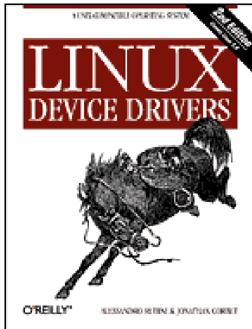

Linux Device Drivers, 2nd Edition

By Alessandro Rubini & Jonathan Corbet
2nd Edition June 2001
0-59600-008-1, Order Number: 0081
586 pages, $39.95

Chapter 12
Loading Block Drivers

Contents:

Registering the Driver
The Header File blk.h
Handling Requests: A Simple Introduction
Handling Requests: The Detailed View
How Mounting and Unmounting Works
The ioctl Method
Removable Devices
Partitionable Devices
Interrupt-Driven Block Drivers
Backward Compatibility
Quick Reference

Our discussion thus far has been limited to char drivers. As we have already mentioned, however, char drivers are not the only type of driver used in Linux systems. Here we turn our attention to block drivers. Block drivers provide access to block-oriented devices -- those that transfer data in randomly accessible, fixed-size blocks. The classic block device is a disk drive, though others exist as well.

The char driver interface is relatively clean and easy to use; the block interface, unfortunately, is a little messier. Kernel developers like to complain about it. There are two reasons for this state of affairs. The first is simple history -- the block interface has been at the core of every version of Linux since the first, and it has proved hard to change. The other reason is performance. A slow char driver is an undesirable thing, but a slow block driver is a drag on the entire system. As a result, the design of the block interface has often been influenced by the need for speed.

The block driver interface has evolved significantly over time. As with the rest of the book, we cover the 2.4 interface in this chapter, with a discussion of the changes at the end. The example drivers work on all kernels between 2.0 and 2.4, however.

This chapter explores the creation of block drivers with two new example drivers. The first, *sbull* (Simple Block Utility for Loading Localities) implements a block device using system memory -- a RAM-disk driver, essentially. Later on, we'll introduce a variant called *spull* as a way of showing how to deal with partition tables.

As always, these example drivers gloss over many of the issues found in real block drivers; their purpose is to demonstrate the interface that such drivers must work with. Real drivers will have to deal with hardware, so the material covered in Chapter 8, "Hardware Management" and Chapter 9, "Interrupt Handling" will be useful as well.

One quick note on terminology: the word *block* as used in this book refers to a block of data as determined by the kernel. The size of blocks can be different in different disks, though they are always a power of two. A *sector*is a fixed-size unit of data as determined by the underlying hardware. Sectors are almost always 512 bytes long.

Registering the Driver

Like char drivers, block drivers in the kernel are identified by major numbers. Block major numbers are entirely distinct from char major numbers, however. A block device with major number 32 can coexist with a char device using the same major number since the two ranges are separate.

The functions for registering and unregistering block devices look similar to those for char devices:

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

The arguments have the same general meaning as for char devices, and major numbers can be assigned dynamically in the same way. So the *sbull* device registers itself in almost exactly the same way as *scull* did:

```
result = register_blkdev(sbull_major, "sbull", &sbull_bdops);
if (result < 0) {
    printk(KERN_WARNING "sbull: can't get major %d\n",sbull_major);
    return result;
}
if (sbull_major == 0) sbull_major = result; /* dynamic */
major = sbull_major; /* Use `major' later on to save typing */
```

The similarity stops here, however. One difference is already evident: *register_chrdev* took a pointer to a `file_operations` structure, but *register_blkdev* uses a structure of type `block_device_operations` instead -- as it has since kernel version 2.3.38. The structure is still sometimes referred to by the name `fops` in block drivers; we'll call it `bdops` to be more faithful to what the structure is and to follow the suggested naming. The definition of this structure is as follows:

```
struct block_device_operations {
    int (*open) (struct inode *inode, struct file *filp);
    int (*release) (struct inode *inode, struct file *filp);
    int (*ioctl) (struct inode *inode, struct file *filp,
                    unsigned command, unsigned long argument);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
};
```

The *open*, *release*, and *ioctl* methods listed here are exactly the same as their char device counterparts.

The other two methods are specific to block devices and are discussed later in this chapter. Note that there is no `owner` field in this structure; block drivers must still maintain their usage count manually, even in the 2.4 kernel.

The `bdops` structure used in *sbull* is as follows:

```
struct block_device_operations sbull_bdops = {
    open:               sbull_open,
    release:            sbull_release,
    ioctl:              sbull_ioctl,
    check_media_change: sbull_check_change,
    revalidate:         sbull_revalidate,
};
```

Note that there are no read or write operations provided in the `block_device_operations` structure. All I/O to block devices is normally buffered by the system (the only exception is with "raw" devices, which we cover in the next chapter); user processes do not perform direct I/O to these devices. User-mode access to block devices usually is implicit in filesystem operations they perform, and those operations clearly benefit from I/O buffering. However, even "direct" I/O to a block device, such as when a filesystem is created, goes through the Linux buffer cache.[47] As a result, the kernel provides a single set of read and write functions for block devices, and drivers do not need to worry about them.

> [47] Actually, the 2.3 development series added the raw I/O capability, allowing user processes to write to block devices without involving the buffer cache. Block drivers, however, are entirely unaware of raw I/O, so we defer the discussion of that facility to the next chapter.

Clearly, a block driver must eventually provide some mechanism for actually doing block I/O to a device. In Linux, the method used for these I/O operations is called *request*; it is the equivalent of the "strategy" function found on many Unix systems. The *request* method handles both read and write operations and can be somewhat complex. We will get into the details of *request* shortly.

For the purposes of block device registration, however, we must tell the kernel where our *request* method is. This method is not kept in the `block_device_operations` structure, for both historical and performance reasons; instead, it is associated with the queue of pending I/O operations for the device. By default, there is one such queue for each major number. A block driver must initialize that queue with *blk_init_queue*. Queue initialization and cleanup is defined as follows:

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

The *init* function sets up the queue, and associates the driver's *request* function (passed as the second parameter) with the queue. It is necessary to call *blk_cleanup_queue* at module cleanup time. The *sbull* driver initializes its queue with this line of code:

```
blk_init_queue(BLK_DEFAULT_QUEUE(major), sbull_request);
```

Each device has a request queue that it uses by default; the macro `BLK_DEFAULT_QUEUE(major)` is used to indicate that queue when needed. This macro looks into a global array of `blk_dev_struct`

structures called blk_dev, which is maintained by the kernel and indexed by major number. The structure looks like this:

```
struct blk_dev_struct {
    request_queue_t      request_queue;
    queue_proc           *queue;
    void                 *data;
};
```

The request_queue member contains the I/O request queue that we have just initialized. We will look at the queue member shortly. The data field may be used by the driver for its own data -- but few drivers do so.

Figure 12-1 visualizes the main steps a driver module performs to register with the kernel proper and deregister. If you compare this figure with Figure 2-1, similarities and differences should be clear.

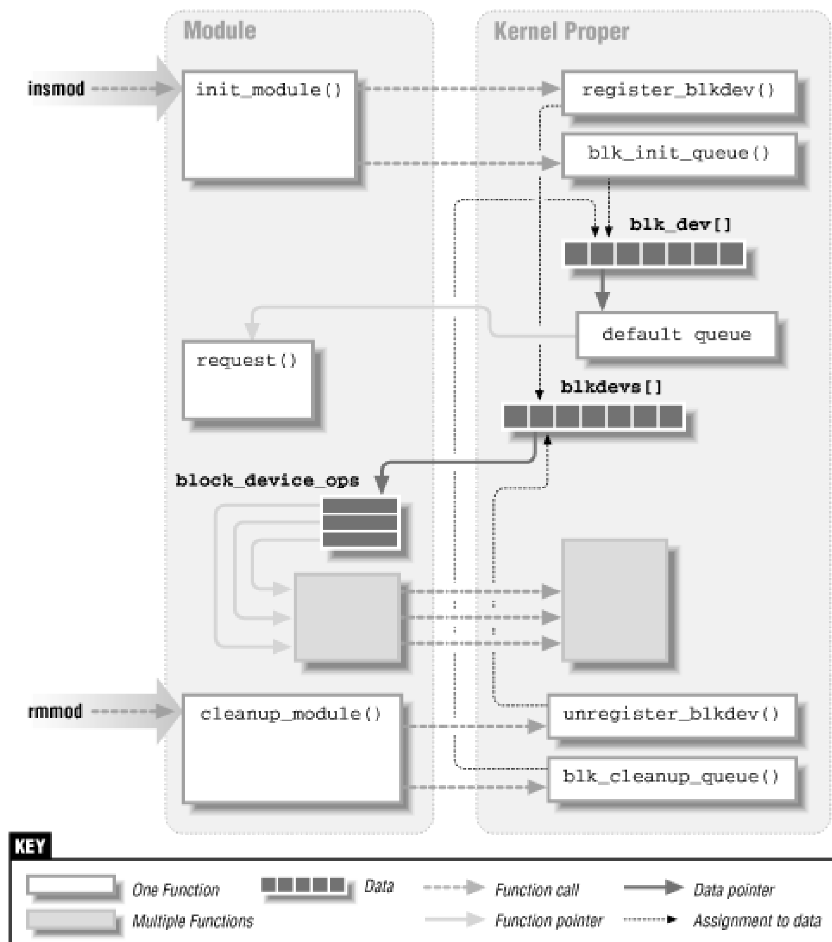

Figure 12-1. Registering a Block Device Driver

In addition to blk_dev, several other global arrays hold information about block drivers. These arrays are indexed by the major number, and sometimes also the minor number. They are declared and described in *drivers/block/ll_rw_block.c*.

int blk_size[][];

This array is indexed by the major and minor numbers. It describes the size of each device, in

kilobytes. If `blk_size[major]` is `NULL`, no checking is performed on the size of the device (i.e., the kernel might request data transfers past end-of-device).

