CHAPTER 16
Block Drivers

So far, our discussion has been limited to char drivers. There are other types of drivers in Linux systems, however, and the time has come for us to widen our focus somewhat. Accordingly, this chapter discusses block drivers.

A block driver provides access to devices that transfer randomly accessible data in fixed-size blocks—disk drives, primarily. The Linux kernel sees block devices as being fundamentally different from char devices; as a result, block drivers have a distinct interface and their own particular challenges.

Efficient block drivers are critical for performance—and not just for explicit reads and writes in user applications. Modern systems with virtual memory work by shifting (hopefully) unneeded data to secondary storage, which is usually a disk drive. Block drivers are the conduit (通道) between core memory and secondary storage; therefore, they can be seen as making up part of the virtual memory subsystem. While it is possible to write a block driver without knowing about `struct page` and other important memory concepts, anybody needing to write a high-performance driver has to draw upon the material covered in Chapter 15.

Much of the design of the block layer is centered on performance. Many char devices can run below their maximum speed, and the performance of the system as a whole is not affected. The system cannot run well, however, if its block I/O subsystem is not well-tuned. The Linux block driver interface allows you to get the most out of a block device but imposes, necessarily, a degree of complexity that you must deal with. Happily, the 2.6 block interface is much improved over what was found in older kernels.

The discussion in this chapter is, as one would expect, centered on an example driver that implements a block-oriented, memory-based device. It is, essentially, a ramdisk. The kernel already contains a far superior ramdisk implementation, but our driver (called *sbull*) lets us demonstrate the creation of a block driver while minimizing unrelated complexity.

Before getting into the details, let's define a couple of terms precisely. A *block* is a fixed-size chunk of data, the size being determined by the kernel. Blocks are often 4096 bytes, but that value can vary depending on the architecture and the exact filesystem being used. A *sector*, in contrast, is a small block whose size is usually determined by the underlying hardware. The kernel expects to be dealing with devices that implement 512-byte sectors. If your device uses a different size, the kernel adapts and avoids generating I/O requests that the hardware cannot handle. It is worth keeping in mind, however, that any time the kernel presents you with a sector number, it is working in a world of 512-byte sectors. If you are using a different hardware sector size, you have to scale the kernel's sector numbers accordingly. We see how that is done in the *sbull* driver.

Registration

Block drivers, like char drivers, must use a set of registration interfaces to make their devices available to the kernel. The concepts are similar, but the details of block device registration are all different. You have a whole new set of data structures and device operations to learn.

Block Driver Registration

The first step taken by most block drivers is to register themselves with the kernel. The function for this task is *register_blkdev* (which is declared in <*linux/fs.h*>):

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

The arguments are the major number that your device will be using and the associated name (which the kernel will display in */proc/devices*). If `major` is passed as `0`, the kernel allocates a new major number and returns it to the caller. As always, a negative return value from *register_blkdev* indicates that an error has occurred.

The corresponding function for canceling a block driver registration is:

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

Here, the arguments must match those passed to *register_blkdev*, or the function returns `-EINVAL` and not unregister anything.

In the 2.6 kernel, the call to *register_blkdev* is entirely optional. The functions performed by *register_blkdev* have been decreasing over time; the only tasks performed by this call at this point are (1) allocating a dynamic major number if requested, and (2) creating an entry in */proc/devices*. In future kernels, *register_blkdev* may be removed altogether. Meanwhile, however, most drivers still call it; it's traditional.

Disk Registration

While *register_blkdev* can be used to obtain a major number, it does not make any disk drives available to the system. There is a separate registration interface that you must use to manage individual drives. Using this interface requires familiarity with a pair of new structures, so that is where we start.

Block device operations

Char devices make their operations available to the system by way of the `file_operations` structure. A similar structure is used with block devices; it is `struct block_device_operations`, which is declared in *<linux/fs.h>*. The following is a brief overview of the fields found in this structure; we revisit them in more detail when we get into the details of the *sbull* driver:

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

Functions that work just like their char driver equivalents; they are called whenever the device is opened and closed. A block driver might respond to an open call by spinning up the device, locking the door (for removable media), etc. If you lock media into the device, you should certainly unlock it in the *release* method.

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

Method that implements the *ioctl* system call. The block layer first intercepts a large number of standard requests, however; so most block driver *ioctl* methods are fairly short.

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

Method called by the kernel to check whether the user has changed the media in the drive, returning a nonzero value if so. Obviously, this method is only applicable to drives that support removable media (and that are smart enough to make a "media changed" flag available to the driver); it can be omitted in other cases.

The `struct gendisk` argument is how the kernel represents a single disk; we will be looking at that structure in the next section.

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

The *revalidate_disk* method is called in response to a media change; it gives the driver a chance to perform whatever work is required to make the new media ready for use. The function returns an `int` value, but that value is ignored by the kernel.

```
struct module *owner;
```

A pointer to the module that owns this structure; it should usually be initialized to `THIS_MODULE`.

Attentive readers may have noticed an interesting omission from this list: there are no functions that actually read or write data. In the block I/O subsystem, these operations are handled by the *request* function, which deserves a large section of its own and is discussed later in the chapter. Before we can talk about servicing requests, we must complete our discussion of disk registration.

The gendisk structure

`struct gendisk` (declared in *<linux/genhd.h>*) is the kernel's representation of an individual disk device. In fact, the kernel also uses `gendisk` structures to represent partitions, but driver authors need not be aware of that. There are several fields in `struct gendisk` that must be initialized by a block driver:

`int major;`
`int first_minor;`
`int minors;`
: Fields that describe the device number(s) used by the disk. At a minimum, a drive must use at least one minor number. If your drive is to be partitionable, however (and most should be), you want to allocate one minor number for each possible partition as well. A common value for `minors` is 16, which allows for the "full disk" device and 15 partitions. Some disk drivers use 64 minor numbers for each device.

`char disk_name[32];`
: Field that should be set to the name of the disk device. It shows up in */proc/partitions* and sysfs.

`struct block_device_operations *fops;`
: Set of device operations from the previous section.

`struct request_queue *queue;`
: Structure used by the kernel to manage I/O requests for this device; we examine it in the section "Request Processing."

`int flags;`
: A (little-used) set of flags describing the state of the drive. If your device has removable media, you should set `GENHD_FL_REMOVABLE`. CD-ROM drives can set `GENHD_FL_CD`. If, for some reason, you do not want partition information to show up in */proc/partitions*, set `GENHD_FL_SUPPRESS_PARTITION_INFO`.

`sector_t capacity;`
: The capacity of this drive, in 512-byte sectors. The `sector_t` type can be 64 bits wide. Drivers should not set this field directly; instead, pass the number of sectors to *set_capacity*.

`void *private_data;`
: Block drivers may use this field for a pointer to their own internal data.

