第十六章 块设备驱动

到目前为止，我们的讨论仅限于字符驱动程序。然而，在Linux系统中还有其他类型的驱动程序，现在是时候我们扩大一些焦点了。因此，这一章将讨论块驱动程序。

块驱动程序提供对设备的访问，这些设备以固定大小的块传输随机可访问的数据——主要是磁盘驱动程序。Linux内核将块设备视为与字符设备根本不同；因此，块驱动程序有一个独特的接口和它们自己特定的挑战。

高效的块驱动程序对于性能至关重要——并不仅仅是对用户应用程序中的显式读取和写入。现代的具有虚拟内存的系统通过将（希望）不需要的数据转移到次级存储（通常是磁盘驱动程序）来工作。块驱动程序是核心内存和次级存储之间的通道；因此，它们可以被视为构成虚拟内存子系统的一部分。虽然可以在不了解`struct page`和其他重要内存概念的情况下编写块驱动程序，但任何需要编写高性能驱动程序的人都必须借鉴第15章中涵盖的材料。

块层的设计大部分都集中在性能上。许多字符设备可以在其最大速度以下运行，而整个系统的性能不受影响。然而，如果其块I/O子系统没有调整好，系统就无法运行良好。Linux块驱动程序接口允许你从块设备中获得最大的性能，但必然会带来一定的复杂性，你必须处理。幸运的是，2.6块接口比旧内核中的接口有很大的改进。

这一章的讨论，正如人们所期望的，集中在一个示例驱动程序上，该驱动程序实现了一个块定向的、基于内存的设备。它本质上是一个ramdisk。内核已经包含了一个远超过我们的驱动程序（称为sbull）的ramdisk实现，但我们的驱动程序让我们展示了创建块驱动程序的过程，同时最大程度地减少了无关的复杂性。

在深入细节之前，让我们准确地定义一下几个术语。块是一个固定大小的数据块，大小由内核确定。块通常是4096字节，但这个值可以根据架构和正在使用的确切文件系统而变化。相比之下，扇区是一个小块，其大小通常由底层硬件确定。内核期望处理实现512字节扇区的设备。如果你的设备使用不同的大小，内核会适应并避免生成硬件无法处理的I/O请求。然而，值得记住的是，任何时候内核向你提供一个扇区号，它都是在512字节扇区的世界中工作。如果你使用的是不同的硬件扇区大小，你必须相应地调整内核的扇区号。我们在sbull驱动程序中看到了这是如何做到的。

16.1. 注册

块驱动程序，像字符驱动程序一样，必须使用一组注册接口来使它们的设备对内核可用。概念是相似的，但块设备注册的细节都是不同的。你有一整套新的数据结构和设备

操作需要学习。

块设备驱动程序的注册通常涉及以下步骤：

1. 定义一个块设备操作结构（`struct block_device_operations`），这个结构包含了驱动程序需要实现的一组回调函数。
2. 创建一个或多个`gendisk`结构，每个结构代表一个物理或逻辑磁盘设备。
3. 使用`register_blkdev`函数注册你的块设备驱动程序。
4. 使用`add_disk`函数将你的`gendisk`结构添加到系统中。

这些步骤的具体实现可能会因驱动程序的具体需求而有所不同。例如，你可能需要实现不同的块设备操作，或者你可能需要为你的设备创建多个`gendisk`结构。

16.1.1. 块驱动注册

大多数块驱动程序的第一步是向内核注册自己。执行此任务的函数是`register_blkdev`（在`<linux/fs.h>`中声明）：

```c
int register_blkdev(unsigned int major, const char *name);
```

参数是你的设备将使用的主要号码和关联的名称（内核将在`/proc/devices`中显示）。如果`major`作为0传递，内核分配一个新的主要号码并将其返回给调用者。像往常一样，`register_blkdev`的负返回值表示发生了错误。

取消块驱动程序注册的对应函数是：

```c
int unregister_blkdev(unsigned int major, const char *name);
```

这里，参数必须与传递给`register_blkdev`的参数匹配，否则函数返回`-EINVAL`并且不注销任何东西。

在2.6内核中，调用`register_blkdev`完全是可选的。`register_blkdev`执行的函数随着时间的推移而减少；此时此调用执行的唯一任务是（1）如果请求，分配一个动

态主要号码，和（2）在/proc/devices中创建一个条目。在未来的内核中，register_blkdev可能会被完全移除。然而，与此同时，大多数驱动程序仍然调用它；这是传统。

16.1.2. 磁盘注册

虽然register_blkdev可以用来获取主要号码，但它并不能使任何磁盘驱动程序对系统可用。你必须使用一个单独的注册接口来管理单个驱动程序。使用这个接口需要熟悉一对新的结构，所以我们从这里开始。

16.1.2.1. 块设备操作

字符设备通过file_operations结构将其操作提供给系统。一个类似的结构用于块设备；它是struct block_device_operations，在<linux/fs.h>中声明。以下是在此结构中找到的字段的简要概述；当我们深入了解sbull驱动程序的细节时，我们将更详细地回顾它们：

```c
int (*open)(struct inode *inode, struct file *filp);

int (*release)(struct inode *inode, struct file *filp);
```

这些函数的工作方式与它们的字符驱动程序等效相同；每当设备被打开和关闭时，它们都会被调用。块驱动程序可能会通过启动设备、锁定门（对于可移动媒体等）来响应一个打开调用。如果你将媒体锁定到设备中，你应该在release方法中解锁它。

```c
int (*ioctl)(struct inode *inode, struct file *filp,
unsigned int cmd, unsigned long arg);
```

实现ioctl系统调用的方法。然而，块层首先拦截大量的标准请求；因此，大多数块驱动程序的ioctl方法都相当短。

```c
int (*media_changed) (struct gendisk *gd);
```

内核调用此方法以检查用户是否已更改驱动程序中的媒体，如果是，则返回非零值。显然，此方法仅适用于支持可移动媒体的驱动程序（并且足够智能以将“媒体已更改”标志提供给驱动程序）；在其他情况下，可以省略它。

`struct gendisk` 参数是内核表示单个磁盘的方式；我们将在下一节中查看该结构。

```c
int (*revalidate_disk) (struct gendisk *gd);
```

`revalidate_disk` 方法是对媒体更改的响应；它给驱动程序一个机会来执行使新媒体准备好使用所需的任何工作。函数返回一个 `int` 值，但内核忽略该值。

```c
struct module *owner;
```

指向拥有此结构的模块的指针；它通常应初始化为 `THIS_MODULE`。

细心的读者可能已经注意到这个列表中有一个有趣的遗漏：没有实际读取或写入数据的函数。在块I/O子系统中，这些操作由 `request` 函数处理，这个函数值得拥有自己的大部分内容，并在本章后面讨论。在我们讨论服务请求之前，我们必须完成对磁盘注册的讨论。

16.1.2.2. gendisk 结构体

`struct gendisk`（在 `<linux/genhd.h>` 中声明）是内核对单个磁盘设备的表示。实际上，内核也使用 `gendisk` 结构来表示分区，但驱动程序作者不需要知道这一点。`struct gendisk` 中有几个字段必须由块驱动程序初始化：

```c
int major;

int first_minor;

int minors;
```

这些字段描述了磁盘使用的设备号。至少，驱动程序必须使用至少一个次要号码。然而，如果你的驱动程序可以分区（大多数都应该可以），你也希望为每个可能的分区分配一个次要号码。`minors`的常见值是16，这允许“全磁盘”设备和15个分区。一些磁盘驱动程序为每个设备使用64个次要号码。

```c
char disk_name[32];
```

应设置为磁盘设备的名称的字段。它出现在`/proc/partitions`和`sysfs`中。

```c
struct block_device_operations *fops;
```

上一节中的设备操作集。

```c
struct request_queue *queue;
```

内核用于管理此设备的I/O请求的结构；我们在“请求处理”部分检查它。

```c
int flags;
```

描述驱动状态的（很少使用的）一组标志。如果你的设备有可移动媒体，你应该设置`GENHD_FL_REMOVABLE`。CD-ROM驱动程序可以设置`GENHD_FL_CD`。如果，由于某种原因，你不希望分区信息出现在`/proc/partitions`中，设置`GENHD_FL_SUPPRESS_PARTITION_INFO`。

```c
sector_t capacity;
```

此驱动的容量，以512字节的扇区为单位。 `sector_t` 类型可以是64位宽。驱动程序不应直接设置此字段；相反，将扇区数传递给 `set_capacity` 。

```c
void *private_data;
```

块驱动程序可以使用此字段作为指向他们自己内部数据的指针。

内核提供了一小组用于处理 `gendisk` 结构的函数。我们在这里介绍它们，然后看看 `sbull` 如何使用它们使其磁盘设备对系统可用。

`struct gendisk` 是一个动态分配的结构，需要特殊的内核操作才能初始化；驱动程序不能自己分配结构。相反，你必须调用：

```c
struct gendisk *alloc_disk(int minors);
```