`int blksize_size[][];`

The size of the block used by each device, in bytes. Like the previous one, this bidimensional array is indexed by both major and minor numbers. If `blksize_size[major]` is a null pointer, a block size of `BLOCK_SIZE` (currently 1 KB) is assumed. The block size for the device must be a power of two, because the kernel uses bit-shift operators to convert offsets to block numbers.

`int hardsect_size[][];`

Like the others, this data structure is indexed by the major and minor numbers. The default value for the hardware sector size is 512 bytes. With the 2.2 and 2.4 kernels, different sector sizes are supported, but they must always be a power of two greater than or equal to 512 bytes.

`int read_ahead[];`
`int max_readahead[][];`

These arrays define the number of sectors to be read in advance by the kernel when a file is being read sequentially. `read_ahead` applies to all devices of a given type and is indexed by major number; `max_readahead` applies to individual devices and is indexed by both the major and minor numbers.

Reading data before a process asks for it helps system performance and overall throughput. A slower device should specify a bigger read-ahead value, while fast devices will be happy even with a smaller value. The bigger the read-ahead value, the more memory the buffer cache uses.

The primary difference between the two arrays is this: `read_ahead` is applied at the block I/O level and controls how many blocks may be read sequentially *from the disk* ahead of the current request. `max_readahead` works at the filesystem level and refers to blocks *in the file*, which may not be sequential on disk. Kernel development is moving toward doing read ahead at the filesystem level, rather than at the block I/O level. In the 2.4 kernel, however, read ahead is still done at both levels, so both of these arrays are used.

There is one `read_ahead[]` value for each major number, and it applies to all its minor numbers. `max_readahead`, instead, has a value for every device. The values can be changed via the driver's *ioctl* method; hard-disk drivers usually set `read_ahead` to 8 sectors, which corresponds to 4 KB. The `max_readahead` value, on the other hand, is rarely set by the drivers; it defaults to `MAX_READAHEAD`, currently 31 pages.

`int max_sectors[][];`

This array limits the maximum size of a single request. It should normally be set to the largest transfer that your hardware can handle.

`int max_segments[];`

This array controlled the number of individual segments that could appear in a clustered request; it was removed just before the release of the 2.4 kernel, however. (See "Section 12.4.2, "Clustered Requests"" later in this chapter for information on clustered requests).

The *sbull* device allows you to set these values at load time, and they apply to all the minor numbers of the sample driver. The variable names and their default values in *sbull* are as follows:

`size=2048` (kilobytes)

Each RAM disk created by *sbull* takes two megabytes of RAM.

`blksize=1024` (bytes)

The software "block" used by the module is one kilobyte, like the system default.

`hardsect=512` (bytes)

The *sbull* sector size is the usual half-kilobyte value.

`rahead=2` (sectors)

Because the RAM disk is a fast device, the default read-ahead value is small.

The *sbull* device also allows you to choose the number of devices to install. `devs`, the number of devices, defaults to 2, resulting in a default memory usage of four megabytes -- two disks at two megabytes each.

The initialization of these arrays in *sbull*is done as follows:

```
read_ahead[major] = sbull_rahead;
result = -ENOMEM; /* for the possible errors */

sbull_sizes = kmalloc(sbull_devs * sizeof(int), GFP_KERNEL);
if (!sbull_sizes)
    goto fail_malloc;
for (i=0; i < sbull_devs; i++) /* all the same size */
    sbull_sizes[i] = sbull_size;
blk_size[major]=sbull_sizes;

sbull_blksizes = kmalloc(sbull_devs * sizeof(int), GFP_KERNEL);
if (!sbull_blksizes)
    goto fail_malloc;
for (i=0; i < sbull_devs; i++) /* all the same blocksize */
    sbull_blksizes[i] = sbull_blksize;
blksize_size[major]=sbull_blksizes;

sbull_hardsects = kmalloc(sbull_devs * sizeof(int), GFP_KERNEL);
if (!sbull_hardsects)
    goto fail_malloc;
for (i=0; i < sbull_devs; i++) /* all the same hardsect */
    sbull_hardsects[i] = sbull_hardsect;
hardsect_size[major]=sbull_hardsects;
```

For brevity, the error handling code (the target of the `fail_malloc goto`) has been omitted; it simply frees anything that was successfully allocated, unregisters the device, and returns a failure status.

One last thing that must be done is to register every "disk" device provided by the driver. *sbull* calls the necessary function (*register_disk*) as follows:

```
for (i = 0; i < sbull_devs; i++)
        register_disk(NULL, MKDEV(major, i), 1, &sbull_bdops,
                        sbull_size << 1);
```

In the 2.4.0 kernel, *register_disk* does nothing when invoked in this manner. The real purpose of *register_disk* is to set up the partition table, which is not supported by *sbull*. All block drivers, however, make this call whether or not they support partitions, indicating that it may become necessary for all block devices in the future. A block driver without partitions will work without this call in 2.4.0, but it is safer to include it. We revisit *register_disk* in detail later in this chapter, when we cover partitions.

The cleanup function used by *sbull* looks like this:

```
for (i=0; i<sbull_devs; i++)
    fsync_dev(MKDEV(sbull_major, i)); /* flush the devices */
unregister_blkdev(major, "sbull");
/*
 * Fix up the request queue(s)
 */
blk_cleanup_queue(BLK_DEFAULT_QUEUE(major));

/* Clean up the global arrays */
read_ahead[major] = 0;
kfree(blk_size[major]);
blk_size[major] = NULL;
kfree(blksize_size[major]);
blksize_size[major] = NULL;
kfree(hardsect_size[major]);
hardsect_size[major] = NULL;
```

Here, the call to *fsync_dev* is needed to free all references to the device that the kernel keeps in various caches. *fsync_dev* is the implementation of *block_fsync*, which is the *fsync* "method" for block devices.

The Header File blk.h

All block drivers should include the header file `<linux/blk.h>`. This file defines much of the common code that is used in block drivers, and it provides functions for dealing with the I/O request queue.

Actually, the *blk.h* header is quite unusual, because it defines several symbols based on the symbol `MAJOR_NR`, which must be declared by the driver *before* it includes the header. This convention was developed in the early days of Linux, when all block devices had preassigned major numbers and modular block drivers were not supported.

If you look at *blk.h*, you'll see that several device-dependent symbols are declared according to the value of `MAJOR_NR`, which is expected to be known in advance. However, if the major number is dynamically assigned, the driver has no way to know its assigned number at compile time and cannot correctly define `MAJOR_NR`. If `MAJOR_NR` is undefined, *blk.h*can't set up some of the macros used with the

request queue. Fortunately, `MAJOR_NR` can be defined as an integer variable and all will work fine for add-on block drivers.

blk.h makes use of some other predefined, driver-specific symbols as well. The following list describes the symbols in `<linux/blk.h>` that must be defined in advance; at the end of the list, the code used in *sbull* is shown.

MAJOR_NR

> This symbol is used to access a few arrays, in particular `blk_dev` and `blksize_size`. A custom driver like *sbull*, which is unable to assign a constant value to the symbol, should `#define` it to the variable holding the major number. For *sbull*, this is `sbull_major`.

DEVICE_NAME

> The name of the device being created. This string is used in printing error messages.

DEVICE_NR(kdev_t device)

> This symbol is used to extract the ordinal number of the physical device from the `kdev_t` device number. This symbol is used in turn to declare `CURRENT_DEV`, which can be used within the *request* function to determine which hardware device owns the minor number involved in a transfer request.
>
> The value of this macro can be `MINOR(device)` or another expression, according to the convention used to assign minor numbers to devices and partitions. The macro should return the same device number for all partitions on the same physical device -- that is, `DEVICE_NR` represents the disk number, not the partition number. Partitionable devices are introduced later in this chapter.

DEVICE_INTR

> This symbol is used to declare a pointer variable that refers to the current bottom-half handler. The macros `SET_INTR(intr)` and `CLEAR_INTR` are used to assign the variable. Using multiple handlers is convenient when the device can issue interrupts with different meanings.

DEVICE_ON(kdev_t device)
DEVICE_OFF(kdev_t device)

> These macros are intended to help devices that need to perform processing before or after a set of transfers is performed; for example, they could be used by a floppy driver to start the drive motor before I/O and to stop it afterward. Modern drivers no longer use these macros, and `DEVICE_ON` does not even get called anymore. Portable drivers, though, should define them (as empty symbols), or compilation errors will result on 2.0 and 2.2 kernels.

DEVICE_NO_RANDOM

> By default, the function *end_request* contributes to system entropy (the amount of collected "randomness"), which is used by */dev/random*. If the device isn't able to contribute significant entropy to the random device, `DEVICE_NO_RANDOM` should be defined. */dev/random* was introduced in "Section 9.3, "Installing an Interrupt Handler"" in Chapter 9, "Interrupt Handling", where `SA_SAMPLE_RANDOM` was explained.

DEVICE_REQUEST

Used to specify the name of the *request* function used by the driver. The only effect of defining `DEVICE_REQUEST` is to cause a forward declaration of the *request* function to be done; it is a holdover from older times, and most (or all) drivers can leave it out.

The *sbull* driver declares the symbols in the following way:

```
#define MAJOR_NR sbull_major /* force definitions on in blk.h */
static int sbull_major; /* must be declared before including blk.h */

#define DEVICE_NR(device) MINOR(device)  /* has no partition bits */
#define DEVICE_NAME "sbull"              /* name for messaging */
#define DEVICE_INTR sbull_intrptr        /* pointer to bottom half */
#define DEVICE_NO_RANDOM                 /* no entropy to contribute */
#define DEVICE_REQUEST sbull_request
#define DEVICE_OFF(d) /* do-nothing */

#include <linux/blk.h>

#include "sbull.h"        /* local definitions */
```

The *blk.h* header uses the macros just listed to define some additional macros usable by the driver. We'll describe those macros in the following sections.