The kernel provides a small set of functions for working with gendisk structures. We introduce them here, then see how *sbull* uses them to make its disk devices available to the system.

`struct gendisk` is a dynamically allocated structure that requires special kernel manipulation to be initialized; drivers cannot allocate the structure on their own. Instead, you must call:

```
struct gendisk *alloc_disk(int minors);//initialized
```

The `minors` argument should be the number of minor numbers this disk uses; note that you cannot change the `minors` field later and expect things to work properly.

When a disk is no longer needed, it should be freed with:

```
void del_gendisk(struct gendisk *gd);//freed
```

A gendisk is a reference-counted structure (it contains a kobject). There are *get_disk* and *put_disk* functions available to manipulate the reference count, but drivers should never need to do that. Normally, the call to *del_gendisk* removes the final reference to a gendisk, but there are no guarantees of that. Thus, it is possible that the structure could continue to exist (and your methods could be called) after a call to *del_gendisk*. If you delete the structure when there are no users (that is, after the final *release* or in your module *cleanup* function), however, you can be sure that you will not hear from it again.

Allocating a gendisk structure does not make the disk available to the system. To do that, you must initialize the structure and call *add_disk*:

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

Keep one important thing in mind here: as soon as you call *add_disk*, the disk is "live" and its methods can be called at any time. In fact, the first such calls will probably happen even before *add_disk* returns; the kernel will read the first few blocks in an attempt to find a partition table. So you should not call *add_disk* until your driver is completely initialized and ready to respond to requests on that disk.

Initialization in sbull

It is time to get down to some examples. The *sbull* driver (available from O'Reilly's FTP site with the rest of the example source) implements a set of in-memory virtual disk drives. For each drive, *sbull* allocates (with *vmalloc*, for simplicity) an array of memory; it then makes that array available via block operations. The *sbull* driver can be tested by partitioning the virtual device, building filesystems on it, and mounting it in the system hierarchy.

Like our other example drivers, *sbull* allows a major number to be specified at compile or module load time. If no number is specified, one is allocated dynamically. Since a call to *register_blkdev* is required for dynamic allocation, *sbull* does so:

```
sbull_major = register_blkdev(sbull_major, "sbull");
if (sbull_major <= 0) {
```

```
    printk(KERN_WARNING "sbull: unable to get major number\n");
    return -EBUSY;
}
```

Also, like the other virtual devices we have presented in this book, the *sbull* device is described by an internal structure:

```
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 */
};
```

Several steps are required to initialize this structure and make the associated device available to the system. We start with basic initialization and allocation of the underlying memory:

```
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);
```

It's important to allocate and initialize a spinlock before the next step, which is the allocation of the request queue. We look at this process in more detail when we get to request processing; for now, suffice it to say that the necessary call is:

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

Here, *sbull_request* is our *request* function—the function that actually performs block read and write requests. When we allocate a request queue, we must provide a spinlock that controls access to that queue. The lock is provided by the driver rather than the general parts of the kernel because, often, the request queue and other driver data structures fall within the same critical section; they tend to be accessed together. As with any function that allocates memory, *blk_init_queue* can fail, so you must check the return value before continuing.

Once we have our device memory and request queue in place, we can allocate, initialize, and install the corresponding `gendisk` structure. The code that does this work is:

```
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);
```

Here, `SBULL_MINORS` is the number of minor numbers each *sbull* device supports. When we set the first minor number for each device, we must take into account all of the numbers taken by prior devices. The name of the disk is set such that the first one is *sbulla*, the second *sbullb*, and so on. User space can then add partition numbers so that the third partition on the second device might be */dev/sbullb3*.

Once everything is set up, we finish with a call to *add_disk*. Chances are that several of our methods will have been called for that disk by the time *add_disk* returns, so we take care to make that call the very last step in the initialization of our device.

A Note on Sector Sizes

As we have mentioned before, the kernel treats every disk as a linear array of 512-byte sectors. Not all hardware uses that sector size, however. Getting a device with a different sector size to work is not particularly hard; it is just a matter of taking care of a few details. The *sbull* device exports a `hardsect_size` parameter that can be used to change the "hardware" sector size of the device; by looking at its implementation, you can see how to add this sort of support to your own drivers.

The first of those details is to inform the kernel of the sector size your device supports. The hardware sector size is a parameter in the request queue, rather than in the `gendisk` structure. This size is set with a call to *blk_queue_hardsect_size* immediately after the queue is allocated:

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

Once that is done, the kernel adheres to your device's hardware sector size. All I/O requests are properly aligned at the beginning of a hardware sector, and the length of each request is an integral number of sectors. You must remember, however, that the kernel always expresses itself in 512-byte sectors; thus, it is necessary to translate all sector numbers accordingly. So, for example, when *sbull* sets the capacity of the device in its `gendisk` structure, the call looks like:

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

`KERNEL_SECTOR_SIZE` is a locally-defined constant that we use to scale between the kernel's 512-byte sectors and whatever size we have been told to use. This sort of calculation pops up frequently as we look at the *sbull* request processing logic.

The Block Device Operations

We had a brief introduction to the `block_device_operations` structure in the previous section. Now we take some time to look at these operations in a bit more detail before getting into request processing. To that end, it is time to mention one other feature of the *sbull* driver: it pretends to be a removable device. Whenever the last user closes the device, a 30-second timer is set; if the device is not opened during that time, the contents of the device are cleared, and the kernel will be told that the media has been changed. The 30-second delay gives the user time to, for example, mount an *sbull* device after creating a filesystem on it.

The open and release Methods

To implement the simulated media removal, *sbull* must know when the last user has closed the device. A count of users is maintained by the driver. It is the job of the *open* and *close* methods to keep that count current.

The *open* method looks very similar to its char-driver equivalent; it takes the relevant `inode` and `file` structure pointers as arguments. When an inode refers to a block device, the field `i_bdev->bd_disk` contains a pointer to the associated `gendisk` structure; this pointer can be used to get to a driver's internal data structures for the device. That is, in fact, the first thing that the *sbull open* method does:

```
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);//check whether a media change has happened
    dev->users++;
    spin_unlock(&dev->lock);
    return 0;
}
```

Once *sbull_open* has its device structure pointer, it calls *del_timer_sync* to remove the "media removal" timer, if any is active. Note that we do not lock the device spinlock until after the timer has been deleted; doing otherwise invites deadlock if the timer function runs before we can delete it. With the device locked, we call a kernel function called *check_disk_change* to check whether a media change has happened. One might argue that the kernel should make that call, but the standard pattern is for drivers to handle it at *open* time.

The last step is to increment the user count and return.

The task of the *release* method is, in contrast, to decrement the user count and, if indicated, start the media removal timer:

```
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;
}
```

In a driver that handles a real, hardware device, the *open* and *release* methods would set the state of the driver and hardware accordingly. This work could involve spinning the disk up or down, locking the door of a removable device, allocating DMA buffers, etc.