`minors` 参数应该是此磁盘使用的次要号码的数量；注意，你不能稍后更改 `minors` 字段并期望事情能正常工作。当不再需要磁盘时，应使用以下方法释放它：

```c
void del_gendisk(struct gendisk *gd);
```

`gendisk` 是一个引用计数结构（它包含一个 `kobject` ）。有 `get_disk` 和 `put_disk` 函数可用于操作引用计数，但驱动程序永远不需要这样做。通常，对 `del_gendisk` 的调用会删除对 `gendisk` 的最后一个引用，但不能保证这一点。因此，即使在调用 `del_gendisk` 后，结构可能继续存在（并且可能会调用你的方法）。然而，如果你在没有用户的情况下删除结构（即，在最后的 `release` 或在你的模块清理函数中），你可以确保你不会再次听到它。

分配一个 `gendisk` 结构并不能使磁盘对系统可用。为此，你必须初始化结构并调用 `add_disk` ：

```c
void add_disk(struct gendisk *gd);
```

在这里要记住一件重要的事情：一旦你调用`add_disk`，磁盘就会“活跃”，并且可以随时调用它的方法。实际上，第一次这样的调用可能会在`add_disk`返回之前就发生；内核将读取前几个块，试图找到一个分区表。因此，你不应该在驱动程序完全初始化并准备好响应该磁盘上的请求之前调用`add_disk`。

16.1.3. 在 sbull 中的初始化

现在是时候看一些例子了。`sbull`驱动程序（可以从O'Reilly的FTP站点与其他示例源代码一起获取）实现了一组内存中的虚拟磁盘驱动程序。对于每个驱动程序，`sbull`分配（使用`vmalloc`，为了简单）一个内存数组；然后通过块操作使该数组可用。可以通过对虚拟设备进行分区，构建文件系统，并将其挂载到系统层次结构中来测试`sbull`驱动程序。

像我们的其他示例驱动程序一样，`sbull`允许在编译或模块加载时指定主要号码。如果没有指定号码，将动态分配一个。由于需要调用`register_blkdev`进行动态分配，所以`sbull`这样做：

```c
sbull_major = register_blkdev(sbull_major, "sbull");

if (sbull_major <= 0) {

    printk(KERN_WARNING "sbull: unable to get major
number\n");

    return -EBUSY;

}
```

此外，像我们在这本书中介绍的其他虚拟设备一样，`sbull`设备由内部结构描述：

```c
struct sbull_dev {

    int size;                       /* Device size in
sectors */

    u8 *data;                       /* The data array */

    short users;                    /* How many users */

    short media_change;             /* Flag a media
change? */

    spinlock_t lock;                /* For mutual
exclusion */

    struct request_queue *queue;    /* The device request
queue */

    struct gendisk *gd;             /* The gendisk
structure */

    struct timer_list timer;        /* For simulated
media changes */

};
```

需要几个步骤来初始化此结构并使关联的设备对系统可用。我们从基本初始化和底层内存的分配开始：

```c
memset (dev, 0, sizeof (struct sbull_dev));
dev->size = nsectors*hardsect_size;
dev->data = vmalloc(dev->size);
if (dev->data == NULL) {
    printk (KERN_NOTICE "vmalloc failure.\n");
    return;
}
spin_lock_init(&dev->lock);
```

在下一步之前分配和初始化一个自旋锁是很重要的，这一步是请求队列的分配。当我们处理请求时，我们会更详细地看这个过程；现在，只需说必要的调用是：

```c
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

这里，`sbull_request`是我们的请求函数——实际执行块读写请求的函数。当我们分配一个请求队列时，我们必须提供一个控制对该队列的访问的自旋锁。这个锁是由驱动程序提供的，而不是内核的一般部分，因为通常，请求队列和其他驱动程序数据结构在同一个关键部分内；它们倾向于一起被访问。与任何分配内存的函数一样，`blk_init_queue`可能会失败，所以你必须在继续之前检查返回值。

一旦我们的设备内存和请求队列就绪，我们就可以分配、初始化并安装相应的`gendisk`结构。执行这项工作的代码是：

```c
dev->gd = alloc_disk(SBULL_MINORS);
if (! dev->gd) {
    printk (KERN_NOTICE "alloc_disk failure\n");
    goto out_vfree;
}
dev->gd->major = sbull_major;
dev->gd->first_minor = which*SBULL_MINORS;
dev->gd->fops = &sbull_ops;
dev->gd->queue = dev->queue;
dev->gd->private_data = dev;
snprintf (dev->gd->disk_name, 32, "sbull%c", which +
'a');
set_capacity(dev->gd, nsectors*
(hardsect_size/KERNEL_SECTOR_SIZE));
add_disk(dev->gd);
```

这里，`SBULL_MINORS`是每个`sbull`设备支持的次要号码的数量。当我们为每个设备设置第一个次要号码时，我们必须考虑到之前设备占用的所有号码。磁盘的名称设置为第一个是`sbulla`，第二个是`sbullb`，依此类推。用户空间可以添加分区号，以便第二个设备上的第三个分区可能是`/dev/sbullb3`。

一旦所有事情都设置好，我们就用调用`add_disk`来结束。很可能在`add_disk`返回时，我们的几个方法已经被那个磁盘调用过了，所以我们要小心，使这个调用成为我们设备初始化的最后一步。

16.1.4. 注意扇区大小

正如我们之前提到的，内核将每个磁盘视为一个线性数组，每个数组由512字节的扇区组成。然而，并非所有的硬件都使用这个扇区大小。让一个具有不同扇区大小的设备工作并不特别困难；这只是处理一些细节的问题。`sbull`设备导出了一个`hardsect_size`参数，可以用来改变设备的"硬件"扇区大小；通过查看其实现，你可以看到如何向你自己的驱动程序添加这种支持。

这些细节中的第一个是通知内核你的设备支持的扇区大小。硬件扇区大小是请求队列中的一个参数，而不是`gendisk`结构中的参数。这个大小是在队列分配后立即通过调用`blk_queue_hardsect_size`设置的：

```c
blk_queue_hardsect_size(dev->queue, hardsect_size);
```

一旦完成，内核将遵循你的设备的硬件扇区大小。所有的I/O请求都在硬件扇区的开始处正确对齐，每个请求的长度是扇区的整数倍。然而，你必须记住，内核总是以512字节的扇区来表达自己；因此，有必要相应地转换所有的扇区号。所以，例如，当`sbull`在其`gendisk`结构中设置设备的容量时，调用看起来像这样：

```c
set_capacity(dev->gd, nsectors*
(hardsect_size/KERNEL_SECTOR_SIZE));
```

`KERNEL_SECTOR_SIZE`是我们定义的一个本地常量，我们用它来在内核的512字节扇区和我们被告知使用的任何大小之间进行缩放。这种类型的计算在我们查看`sbull`请求处理逻辑时经常出现。

16.2. 块设备操作

在上一节中，我们对`block_device_operations`结构进行了简单的介绍。现在，在进入请求处理之前，我们花些时间更详细地看看这些操作。为此，现在是时候提及`sbull`驱动程序的另一个特性：它假装是一个可移动设备。每当最后一个用户关闭设备时，就会设置一个30秒的定时器；如果在那段时间内没有打开设备，设备的内容将被清除，内核将被告知媒体已经改变。30秒的延迟给了用户时间，例如，在一个`sbull`设备上创建文件系统后挂载它。

16.2.1. open 和 release 方法

为了实现模拟媒体移除，`sbull`必须知道最后一个用户何时关闭了设备。驱动程序维护了一个用户计数。`open`和`close`方法的任务是保持该计数的当前状态。

`open`方法看起来非常类似于其字符驱动程序的等价物；它将相关的`inode`和`file`结构指针作为参数。当一个`inode`引用一个块设备时，字段`i_bdev->bd_disk`包含一个指向关联的`gendisk`结构的指针；这个指针可以用来获取驱动程序的设备内部数据结构。实际上，这是`sbull open`方法做的第一件事：

```c
static int sbull_open(struct inode *inode, struct file
*filp)

{

    struct sbull_dev *dev = inode->i_bdev->bd_disk-
>private_data;

    del_timer_sync(&dev->timer);

    filp->private_data = dev;

    spin_lock(&dev->lock);

    if (! dev->users)

        check_disk_change(inode->i_bdev);

    dev->users++;

    spin_unlock(&dev->lock);

    return 0;

}
```

一旦 `sbull_open` 有了它的设备结构指针，它就调用 `del_timer_sync` 来移除“媒体移除”定时器，如果有任何活动的定时器的话。注意，我们在删除定时器之后才锁定设备自旋锁；否则，如果定时器函数在我们能删除它之前运行，就会邀请死锁。在设备锁定后，我们调用一个名为 `check_disk_change` 的内核函数来检查是否发生了媒体更改。有人可能会争辩说内核应该做那个调用，但是标准的模式是让驱动程序在打开时处理它。

最后一步是增加用户计数并返回。

相比之下， `release` 方法的任务是减少用户计数，如果需要，启动媒体移除定时器：

```c
static int sbull_release(struct inode *inode, struct file
*filp)