Handling Requests: A Simple Introduction

The most important function in a block driver is the *request* function, which performs the low-level operations related to reading and writing data. This section discusses the basic design of the *request*procedure.

The Request Queue

When the kernel schedules a data transfer, it queues the request in a list, ordered in such a way that it maximizes system performance. The queue of requests is then passed to the driver's *request* function, which has the following prototype:

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

The *request* function should perform the following tasks for each request in the queue:

1. Check the validity of the request. This test is performed by the macro `INIT_REQUEST`, defined in *blk.h*; the test consists of looking for problems that could indicate a bug in the system's request queue handling.

2. Perform the actual data transfer. The `CURRENT` variable (a macro, actually) can be used to retrieve the details of the current request. `CURRENT` is a pointer to `struct request`, whose fields are described in the next section.

3. Clean up the request just processed. This operation is performed by *end_request*, a static function whose code resides in *blk.h*. *end_request*handles the management of the request queue and wakes

up processes waiting on the I/O operation. It also manages the `CURRENT` variable, ensuring that it points to the next unsatisfied request. The driver passes the function a single argument, which is 1 in case of success and 0 in case of failure. When *end_request* is called with an argument of 0, an "I/O error" message is delivered to the system logs (via *printk*).

4. Loop back to the beginning, to consume the next request.

Based on the previous description, a minimal *request* function, which does not actually transfer any data, would look like this:

```
void sbull_request(request_queue_t *q)
{
    while(1) {
        INIT_REQUEST;
        printk("<1>request %p: cmd %i sec %li (nr. %li)\n", CURRENT,
               CURRENT->cmd,
               CURRENT->sector,
               CURRENT->current_nr_sectors);
        end_request(1); /* success */
    }
}
```

Although this code does nothing but print messages, running this function provides good insight into the basic design of data transfer. It also demonstrates a couple of features of the macros defined in `<linux/blk.h>`. The first is that, although the `while` loop looks like it will never terminate, the fact is that the `INIT_REQUEST` macro performs a `return` when the request queue is empty. The loop thus iterates over the queue of outstanding requests and then returns from the *request* function. Second, the `CURRENT` macro always describes the request to be processed. We get into the details of `CURRENT` in the next section.

A block driver using the *request* function just shown will actually work -- for a short while. It is possible to make a filesystem on the device and access it for as long as the data remains in the system's buffer cache.

This empty (but verbose) function can still be run in *sbull* by defining the symbol `SBULL_EMPTY_REQUEST` at compile time. If you want to understand how the kernel handles different block sizes, you can experiment with `blksize=` on the *insmod* command line. The empty *request* function shows the internal workings of the kernel by printing the details of each request.

The *request* function has one very important constraint: it must be atomic. *request* is not usually called in direct response to user requests, and it is not running in the context of any particular process. It can be called at interrupt time, from tasklets, or from any number of other places. Thus, it must not sleep while carrying out its tasks.

Performing the Actual Data Transfer

To understand how to build a working *request*function for *sbull*, let's look at how the kernel describes a request within a `struct request`. The structure is defined in `<linux/blkdev.h>`. By accessing the fields in the `request` structure, usually by way of `CURRENT`, the driver can retrieve all the information needed to transfer data between the buffer cache and the physical block device.[48] `CURRENT` is just a

pointer into `blk_dev[MAJOR_NR].request_queue`. The following fields of a request hold information that is useful to the *request* function:

[48]Actually, not all blocks passed to a block driver need be in the buffer cache, but that's a topic beyond the scope of this chapter.

`kdev_t rq_dev;`

The device accessed by the request. By default, the same *request* function is used for every device managed by the driver. A single *request* function deals with all the minor numbers; `rq_dev` can be used to extract the minor device being acted upon. The `CURRENT_DEV` macro is simply defined as `DEVICE_NR(CURRENT->rq_dev)`.

`int cmd;`

This field describes the operation to be performed; it is either `READ` (from the device) or `WRITE` (to the device).

`unsigned long sector;`

The number of the first sector to be transferred in this request.

`unsigned long current_nr_sectors;`
`unsigned long nr_sectors;`

The number of sectors to transfer for the current request. The driver should refer to `current_nr_sectors` and ignore `nr_sectors` (which is listed here just for completeness). See "Section 12.4.2, "Clustered Requests"" later in this chapter for more detail on `nr_sectors`.

`char *buffer;`

The area in the buffer cache to which data should be written (`cmd==READ`) or from which data should be read (`cmd==WRITE`).

`struct buffer_head *bh;`

The structure describing the first buffer in the list for this request. Buffer heads are used in the management of the buffer cache; we'll look at them in detail shortly in "Section 12.4.1.1, "The request structure and the buffer cache"."

There are other fields in the structure, but they are primarily meant for internal use in the kernel; the driver is not expected to use them.

The implementation for the working *request*function in the *sbull* device is shown here. In the following code, the `Sbull_Dev` serves the same function as `Scull_Dev`, introduced in "Section 3.6, "scull's Memory Usage"" in Chapter 3, "Char Drivers".

```
void sbull_request(request_queue_t *q)
{
    Sbull_Dev *device;
    int status;
```

```
    while(1) {
        INIT_REQUEST;  /* returns when queue is empty */

        /* Which "device" are we using? */
        device = sbull_locate_device (CURRENT);
        if (device == NULL) {
            end_request(0);
            continue;
        }

        /* Perform the transfer and clean up. */
        spin_lock(&device->lock);
        status = sbull_transfer(device, CURRENT);
        spin_unlock(&device->lock);
        end_request(status);
    }
}
```

This code looks little different from the empty version shown earlier; it concerns itself with request queue management and pushes off the real work to other functions. The first, *sbull_locate_device*, looks at the device number in the request and finds the right `Sbull_Dev` structure:

```
static Sbull_Dev *sbull_locate_device(const struct request *req)
{
    int devno;
    Sbull_Dev *device;

    /* Check if the minor number is in range */
    devno = DEVICE_NR(req->rq_dev);
    if (devno >= sbull_devs) {
        static int count = 0;
        if (count++ < 5) /* print the message at most five times */
            printk(KERN_WARNING "sbull: request for unknown device\n");
        return NULL;
    }
    device = sbull_devices + devno; /* Pick it out of device array */
    return device;
}
```

The only "strange" feature of the function is the conditional statement that limits it to reporting five errors. This is intended to avoid clobbering the system logs with too many messages, since `end_request(0)` already prints an "I/O error" message when the request fails. The `static` counter is a standard way to limit message reporting and is used several times in the kernel.

The actual I/O of the request is handled by *sbull_transfer*:

```
static int sbull_transfer(Sbull_Dev *device, const struct request *req)
{
    int size;
    u8 *ptr;
```

```
        ptr = device->data + req->sector * sbull_hardsect;
        size = req->current_nr_sectors * sbull_hardsect;

        /* Make sure that the transfer fits within the device. */
        if (ptr + size > device->data + sbull_blksize*sbull_size) {
            static int count = 0;
            if (count++ < 5)
                printk(KERN_WARNING "sbull: request past end of device\n");
            return 0;
        }

        /* Looks good, do the transfer. */
        switch(req->cmd) {
            case READ:
                memcpy(req->buffer, ptr, size); /* from sbull to buffer */
                return 1;
            case WRITE:
                memcpy(ptr, req->buffer, size); /* from buffer to sbull */
                return 1;
            default:
                /* can't happen */
                return 0;
        }
    }
```

Since *sbull* is just a RAM disk, its "data transfer" reduces to a *memcpy* call.

Handling Requests: The Detailed View

The *sbull* driver as described earlier works very well. In simple situations (as with *sbull*), the macros from `<linux/blk.h>` can be used to easily set up a *request* function and get a working driver. As has already been mentioned, however, block drivers are often a performance-critical part of the kernel. Drivers based on the simple code shown earlier will likely not perform very well in many situations, and can also be a drag on the system as a whole. In this section we get into the details of how the I/O request queue works with an eye toward writing a faster, more efficient driver.

The I/O Request Queue

Each block driver works with at least one I/O request queue. This queue contains, at any given time, all of the I/O operations that the kernel would like to see done on the driver's devices. The management of this queue is complicated; the performance of the system depends on how it is done.

The queue is designed with physical disk drives in mind. With disks, the amount of time required to transfer a block of data is typically quite small. The amount of time required to position the head (*seek*) to do that transfer, however, can be very large. Thus the Linux kernel works to minimize the number and extent of the seeks performed by the device.

Two things are done to achieve those goals. One is the clustering of requests to adjacent sectors on the disk. Most modern filesystems will attempt to lay out files in consecutive sectors; as a result, requests to adjoining parts of the disk are common. The kernel also applies an "elevator" algorithm to the requests. An elevator in a skyscraper is either going up or down; it will continue to move in those directions until

all of its "requests" (people wanting on or off) have been satisfied. In the same way, the kernel tries to keep the disk head moving in the same direction for as long as possible; this approach tends to minimize seek times while ensuring that all requests get satisfied eventually.

A Linux I/O request queue is represented by a structure of type `request_queue`, declared in `<linux/blkdev.h>`. The `request_queue` structure looks somewhat like `file_operations` and other such objects, in that it contains pointers to a number of functions that operate on the queue -- for example, the driver's *request*function is stored there. There is also a queue head (using the functions from `<linux/list.h>` described in "Section 10.5, "Linked Lists"" in Chapter 10, "Judicious Use of Data Types"), which points to the list of outstanding requests to the device.

