设备是否是一个分区（即bio->bi_dev与bio->bi_dev->bd_contains不相等）。若是一个分区，则获取从bio->bi_dev中获取hd_struct描述符（）（1692行）。

（4）调用q->make_request_fn方法，将bio请求插入请求队列q中。

对于所有的块设备来说，q->make_request_fn方法就是__make_request（），代码也在文件driver/block/ll_rw_blk.c中。我们会在Linux内核I/O调度层内容中详细介绍该函数。

下面是bio_endio（）和blk_partition_remap（）两个函数源码及解释。

```
01411: /**
01412:  * bio_endio - end I/O on a bio
01413:  * @bio: bio
01414:  * @error:    error, if any
01415:  *
01416:  * Description:
01417:  *     bio_endio() will end I/O on the whole bio. bio_endio() is the
01418:  *     preferred way to end I/O on a bio, it takes care of clearing
01419:  *     BIO_UPTODATE on error. @error is 0 on success, and and one of the
01420:  *     established -Exxxx (-EIO, for instance) error values in case
01421:  *     something went wrong. Noone should call bi_end_io() directly on a
01422:  *     bio unless they own it and thus know that it has an end_io
01423:  *     function.
01424:  **/
01425: void bio_endio(struct bio *bio, int error)
01426: {
01427:     if (error)
01428:         clear_bit(BIO_UPTODATE, &bio->bi_flags);
01429:     else if (!test_bit(BIO_UPTODATE, &bio->bi_flags))
01430:         error = -EIO;
01431:
01432:     if (bio->bi_end_io)
01433:         bio->bi_end_io(bio,   error);
01434: }
01435: EXPORT_SYMBOL(bio_endio);
01436:
```

bio_endio（）更新bi_size和bi_sector的值，然后调用bio的bi_end_io方法。

```
01517: /*
01518:  * If bio->bi_dev is a partition, remap the location
01519:  */
01520: static inline void blk_partition_remap(struct bio *bio)
01521: {
01522:     struct block_device *bdev = bio->bi_bdev;
01523:
01524:     if (bio_sectors(bio)   && bdev != bdev->bd_contains) {
01525:         struct hd_struct *p = bdev->bd_part;
01526:
01527:         bio->bi_sector += p->start_sect;
01528:         bio->bi_bdev = bdev->bd_contains;
01529:
01530:         trace_block_remap(bdev_get_queue(bio->bi_bdev), bio,
01531:                 bdev->bd_dev,
01532:                 bio->bi_sector - p->start_sect);
01533:     }
01534: }
01535:
```

更新读写统计信息，然后调整bio->bi_setor的值，即将分区内的扇区数转换为整个磁盘设备的扇区数；最后将由原来的磁盘分区bio->bi_bdev设置为整个磁盘。