{

    struct sbull_dev *dev = inode->i_bdev->bd_disk-
>private_data;

    spin_lock(&dev->lock);

    dev->users--;

    if (!dev->users) {

        dev->timer.expires = jiffies + INVALIDATE_DELAY;

        add_timer(&dev->timer);

    }

    spin_unlock(&dev->lock);

    return 0;

}
```

在处理真实硬件设备的驱动程序中，`open`和`release`方法会相应地设置驱动程序和硬件的状态。这项工作可能涉及到启动或关闭磁盘，锁定可移动设备的门，分配DMA缓冲区等。

你可能会想知道谁实际上打开了一个块设备。有一些操作会导致用户空间直接打开一个块设备；这些操作包括对磁盘进行分区，在分区上建立文件系统，或运行文件系统检查器。当一个分区被挂载时，块驱动程序也会看到一个`open`调用。在这种情况下，没有用户空间进程持有设备的打开文件描述符；相反，打开的文件由内核本身持有。块驱动程序无法区分挂载操作（从内核空间打开设备）和调用如`mkfs`这样的实用程序（从用户空间打开它）。

16.2.2. 支持可移出的介质

`block_device_operations`结构包括两种方法来支持可移动媒体。如果你正在为一个不可移动的设备编写驱动程序，你可以安全地省略这些方法。它们的实现相对直接。

`media_changed`方法被调用（从`check_disk_change`）来查看媒体是否已经改变；如果发生了这种情况，它应该返回一个非零值。`sbull`的实现很简单；它查询一个标志，如果媒体移除定时器已经过期，这个标志就会被设置：

```c
int sbull_media_changed(struct gendisk *gd)
{
    struct sbull_dev *dev = gd->private_data;
    return dev->media_change;
}
```

在媒体更改后，会调用`revalidate`方法；它的任务是做任何必要的准备工作，以便驱动程序对新媒体（如果有的话）进行操作。在调用`revalidate`之后，内核试图重新读取分区表，并重新开始使用设备。`sbull`的实现只是重置`media_change`标志，并将设备内存清零，以模拟插入一个空白磁盘。

```c
int sbull_revalidate(struct gendisk *gd)
{
    struct sbull_dev *dev = gd->private_data;
    if (dev->media_change) {
        dev->media_change = 0;
        memset (dev->data, 0, dev->size);
    }
    return 0;
}
```

16.2.3. ioctl 方法

块设备可以提供一个 `ioctl` 方法来执行设备控制函数。然而，更高级别的块子系统代码在你的驱动程序有机会看到它们之前，就会拦截一些 `ioctl` 命令（在内核源码的 `drivers/block/ioctl.c` 中可以看到完整的集合）。实际上，现代的块驱动程序可能不需要实现很多 `ioctl` 命令。

`sbull` 的 `ioctl` 方法只处理一个命令——请求设备的几何信息：

```c
int sbull_ioctl (struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg)
{
    long size;
    struct hd_geometry geo;
    struct sbull_dev *dev = filp->private_data;
    switch(cmd) {
        case HDIO_GETGEO:
        /*
         * Get geometry: since we are a virtual device, we have to make
         * up something plausible.  So we claim 16 sectors, four heads,
         * and calculate the corresponding number of cylinders.  We set the
         * start of data at sector four.
         */
        size = dev->size*(hardsect_size/KERNEL_SECTOR_SIZE);
        geo.cylinders = (size & ~0x3f) >> 6;
        geo.heads = 4;
```

```
            geo.sectors = 16;

            geo.start = 4;

            if (copy_to_user((void __user *) arg, &geo,
    sizeof(geo)))

                return -EFAULT;

                return 0;

        }

        return -ENOTTY; /* unknown command */

    }
```

提供几何信息可能看起来像一个奇怪的任务，因为我们的设备纯粹是虚拟的，与磁道和柱面无关。即使是大多数真实的块硬件，多年来也已经配备了更复杂的结构。内核并不关心块设备的几何形状；它只把它看作是一个线性的扇区数组。然而，有一些用户空间的实用程序仍然期望能够查询磁盘的几何形状。特别是，`fdisk`工具，它编辑分区表，依赖于柱面信息，如果没有这些信息，它就无法正常工作。

我们希望`sbull`设备能够被分区，即使是使用较旧的，简单的工具。所以，我们提供了一个`ioctl`方法，它提供了一个可信的虚构的几何形状，这个形状可以匹配我们设备的容量。大多数磁盘驱动程序都做了类似的事情。注意，如同往常一样，如果需要，扇区计数会被转换，以匹配内核使用的512字节的约定。

16.3. 请求处理

每个块驱动程序的核心都是它的请求函数。这个函数是真正的工作开始的地方——或者至少是开始；所有其他的都是开销。因此，我们花了相当多的时间来研究块驱动程序中的请求处理。

磁盘驱动程序的性能可能是整个系统性能的关键部分。因此，内核的块子系统已经考虑到性能；它尽可能地使你的驱动程序能够最大限度地利用它控制的设备。这是一件好事，因为它使得I/O速度极快。另一方面，块子系统在驱动程序API中不必要地暴露了大

量的复杂性。写一个非常简单的请求函数是可能的（我们将很快看到一个），但是如果你的驱动程序必须在复杂的硬件上高效运行，那么它将会非常复杂。

16.3.1. 对请求方法的介绍

块驱动程序的请求方法有以下原型：

```c
void request(request_queue_t *queue);
```

每当内核认为你的驱动程序应该处理一些读取、写入或其他设备操作时，就会调用这个函数。请求函数不需要在返回之前实际完成队列上的所有请求；实际上，对于大多数真实的设备，它可能不会完成任何请求。然而，它必须开始处理这些请求，并确保它们最终都被驱动程序处理。

每个设备都有一个请求队列。这是因为实际的磁盘读写操作可以在内核请求它们的时间很远的地方发生，而且因为内核需要灵活性，以便在最适当的时刻安排每次传输（例如，将影响磁盘上相邻扇区的请求组合在一起）。并且，你可能记得，请求函数在创建队列时与请求队列关联。让我们回顾一下 `sbull` 是如何创建它的队列的：

```c
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

因此，当队列被创建时，请求函数就与它关联了。我们还在创建队列的过程中提供了一个自旋锁。每当我们的请求函数被调用时，那个锁就被内核持有。因此，请求函数在原子上下文中运行；它必须遵循第5章中讨论的所有原子代码的常规规则。

队列锁也阻止内核在你的请求函数持有锁的时候为你的设备排队任何其他请求。在某些条件下，你可能会考虑在请求函数运行时放弃那个锁。然而，如果你这样做，你必须确保在没有持有锁的时候，不要访问请求队列，或者任何其他由锁保护的数据结构。你还必须在请求函数返回之前重新获取锁。

最后，请求函数的调用（通常）完全与任何用户空间进程的行为异步。你不能假设内核正在运行的上下文是启动当前请求的进程。你不知道请求提供的I/O缓冲区是在内核空间还是用户空间。所以，任何明确访问用户空间的操作都是错误的，肯定会导致问题。你会看到，你的驱动程序需要知道的关于请求的所有信息都包含在通过请求队列传递给你的结构中。

16.3.2. 一个简单的请求方法

`sbull`示例驱动程序提供了几种不同的请求处理方法。默认情况下，`sbull`使用一个叫做`sbull_request`的方法，这是最简单的请求方法的一个例子。下面就是它的代码：

```c
static void sbull_request(request_queue_t *q)
{
    struct request *req;
    while ((req = elv_next_request(q)) != NULL) {
        struct sbull_dev *dev = req->rq_disk->private_data;
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
        sbull_transfer(dev, req->sector, req->current_nr_sectors,
                req->buffer, rq_data_dir(req));
        end_request(req, 1);
    }
}
```

这个函数引入了`struct request`结构。我们稍后会详细研究`struct request`；现在，只需要知道它代表了我们要执行的一个块I/O请求。

内核提供了`elv_next_request`函数来获取队列上的第一个未完成的请求；当没有请求需要处理时，该函数返回NULL。注意，`elv_next_request`并不会从队列中移除请求。如果你在没有其他操作的情况下连续两次调用它，它会两次返回相同的请求结构。在这种简单的操作模式下，只有当请求完成时，请求才会从队列中被取出。

块请求队列可以包含并不实际将块移动到磁盘和从磁盘移动的请求。这样的请求可以包括特定于供应商的、低级别的诊断操作或与特殊设备模式相关的指令，例如可记录媒体的数据包写入模式。大多数块驱动程序不知道如何处理这样的请求，只是简单地使它们失败；`sbull`也是这样工作的。调用`blk_fs_request`告诉我们我们是否正在查看一个文件系统请求——一个移动数据块的请求。如果一个请求不是文件系统请求，我们将它传递给`end_request`：

```c
void end_request(struct request *req, int succeeded);
```