These requests are, of course, of type `struct request`; we have already looked at some of the fields in this structure. The reality of the `request` structure is a little more complicated, however; understanding it requires a brief digression into the structure of the Linux buffer cache.

The request structure and the buffer cache

The design of the `request` structure is driven by the Linux memory management scheme. Like most Unix-like systems, Linux maintains a *buffer cache*, a region of memory that is used to hold copies of blocks stored on disk. A great many "disk" operations performed at higher levels of the kernel -- such as in the filesystem code -- act only on the buffer cache and do not generate any actual I/O operations. Through aggressive caching the kernel can avoid many read operations altogether, and multiple writes can often be merged into a single physical write to disk.

One unavoidable aspect of the buffer cache, however, is that blocks that are adjacent on disk are almost certainly *not* adjacent in memory. The buffer cache is a dynamic thing, and blocks end up being scattered widely. In order to keep track of everything, the kernel manages the buffer cache through `buffer_head` structures. One `buffer_head` is associated with each data buffer. This structure contains a great many fields, most of which do not concern a driver writer. There are a few that are important, however, including the following:

`char *b_data;`

> The actual data block associated with this buffer head.

`unsigned long b_size;`

> The size of the block pointed to by `b_data`.

`kdev_t b_rdev;`

> The device holding the block represented by this buffer head.

`unsigned long b_rsector;`

> The sector number where this block lives on disk.

`struct buffer_head *b_reqnext;`

> A pointer to a linked list of buffer head structures in the request queue.

`void (*b_end_io)(struct buffer_head *bh, int uptodate);`

A pointer to a function to be called when I/O on this buffer completes. `bh` is the buffer head itself, and `uptodate` is nonzero if the I/O was successful.

Every block passed to a driver's *request* function either lives in the buffer cache, or, on rare occasion, lives elsewhere but has been made to look as if it lived in the buffer cache.[49] As a result, every request passed to the driver deals with one or more `buffer_head` structures. The `request` structure contains a member (called simply `bh`) that points to a linked list of these structures; satisfying the request requires performing the indicated I/O operation on each buffer in the list. Figure 12-2 shows how the request queue and `buffer_head` structures fit together.

[49]The RAM-disk driver, for example, makes its memory look as if it were in the buffer cache. Since the "disk" buffer is already in system RAM, there's no need to keep a copy in the buffer cache. Our sample code is thus much less efficient than a properly implemented RAM disk, not being concerned with RAM-disk-specific performance issues.

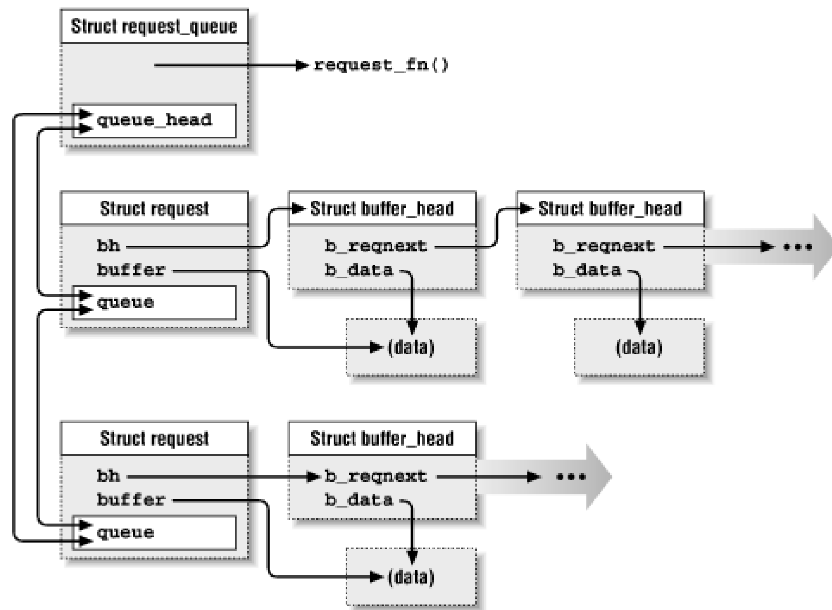

Figure 12-2. Buffers in the I/O Request Queue

Requests are not made of random lists of buffers; instead, all of the buffer heads attached to a single request will belong to a series of adjacent blocks on the disk. Thus a request is, in a sense, a single operation referring to a (perhaps long) group of blocks on the disk. This grouping of blocks is called *clustering*, and we will look at it in detail after completing our discussion of how the request list works.

Request queue manipulation

The header `<linux/blkdev.h>` defines a small number of functions that manipulate the request queue, most of which are implemented as preprocessor macros. Not all drivers will need to work with the queue at this level, but a familiarity with how it all works can be helpful. Most request queue functions will be introduced as we need them, but a few are worth mentioning here.

`struct request *blkdev_entry_next_request(struct list_head *head);`

Returns the next entry in the request list. Usually the `head` argument is the `queue_head` member of the `request_queue` structure; in this case the function returns the first entry in the queue. The

function uses the *list_entry* macro to look in the list.

struct request *blkdev_next_request(struct request *req);
struct request *blkdev_prev_request(struct request *req);

Given a request structure, return the next or previous structure in the request queue.

blkdev_dequeue_request(struct request *req);

Removes a request from its request queue.

blkdev_release_request(struct request *req);

Releases a request structure back to the kernel when it has been completely executed. Each request queue maintains its own free list of request structures (two, actually: one for reads and one for writes); this function places a structure back on the proper free list. *blkdev_release_request* will also wake up any processes that are waiting on a free request structure.

All of these functions require that the `io_request_lock` be held, which we will discuss next.

The I/O request lock

The I/O request queue is a complex data structure that is accessed in many places in the kernel. It is entirely possible that the kernel needs to add more requests to the queue at the same time that your driver is taking requests off. The queue is thus subject to the usual sort of race conditions, and must be protected accordingly.

In Linux 2.2 and 2.4, all request queues are protected with a single global spinlock called `io_request_lock`. Any code that manipulates a request queue must hold that lock *and* disable interrupts, with one small exception: the very first entry in the request queue is (by default) considered to be owned by the driver. Failure to acquire the `io_request_lock` prior to working with the request queue can cause the queue to be corrupted, with a system crash following shortly thereafter.

The simple *request* function shown earlier did not need to worry about this lock because the kernel always calls the *request* function with the `io_request_lock` held. A driver is thus protected against corrupting the request queue; it is also protected against reentrant calls to the *request* function. This scheme was designed to enable drivers that are not SMP aware to function on multiprocessor systems.

Note, however, that the `io_request_lock` is an expensive resource to hold. As long as your driver holds this lock, no other requests may be queued to any block driver in the system, and no other *request* functions may be called. A driver that holds this lock for a long time may well slow down the system as a whole.

Thus, well-written block drivers often drop this lock as soon as possible. We will see an example of how this can be done shortly. Block drivers that drop the `io_request_lock` must be written with a couple of important things in mind, however. First is that the *request* function must always reacquire this lock before returning, since the calling code expects it to still be held. The other concern is that, as soon as the `io_request_lock` is dropped, the possibility of reentrant calls to the *request* function is very real; the function must be written to handle that eventuality.

A variant of this latter case can also occur if your *request* function returns while an I/O request is still active. Many drivers for real hardware will start an I/O operation, then return; the work is completed in

the driver's interrupt handler. We will look at interrupt-driven block I/O in detail later in this chapter; for now it is worth mentioning, however, that the *request* function can be called while these operations are still in progress.

Some drivers handle *request* function reentrancy by maintaining an internal request queue. The *request* function simply removes any new requests from the I/O request queue and adds them to the internal queue, which is then processed through a combination of tasklets and interrupt handlers.

How the blk.h macros and functions work

In our simple *request* function earlier, we were not concerned with `buffer_head` structures or linked lists. The macros and functions in `<linux/blk.h>` hide the structure of the I/O request queue in order to make the task of writing a block driver simpler. In many cases, however, getting reasonable performance requires a deeper understanding of how the queue works. In this section we look at the actual steps involved in manipulating the request queue; subsequent sections show some more advanced techniques for writing block *request* functions.

The fields of the `request` structure that we looked at earlier -- `sector`, `current_nr_sectors`, and `buffer` -- are really just copies of the analogous information stored in the first `buffer_head` structure on the list. Thus, a *request* function that uses this information from the `CURRENT` pointer is just processing the first of what might be many buffers within the request. The task of splitting up a multibuffer request into (seemingly) independent, single-buffer requests is handled by two important definitions in `<linux/blk.h>`: the `INIT_REQUEST` macro and the *end_request* function.

Of the two, `INIT_REQUEST` is the simpler; all it really does is make a couple of consistency checks on the request queue and cause a return from the *request*function if the queue is empty. It is simply making sure that there is still work to do.

The bulk of the queue management work is done by *end_request*. This function, remember, is called when the driver has processed a single "request" (actually one buffer); it has several tasks to perform:

1. Complete the I/O processing on the current buffer; this involves calling the *b_end_io* function with the status of the operation, thus waking any process that may be sleeping on the buffer.

2. Remove the buffer from the request's linked list. If there are further buffers to be processed, the `sector`, `current_nr_sectors`, and `buffer` fields in the request structure are updated to reflect the contents of the next `buffer_head` structure in the list. In this case (there are still buffers to be transferred), *end_request* is finished for this iteration and steps 3 to 5 are not executed.

3. Call *add_blkdev_randomness* to update the entropy pool, unless `DEVICE_NO_RANDOM` has been defined (as is done in the *sbull* driver).

4. Remove the finished request from the request queue by calling *blkdev_dequeue_request*. This step modifies the request queue, and thus must be performed with the `io_request_lock` held.