You may be wondering who actually opens a block device. There are some operations that cause a block device to be opened directly from user space; these include partitioning a disk, building a filesystem on a partition, or running a filesystem checker. A block driver also sees an *open* call when a partition is mounted. In this case, there is no user-space process holding an open file descriptor for the device; the open file is, instead, held by the kernel itself. A block driver cannot tell the difference between a *mount* operation (which opens the device from kernel space) and the invocation of a utility such as *mkfs* (which opens it from user space).

Supporting Removable Media

The `block_device_operations` structure includes two methods for supporting removable media. If you are writing a driver for a nonremovable device, you can safely omit these methods. Their implementation is relatively straightforward.

The *media_changed* method is called (from *check_disk_change*) to see whether the media has been changed; it should return a nonzero value if this has happened. The *sbull* implementation is simple; it queries a flag that has been set if the media removal timer has expired:

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

    return dev->media_change;
}
```

The *revalidate* method is called after a media change; its job is to do whatever is required to prepare the driver for operations on the new media, if any. After the call to *revalidate*, the kernel attempts to reread the partition table and start over with the device. The *sbull* implementation simply resets the `media_change` flag and zeroes out the device memory to simulate the insertion of a blank disk.

```
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;
}
```

The ioctl Method

Block devices can provide an *ioctl* method to perform device control functions. The higher-level block subsystem code intercepts a number of *ioctl* commands before your driver ever gets to see them, however (see *drivers/block/ioctl.c* in the kernel source for the full set). In fact, a modern block driver may not have to implement very many *ioctl* commands at all.

The *sbull ioctl* method handles only one command—a request for the device's geometry:

```
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 */
    }
```

Providing geometry information may seem like a curious task, since our device is purely virtual and has nothing to do with tracks and cylinders. Even most real-block hardware has been furnished with much more complicated structures for many years. The kernel is not concerned with a block device's geometry; it sees it simply as a linear array of sectors. There are certain user-space utilities that still expect to be able to query a disk's geometry, however. In particular, the *fdisk* tool, which edits partition tables, depends on cylinder information and does not function properly if that information is not available.

We would like the *sbull* device to be partitionable, even with older, simple-minded tools. So, we have provided an *ioctl* method that comes up with a credible fiction for a geometry that could match the capacity of our device. Most disk drivers do something similar. Note that, as usual, the sector count is translated, if need be, to match the 512-byte convention used by the kernel.

Request Processing

The core of every block driver is its *request* function. This function is where the real work gets done—or at least started; all the rest is overhead. Consequently, we spend a fair amount of time looking at request processing in block drivers.

A disk driver's performance can be a critical part of the performance of the system as a whole. Therefore, the kernel's block subsystem has been written with performance very much in mind; it does everything possible to enable your driver to get the most out of the devices it controls. This is a good thing, in that it enables blindingly fast I/O. On the other hand, the block subsystem unnecessarily exposes a great deal of complexity in the driver API. It is possible to write a very simple *request* function (we will see one shortly), but if your driver must perform at a high level on complex hardware, it will be anything but simple.

Introduction to the request Method

The block driver *request* method has the following prototype:

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

This function is called whenever the kernel believes it is time for your driver to process some reads, writes, or other operations on the device. The *request* function does not need to actually complete all of the requests on the queue before it returns; indeed, it probably does not complete any of them for most real devices. It must,

however, make a start on those requests and ensure that they are all, eventually, processed by the driver.

Every device has a request queue. This is because actual transfers to and from a disk can take place far away from the time the kernel requests them, and because the kernel needs the flexibility to schedule each transfer at the most propitious moment (grouping together, for instance, requests that affect sectors close together on the disk). And the *request* function, you may remember, is associated with a request queue when that queue is created. Let us look back at how *sbull* makes its queue:

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

Thus, when the queue is created, the *request* function is associated with it. We also provided a spinlock as part of the queue creation process. Whenever our *request* function is called, that lock is held by the kernel. As a result, the *request* function is running in an atomic context; it must follow all of the usual rules for atomic code discussed in Chapter 5.

The queue lock also prevents the kernel from queuing any other requests for your device while your *request* function holds the lock. Under some conditions, you may want to consider dropping that lock while the *request* function runs. If you do so, however, you must be sure not to access the request queue, or any other data structure protected by the lock, while the lock is not held. You must also reacquire the lock before the *request* function returns.

Finally, the invocation of the *request* function is (usually) entirely asynchronous with respect to the actions of any user-space process. You cannot assume that the kernel is running in the context of the process that initiated the current request. You do not know if the I/O buffer provided by the request is in kernel or user space. So any sort of operation that explicitly accesses user space is in error and will certainly lead to trouble. As you will see, everything your driver needs to know about the request is contained within the structures passed to you via the request queue.

A Simple request Method

The *sbull* example driver provides a few different methods for request processing. By default, *sbull* uses a method called *sbull_request*, which is meant to be an example of the simplest possible *request* method. Without further ado, here it is:

```
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);
        }
    }
```

This function introduces the `struct request` structure. We will examine `struct request` in great detail later on; for now, suffice it to say that it represents a block I/O request for us to execute.

The kernel provides the function *elv_next_request* to obtain the first incomplete request on the queue; that function returns `NULL` when there are no requests to be processed. Note that *elv_next_request* does not remove the request from the queue. If you call it twice with no intervening operations, it returns the same `request` structure both times. In this simple mode of operation, requests are taken off the queue only when they are complete.

A block request queue can contain requests that do not actually move blocks to and from a disk. Such requests can include vendor-specific, low-level diagnostics operations or instructions relating to specialized device modes, such as the packet writing mode for recordable media. Most block drivers do not know how to handle such requests and simply fail them; *sbull* works in this way as well. The call to *block_fs_request* tells us whether we are looking at a filesystem request—one that moves blocks of data. If a request is not a filesystem request, we pass it to *end_request*:

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

When we dispose of nonfilesystem requests, we pass `succeeded` as `0` to indicate that we did not successfully complete the request. Otherwise, we call *sbull_transfer* to actually move the data, using a set of fields provided in the `request` structure:

`sector_t sector;`
: The index of the beginning sector on our device. Remember that this sector number, like all such numbers passed between the kernel and the driver, is expressed in 512-byte sectors. If your hardware uses a different sector size, you need to scale `sector` accordingly. For example, if the hardware uses 2048-byte sectors, you need to divide the beginning sector number by four before putting it into a request for the hardware.

`unsigned long nr_sectors;`
: The number of (512-byte) sectors to be transferred.

`char *buffer;`
: A pointer to the buffer to or from which the data should be transferred. This pointer is a kernel virtual address and can be dereferenced directly by the driver if need be.

`rq_data_dir(struct request *req);`
: This macro extracts the direction of the transfer from the request; a zero return value denotes a read from the device, and a nonzero return value denotes a write to the device.

Given this information, the *sbull* driver can implement the actual data transfer with a simple *memcpy* call—our data is already in memory, after all. The function that performs this copy operation (*sbull_transfer*) also handles the scaling of sector sizes and ensures that we do not try to copy beyond the end of our virtual device:

```
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);
}
```

With the code, *sbull* implements a complete, simple RAM-based disk device. It is not, however, a realistic driver for many types of devices, for a couple of reasons.

The first of those reasons is that *sbull* executes requests synchronously, one at a time. High-performance disk devices are capable of having numerous requests outstanding at the same time; the disk's onboard controller can then choose to execute them in the optimal order (one hopes). As long as we process only the first request in the queue, we can never have multiple requests being fulfilled at a given time. Being able to work with more than one request requires a deeper understanding of request queues and the `request` structure; the next few sections help build that understanding.

There is another issue to consider, however. The best performance is obtained from disk devices when the system performs large transfers involving multiple sectors that are located together on the disk. The highest cost in a disk operation is always the positioning of the read and write heads; once that is done, the time required to actually read or write the data is almost insignificant. The developers who design and implement filesystems and virtual memory subsystems understand this, so they do their best to locate related data contiguously on the disk and to transfer as many sectors as possible in a single request. The block subsystem also helps in this regard;

request queues contain a great deal of logic aimed at finding adjacent requests and coalescing them into larger operations.

The *sbull* driver, however, takes all that work and simply ignores it. Only one buffer is transferred at a time, meaning that the largest single transfer is almost never going to exceed the size of a single page. A block driver can do much better than that, but it requires a deeper understanding of `request` structures and the `bio` structures from which requests are built.

The next few sections delve more deeply into how the block layer does its job and the data structures that result from that work.

Request Queues

In the simplest sense, a block request queue is exactly that: a queue of block I/O requests. If you look under the hood, a request queue turns out to be a surprisingly complex data structure. Fortunately, drivers need not worry about most of that complexity.

Request queues keep track of outstanding block I/O requests. But they also play a crucial role in the creation of those requests. The request queue stores parameters that describe what kinds of requests the device is able to service: their maximum size, how many separate segments may go into a request, the hardware sector size, alignment requirements, etc. If your request queue is properly configured, it should never present you with a request that your device cannot handle.

Request queues also implement a plug-in interface that allows the use of multiple *I/O schedulers* (or *elevators*) to be used. An I/O scheduler's job is to present I/O requests to your driver in a way that maximizes performance. To this end, most I/O schedulers accumulate a batch of requests, sort them into increasing (or decreasing) block index order, and present the requests to the driver in that order. The disk head, when given a sorted list of requests, works its way from one end of the disk to the other, much like a full elevator moves in a single direction until all of its "requests" (people waiting to get off) have been satisfied. The 2.6 kernel includes a "deadline scheduler," which makes an effort to ensure that every request is satisfied within a preset maximum time, and an "anticipatory scheduler," which actually stalls a device briefly after a read request in anticipation that another, adjacent read will arrive almost immediately. As of this writing, the default scheduler is the anticipatory scheduler, which seems to give the best interactive system performance.

The I/O scheduler is also charged with merging adjacent requests. When a new I/O request is handed to the scheduler, it searches the queue for requests involving adjacent sectors; if one is found and if the resulting request would not be too large, the two requests are merged.

Request queues have a type of `struct request_queue` or `request_queue_t`. This type, and the many functions that operate on it, are defined in *<linux/blkdev.h>*. If you are

interested in the implementation of request queues, you can find most of the code in *drivers/block/ll_rw_block.c* and *elevator.c*.

Queue creation and deletion

As we saw in our example code, a request queue is a dynamic data structure that must be created by the block I/O subsystem. The function to create and initialize a request queue is:

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

The arguments are, of course, the *request* function for this queue and a spinlock that controls access to the queue. This function allocates memory (quite a bit of memory, actually) and can fail because of this; you should always check the return value before attempting to use the queue.

As part of the initialization of a request queue, you can set the field `queuedata` (which is a `void *` pointer) to any value you like. This field is the request queue's equivalent to the `private_data` we have seen in other structures.

To return a request queue to the system (at module unload time, generally), call *blk_cleanup_queue*:

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

After this call, your driver sees no more requests from the given queue and should not reference it again.

Queueing functions

There is a very small set of functions for the manipulation of requests on queues—at least, as far as drivers are concerned. You must hold the queue lock before you call these functions.

The function that returns the next request to process is *elv_next_request*:

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

We have already seen this function in the simple *sbull* example. It returns a pointer to the next request to process (as determined by the I/O scheduler) or `NULL` if no more requests remain to be processed. *elv_next_request* leaves the request on the queue but marks it as being active; this mark prevents the I/O scheduler from attempting to merge other requests with this one once you start to execute it.

To actually remove a request from a queue, use *blkdev_dequeue_request*:

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

If your driver operates on multiple requests from the same queue simultaneously, it must dequeue them in this manner.

Should you need to put a dequeued request back on the queue for some reason, you can call:

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

Queue control functions

The block layer exports a set of functions that can be used by a driver to control how a request queue operates. These functions include:

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

If your device has reached a state where it can handle no more outstanding commands, you can call *blk_stop_queue* to tell the block layer. After this call, your *request* function will not be called until you call *blk_start_queue*. Needless to say, you should not forget to restart the queue when your device can handle more requests. The queue lock must be held when calling either of these functions.

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

Function that tells the kernel the highest physical address to which your device can perform DMA. If a request comes in containing a reference to memory above the limit, a bounce buffer will be used for the operation; this is, of course, an expensive way to perform block I/O and should be avoided whenever possible. You can provide any reasonable physical address in this argument, or make use of the predefined symbols `BLK_BOUNCE_HIGH` (use bounce buffers for high-memory pages), `BLK_BOUNCE_ISA` (the driver can DMA only into the 16-MB ISA zone), or `BLK_BOUNCE_ANY` (the driver can perform DMA to any address). The default value is `BLK_BOUNCE_HIGH`.