当我们处理非文件系统请求时，我们将`succeeded`作为0传递，表示我们没有成功完成请求。否则，我们调用`sbull_transfer`来实际移动数据，使用请求结构中提供的一组字段：

```c
sector_t sector; // 我们设备上开始扇区的索引。记住，这个扇区号，像所有这样的号码一样，是以512字节的扇区表示的。如果你的硬件使用不同的扇区大小，你需要相应地调整扇区。例如，如果硬件使用2048字节的扇区，你需要在将它放入硬件的请求之前，将开始扇区号除以四。

unsigned long nr_sectors; // 要传输的（512字节）扇区的数量。

char *buffer; // 指向要从中或向其中传输数据的缓冲区的指针。这个指针是一个内核虚拟地址，如果需要，驱动程序可以直接解引用它。

rq_data_dir(struct request *req); // 这个宏从请求中提取传输的方向；返回值为零表示从设备读取，非零返回值表示写入设备。
```

有了这些信息，`sbull`驱动程序可以用一个简单的`memcpy`调用来实现实际的数据传输——毕竟我们的数据已经在内存中了。执行这个复制操作的函数

（sbull_transfer）也处理扇区大小的缩放，并确保我们不会试图复制超出我们的虚拟设备的末尾：

```c
static void sbull_transfer(struct sbull_dev *dev,
unsigned long sector,
        unsigned long nsect, char *buffer, int write)
{
    unsigned long offset = sector*KERNEL_SECTOR_SIZE;
    unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;
    if ((offset + nbytes) > dev->size) {
        printk (KERN_NOTICE "Beyond-end write (%ld
%ld)\n", offset, nbytes);
        return;
    }
    if (write)
        memcpy(dev->data + offset, buffer, nbytes);
    else
        memcpy(buffer, dev->data + offset, nbytes);
}
```

有了这段代码，sbull实现了一个完整的、简单的基于RAM的磁盘设备。然而，由于几个原因，它并不是许多类型设备的现实驱动程序。

这些原因中的第一个是，`sbull`一次同步执行一个请求。高性能的磁盘设备能够同时有多个请求未完成；然后磁盘的板载控制器可以选择以最优的顺序执行它们（希望如此）。只要我们只处理队列中的第一个请求，我们就永远不可能在给定的时间内有多个请求被满足。能够处理多个请求需要对请求队列和请求结构有更深入的理解；接下来的几节将帮助建立这种理解。

然而，还有另一个问题需要考虑。当系统执行涉及多个位于磁盘上的连续扇区的大型传输时，可以从磁盘设备获得最佳性能。磁盘操作中的最高成本总是读写头的定位；一旦完成，实际读取或写入数据所需的时间几乎可以忽略不计。设计和实现文件系统和虚拟内存子系统的开发人员理解这一点，所以他们尽力将相关数据连续地定位在磁盘上，并尽可能在一个请求中传输尽可能多的扇区。块子系统在这方面也有所帮助；请求队列包含大量的逻辑，目的是找到相邻的请求并将它们合并成更大的操作。

然而，`sbull`驱动程序却忽略了所有这些工作。一次只传输一个缓冲区，这意味着最大的单个传输几乎永远不会超过单个页面的大小。一个块驱动程序可以做得比这更好，但是它需要对请求结构和构建请求的`bio`结构有更深入的理解。

接下来的几节将更深入地探讨块层如何完成其工作，以及由此产生的数据结构。

16.3.3. 请求队列

在最简单的意义上，块请求队列就是这样：一个块I/O请求的队列。但如果你深入了解，会发现请求队列实际上是一个相当复杂的数据结构。幸运的是，驱动程序不需要担心大部分的复杂性。

请求队列跟踪未完成的块I/O请求。但它们在创建这些请求时也起着关键作用。请求队列存储描述设备能够服务的请求类型的参数：它们的最大大小，一个请求中可能包含的独立段的数量，硬件扇区大小，对齐要求等。如果你的请求队列配置正确，它应该永远不会向你提出你的设备无法处理的请求。

请求队列还实现了一个插件接口，允许使用多个I/O调度器（或电梯）。I/O调度器的工作是以一种最大化性能的方式向你的驱动程序提出I/O请求。为此，大多数I/O调度器会积累一批请求，将它们按照增加（或减少）的块索引顺序排序，并按照这个顺序向驱动程序提出请求。当磁盘头得到一个排序后的请求列表时，它会从磁盘的一端工作到另一端，就像一个满载的电梯在满足所有的“请求”（等待下车的人）之前只会向一个方向移动。2.6内核包括一个“截止日期调度器”，它努力确保每个请求在预设的最大时间内得到满足，以及一个“预期调度器”，它在读取请求后会短暂地阻塞设备，预期另一个相邻的读取请求将几乎立即到达。在写这篇文章的时候，默认的调度器是预期调度器，它似乎提供了最好的交互系统性能。

I/O调度器还负责合并相邻的请求。当一个新的I/O请求交给调度器时，它会在队列中搜索涉及相邻扇区的请求；如果找到一个，并且结果请求不会太大，那么这两个请求就会被合并。

请求队列有一个类型为`struct request_queue`或`request_queue_t`。这个类型，以及许多在它上面操作的函数，都在`<linux/blkdev.h>`中定义。如果你对请求队列的实现感兴趣，你可以在`drivers/block/ll_rw_block.c`和`elevator.c`中找到大部分的代码。

16.3.3.1. 队列的创建和删除

如我们在示例代码中看到的，请求队列是一个动态数据结构，必须由块I/O子系统创建。创建和初始化请求队列的函数是：

```c
request_queue_t *blk_init_queue(request_fn_proc *request,
spinlock_t *lock);
```

参数当然是这个队列的请求函数和控制对队列的访问的自旋锁。这个函数分配内存（实际上是相当多的内存），因此可能会失败；在尝试使用队列之前，你应该始终检查返回值。

作为初始化请求队列的一部分，你可以将字段`queuedata`（它是一个`void *`指针）设置为你喜欢的任何值。这个字段是请求队列等同于我们在其他结构中看到的`private_data`。

要将请求队列返回给系统（通常在模块卸载时），调用`blk_cleanup_queue`：

```c
void blk_cleanup_queue(request_queue_t *);
```

在这个调用之后，你的驱动程序不再从给定的队列中看到任何请求，也不应再引用它。

16.3.3.2. 排队函数

对于驱动程序来说，用于操作队列上的请求的函数集非常小。在调用这些函数之前，你必须持有队列锁。

返回要处理的下一个请求的函数是`elv_next_request`：

```c
struct request *elv_next_request(request_queue_t *queue);
```

我们已经在简单的`sbull`示例中看到了这个函数。它返回一个指向要处理的下一个请求的指针（由I/O调度器确定），或者如果没有更多的请求需要处理，则返回NULL。`elv_next_request`将请求留在队列上，但将其标记为活动状态；这个标记防止I/O调度器在你开始执行它之后尝试将其他请求与这个请求合并。

要实际从队列中移除一个请求，使用`blkdev_dequeue_request`：

```c
void blkdev_dequeue_request(struct request *req);
```

如果你的驱动程序同时操作来自同一队列的多个请求，它必须以这种方式出队它们。

如果你需要出于某种原因将出队的请求放回队列，你可以调用：

```c
void elv_requeue_request(request_queue_t *queue, struct
request *req);
```

16.3.3.3. 队列控制函数

块层导出了一组函数，驱动程序可以使用这些函数来控制请求队列的操作。这些函数包括：

```c
void blk_stop_queue(request_queue_t *queue);

void blk_start_queue(request_queue_t *queue);
```

如果你的设备已经达到了不能处理更多未完成命令的状态，你可以调用 `blk_stop_queue` 来告诉块层。在这个调用之后，你的请求函数将不会被调用，直到你调用 `blk_start_queue` 。不用说，当你的设备可以处理更多请求时，你不应该忘记重新启动队列。在调用这两个函数时，必须持有队列锁。

```c
void blk_queue_bounce_limit(request_queue_t *queue, u64
dma_addr);
```

这个函数告诉内核你的设备可以执行DMA的最高物理地址。如果一个请求包含对超过限制的内存的引用，那么将使用一个弹跳缓冲区来进行操作；这当然是执行块I/O的一种昂贵方式，应尽可能避免。你可以在这个参数中提供任何合理的物理地址，或者使用预定义的符号 `BLK_BOUNCE_HIGH` （对高内存页面使用弹跳缓冲区）、 `BLK_BOUNCE_ISA` （驱动程序只能在16-MB ISA区域内进行DMA）或 `BLK_BOUNCE_ANY` （驱动程序可以对任何地址进行DMA）。默认值是 `BLK_BOUNCE_HIGH` 。

```c
void blk_queue_max_sectors(request_queue_t *queue,
unsigned short max);

void blk_queue_max_phys_segments(request_queue_t *queue,
unsigned short max);

void blk_queue_max_hw_segments(request_queue_t *queue,
unsigned short max);

void blk_queue_max_segment_size(request_queue_t *queue,
unsigned int max);
```