5. Release the finished request back to the system; `io_request_lock` is required here too.

The kernel defines a couple of helper functions that are used by *end_request* to do most of this work. The first one is called *end_that_request_first*, which handles the first two steps just described. Its prototype is

```
int end_that_request_first(struct request *req, int status, char *name);
```

`status` is the status of the request as passed to *end_request*; the `name` parameter is the device name, to be used when printing error messages. The return value is nonzero if there are more buffers to be processed in the current request; in that case the work is done. Otherwise, the request is dequeued and released with *end_that_request_last*:

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

In *end_request* this step is handled with this code:

```
struct request *req = CURRENT;
blkdev_dequeue_request(req);
end_that_request_last(req);
```

That is all there is to it.

Clustered Requests

The time has come to look at how to apply all of that background material to the task of writing better block drivers. We'll start with a look at the handling of clustered requests. Clustering, as mentioned earlier, is simply the practice of joining together requests that operate on adjacent blocks on the disk. There are two advantages to doing things this way. First, clustering speeds up the transfer; clustering can also save some memory in the kernel by avoiding allocation of redundant `request` structures.

As we have seen, block drivers need not be aware of clustering at all; `<linux/blk.h>` transparently splits each clustered request into its component pieces. In many cases, however, a driver can do better by explicitly acting on clustering. It is often possible to set up the I/O for several consecutive blocks at the same time, with an improvement in throughput. For example, the Linux floppy driver attempts to write an entire track to the diskette in a single operation. Most high-performance disk controllers can do "scatter/gather" I/O as well, leading to large performance gains.

To take advantage of clustering, a block driver must look directly at the list of `buffer_head` structures attached to the request. This list is pointed to by `CURRENT->bh`; subsequent buffers can be found by following the `b_reqnext` pointers in each `buffer_head` structure. A driver performing clustered I/O should follow roughly this sequence of operations with each buffer in the cluster:

1. Arrange to transfer the data block at address `bh->b_data`, of size `bh->b_size` bytes. The direction of the data transfer is `CURRENT->cmd` (i.e., either `READ` or `WRITE`).

2. Retrieve the next buffer head in the list: `bh->b_reqnext`. Then detach the buffer just transferred from the list, by zeroing its `b_reqnext` -- the pointer to the new buffer you just retrieved.

3. Update the `request` structure to reflect the I/O done with the buffer that has just been removed. Both `CURRENT->hard_nr_sectors` and `CURRENT->nr_sectors` should be decremented by the number of sectors (not blocks) transferred from the buffer. The sector numbers `CURRENT->hard_sector` and `CURRENT->sector` should be incremented by the same amount. Performing these operations keeps the `request` structure consistent.

4. Loop back to the beginning to transfer the next adjacent block.

When the I/O on each buffer completes, your driver should notify the kernel by calling the buffer's I/O completion routine:

```
bh->b_end_io(bh, status);
```

`status` is nonzero if the operation was successful. You also, of course, need to remove the `request` structure for the completed operations from the queue. The processing steps just described can be done without holding the `io_request_lock`, but that lock must be reacquired before changing the queue itself.

Your driver can still use *end_request* (as opposed to manipulating the queue directly) at the completion of the I/O operation, as long as it takes care to set the `CURRENT->bh` pointer properly. This pointer should either be `NULL` or it should point to the last `buffer_head` structure that was transferred. In the latter case, the *b_end_io* function should *not* have been called on that last buffer, since *end_request* will make that call.

A full-featured implementation of clustering appears in *drivers/block/floppy.c*, while a summary of the operations required appears in *end_request*, in *blk.h*. Neither *floppy.c* nor *blk.h* are easy to understand, but the latter is a better place to start.

The active queue head

One other detail regarding the behavior of the I/O request queue is relevant for block drivers that are dealing with clustering. It has to do with the queue head -- the first request on the queue. For historical compatibility reasons, the kernel (almost) always assumes that a block driver is processing the first entry in the request queue. To avoid corruption resulting from conflicting activity, the kernel will never modify a request once it gets to the head of the queue. No further clustering will happen on that request, and the elevator code will not put other requests in front of it.

Many block drivers remove requests from the queue entirely before beginning to process them. If your driver works this way, the request at the head of the queue should be fair game for the kernel. In this case, your driver should inform the kernel that the head of the queue is not active by calling *blk_queue_headactive*:

```
blk_queue_headactive(request_queue_t *queue, int active);
```

If `active` is 0, the kernel will be able to make changes to the head of the request queue.

Multiqueue Block Drivers

As we have seen, the kernel, by default, maintains a single I/O request queue for each major number. The single queue works well for devices like *sbull*, but it is not always optimal for real-world situations.

Consider a driver that is handling real disk devices. Each disk is capable of operating independently; the performance of the system is sure to be better if the drives could be kept busy in parallel. A simple driver based on a single queue will not achieve that -- it will perform operations on a single device at a time.

It would not be all that hard for a driver to walk through the request queue and pick out requests for independent drives. But the 2.4 kernel makes life easier by allowing the driver to set up independent queues for each device. Most high-performance drivers take advantage of this multiqueue capability.

Doing so is not difficult, but it does require moving beyond the simple `<linux/blk.h>` definitions.

The *sbull* driver, when compiled with the `SBULL_MULTIQUEUE` symbol defined, operates in a multiqueue mode. It works without the `<linux/blk.h>` macros, and demonstrates a number of the features that have been described in this section.

To operate in a multiqueue mode, a block driver must define its own request queues. *sbull* does this by adding a `queue` member to the `Sbull_Dev` structure:

```
request_queue_t queue;
int busy;
```

The `busy` flag is used to protect against *request* function reentrancy, as we will see.

Request queues must be initialized, of course. *sbull* initializes its device-specific queues in this manner:

```
for (i = 0; i < sbull_devs; i++) {
    blk_init_queue(&sbull_devices[i].queue, sbull_request);
    blk_queue_headactive(&sbull_devices[i].queue, 0);
}
blk_dev[major].queue = sbull_find_queue;
```

The call to *blk_init_queue* is as we have seen before, only now we pass in the device-specific queues instead of the default queue for our major device number. This code also marks the queues as not having active heads.

You might be wondering how the kernel manages to find the request queues, which are buried in a device-specific, private structure. The key is the last line just shown, which sets the `queue` member in the global `blk_dev` structure. This member points to a function that has the job of finding the proper request queue for a given device number. Devices using the default queue have no such function, but multiqueue devices must implement it. *sbull*'s queue function looks like this:

```
request_queue_t *sbull_find_queue(kdev_t device)
{
    int devno = DEVICE_NR(device);

    if (devno >= sbull_devs) {
        static int count = 0;
        if (count++ < 5) /* print the message at most five times */
            printk(KERN_WARNING "sbull: request for unknown device\n");
        return NULL;
    }
    return &sbull_devices[devno].queue;
}
```

Like the *request* function, *sbull_find_queue* must be atomic (no sleeping allowed).

Each queue has its own *request* function, though usually a driver will use the same function for all of its queues. The kernel passes the actual request queue into the *request* function as a parameter, so the function can always figure out which device is being operated on. The multiqueue *request* function used

in *sbull* looks a little different from the ones we have seen so far because it manipulates the request queue directly. It also drops the `io_request_lock` while performing transfers to allow the kernel to execute other block operations. Finally, the code must take care to avoid two separate perils: multiple calls of the *request* function and conflicting access to the device itself.

```
void sbull_request(request_queue_t *q)
{
    Sbull_Dev *device;
    struct request *req;
    int status;

    /* Find our device */
    device = sbull_locate_device (blkdev_entry_next_request(&q->queue_head));
    if (device->busy) /* no race here - io_request_lock held */
        return;
    device->busy = 1;

    /* Process requests in the queue */
    while(! list_empty(&q->queue_head)) {

    /* Pull the next request off the list. */
        req = blkdev_entry_next_request(&q->queue_head);
        blkdev_dequeue_request(req);
        spin_unlock_irq (&io_request_lock);
        spin_lock(&device->lock);

    /* Process all of the buffers in this (possibly clustered) request. */
        do {
            status = sbull_transfer(device, req);
        } while (end_that_request_first(req, status, DEVICE_NAME));
        spin_unlock(&device->lock);
        spin_lock_irq (&io_request_lock);
        end_that_request_last(req);
    }
    device->busy = 0;
}
```

Instead of using `INIT_REQUEST`, this function tests its specific request queue with the list function *list_empty*. As long as requests exist, it removes each one in turn from the queue with *blkdev_dequeue_request*. Only then, once the removal is complete, is it able to drop `io_request_lock` and obtain the device-specific lock. The actual transfer is done using *sbull_transfer*, which we have already seen.

Each call to *sbull_transfer* handles exactly one `buffer_head` structure attached to the request. The function then calls *end_that_request_first* to dispose of that buffer, and, if the request is complete, goes on to *end_that_request_last* to clean up the request as a whole.

The management of concurrency here is worth a quick look. The `busy` flag is used to prevent multiple invocations of *sbull_request*. Since *sbull_request* is always called with the `io_request_lock` held, it is safe to test and set the `busy` flag with no additional protection. (Otherwise, an `atomic_t` could have been used). The `io_request_lock` is dropped before the device-specific lock is acquired. It is

possible to acquire multiple locks without risking deadlock, but it is harder; when the constraints allow, it is better to release one lock before obtaining another.

end_that_request_first is called without the `io_request_lock` held. Since this function operates only on the given request structure, calling it this way is safe -- as long as the request is not on the queue. The call to *end_that_request_last*, however, requires that the lock be held, since it returns the request to the request queue's free list. The function also always exits from the outer loop (and the function as a whole) with the `io_request_lock` held and the device lock released.