```
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);
```

Functions that set parameters describing the requests that can be satisfied by this device. *blk_queue_max_sectors* can be used to set the maximum size of any request in (512-byte) sectors; the default is 255. *blk_queue_max_phys_segments* and *blk_queue_max_hw_segments* both control how many physical segments (nonadjacent areas in system memory) may be contained within a single request. Use *blk_queue_max_phys_segments* to say how many segments your driver is prepared to cope with; this may be the size of a staticly allocated scatterlist, for example. *blk_queue_max_hw_segments*, in contrast, is the maximum number of segments that the device itself can handle. Both of these parameters default to 128. Finally, *blk_queue_max_segment_size* tells the kernel how large any individual segment of a request can be in bytes; the default is 65,536 bytes.

`blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);`
: Some devices cannot handle requests that cross a particular size memory boundary; if your device is one of those, use this function to tell the kernel about that boundary. For example, if your device has trouble with requests that cross a 4-MB boundary, pass in a mask of `0x3fffff`. The default mask is `0xffffffff`.

`void blk_queue_dma_alignment(request_queue_t *queue, int mask);`
: Function that tells the kernel about the memory alignment constraints your device imposes on DMA transfers. All requests are created with the given alignment, and the length of the request also matches the alignment. The default mask is `0x1ff`, which causes all requests to be aligned on 512-byte boundaries.

`void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);`
: Tells the kernel about your device's hardware sector size. All requests generated by the kernel are a multiple of this size and are properly aligned. All communications between the block layer and the driver continues to be expressed in 512-byte sectors, however.

The Anatomy of a Request

In our simple example, we encountered the `request` structure. However, we have barely scratched the surface of that complicated data structure. In this section, we look, in some detail, at how block I/O requests are represented in the Linux kernel.

Each `request` structure represents one block I/O request, although it may have been formed through a merger of several independent requests at a higher level. The sectors to be transferred for any particular request may be distributed throughout main memory, although they always correspond to a set of consecutive sectors on the block device. The request is represented as a set of segments, each of which corresponds to one in-memory buffer. The kernel may join multiple requests that involve adjacent sectors on the disk, but it never combines read and write operations within a single `request` structure. The kernel also makes sure not to combine requests if the result would violate any of the request queue limits described in the previous section.

A `request` structure is implemented, essentially, as a linked list of `bio` structures combined with some housekeeping information to enable the driver to keep track of its position as it works through the request. The `bio` structure is a low-level description of a portion of a block I/O request; we take a look at it now.

The bio structure

When the kernel, in the form of a filesystem, the virtual memory subsystem, or a system call, decides that a set of blocks must be transferred to or from a block I/O device; it puts together a `bio` structure to describe that operation. That structure is then handed to the block I/O code, which merges it into an existing `request` structure or, if need be, creates a new one. The `bio` structure contains everything that a

block driver needs to carry out the request without reference to the user-space process that caused that request to be initiated.

The `bio` structure, which is defined in *<linux/bio.h>*, contains a number of fields that may be of use to driver authors:

`sector_t bi_sector;`
: The first (512-byte) sector to be transferred for this `bio`.

`unsigned int bi_size;`
: The size of the data to be transferred, in bytes. Instead, it is often easier to use `bio_sectors(bio)`, a macro that gives the size in sectors.

`unsigned long bi_flags;`
: A set of flags describing the `bio`; the least significant bit is set if this is a write request (although the macro `bio_data_dir(bio)` should be used instead of looking at the flags directly).

`unsigned short bio_phys_segments;`
`unsigned short bio_hw_segments;`
: The number of physical segments contained within this BIO and the number of segments seen by the hardware after DMA mapping is done, respectively.

The core of a `bio`, however, is an array called `bi_io_vec`, which is made up of the following structure:

```
struct bio_vec {
        struct page     *bv_page;
        unsigned int    bv_len;
        unsigned int    bv_offset;
};
```

Figure 16-1 shows how these structures all tie together. As you can see, by the time a block I/O request is turned into a `bio` structure, it has been broken down into individual pages of physical memory. All a driver needs to do is to step through this array of structures (there are `bi_vcnt` of them), and transfer data within each page (but only `len` bytes starting at `offset`).

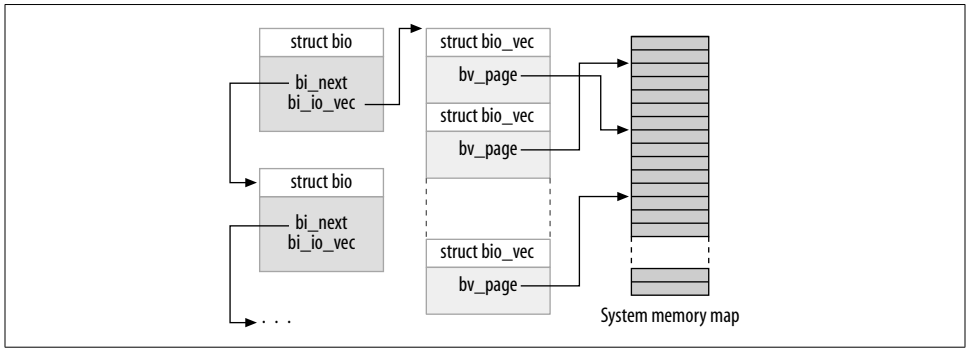

Figure 16-1. The bio structure

Working directly with the bi_io_vec array is discouraged in the interest of kernel developers being able to change the bio structure in the future without breaking things. To that end, a set of macros has been provided to ease the process of working with the bio structure. The place to start is with bio_for_each_segment, which simply loops through every unprocessed entry in the bi_io_vec array. This macro should be used as follows:

```
int segno;
struct bio_vec *bvec;

bio_for_each_segment(bvec, bio, segno) {
    /* Do something with this segment
}
```

Within this loop, bvec points to the current bio_vec entry, and segno is the current segment number. These values can be used to set up DMA transfers (an alternative way using *blk_rq_map_sg* is described in the section "Block requests and DMA"). If you need to access the pages directly, you should first ensure that a proper kernel virtual address exists; to that end, you can use:

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

This low-level function allows you to directly map the buffer found in a given bio_vec, as indicated by the index i. An atomic kmap is created; the caller must provide the appropriate slot to use (as described in the section "The Memory Map and Struct Page" in Chapter 15).

The block layer also maintains a set of pointers within the bio structure to keep track of the current state of request processing. Several macros exist to provide access to that state:

`struct page *bio_page(struct bio *bio);`
: Returns a pointer to the page structure representing the page to be transferred next.

`int bio_offset(struct bio *bio);`
: Returns the offset within the page for the data to be transferred.

`int bio_cur_sectors(struct bio *bio);`
: Returns the number of sectors to be transferred out of the current page.

`char *bio_data(struct bio *bio);`
: Returns a kernel logical address pointing to the data to be transferred. Note that this address is available only if the page in question is not located in high memory; calling it in other situations is a bug. By default, the block subsystem does not pass high-memory buffers to your driver, but if you have changed that setting with *blk_queue_bounce_limit*, you probably should not be using bio_data.