这些函数设置描述这个设备可以满足的请求的参数。 `blk_queue_max_sectors` 可以用来设置任何请求的最大大小（以512字节的扇区为单位）；默认值是255。 `blk_queue_max_phys_segments` 和 `blk_queue_max_hw_segments` 都控制一个单一请求中可能包含的物理段（系统内存中的非相邻区域）的数量。使用 `blk_queue_max_phys_segments` 来说明你的驱动程序准备处理的段的数量；例如，这可能是静态分配的scatterlist的大小。相反， `blk_queue_max_hw_segments`

是设备本身可以处理的最大段数量。这两个参数的默认值都是128。最后，`blk_queue_max_segment_size`告诉内核一个请求的任何单独段的大小可以是多少字节；默认值是65536字节。

```c
blk_queue_segment_boundary(request_queue_t *queue,
unsigned long mask);
```

有些设备不能处理跨越特定大小内存边界的请求；如果你的设备是其中之一，使用这个函数告诉内核那个边界。例如，如果你的设备在处理跨越4MB边界的请求时有问题，传入一个掩码0x3fffff。默认的掩码是0xffffffff。

```c
void blk_queue_dma_alignment(request_queue_t *queue, int
mask);
```

这个函数告诉内核你的设备对DMA传输施加的内存对齐约束。所有的请求都以给定的对齐创建，请求的长度也符合对齐。默认的掩码是0x1ff，这使得所有的请求都在512字节的边界上对齐。

```c
void blk_queue_hardsect_size(request_queue_t *queue,
unsigned short max);
```

告诉内核你的设备的硬件扇区大小。内核生成的所有请求都是这个大小的倍数，并且正确对齐。然而，块层和驱动程序之间的所有通信仍然以512字节的扇区表示。

16.3.4. 请求的分析

在我们的简单示例中，我们遇到了请求结构。然而，我们只是刚刚开始探索这个复杂的数据结构。在这一节中，我们将详细地看一下Linux内核中如何表示块I/O请求。

每个请求结构代表一个块I/O请求，尽管它可能是通过合并高级别的几个独立请求形成的。任何特定请求要传输的扇区可能分布在主内存中，尽管它们总是对应于块设备上的一组连续扇区。请求被表示为一组段，每个段对应一个内存缓冲区。内核可能会合并涉

及磁盘上相邻扇区的多个请求，但它永远不会在一个单独的请求结构中合并读和写操作。内核也确保不合并请求，如果结果会违反前一节中描述的任何请求队列限制。

请求结构基本上是实现为一个bio结构的链表，加上一些管理信息，以便驱动程序在处理请求时跟踪其位置。bio结构是对块I/O请求的一部分的低级描述；我们现在来看一下它。

```c
struct request {
    struct list_head queuelist;
    struct call_single_data csd;
    struct request_queue *q;
    struct bio *bio;
    struct bio *biotail;
    // ... other fields ...
};
```

在这个结构中，`queuelist`是用于将请求链接到请求队列的链表节点。`csd`是一个用于处理请求完成的数据结构。`q`是一个指向请求队列的指针。`bio`和`biotail`是指向请求的第一个和最后一个bio的指针。其他字段包括用于跟踪请求状态和位置的信息，以及用于处理请求完成的回调函数。

16.3.4.1. bio 结构

当内核，以文件系统、虚拟内存子系统或系统调用的形式，决定必须将一组块传输到块I/O设备或从块I/O设备传输时，它会组装一个bio结构来描述该操作。然后，该结构被交给块I/O代码，该代码将其合并到现有的请求结构中，或者如果需要，创建一个新的请求结构。bio结构包含了块驱动程序执行请求所需的所有内容，而无需参考启动该请求的用户空间进程。

bio结构在`<linux/bio.h>`中定义，包含许多可能对驱动程序作者有用的字段：

```c
sector_t bi_sector; // 这个bio要传输的第一个（512字节）扇区。

unsigned int bi_size; // 要传输的数据的大小，以字节为单位。相反，通常更容易使用bio_sectors(bio)，这是一个以扇区为单位给出大小的宏。

unsigned long bi_flags; // 描述bio的一组标志；如果这是一个写请求，最低有效位被设置（尽管应该使用宏bio_data_dir(bio)而不是直接查看标志）。

unsigned short bio_phys_segments;

unsigned short bio_hw_segments; // 分别是这个BIO中包含的物理段的数量和DMA映射完成后硬件看到的段的数量。
```

然而，bio的核心是一个名为 `bi_io_vec` 的数组，它由以下结构组成：

```c
struct bio_vec {

    struct page     *bv_page;

    unsigned int    bv_len;

    unsigned int    bv_offset;

};```
```

如图16-1所示，这些结构是如何全部关联在一起的。如你所见，当一个块I/O请求被转换为一个bio结构时，它已经被分解为单独的物理内存页面。驱动程序需要做的就是遍历这个结构数组（有`bi_vcnt`个），并在每个页面内传输数据（但只有从偏移开始的`len`字节）。

直接操作`bi_io_vec`数组是不被鼓励的，因为这可能会阻碍内核开发者在未来改变`bio`结构而不破坏现有代码。为此，提供了一组宏来简化与`bio`结构的交互。开始的地方是`bio_for_each_segment`，它简单地遍历`bi_io_vec`数组中的每个未处理条目。这个宏应该这样使用：

```c
int segno;

struct bio_vec *bvec;

bio_for_each_segment(bvec, bio, segno) {

    /* Do something with this segment */

}
```

在这个循环中，`bvec`指向当前的`bio_vec`条目，`segno`是当前的段号。这些值可以用来设置DMA传输（使用`blk_rq_map_sg`的另一种方式在“块请求和DMA”一节中描

述）。如果你需要直接访问页面，你应该首先确保存在一个适当的内核虚拟地址；为此，你可以使用：

```c
char *__bio_kmap_atomic(struct bio *bio, int i, enum
km_type type);

void __bio_kunmap_atomic(char *buffer, enum km_type
type);
```

这个低级函数允许你直接映射在给定的`bio_vec`中找到的缓冲区，如索引`i`所示。创建一个原子`kmap`；调用者必须提供要使用的适当的插槽（如第15章的“The Memory Map and Struct Page”一节中所述）。

块层还在`bio`结构中维护了一组指针，以跟踪请求处理的当前状态。存在几个宏来提供对该状态的访问：

```c
struct page *bio_page(struct bio *bio); // 返回一个指向要传输的页面的页面结构的指针。

int bio_offset(struct bio *bio); // 返回要传输的数据在页面内的偏移量。

int bio_cur_sectors(struct bio *bio); // 返回要从当前页面传输的扇区数。

char *bio_data(struct bio *bio); // 返回一个指向要传输的数据的内核逻辑地址。注意，只有当问题页面不位于高内存时，这个地址才可用；在其他情况下调用它是一个错误。默认情况下，块子系统不会将高内存缓冲区传递给你的驱动程序，但是如果你已经使用`blk_queue_bounce_limit`改变了这个设置，你可能不应该使用`bio_data`。

char *bio_kmap_irq(struct bio *bio, unsigned long *flags);

void bio_kunmap_irq(char *buffer, unsigned long *flags); // `bio_kmap_irq`返回任何缓冲区的内核虚拟地址，无论它是否位于高或低内存。使用了原子`kmap`，所以你的驱动程序不能在这个映射活动时睡眠。使用`bio_kunmap_irq`来取消映射缓冲区。注意，这里的`flags`参数是通过指针传递的。还要注意，由于使用了原子`kmap`，你不能一次映射多个段。
```

所有刚刚描述的函数都访问“当前”缓冲区——即内核知道的第一个未被传输的缓冲区。驱动程序通常希望在`bio`中处理几个缓冲区，然后再对任何一个缓冲区发出完成信号（使用`end_that_request_first`，将在稍后描述），所以这些函数通常不是很有用。还存在几个其他的宏用于处理`bio`结构的内部（详见`<linux/bio.h>`）。

16.3.4.2. 请求结构成员

现在我们对`bio`结构的工作方式有了一些了解，我们可以深入到`struct request`，看看请求处理是如何工作的。这个结构的字段包括：

```c
sector_t hard_sector;

unsigned long hard_nr_sectors;

unsigned int hard_cur_sectors;
```

这些字段跟踪驱动程序尚未完成的扇区。尚未传输的第一个扇区存储在`hard_sector`中，尚未传输的扇区总数在`hard_nr_sectors`中，当前`bio`中剩余的扇区数在`hard_cur_sectors`中。这些字段仅供块子系统内部使用；驱动程序不应使用它们。

```c
struct bio *bio;
```

`bio`是此请求的`bio`结构的链表。你不应直接访问此字段；应使用`rq_for_each_bio`（稍后描述）。

```c
char *buffer;
```