Multiqueue drivers must, of course, clean up all of their queues at module removal time:

```
for (i = 0; i < sbull_devs; i++)
        blk_cleanup_queue(&sbull_devices[i].queue);
blk_dev[major].queue = NULL;
```

It is worth noting, briefly, that this code could be made more efficient. It allocates a whole set of request queues at initialization time, even though some of them may never be used. A request queue is a large structure, since many (perhaps thousands) of `request` structures are allocated when the queue is initialized. A more clever implementation would allocate a request queue when needed in either the *open* method or the *queue* function. We chose a simpler implementation for *sbull* in order to avoid complicating the code.

That covers the mechanics of multiqueue drivers. Drivers handling real hardware may have other issues to deal with, of course, such as serializing access to a controller. But the basic structure of multiqueue drivers is as we have seen here.

Doing Without the Request Queue

Much of the discussion to this point has centered around the manipulation of the I/O request queue. The purpose of the request queue is to improve performance by allowing the driver to act asynchronously and, crucially, by allowing the merging of contiguous (on the disk) operations. For normal disk devices, operations on contiguous blocks are common, and this optimization is necessary.

Not all block devices benefit from the request queue, however. *sbull*, for example, processes requests synchronously and has no problems with seek times. For *sbull*, the request queue actually ends up slowing things down. Other types of block devices also can be better off without a request queue. For example, RAID devices, which are made up of multiple disks, often spread "contiguous" blocks across multiple physical devices. Block devices implemented by the logical volume manager (LVM) capability (which first appeared in 2.4) also have an implementation that is more complex than the block interface that is presented to the rest of the kernel.

In the 2.4 kernel, block I/O requests are placed on the queue by the function *__make_request*, which is also responsible for invoking the driver's *request*function. Block drivers that need more control over request queueing, however, can replace that function with their own "make request" function. The RAID and LVM drivers do so, providing their own variant that, eventually, requeues each I/O request (with different block numbers) to the appropriate low-level device (or devices) that make up the higher-level device. A RAM-disk driver, instead, can execute the I/O operation directly.

sbull, when loaded with the `noqueue=1` option on 2.4 systems, will provide its own "make request" function and operate without a request queue. The first step in this scenario is to replace

__make_request. The "make request" function pointer is stored in the request queue, and can be changed with *blk_queue_make_request*:

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

The `make_request_fn` type, in turn, is defined as follows:

```
typedef int (make_request_fn) (request_queue_t *q, int rw,
              struct buffer_head *bh);
```

The "make request" function must arrange to transfer the given block, and see to it that the *b_end_io* function is called when the transfer is done. The kernel does *not* hold the `io_request_lock` lock when calling the *make_request_fn* function, so the function must acquire the lock itself if it will be manipulating the request queue. If the transfer has been set up (not necessarily completed), the function should return 0.

The phrase "arrange to transfer" was chosen carefully; often a driver-specific make request function will not actually transfer the data. Consider a RAID device. What the function really needs to do is to map the I/O operation onto one of its constituent devices, then invoke that device's driver to actually do the work. This mapping is done by setting the `b_rdev` member of the `buffer_head` structure to the number of the "real" device that will do the transfer, then signaling that the block still needs to be written by returning a nonzero value.

When the kernel sees a nonzero return value from the make request function, it concludes that the job is not done and will try again. But first it will look up the make request function for the device indicated in the `b_rdev` field. Thus, in the RAID case, the RAID driver's "make request" function will *not* be called again; instead, the kernel will pass the block to the appropriate function for the underlying device.

sbull, at initialization time, sets up its make request function as follows:

```
if (noqueue)
    blk_queue_make_request(BLK_DEFAULT_QUEUE(major), sbull_make_request);
```

It does not call *blk_init_queue* when operating in this mode, because the request queue will not be used.

When the kernel generates a request for an *sbull* device, it will call *sbull_make_request*, which is as follows:

```
int sbull_make_request(request_queue_t *queue, int rw,
                       struct buffer_head *bh)
{
    u8 *ptr;

    /* Figure out what we are doing */
    Sbull_Dev *device = sbull_devices + MINOR(bh->b_rdev);
    ptr = device->data + bh->b_rsector * sbull_hardsect;

    /* Paranoid check; this apparently can really happen */
    if (ptr + bh->b_size > device->data + sbull_blksize*sbull_size) {
        static int count = 0;
```

```
            if (count++ < 5)
                printk(KERN_WARNING "sbull: request past end of device\n");
            bh->b_end_io(bh, 0);
            return 0;
        }

        /* This could be a high-memory buffer; shift it down */
    #if CONFIG_HIGHMEM
        bh = create_bounce(rw, bh);
    #endif

        /* Do the transfer */
        switch(rw) {
        case READ:
        case READA:  /* Read ahead */
            memcpy(bh->b_data, ptr, bh->b_size); /* from sbull to buffer */
            bh->b_end_io(bh, 1);
            break;
        case WRITE:
            refile_buffer(bh);
            memcpy(ptr, bh->b_data, bh->b_size); /* from buffer to sbull */
            mark_buffer_uptodate(bh, 1);
            bh->b_end_io(bh, 1);
            break;
        default:
            /* can't happen */
            bh->b_end_io(bh, 0);
            break;
        }

        /* Nonzero return means we're done */
        return 0;
    }
```

For the most part, this code should look familiar. It contains the usual calculations to determine where the block lives within the *sbull* device and uses *memcpy* to perform the operation. Because the operation completes immediately, it is able to call `bh->b_end_io` to indicate the completion of the operation, and it returns 0 to the kernel.

There is, however, one detail that the "make request" function must take care of. The buffer to be transferred could be resident in high memory, which is not directly accessible by the kernel. High memory is covered in detail in Chapter 13, "mmap and DMA". We won't repeat the discussion here; suffice it to say that one way to deal with the problem is to replace a high-memory buffer with one that is in accessible memory. The function *create_bounce*will do so, in a way that is transparent to the driver. The kernel normally uses *create_bounce* before placing buffers in the driver's request queue; if the driver implements its own *make_request_fn*, however, it must take care of this task itself.

How Mounting and Unmounting Works

Block devices differ from char devices and normal files in that they can be mounted on the computer's filesystem. Mounting provides a level of indirection not seen with char devices, which are accessed

through a `struct file` pointer that is held by a specific process. When a filesystem is mounted, there is no process holding that `file` structure.

When the kernel mounts a device in the filesystem, it invokes the normal *open* method to access the driver. However, in this case both the `filp` and `inode` arguments to *open* are dummy variables. In the `file` structure, only the `f_mode` and `f_flags` fields hold anything meaningful; in the `inode` structure only `i_rdev` may be used. The remaining fields hold random values and should not be used. The value of `f_mode` tells the driver whether the device is to be mounted read-only (`f_mode == FMODE_READ`) or read/write (`f_mode == (FMODE_READ|FMODE_WRITE)`).

This interface may seem a little strange; it is done this way for two reasons. First is that the *open* method can still be called normally by a process that accesses the device directly -- the *mkfs* utility, for example. The other reason is a historical artifact: block devices once used the same `file_operations` structure as char devices, and thus had to conform to the same interface.

Other than the limitations on the arguments to the *open* method, the driver does not really see anything unusual when a filesystem is mounted. The device is opened, and then the *request* method is invoked to transfer blocks back and forth. The driver cannot really tell the difference between operations that happen in response to an individual process (such as *fsck*) and those that originate in the filesystem layers of the kernel.

As far as *umount* is concerned, it just flushes the buffer cache and calls the *release* driver method. Since there is no meaningful `filp` to pass to the *release* method, the kernel uses `NULL`. Since the *release*implementation of a block driver can't use `filp->private_data` to access device information, it uses `inode->i_rdev` to differentiate between devices instead. This is how *sbull*implements *release*:

```
int sbull_release (struct inode *inode, struct file *filp)
{
    Sbull_Dev *dev = sbull_devices + MINOR(inode->i_rdev);

    spin_lock(&dev->lock);
    dev->usage--;
    MOD_DEC_USE_COUNT;
    spin_unlock(&dev->lock);
    return 0;
}
```

Other driver functions are not affected by the "missing `filp`" problem because they aren't involved with mounted filesystems. For example, *ioctl* is issued only by processes that explicitly *open* the device.

The ioctl Method

Like char devices, block devices can be acted on by using the *ioctl* system call. The only relevant difference between block and char *ioctl* implementations is that block drivers share a number of common *ioctl*commands that most drivers are expected to support.

The commands that block drivers usually handle are the following, declared in `<linux/fs.h>`.

BLKGETSIZE

Retrieve the size of the current device, expressed as the number of sectors. The value of `arg` passed in by the system call is a pointer to a `long` value and should be used to copy the size to a user-space variable. This *ioctl* command is used, for instance, by *mkfs* to know the size of the filesystem being created.

BLKFLSBUF

Literally, "flush buffers." The implementation of this command is the same for every device and is shown later with the sample code for the whole *ioctl* method.

BLKRRPART

Reread the partition table. This command is meaningful only for partitionable devices, introduced later in this chapter.

BLKRAGET
BLKRASET

Used to get and change the current block-level read-ahead value (the one stored in the `read_ahead` array) for the device. For `GET`, the current value should be written to user space as a `long` item using the pointer passed to *ioctl* in `arg`; for `SET`, the new value is passed as an argument.

BLKFRAGET
BLKFRASET

Get and set the filesystem-level read-ahead value (the one stored in `max_readahead`) for this device.

BLKROSET
BLKROGET

These commands are used to change and check the read-only flag for the device.

BLKSECTGET
BLKSECTSET

These commands retrieve and set the maximum number of sectors per request (as stored in `max_sectors`).