```
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
```

bio_kmap_irq returns a kernel virtual address for any buffer, regardless of whether it resides in high or low memory. An atomic kmap is used, so your driver cannot sleep while this mapping is active. Use *bio_kunmap_irq* to unmap the buffer. Note that the `flags` argument is passed by pointer here. Note also that since an atomic kmap is used, you cannot map more than one segment at a time.

All of the functions just described access the "current" buffer—the first buffer that, as far as the kernel knows, has not been transferred. Drivers often want to work through several buffers in the `bio` before signaling completion on any of them (with *end_that_request_first*, to be described shortly), so these functions are often not useful. Several other macros exist for working with the internals of the `bio` structure (see *<linux/bio.h>* for details).

Request structure fields

Now that we have an idea of how the `bio` structure works, we can get deep into `struct request` and see how request processing works. The fields of this structure include:

```
sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;
```

Fields that track the sectors that the driver has yet to complete. The first sector that has *not* been transferred is stored in `hard_sector`, the total number of sectors yet to transfer is in `hard_nr_sectors`, and the number of sectors remaining in the current `bio` is `hard_cur_sectors`. These fields are intended for use only within the block subsystem; drivers should not make use of them.

```
struct bio *bio;
```

`bio` is the linked list of `bio` structures for this request. You should not access this field directly; use *rq_for_each_bio* (described later) instead.

```
char *buffer;
```

The simple driver example earlier in this chapter used this field to find the buffer for the transfer. With our deeper understanding, we can now see that this field is simply the result of calling *bio_data* on the current `bio`.

```
unsigned short nr_phys_segments;
```

The number of distinct segments occupied by this request in physical memory after adjacent pages have been merged.

```
struct list_head queuelist;
```

The linked-list structure (as described in the section "Linked Lists" in Chapter 11) that links the request into the request queue. If (and only if) you

remove the request from the queue with *blkdev_dequeue_request*, you may use this list head to track the request in an internal list maintained by your driver.

Figure 16-2 shows how the request structure and its component bio structures fit together. In the figure, the request has been partially satisfied; the `cbio` and `buffer` fields point to the first bio that has not yet been transferred.

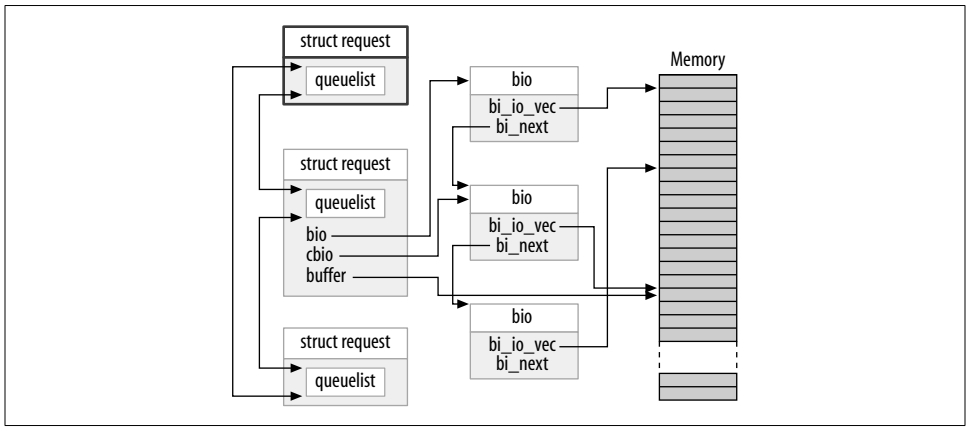

Figure 16-2. A request queue with a partially processed request

There are many other fields inside the `request` structure, but the list in this section should be enough for most driver writers.

Barrier requests

The block layer reorders requests before your driver sees them to improve I/O performance. Your driver, too, can reorder requests if there is a reason to do so. Often, this reordering happens by passing multiple requests to the drive and letting the hardware figure out the optimal ordering. There is a problem with unrestricted reordering of requests, however: some applications require guarantees that certain operations will complete before others are started. Relational database managers, for example, must be absolutely sure that their journaling information has been flushed to the drive before executing a transaction on the database contents. Journaling filesystems, which are now in use on most Linux systems, have very similar ordering constraints. If the wrong operations are reordered, the result can be severe, undetected data corruption.

The 2.6 block layer addresses this problem with the concept of a *barrier request*. If a request is marked with the `REQ_HARDBARRER` flag, it must be written to the drive before any following request is initiated. By "written to the drive," we mean that the data must actually reside and be persistent on the physical media. Many drives perform caching of write requests; this caching improves performance, but it can defeat the purpose of barrier requests. If a power failure occurs when the critical data is still sitting in

the drive's cache, that data is still lost even if the drive has reported completion. So a driver that implements barrier requests must take steps to force the drive to actually write the data to the media.

If your driver honors barrier requests, the first step is to inform the block layer of this fact. Barrier handling is another of the request queues; it is set with:

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

To indicate that your driver implements barrier requests, set the `flag` parameter to a nonzero value.

The actual implementation of barrier requests is simply a matter of testing for the associated flag in the `request` structure. A macro has been provided to perform this test:

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

If this macro returns a nonzero value, the request is a barrier request. Depending on how your hardware works, you may have to stop taking requests from the queue until the barrier request has been completed. Other drives can understand barrier requests themselves; in this case, all your driver has to do is to issue the proper operations for those drives.

Nonretryable requests

Block drivers often attempt to retry requests that fail the first time. This behavior can lead to a more reliable system and help to avoid data loss. The kernel, however, sometimes marks requests as not being retryable. Such requests should simply fail as quickly as possible if they cannot be executed on the first try.

If your driver is considering retrying a failed request, it should first make a call to:

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

If this macro returns a nonzero value, your driver should simply abort the request with an error code instead of retrying it.

Request Completion Functions

There are, as we will see, several different ways of working through a `request` structure. All of them make use of a couple of common functions, however, which handle the completion of an I/O request or parts of a request. Both of these functions are atomic and can be safely called from an atomic context.

When your device has completed transferring some or all of the sectors in an I/O request, it must inform the block subsystem with:

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

This function tells the block code that your driver has finished with the transfer of `count` sectors starting where you last left off. If the I/O was successful, pass `success`

as 1; otherwise pass 0. Note that you must signal completion in order from the first sector to the last; if your driver and device somehow conspire to complete requests out of order, you have to store the out-of-order completion status until the intervening sectors have been transferred.

The return value from *end_that_request_first* is an indication of whether all sectors in this request have been transferred or not. A return value of 0 means that all sectors have been transferred and that the request is complete. At that point, you must dequeue the request with *blkdev_dequeue_request* (if you have not already done so) and pass it to:

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

end_that_request_last informs whoever is waiting for the request that it has completed and recycles the request structure; it must be called with the queue lock held.

In our simple *sbull* example, we didn't use any of the above functions. That example, instead, is called *end_request*. To show the effects of this call, here is the entire *end_request* function as seen in the 2.6.10 kernel:

```
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);
    }
}
```

The function *add_disk_randomness* uses the timing of block I/O requests to contribute entropy to the system's random number pool; it should be called only if the disk's timing is truly random. That is true for most mechanical devices, but it is not true for a memory-based virtual device, such as *sbull*. For this reason, the more complicated version of *sbull* shown in the next section does not call *add_disk_randomness*.

Working with bios

You now know enough to write a block driver that works directly with the bio structures that make up a request. An example might help, however. If the *sbull* driver is loaded with the request_mode parameter set to 1, it registers a bio-aware *request* function instead of the simple function we saw above. That function looks like this:

```
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);
        }
    }
}
```

This function simply takes each request, passes it to *sbull_xfer_request*, then completes it with *end_that_request_first* and, if necessary, *end_that_request_last*. Thus, this function is handling the high-level queue and request management parts of the problem. The job of actually executing a request, however, falls to *sbull_xfer_request*:

```
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;
}
```

Here we introduce another macro: *rq_for_each_bio*. As you might expect, this macro simply steps through each `bio` structure in the request, giving us a pointer that we can pass to *sbull_xfer_bio* for the transfer. That function looks like:

```
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" */
}
```

This function simply steps through each segment in the `bio` structure, gets a kernel virtual address to access the buffer, then calls the same *sbull_transfer* function we saw earlier to copy the data over.

Each device has its own needs, but, as a general rule, the code just shown should serve as a model for many situations where digging through the `bio` structures is needed.

Block requests and DMA

If you are working on a high-performance block driver, chances are you will be using DMA for the actual data transfers. A block driver can certainly step through the `bio` structures, as described above, create a DMA mapping for each one, and pass the result to the device. There is an easier way, however, if your device can do scatter/gather I/O. The function:

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

fills in the given `list` with the full set of segments from the given request. Segments that are adjacent in memory are coalesced prior to insertion into the scatterlist, so you need not try to detect them yourself. The return value is the number of entries in the list. The function also passes back, in its third argument, a scatterlist suitable for passing to *dma_map_sg*. (See the section "Scatter-gather mappings" in Chapter 15 for more information on *dma_map_sg*.)

Your driver must allocate the storage for the scatterlist before calling *blk_rq_map_sg*. The list must be able to hold at least as many entries as the request has physical segments; the `struct request` field `nr_phys_segments` holds that count, which will not exceed the maximum number of physical segments specified with *blk_queue_max_phys_segments*.

If you do not want *blk_rq_map_sg* to coalesce adjacent segments, you can change the default behavior with a call such as:

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

Some SCSI disk drivers mark their request queue in this way, since they do not benefit from the coalescing of requests.

Doing without a request queue

Previously, we have discussed the work the kernel does to optimize the order of requests in the queue; this work involves sorting requests and, perhaps, even stalling the queue to allow an anticipated request to arrive. These techniques help the system's performance when dealing with a real, spinning disk drive. They are completely wasted, however, with a device like *sbull*. Many block-oriented devices, such as flash memory arrays, readers for media cards used in digital cameras, and RAM disks have truly random-access performance and do not benefit from advanced-request queueing logic. Other devices, such as software RAID arrays or virtual disks created by logical volume managers, do not have the performance characteristics for which the block layer's request queues are optimized. For this kind of device, it

would be better to accept requests directly from the block layer and not bother with the request queue at all.

For these situations, the block layer supports a "no queue" mode of operation. To make use of this mode, your driver must provide a "make request" function, rather than a *request* function. The *make_request* function has this prototype:

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

Note that a request queue is still present, even though it will never actually hold any requests. The *make_request* function takes as its main parameter a `bio` structure, which represents one or more buffers to be transferred. The *make_request* function can do one of two things: it can either perform the transfer directly, or it can redirect the request to another device.

Performing the transfer directly is just a matter of working through the `bio` with the accessor methods we described earlier. Since there is no `request` structure to work with, however, your function should signal completion directly to the creator of the `bio` structure with a call to *bio_endio*:

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

Here, `bytes` is the number of bytes you have transferred so far. It can be less than the number of bytes represented by the `bio` as a whole; in this way, you can signal partial completion, and update the internal "current buffer" pointers within the `bio`. You should either call *bio_endio* again as your device makes further process, or signal an error if you are unable to complete the request. Errors are indicated by providing a nonzero value for the `error` parameter; this value is normally an error code such as `-EIO`. The *make_request* should return `0`, regardless of whether the I/O is successful.

If *sbull* is loaded with `request_mode=2`, it operates with a *make_request* function. Since *sbull* already has a function that can transfer a single `bio`, the *make_request* function is simple:

```
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;
}
```

Please note that you should never call *bio_endio* from a regular *request* function; that job is handled by *end_that_request_first* instead.

Some block drivers, such as those implementing volume managers and software RAID arrays, really need to redirect the request to another device that handles the actual I/O. Writing such a driver is beyond the scope of this book. We note, however, that if the *make_request* function returns a nonzero value, the `bio` is submitted

again. A "stacking" driver can, therefore, modify the `bi_bdev` field to point to a different device, change the starting sector value, then return; the block system then passes the `bio` to the new device. There is also a *bio_split* call that can be used to split a `bio` into multiple chunks for submission to more than one device. Although if the queue parameters are set up correctly, splitting a `bio` in this way should almost never be necessary.

Either way, you must tell the block subsystem that your driver is using a custom *make_request* function. To do so, you must allocate a request queue with:

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

This function differs from *blk_init_queue* in that it does not actually set up the queue to hold requests. The `flags` argument is a set of allocation flags to be used in allocating memory for the queue; usually the right value is `GFP_KERNEL`. Once you have a queue, pass it and your *make_request* function to *blk_queue_make_request*:

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

The *sbull* code to set up the *make_request* function looks like:

```
dev->queue = blk_alloc_queue(GFP_KERNEL);
if (dev->queue == NULL)
    goto out_vfree;
blk_queue_make_request(dev->queue, sbull_make_request);
```

For the curious, some time spent digging through *drivers/block/ll_rw_block.c* shows that all queues have a *make_request* function. The default version, *generic_make_request*, handles the incorporation of the `bio` into a `request` structure. By providing a *make_request* function of its own, a driver is really just overriding a specific *request queue* method and sorting out much of the work.

Some Other Details

This section covers a few other aspects of the block layer that may be of interest for advanced drivers. None of the following facilities need to be used to write a correct driver, but they may be helpful in some situations.

Command Pre-Preparation

The block layer provides a mechanism for drivers to examine and preprocess requests before they are returned from *elv_next_request*. This mechanism allows drivers to set up the actual drive commands ahead of time, decide whether the request can be handled at all, or perform other sorts of housekeeping.

If you want to use this feature, create a command preparation function that fits this prototype:

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

The request structure includes a field called cmd, which is an array of BLK_MAX_CDB bytes; this array may be used by the preparation function to store the actual hardware command (or any other useful information). This function should return one of the following values:

BLKPREP_OK
: Command preparation went normally, and the request can be handed to your driver's *request* function.

BLKPREP_KILL
: This request cannot be completed; it is failed with an error code.

BLKPREP_DEFER
: This request cannot be completed at this time. It stays at the front of the queue but is not handed to the *request* function.

The preparation function is called by *elv_next_request* immediately before the request is returned to your driver. If this function returns BLKPREP_DEFER, the return value from *elv_next_request* to your driver is NULL. This mode of operation can be useful if, for example, your device has reached the maximum number of requests it can have outstanding.

To have the block layer call your preparation function, pass it to:

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

By default, request queues have no preparation function.

Tagged Command Queueing

Hardware that can have multiple requests active at once usually supports some form of *tagged command queueing* (TCQ). TCQ is simply the technique of attaching an integer "tag" to each request so that when the drive completes one of those requests, it can tell the driver which one. In previous versions of the kernel, block drivers that implemented TCQ had to do all of the work themselves; in 2.6, a TCQ support infrastructure has been added to the block layer for all drivers to use.

If your drive performs tagged command queueing, you should inform the kernel of that fact at initialization time with a call to:

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

Here, queue is your request queue, and depth is the number of tagged requests your device can have outstanding at any given time. tags is an optional pointer to an array of struct blk_queue_tag structures; there must be depth of them. Normally, tags can be passed as NULL, and *blk_queue_init_tags* allocates the array. If, however, you need to share the same tags between multiple devices, you can pass the tags array pointer (stored in the queue_tags field) from another request queue. You should never actually

allocate the tags array yourself; the block layer needs to initialize the array and does not export the initialization function to modules.

Since *blk_queue_init_tags* allocates memory, it can fail; it returns a negative error code to the caller in that case.

If the number of tags your device can handle changes, you can inform the kernel with:

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

The queue lock must be held during the call. This call can fail, returning a negative error code in that case.

The association of a tag with a request structure is done with *blk_queue_start_tag*, which must be called with the queue lock held:

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

If a tag is available, this function allocates it for this request, stores the tag number in req->tag, and returns 0. It also dequeues the request from the queue and links it into its own tag-tracking structure, so your driver should take care not to dequeue the request itself if it's using tags. If no more tags are available, *blk_queue_start_tag* leaves the request on the queue and returns a nonzero value.

When all transfers for a given request have been completed, your driver should return the tag with:

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

Once again, you must hold the queue lock before calling this function. The call should be made after *end_that_request_first* returns 0 (meaning that the request is complete) but before calling *end_that_request_last*. Remember that the request is already dequeued, so it would be a mistake for your driver to do so at this point.

If you need to find the request associated with a given tag (when the drive reports completion, for example), use *blk_queue_find_tag*:

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

The return value is the associated request structure, unless something has gone truly wrong.

If things really do go wrong, your driver may find itself having to reset or perform some other act of violence against one of its devices. In that case, any outstanding tagged commands will not be completed. The block layer provides a function that can help with the recovery effort in such situations:

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

This function returns all outstanding tags to the pool and puts the associated requests back into the request queue. The queue lock must be held when you call this function.

Quick Reference

```
#include <linux/fs.h>
int register_blkdev(unsigned int major, const char *name);
int unregister_blkdev(unsigned int major, const char *name);
```
register_blkdev registers a block driver with the kernel and, optionally, obtains a major number. A driver can be unregistered with *unregister_blkdev*.

```
struct block_device_operations
```
Structure that holds most of the methods for block drivers.

```
#include <linux/genhd.h>
struct gendisk;
```
Structure that describes a single block device within the kernel.

```
struct gendisk *alloc_disk(int minors);
void add_disk(struct gendisk *gd);
```
Functions that allocate `gendisk` structures and return them to the system.

```
void set_capacity(struct gendisk *gd, sector_t sectors);
```
Stores the capacity of the device (in 512-byte sectors) within the `gendisk` structure.

```
void add_disk(struct gendisk *gd);
```
Adds a disk to the kernel. As soon as this function is called, your disk's methods can be invoked by the kernel.

```
int check_disk_change(struct block_device *bdev);
```
A kernel function that checks for a media change in the given disk drive and takes the required cleanup action when such a change is detected.

```
#include <linux/blkdev.h>
request_queue_t blk_init_queue(request_fn_proc *request, spinlock_t *lock);
void blk_cleanup_queue(request_queue_t *);
```
Functions that handle the creation and deletion of block request queues.

```
struct request *elv_next_request(request_queue_t *queue);
void end_request(struct request *req, int success);
```
elv_next_request obtains the next request from a request queue; *end_request* may be used in very simple drivers to mark the completion of (or part of) a request.

```
void blkdev_dequeue_request(struct request *req);
void elv_requeue_request(request_queue_t *queue, struct request *req);
```
Functions that remove a request from a queue and put it back on if necessary.

```
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
```
If you need to prevent further calls to your *request* method, a call to *blk_stop_queue* does the trick. A call to *blk_start_queue* is necessary to cause your *request* method to be invoked again.

```
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);
```

Functions that set various queue parameters that control how requests are created for a particular device; the parameters are described in the section "Queue control functions."

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

Low-level structure representing a portion of a block I/O request.

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

Two macros that yield the size and direction of a transfer described by a `bio` structure.

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

A pseudocontrol structure used to loop through the segments that make up a `bio` structure.

```
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 may be used to create a kernel virtual address for a given segment within a `bio` structure. The mapping must be undone with *__bio_kunmap_atomic*.

```
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);
```

A set of accessor macros that provide access to the "current" segment within a `bio` structure.

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

Call *blk_queue_ordered* if your driver implements barrier requests—as it should. The macro *blk_barrier_rq* returns a nonzero value if the current request is a barrier request.

```
int blk_noretry_request(struct request *req);
```
This macro returns a nonzero value if the given request should not be retried on errors.

```
int end_that_request_first(struct request *req, int success, int count);
void end_that_request_last(struct request *req);
```
Use *end_that_request_first* to indicate completion of a portion of a block I/O request. When that function returns `0`, the request is complete and should be passed to *end_that_request_last*.

```
rq_for_each_bio(bio, request)
```
Another macro-implemented control structure; it steps through each `bio` that makes up a request.

```
int blk_rq_map_sg(request_queue_t *queue, struct request *req, struct
  scatterlist *list);
```
Fills the given scatterlist with the information needed to map the buffers in the given request for a DMA transfer.

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```
The prototype for the *make_request* function.

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```
Signal completion for a given `bio`. This function should be used only if your driver obtained the `bio` directly from the block layer via the *make_request* function.

```
request_queue_t *blk_alloc_queue(int flags);
void blk_queue_make_request(request_queue_t *queue, make_request_fn *func);
```
Use *blk_alloc_queue* to allocate a request queue that is used with a custom *make_request* function. That function should be set with *blk_queue_make_request*.

```
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);
```
The prototype and setup functions for a command preparation function, which can be used to prepare the necessary hardware command before the request is passed to your *request* function.

```
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);
```
Support functions for drivers using tagged command queueing.