本章前面的简单驱动程序示例使用此字段来查找传输的缓冲区。有了我们更深入的理解，我们现在可以看到，这个字段只是在当前`bio`上调用`bio_data`的结果。

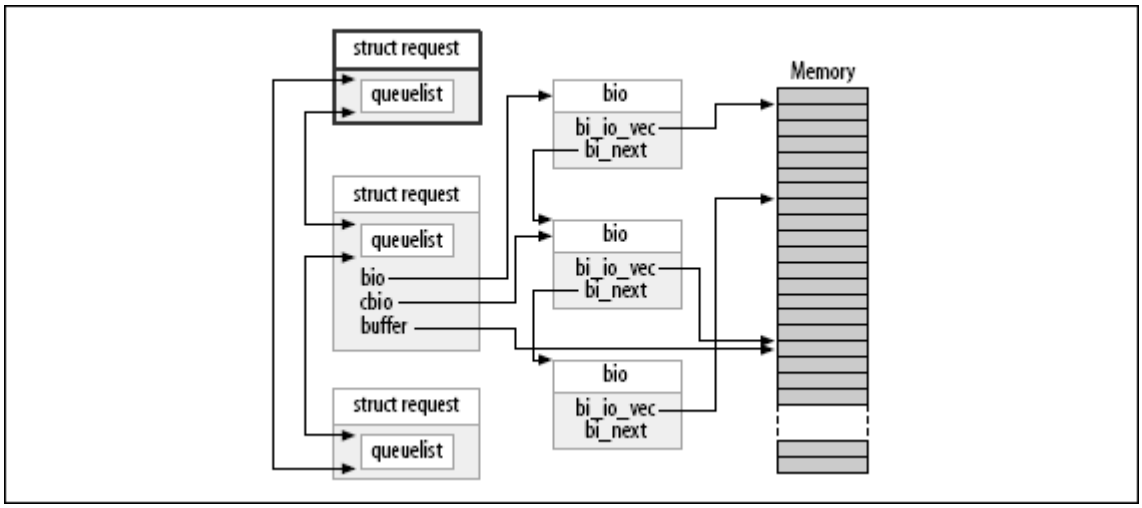
```c
unsigned short nr_phys_segments;
```

此请求在物理内存中占用的不同段的数量，合并了相邻的页面后。

```c
struct list_head queuelist;
```

链接请求到请求队列的链表结构（如第11章的“Linked Lists”一节中所述）。如果（且仅当）你使用`blkdev_dequeue_request`从队列中移除请求，你可以使用这个列表头来在你的驱动程序维护的内部列表中跟踪请求。

图16-2显示了请求结构及其组成的`bio`结构如何配合工作。在图中，请求已经部分满足；`cbio`和`buffer`字段指向尚未传输的第一个`bio`。



16.3.4.3. 屏障Barrier请求

块层在你的驱动程序看到请求之前会重新排序请求，以提高I/O性能。如果有理由这样做，你的驱动程序也可以重新排序请求。通常，这种重新排序是通过将多个请求传递给驱动程序，让硬件确定最优的排序。然而，对请求的无限制重新排序存在一个问题：一些应用程序需要保证某些操作在其他操作开始之前完成。例如，关系数据库管理器必须绝对确定他们的日志信息已经被刷新到驱动程序，然后再在数据库内容上执行事务。现在在大多数Linux系统上使用的日志文件系统，有非常类似的排序约束。如果错误的操作被重新排序，结果可能是严重的、未被检测到的数据损坏。

2.6版的块层通过引入“屏障请求”的概念来解决这个问题。如果一个请求被标记为`REQ_HARDBARRER`标志，那么它必须在启动任何后续请求之前写入驱动程序。我们说“写入驱动程序”，意味着数据必须实际存在并持久化在物理介质上。许多驱动程序执行写请求的缓存；这种缓存提高了性能，但它可能会破坏屏障请求的目的。如果在关键数据仍然在驱动程序的缓存中时发生电源故障，即使驱动程序已经报告完成，那么这些数据仍然会丢失。因此，实现屏障请求的驱动程序必须采取措施强制驱动程序实际将数据写入介质。

如果你的驱动程序支持屏障请求，第一步是通知块层这个事实。屏障处理是请求队列的另一个；它是通过以下方式设置的：

```c
void blk_queue_ordered(request_queue_t *queue, int flag);
```

要表示你的驱动程序实现了屏障请求，将 `flag` 参数设置为非零值。

屏障请求的实际实现只是在请求结构中测试相关标志的问题。提供了一个宏来执行这个测试：

```c
int blk_barrier_rq(struct request *req);
```

如果这个宏返回一个非零值，那么请求就是一个屏障请求。根据你的硬件的工作方式，你可能需要停止从队列中获取请求，直到屏障请求完成。其他驱动程序可以理解屏障请求；在这种情况下，你的驱动程序只需要为这些驱动程序发出适当的操作。

16.3.4.4. 不可重入Nonretryable请求

块驱动程序通常会尝试重试第一次失败的请求。这种行为可以使系统更可靠，帮助避免数据丢失。然而，内核有时会将请求标记为不可重试。如果这样的请求在第一次尝试时无法执行，那么应尽快失败。

如果你的驱动程序正在考虑重试失败的请求，它应首先调用：

```c
int blk_noretry_request(struct request *req);
```

如果这个宏返回一个非零值，你的驱动程序应该简单地用一个错误代码中止请求，而不是重试它。

16.3.5. 请求完成函数

当你的设备完成了I/O请求中的一部分或全部扇区的传输时，必须用以下函数通知块子系统：

```c
int end_that_request_first(struct request *req, int
success, int count);
```

这个函数告诉块代码，你的驱动程序已经完成了从你上次离开的地方开始的`count`个扇区的传输。如果I/O成功，将`success`传递为1；否则传递0。注意，你必须按照从第一个扇区到最后一个扇区的顺序标记完成；如果你的驱动程序和设备以某种方式共谋完成请求的顺序，你必须存储乱序完成的状态，直到传输了中间的扇区。

`end_that_request_first`的返回值是一个指示是否已经传输了此请求中的所有扇区的指示。返回值为0意味着所有扇区都已经传输，请求已经完成。在这一点上，你必须使用`blkdev_dequeue_request`出队请求（如果你还没有这样做），并将其传递给：

```c
void end_that_request_last(struct request *req);
```

`end_that_request_last`通知等待请求的人请求已经完成，并回收请求结构；必须在持有队列锁的情况下调用它。

在我们的简单的`sbull`示例中，我们没有使用上述任何函数。相反，那个例子中调用的是`end_request`。为了显示这个调用的效果，这里是2.6.10内核中看到的整个`end_request`函数：

```c
void end_request(struct request *req, int uptodate)

{

    if (!end_that_request_first(req, uptodate, req-
>hard_cur_sectors)) {

        add_disk_randomness(req→rq_disk);

        blkdev_dequeue_request(req);

        end_that_request_last(req);

    }

}
```

函数`add_disk_randomness`使用块I/O请求的时间来为系统的随机数池提供熵；只有当磁盘的时间真的是随机的时候，才应该调用它。这对于大多数机械设备来说是正确的，但对于基于内存的虚拟设备，如`sbull`，则不是这样。因此，下一节中显示的更复杂的`sbull`版本不调用`add_disk_randomness`。

16.3.5.1. 使用 bios

现在你已经知道足够的信息来编写一个直接与构成请求的`bio`结构体工作的块驱动程序。然而，一个例子可能会有所帮助。如果`sbull`驱动程序以`request_mode`参数设置为1加载，它将注册一个`bio`感知的请求函数，而不是我们之前看到的简单函数。那个函数看起来像这样：

```c
static void sbull_full_request(request_queue_t *q)
{
    struct request *req;
    int sectors_xferred;
    struct sbull_dev *dev = q->queuedata;
    while ((req = elv_next_request(q)) != NULL) {
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
        sectors_xferred = sbull_xfer_request(dev, req);
        if (! end_that_request_first(req, 1,
sectors_xferred)) {
            blkdev_dequeue_request(req);
            end_that_request_last(req);
        }
    }
}
```

这个函数只是接受每个请求，将其传递给 `sbull_xfer_request`，然后使用 `end_that_request_first` 和（如果需要） `end_that_request_last` 完成它。因此，这个函数处理的是问题的高级队列和请求管理部分。然而，实际执行请求的任务，落在 `sbull_xfer_request` 上：

```c
static int sbull_xfer_request(struct sbull_dev *dev,
struct request *req)

{

    struct bio *bio;

    int nsect = 0;

    rq_for_each_bio(bio, req) {

        sbull_xfer_bio(dev, bio);

        nsect += bio->bi_size/KERNEL_SECTOR_SIZE;

    }

    return nsect;

}
```

这里我们引入了另一个宏：`rq_for_each_bio`。正如你可能预期的，这个宏只是遍历请求中的每个 `bio` 结构，给我们一个指针，我们可以将其传递给 `sbull_xfer_bio` 进行传输。那个函数看起来像这样：

```c
static int sbull_xfer_bio(struct sbull_dev *dev, struct
bio *bio)

{

    int i;

    struct bio_vec *bvec;

    sector_t sector = bio→bi_sector;

    /* Do each segment independently. */

    bio_for_each_segment(bvec, bio, i) {

        char *buffer = __bio_kmap_atomic(bio, i,
KM_USER0);

        sbull_transfer(dev, sector, bio_cur_sectors(bio),

                buffer, bio_data_dir(bio) = = WRITE);

        sector += bio_cur_sectors(bio);

        __bio_kunmap_atomic(bio, KM_USER0);

    }

    return 0; /* Always "succeed" */

}
```