BLKSSZGET

Returns the sector size of this block device in the integer variable pointed to by the caller; this size comes directly from the `hardsect_size` array.

BLKPG

The `BLKPG` command allows user-mode programs to add and delete partitions. It is implemented by *blk_ioctl* (described shortly), and no drivers in the mainline kernel provide their own implementation.

BLKELVGET
BLKELVSET

These commands allow some control over how the elevator request sorting algorithm works. As with `BLKPG`, no driver implements them directly.

HDIO_GETGEO

Defined in `<linux/hdreg.h>` and used to retrieve the disk geometry. The geometry should be written to user space in a `struct hd_geometry`, which is declared in *hdreg.h* as well. *sbull* shows the general implementation for this command.

The `HDIO_GETGEO` command is the most commonly used of a series of `HDIO_` commands, all defined in `<linux/hdreg.h>`. The interested reader can look in *ide.c* and *hd.c* for more information about these commands.

Almost all of these *ioctl* commands are implemented in the same way for all block devices. The 2.4 kernel has provided a function, *blk_ioctl*, that may be called to implement the common commands; it is declared in `<linux/blkpg.h>`. Often the only ones that must be implemented in the driver itself are `BLKGETSIZE` and `HDIO_GETGEO`. The driver can then safely pass any other commands to *blk_ioctl* for handling.

The *sbull* device supports only the general commands just listed, because implementing device-specific commands is no different from the implementation of commands for char drivers. The *ioctl* implementation for *sbull* is as follows:

```
int sbull_ioctl (struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg)
{
    int err;
    long size;
    struct hd_geometry geo;

    PDEBUG("ioctl 0x%x 0x%lx\n", cmd, arg);
    switch(cmd) {

      case BLKGETSIZE:
        /* Return the device size, expressed in sectors */
        if (!arg) return -EINVAL; /* NULL pointer: not valid */
        err = ! access_ok (VERIFY_WRITE, arg, sizeof(long));
        if (err) return -EFAULT;
        size = blksize*sbull_sizes[MINOR(inode->i_rdev)]
                / sbull_hardsects[MINOR(inode->i_rdev)];
        if (copy_to_user((long *) arg, &size, sizeof (long)))
            return -EFAULT;
        return 0;

      case BLKRRPART: /* reread partition table: can't do it */
        return -ENOTTY;

      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.
         */
        err = ! access_ok(VERIFY_WRITE, arg, sizeof(geo));
        if (err) return -EFAULT;
        size = sbull_size * blksize / sbull_hardsect;
        geo.cylinders = (size & ~0x3f) >> 6;
        geo.heads = 4;
        geo.sectors = 16;
        geo.start = 4;
        if (copy_to_user((void *) arg, &geo, sizeof(geo)))
            return -EFAULT;
        return 0;

      default:
        /*
         * For ioctls we don't understand, let the block layer
         * handle them.
         */
        return blk_ioctl(inode->i_rdev, cmd, arg);
    }

    return -ENOTTY; /* unknown command */
}
```

The PDEBUG statement at the beginning of the function has been left in so that when you compile the module, you can turn on debugging to see which *ioctl* commands are invoked on the device.

Removable Devices

Thus far, we have ignored the final two file operations in the block_device_operations structure, which deal with devices that support removable media. It's now time to look at them; *sbull* isn't actually removable but it pretends to be, and therefore it implements these methods.

The operations in question are *check_media_change*and *revalidate*. The former is used to find out if the device has changed since the last access, and the latter re-initializes the driver's status after a disk change.

As far as *sbull* is concerned, the data area associated with a device is released half a minute after its usage count drops to zero. Leaving the device unmounted (or closed) long enough simulates a disk change, and the next access to the device allocates a new memory area.

This kind of "timely expiration" is implemented using a kernel timer.

check_media_change

The checking function receives kdev_t as a single argument that identifies the device. The return value is 1 if the medium has been changed and 0 otherwise. A block driver that doesn't support removable devices can avoid declaring the function by setting bdops->check_media_change to NULL.

It's interesting to note that when the device is removable but there is no way to know if it changed,

returning 1 is a safe choice. This is the behavior of the IDE driver when dealing with removable disks.

The implementation in *sbull* returns 1 if the device has already been removed from memory due to the timer expiration, and 0 if the data is still valid. If debugging is enabled, it also prints a message to the system logger; the user can thus verify when the method is called by the kernel.

```
int sbull_check_change(kdev_t i_rdev)
{
    int minor = MINOR(i_rdev);
    Sbull_Dev *dev = sbull_devices + minor;

    PDEBUG("check_change for dev %i\n",minor);
    if (dev->data)
        return 0; /* still valid */
    return 1; /* expired */
}
```

Revalidation

The validation function is called when a disk change is detected. It is also called by the various *stat* system calls implemented in version 2.1 of the kernel. The return value is currently unused; to be safe, return 0 to indicate success and a negative error code in case of error.

The action performed by *revalidate* is device specific, but *revalidate* usually updates the internal status information to reflect the new device.

In *sbull*, the *revalidate* method tries to allocate a new data area if there is not already a valid area.

```
int sbull_revalidate(kdev_t i_rdev)
{
    Sbull_Dev *dev = sbull_devices + MINOR(i_rdev);

    PDEBUG("revalidate for dev %i\n",MINOR(i_rdev));
    if (dev->data)
        return 0;
    dev->data = vmalloc(dev->size);
    if (!dev->data)
        return -ENOMEM;
    return 0;
}
```

Extra Care

Drivers for removable devices should also check for a disk change when the device is opened. The kernel provides a function to cause this check to happen:

```
int check_disk_change(kdev_t dev);
```

The return value is nonzero if a disk change was detected. The kernel automatically calls *check_disk_change* at *mount* time, but not at *open*time.

Some programs, however, directly access disk data without mounting the device: *fsck*, *mcopy*, and *fdisk*are examples of such programs. If the driver keeps status information about removable devices in memory, it should call the kernel *check_disk_change* function when the device is first opened. This function uses the driver methods (*check_media_change* and *revalidate*), so nothing special has to be implemented in *open* itself.

Here is the *sbull* implementation of *open*, which takes care of the case in which there's been a disk change:

```
int sbull_open (struct inode *inode, struct file *filp)
{
    Sbull_Dev *dev; /* device information */
    int num = MINOR(inode->i_rdev);

    if (num >= sbull_devs) return -ENODEV;
    dev = sbull_devices + num;

    spin_lock(&dev->lock);
    /* revalidate on first open and fail if no data is there */
    if (!dev->usage) {
        check_disk_change(inode->i_rdev);
        if (!dev->data)
        {
            spin_unlock (&dev->lock);
            return -ENOMEM;
        }
    }
    dev->usage++;
    spin_unlock(&dev->lock);
    MOD_INC_USE_COUNT;
    return 0;          /* success */
}
```

Nothing else needs to be done in the driver for a disk change. Data is corrupted anyway if a disk is changed while its open count is greater than zero. The only way the driver can prevent this problem from happening is for the usage count to control the door lock in those cases where the physical device supports it. Then *open* and *close* can disable and enable the lock appropriately.

Partitionable Devices

Most block devices are not used in one large chunk. Instead, the system administrator expects to be able to *partition* the device -- to split it into several independent pseudodevices. If you try to create partitions on an *sbull* device with *fdisk*, you'll run into problems. The *fdisk* program calls the partitions */dev/sbull01*, */dev/sbull02*, and so on, but those names don't exist on the filesystem. More to the point, there is no mechanism in place for binding those names to partitions in the *sbull* device. Something more must be done before a block device can be partitioned.

To demonstrate how partitions are supported, we introduce a new device called *spull*, a "Simple Partitionable Utility." It is far simpler than *sbull*, lacking the request queue management and some flexibility (like the ability to change the hard-sector size). The device resides in the *spull* directory and

is completely detached from *sbull*, even though they share some code.

To be able to support partitions on a device, we must assign several minor numbers to each physical device. One number is used to access the whole device (for example, */dev/hda*), and the others are used to access the various partitions (such as */dev/hda1*). Since *fdisk* creates partition names by adding a numerical suffix to the whole-disk device name, we'll follow the same naming convention in the *spull* driver.

The device nodes implemented by *spull* are called `pd`, for "partitionable disk." The four whole devices (also called *units*) are thus named */dev/pda* through */dev/pdd*; each device supports at most 15 partitions. Minor numbers have the following meaning: the least significant four bits represent the partition number (where 0 is the whole device), and the most significant four bits represent the unit number. This convention is expressed in the source file by the following macros:

```
#define MAJOR_NR spull_major /* force definitions on in blk.h */
int spull_major; /* must be declared before including blk.h */

#define SPULL_SHIFT 4                         /* max 16 partitions  */
#define SPULL_MAXNRDEV 4                      /* max 4 device units */
#define DEVICE_NR(device) (MINOR(device)>>SPULL_SHIFT)
#define DEVICE_NAME "pd"                      /* name for messaging */
```

The *spull* driver also hardwires the value of the hard-sector size in order to simplify the code:

```
#define SPULL_HARDSECT 512  /* 512-byte hardware sectors */
```

The Generic Hard Disk

Every partitionable device needs to know how it is partitioned. The information is available in the partition table, and part of the initialization process consists of decoding the partition table and updating the internal data structures to reflect the partition information.

This decoding isn't easy, but fortunately the kernel offers "generic hard disk" support usable by all block drivers. Such support considerably reduces the amount of code needed in the driver for handling partitions. Another advantage of the generic support is that the driver writer doesn't need to understand how the partitioning is done, and new partitioning schemes can be supported in the kernel without requiring changes to driver code.