这个函数只是遍历 `bio` 结构中的每个段，获取一个内核虚拟地址来访问缓冲区，然后调用我们之前看到的相同的 `sbull_transfer` 函数来复制数据。

每个设备都有自己的需求，但是，一般来说，刚刚展示的代码应该作为许多需要通过 `bio` 结构进行操作的情况的模型。这些代码展示了如何处理请求队列，如何遍历每个请求中的 `bio` 结构，以及如何处理每个 `bio` 中的每个段。这为处理块设备I/O提供了一个基本的框架，可以根据特定设备的需求进行修改和扩展。

16.3.5.2. 块请求和 DMA

如果你正在开发一个高性能的块驱动程序，那么你可能会使用DMA进行实际的数据传输。一个块驱动程序当然可以像上面描述的那样遍历 `bio` 结构，为每一个创建一个DMA映射，并将结果传递给设备。然而，如果你的设备可以做散射/聚集I/O，有一个更简单的方法。函数：

```c
int blk_rq_map_sg(request_queue_t *queue, struct request
*req,
                  struct scatterlist *list);
```

用给定请求的所有段填充给定的列表。在插入到散列列表之前，内存中相邻的段会被合并，所以你不需要自己尝试检测它们。返回值是列表中的条目数。该函数还在其第三个参数中返回一个适合传递给 `dma_map_sg` 的散列列表。（有关 `dma_map_sg` 的更多信息，请参见第15章的“散射-聚集映射”部分。）

在调用 `blk_rq_map_sg` 之前，你的驱动程序必须为散列列表分配存储空间。列表必须能够容纳至少与请求具有的物理段一样多的条目； `struct request` 字段 `nr_phys_segments` 保存了这个计数，它不会超过用 `blk_queue_max_phys_segments` 指定的物理段的最大数量。

如果你不希望 `blk_rq_map_sg` 合并相邻的段，你可以通过如下调用改变默认行为：

```c
clear_bit(QUEUE_FLAG_CLUSTER, &queue->queue_flags);
```

一些SCSI磁盘驱动程序以这种方式标记他们的请求队列，因为他们并不从请求的合并中受益。

16.3.5.3. 不用一个请求队列

我们之前讨论过内核为优化队列中请求的顺序所做的工作；这项工作涉及对请求的排序，甚至可能暂停队列以等待预期的请求到达。这些技术在处理真正的旋转磁盘驱动器时有助于系统的性能。然而，对于像 `sbull` 这样的设备，这些技术完全是浪费的。许多块定向设备，如闪存阵列、用于数字相机的媒体卡读取器和RAM磁盘，都具有真正的随机访问性能，不会从高级请求队列逻辑中受益。其他设备，如软件RAID阵列或由逻辑卷管理器创建的虚拟磁盘，没有块层的请求队列优化的性能特性。对于这种设备，最好直接从块层接受请求，而不用管请求队列。

对于这些情况，块层支持“无队列”模式的操作。要使用这种模式，你的驱动程序必须提供一个“make request”函数，而不是一个请求函数。 `make_request` 函数有这样的原型：

```c
typedef int (make_request_fn) (request_queue_t *q, struct
bio *bio);
```

请注意，尽管它实际上永远不会持有任何请求，但请求队列仍然存在。 `make_request` 函数的主要参数是一个 `bio` 结构，它代表一个或多个要传输的缓冲区。 `make_request` 函数可以做两件事：它可以直接执行传输，或者可以将请求重定向到另一个设备。

直接执行传输只是通过我们之前描述的访问器方法处理 `bio` 。然而，由于没有请求结构可以使用，所以你的函数应该直接向 `bio` 结构的创建者发出完成信号，通过调用 `bio_endio` ：

```c
void bio_endio(struct bio *bio, unsigned int bytes, int
error);
```

这里， `bytes` 是你到目前为止传输的字节数。它可以小于 `bio` 整体代表的字节数；这样，你可以发出部分完成的信号，并更新 `bio` 内部的“当前缓冲区”指针。你应该在你的设备进一步处理时再次调用 `bio_endio` ，或者如果你无法完成请求，发出错误信号。错误是通过为错误参数提供一个非零值来指示的；这个值通常是一个错误代码，如 `-EIO` 。 `make_request` 应该返回0，无论I/O是否成功。

如果 sbull 以 request_mode=2 加载，它将使用一个 make_request 函数。由于 sbull 已经有一个可以传输单个 bio 的函数，所以 make_request 函数很简单：

```c
static int sbull_make_request(request_queue_t *q, struct
bio *bio)

{

    struct sbull_dev *dev = q->queuedata;

    int status;

    status = sbull_xfer_bio(dev, bio);

    bio_endio(bio, bio->bi_size, status);

    return 0;

}
```

请注意，你永远不应该从一个常规的请求函数中调用 bio_endio；这个工作由 end_that_request_first 来处理。

一些块驱动程序，如实现卷管理器和软件RAID阵列的驱动程序，真正需要将请求重定向到处理实际I/O的另一个设备。编写这样的驱动程序超出了本书的范围。然而，我们注意到，如果 make_request 函数返回一个非零值， bio 会被再次提交。因此，一个“堆叠”驱动程序可以修改 bi_bdev 字段指向一个不同的设备，改变起始扇区值，然后返回；然后块系统将 bio 传递给新设备。还有一个 bio_split 调用可以用来将 bio 分割成多个块，以提交给多个设备。虽然如果队列参数设置正确，以这种方式分割 bio 几乎永远不必要。

无论哪种方式，你都必须告诉块子系统你的驱动程序正在使用一个自定义的 make_request 函数。为此，你必须使用以下方法分配一个请求队列：

```c
request_queue_t *blk_alloc_queue(int flags);
```

这个函数与`blk_init_queue`不同，它实际上并没有设置队列来持有请求。`flags`参数是用于为队列分配内存的分配标志集；通常正确的值是`GFP_KERNEL`。一旦你有了一个队列，将它和你的`make_request`函数传递给`blk_queue_make_request`：

```c
void blk_queue_make_request(request_queue_t *queue,
make_request_fn *func);
```

设置`make_request`函数的`sbull`代码看起来像这样：

```c
dev->queue = blk_alloc_queue(GFP_KERNEL);

if (dev->queue == NULL)

    goto out_vfree;

blk_queue_make_request(dev->queue, sbull_make_request);
```

对于好奇的人，花些时间深入研究`drivers/block/ll_rw_block.c`会发现所有队列都有一个`make_request`函数。默认版本`generic_make_request`处理将`bio`合并到请求结构中。通过提供自己的`make_request`函数，驱动程序实际上只是覆盖了一个特定的请求队列方法，并解决了大部分工作。

16.4. 一些其他的细节

这一部分将涵盖块层的一些其他方面，这些方面可能对高级驱动程序有兴趣。以下所有设施都不需要用来编写一个正确的驱动程序，但在某些情况下它们可能会有所帮助。

16.4.1. 命令预准备

块层为驱动程序提供了一种机制，可以在请求从 elv_next_request 返回之前检查和预处理请求。这种机制允许驱动程序提前设置实际的驱动命令，决定是否可以处理请求，或者执行其他类型的清理工作。

如果你想使用这个特性，创建一个符合以下原型的命令准备函数：

```c
typedef int (prep_rq_fn) (request_queue_t *queue, struct request *req);
```

请求结构包含一个名为 cmd 的字段，这是一个 BLK_MAX_CDB 字节的数组；这个数组可以被准备函数用来存储实际的硬件命令（或任何其他有用的信息）。这个函数应该返回以下值之一：

- BLKPREP_OK：命令准备正常进行，请求可以交给你的驱动程序的请求函数。
- BLKPREP_KILL：此请求无法完成；它以错误代码失败。
- BLKPREP_DEFER：此请求目前无法完成。它停留在队列的前端，但不交给请求函数。

准备函数在请求返回给你的驱动程序之前，由 elv_next_request 立即调用。如果这个函数返回 BLKPREP_DEFER，那么从 elv_next_request 到你的驱动程序的返回值是 NULL。如果例如你的设备已经达到了它可以有的最大请求数量，这种操作模式可能会很有用。

要让块层调用你的准备函数，将它传递给：

```c
void blk_queue_prep_rq(request_queue_t *queue, prep_rq_fn *func);
```

默认情况下，请求队列没有准备函数。

16.4.2. 被标识的命令排队

可以同时激活多个请求的硬件通常支持某种形式的标记命令队列（TCQ）。TCQ只是将一个整数“标记”附加到每个请求的技术，这样当驱动完成其中一个请求时，它可以告诉

驱动程序哪一个。在内核的早期版本中，实现TCQ的块驱动程序必须自己做所有的工作；在2.6版本中，为所有驱动程序使用的块层添加了TCQ支持基础设施。

如果你的驱动执行标记命令队列，你应该在初始化时通过调用以下函数通知内核：

```c
int blk_queue_init_tags(request_queue_t *queue, int
depth, struct blk_queue_tag *tags);
```

这里，`queue`是你的请求队列，`depth`是你的设备在任何给定时间可以有的标记请求的数量。`tags`是一个可选的指向`struct blk_queue_tag`结构的数组的指针；必须有`depth`个。通常，`tags`可以作为`NULL`传递，`blk_queue_init_tags`分配数组。然而，如果你需要在多个设备之间共享相同的标签，你可以传递另一个请求队列的标签数组指针（存储在`queue_tags`字段中）。你永远不应该自己分配标签数组；块层需要初始化数组，并且不向模块导出初始化函数。

由于`blk_queue_init_tags`分配内存，它可能会失败；在这种情况下，它返回一个负的错误代码给调用者。

如果你的设备可以处理的标签数量发生变化，你可以通过以下函数通知内核：

```c
int blk_queue_resize_tags(request_queue_t *queue, int
new_depth);
```

在调用期间必须持有队列锁。这个调用可能会失败，在这种情况下返回一个负的错误代码。

使用`blk_queue_start_tag`将标签与请求结构关联起来，必须在持有队列锁的情况下调用：

```c
int blk_queue_start_tag(request_queue_t *queue, struct
request *req);
```

如果标签可用，此函数为此请求分配它，将标签号存储在`req->tag`中，并返回0。它还将请求从队列中出队，并将其链接到自己的标签跟踪结构中，所以如果你的驱动程序使用标签，应该小心不要自己出队请求。如果没有更多的标签可用，`blk_queue_start_tag`将请求留在队列上并返回一个非零值。

当给定请求的所有传输都已完成时，你的驱动程序应该返回标签：

```c
void blk_queue_end_tag(request_queue_t *queue, struct
request *req);
```

再次，你必须在调用此函数之前持有队列锁。这个调用应该在`end_that_request_first`返回0（意味着请求已完成）但在调用`end_that_request_last`之前进行。记住，请求已经出队，所以此时你的驱动程序这样做将是一个错误。

如果你需要找到与给定标签关联的请求（例如，当驱动程序报告完成时），使用`blk_queue_find_tag`：

```c
struct request *blk_queue_find_tag(request_queue_t
*queue, int tag);
```

返回值是关联的请求结构，除非出现真正的错误。

如果事情真的出错，你的驱动程序可能会发现自己必须重置或对其设备执行其他暴力行为。在这种情况下，任何未完成的标记命令都不会完成。块层提供了一个函数，可以在这种情况下帮助恢复工作：

```c
void blk_queue_invalidate_tags(request_queue_t *queue);
```

此函数将所有未完成的标签返回到池中，并将关联的请求放回请求队列中。当你调用此函数时，必须持有队列锁。

16.5. 快速参考

```
#include <linux/fs.h>
int register_blkdev(unsigned int major, const char *name);
int unregister_blkdev(unsigned int major, const char
*name);
```

register_blkdev 注册一个块驱动到内核, 并且, 可选地, 获得一个主编号. 一个驱动可被注销, 使用 unregister_blkdev.

```
struct block_device_operations
```

持有大部分块驱动的方法的结构.

```
#include <linux/genhd.h>
struct gendisk;
```

描述内核中单个块设备的结构.

```
struct gendisk *alloc_disk(int minors);
void add_disk(struct gendisk *gd);
```

分配 gendisk 结构的函数, 并且返回它们到系统.

```
void set_capacity(struct gendisk *gd, sector_t sectors);
```

存储设备能力(以 512-字节)在 gendisk 结构中.

```
void add_disk(struct gendisk *gd);
```

添加一个磁盘到内核. 一旦调用这个函数, 你的磁盘的方法可被内核调用.

```
int check_disk_change(struct block_device *bdev);
```

一个内核函数, 检查在给定磁盘驱动器中的介质改变, 并且采取要求的清理动作当检测到这样一个改变.

```
#include <linux/blkdev.h>
request_queue_t blk_init_queue(request_fn_proc *request,
spinlock_t *lock);
void blk_cleanup_queue(request_queue_t *);
```

处理块请求队列的创建和删除的函数.

```
struct request *elv_next_request(request_queue_t *queue);
void end_request(struct request *req, int success);
```

elv_next_request 从一个请求队列中获得下一个请求; end_request 可用在每个简单驱动器中来标识一个(或部分)请求完成.

```
void blkdev_dequeue_request(struct request *req);
void elv_requeue_request(request_queue_t *queue, struct
request *req);
```

从队列中除去一个请求, 并且放回它的函数如果需要.

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```

如果你需要阻止对你的请求函数的进一步调用, 调用 blk_stop_queue 来完成. 调用 blk_start_queue 来使你的请求方法被再次调用.

```
void blk_queue_bounce_limit(request_queue_t *queue, u64
dma_addr);
void blk_queue_max_sectors(request_queue_t *queue,
unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue,
unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue,
unsigned short max);
void blk_queue_max_segment_size(request_queue_t *queue,
unsigned int max);
blk_queue_segment_boundary(request_queue_t *queue,
unsigned long mask);
void blk_queue_dma_alignment(request_queue_t *queue, int
mask);
void blk_queue_hardsect_size(request_queue_t *queue,
unsigned short max);
```

设置各种队列参数的函数, 来控制请求如何被创建给一个特殊设备; 这些参数在"队列控制函数"一节中描述.

```
#include <linux/bio.h>
struct bio;
```

低级函数, 表示一个块 I/O 请求的一部分.

```
bio_sectors(struct bio *bio);
bio_data_dir(struct bio *bio);
```

2 个宏定义, 表示一个由 bio 结构描述的传送的大小和方向.

```
bio_for_each_segment(bvec, bio, segno);
```

一个伪控制结构, 用来循环组成一个 bio 结构的各个段.

```
char *__bio_kmap_atomic(struct bio *bio, int i, enum
km_type type);
void __bio_kunmap_atomic(char *buffer, enum km_type type);
```

bio_kmap_atomic 可用来创建一个内核虚拟地址给一个在 bio 结构中的给定的段. 映射必须使用 bio_kunmap_atomic 来恢复.

```
struct page *bio_page(struct bio *bio);
int bio_offset(struct bio *bio);int bio_cur_sectors(struct
bio *bio);
char *bio_data(struct bio *bio);
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
```

一组存取者宏定义, 提供对一个 bio 结构中的"当前"段的存取.

```
void blk_queue_ordered(request_queue_t *queue, int flag);
int blk_barrier_rq(struct request *req);
```

如果你的驱动实现屏障请求, 调用 blk_queue_ordered -- 如同它应当做的. 宏 blk_barrier_rq 返回一个非零值如果当前请求是一个屏障请求.

```
int blk_noretry_request(struct request *req);
```

这个宏返回一个非零值, 如果给定的请求不应当在出错时重新尝试.

```
int end_that_request_first(struct request *req, int
success, int count);
void end_that_request_last(struct request *req);
```

使用 end_that_request_firest 来指示一个块 I/O 请求的一部分完成. 当那个函数返回 0, 请求完成并且应当被传递给 end_that_request_last.

```
rq_for_each_bio(bio, request)
```

另一个用宏定义来实现的控制结构; 它步入构成一个请求的每个 bio.

```
int blk_rq_map_sg(request_queue_t *queue, struct request
*req, struct scatterlist *list);
```

为一次 DMA 传送填充给定的散布表, 用需要来映射给定请求中的缓冲的信息

```
typedef int (make_request_fn) (request_queue_t *q, struct
bio *bio);
```

make_request 函数的原型.

```
void bio_endio(struct bio *bio, unsigned int bytes, int
error);
```

指示一个给定 bio 的完成. 这个函数应当只用在你的驱动直接获取 bio , 通过 make_request 函数从块层.

```
request_queue_t *blk_alloc_queue(int flags);
void blk_queue_make_request(request_queue_t *queue,
make_request_fn *func);
```

使用 blk_alloc_queue 来分配由定制的 make_request 函数使用的请求队列, . 那个函数应当使用 blk_queue_make_request 来设置.

```
typedef int (prep_rq_fn) (request_queue_t *queue, struct
request *req);
void blk_queue_prep_rq(request_queue_t *queue, prep_rq_fn
*func);
```

一个命令准备函数的原型和设置函数, 它可用来准备必要的硬件命令, 在请求被传递给你的请求函数之前.

```
int blk_queue_init_tags(request_queue_t *queue, int depth,
struct blk_queue_tag *tags);
int blk_queue_resize_tags(request_queue_t *queue, int
new_depth);
int blk_queue_start_tag(request_queue_t *queue, struct
request *req);
void blk_queue_end_tag(request_queue_t *queue, struct
request *req);
struct request *blk_queue_find_tag(request_queue_t *qeue,
int tag);void blk_queue_invalidate_tags(request_queue_t
*queue);
```

驱动使用被标记的命令队列的支持函数.