A block driver that supports partitions must include `<linux/genhd.h>` and should declare a `struct gendisk` structure. This structure describes the layout of the disk(s) provided by the driver; the kernel maintains a global list of such structures, which may be queried to see what disks and partitions are available on the system.

Before we go further, let's look at some of the fields in `struct gendisk`. You'll need to understand them in order to exploit generic device support.

`int major`

The major number for the device that the structure refers to.

`const char *major_name`

The base name for devices belonging to this major number. Each device name is derived from this name by adding a letter for each unit and a number for each partition. For example, "hd" is the base name that is used to build */dev/hda1* and */dev/hdb3*. In modern kernels, the full length of the disk name can be up to 32 characters; the 2.0 kernel, however, was more restricted. Drivers wishing to be backward portable to 2.0 should limit the `major_name` field to five characters. The name for *spull* is `pd` ("partitionable disk").

`int minor_shift`

The number of bit shifts needed to extract the drive number from the device minor number. In *spull* the number is 4. The value in this field should be consistent with the definition of the macro `DEVICE_NR(device)` (see "Section 12.2, "The Header File blk.h""). The macro in *spull*expands to `device>>4`.

`int max_p`

The maximum number of partitions. In our example, `max_p` is 16, or more generally, `1 << minor_shift`.

`struct hd_struct *part`

The decoded partition table for the device. The driver uses this item to determine what range of the disk's sectors is accessible through each minor number. The driver is responsible for allocation and deallocation of this array, which most drivers implement as a static array of `max_nr << minor_shift` structures. The driver should initialize the array to zeros before the kernel decodes the partition table.

`int *sizes`

An array of integers with the same information as the global `blk_size` array. In fact, they are usually the same array. The driver is responsible for allocating and deallocating the `sizes` array. Note that the partition check for the device copies this pointer to `blk_size`, so a driver handling partitionable devices doesn't need to allocate the latter array.

`int nr_real`

The number of real devices (units) that exist.

`void *real_devices`

A private area that may be used by the driver to keep any additional required information.

`void struct gendisk *next`

A pointer used to implement the linked list of generic hard-disk structures.

`struct block_device_operations *fops;`

A pointer to the block operations structure for this device.

Many of the fields in the `gendisk` structure are set up at initialization time, so the compile-time setup

is relatively simple:

```
struct gendisk spull_gendisk = {
    major:              0,            /* Major number assigned later */
    major_name:         "pd",         /* Name of the major device */
    minor_shift:        SPULL_SHIFT,  /* Shift to get device number */
    max_p:              1 << SPULL_SHIFT, /* Number of partitions */
    fops:               &spull_bdops, /* Block dev operations */
/* everything else is dynamic */
};
```

Partition Detection

When a module initializes itself, it must set things up properly for partition detection. Thus, *spull* starts by setting up the `spull_sizes` array for the `gendisk` structure (which also gets stored in `blk_size[MAJOR_NR]` and in the `sizes` field of the `gendisk` structure) and the `spull_partitions` array, which holds the actual partition information (and gets stored in the `part` member of the `gendisk` structure). Both of these arrays are initialized to zeros at this time. The code looks like this:

```
spull_sizes = kmalloc( (spull_devs << SPULL_SHIFT) * sizeof(int),
                      GFP_KERNEL);
if (!spull_sizes)
    goto fail_malloc;

/* Start with zero-sized partitions, and correctly sized units */
memset(spull_sizes, 0, (spull_devs << SPULL_SHIFT) * sizeof(int));
for (i=0; i< spull_devs; i++)
    spull_sizes[i<<SPULL_SHIFT] = spull_size;
blk_size[MAJOR_NR] = spull_gendisk.sizes = spull_sizes;

/* Allocate the partitions array. */
spull_partitions = kmalloc( (spull_devs << SPULL_SHIFT) *
                           sizeof(struct hd_struct), GFP_KERNEL);
if (!spull_partitions)
    goto fail_malloc;

memset(spull_partitions, 0, (spull_devs << SPULL_SHIFT) *
       sizeof(struct hd_struct));
/* fill in whole-disk entries */
for (i=0; i < spull_devs; i++)
    spull_partitions[i << SPULL_SHIFT].nr_sects =
        spull_size*(blksize/SPULL_HARDSECT);
spull_gendisk.part = spull_partitions;
spull_gendisk.nr_real = spull_devs;
```

The driver should also include its `gendisk` structure on the global list. There is no kernel-supplied function for adding `gendisk` structures; it must be done by hand:

```
spull_gendisk.next = gendisk_head;
gendisk_head = &spull_gendisk;
```

In practice, the only thing the system does with this list is to implement */proc/partitions*.

The *register_disk* function, which we have already seen briefly, handles the job of reading the disk's partition table.

```
register_disk(struct gendisk *gd, int drive, unsigned minors,
              struct block_device_operations *ops, long size);
```

Here, `gd` is the `gendisk` structure that we built earlier, `drive` is the device number, `minors` is the number of partitions supported, `ops` is the `block_device_operations` structure for the driver, and `size` is the size of the device in sectors.

Fixed disks might read the partition table only at module initialization time and when `BLKRRPART` is invoked. Drivers for removable drives will also need to make this call in the *revalidate* method. Either way, it is important to remember that *register_disk* will call your driver's *request* function to read the partition table, so the driver must be sufficiently initialized at that point to handle requests. You should also not have any locks held that will conflict with locks acquired in the *request* function. *register_disk* must be called for each disk actually present on the system.

spull sets up partitions in the *revalidate* method:

```
int spull_revalidate(kdev_t i_rdev)
{
  /* first partition, # of partitions */
  int part1 = (DEVICE_NR(i_rdev) << SPULL_SHIFT) + 1;
  int npart = (1 << SPULL_SHIFT) -1;

  /* first clear old partition information */
  memset(spull_gendisk.sizes+part1, 0, npart*sizeof(int));
  memset(spull_gendisk.part +part1, 0, npart*sizeof(struct hd_struct));
  spull_gendisk.part[DEVICE_NR(i_rdev) << SPULL_SHIFT].nr_sects =
          spull_size << 1;

  /* then fill new info */
  printk(KERN_INFO "Spull partition check: (%d) ", DEVICE_NR(i_rdev));
  register_disk(&spull_gendisk, i_rdev, SPULL_MAXNRDEV, &spull_bdops,
                  spull_size << 1);
  return 0;
}
```

It's interesting to note that *register_disk*prints partition information by repeatedly calling

```
printk(" %s", disk_name(hd, minor, buf));
```

That's why *spull* prints a leading string. It's meant to add some context to the information that gets stuffed into the system log.

When a partitionable module is unloaded, the driver should arrange for all the partitions to be flushed, by calling *fsync_dev* for every supported major/minor pair. All of the relevant memory should be freed

as well, of course. The cleanup function for *spull* is as follows:

```
for (i = 0; i < (spull_devs << SPULL_SHIFT); i++)
    fsync_dev(MKDEV(spull_major, i)); /* flush the devices */
blk_cleanup_queue(BLK_DEFAULT_QUEUE(major));
read_ahead[major] = 0;
kfree(blk_size[major]); /* which is gendisk->sizes as well */
blk_size[major] = NULL;
kfree(spull_gendisk.part);
kfree(blksize_size[major]);
blksize_size[major] = NULL;
```

It is also necessary to remove the `gendisk` structure from the global list. There is no function provided to do this work, so it's done by hand:

```
for (gdp = &gendisk_head; *gdp; gdp = &((*gdp)->next))
    if (*gdp == &spull_gendisk) {
        *gdp = (*gdp)->next;
        break;
    }
```

Note that there is no *unregister_disk* to complement the *register_disk* function. Everything done by *register_disk* is stored in the driver's own arrays, so there is no additional cleanup required at unload time.

Partition Detection Using initrd

If you want to mount your root filesystem from a device whose driver is available only in modularized form, you must use the *initrd* facility offered by modern Linux kernels. We won't introduce *initrd* here; this subsection is aimed at readers who know about *initrd* and wonder how it affects block drivers. More information on *initrd* can be found in *Documentation/initrd.txt* in the kernel source.

When you boot a kernel with *initrd*, it establishes a temporary running environment before it mounts the real root filesystem. Modules are usually loaded from within the RAM disk being used as the temporary root file system.

Because the *initrd* process is run after all boot-time initialization is complete (but before the real root filesystem has been mounted), there's no difference between loading a normal module and loading one living in the *initrd* RAM disk. If a driver can be correctly loaded and used as a module, all Linux distributions that have *initrd* available can include the driver on their installation disks without requiring you to hack in the kernel source.

The Device Methods for spull

We have seen how to initialize partitionable devices, but not yet how to access data within the partitions. To do that, we need to make use of the partition information stored in the `gendisk->part` array by *register_disk*. This array is made up of `hd_struct` structures, and is indexed by the minor number. The `hd_struct` has two fields of interest: `start_sect` tells where a given partition starts on the disk, and `nr_sects` gives the size of that partition.

Here we will show how *spull* makes use of that information. The following code includes only those parts of *spull* that differ from *sbull*, because most of the code is exactly the same.

First of all, *open* and *close*must keep track of the usage count for each device. Because the usage count refers to the physical device (unit), the following declaration and assignment is used for the `dev` variable:

```
Spull_Dev *dev = spull_devices + DEVICE_NR(inode->i_rdev);
```

The `DEVICE_NR` macro used here is the one that must be declared before `<linux/blk.h>` is included; it yields the physical device number without taking into account which partition is being used.

Although almost every device method works with the physical device as a whole, *ioctl* should access specific information for each partition. For example, when *mkfs*calls *ioctl* to retrieve the size of the device on which it will build a filesystem, it should be told the size of the partition of interest, not the size of the whole device. Here is how the `BLKGETSIZE` *ioctl* command is affected by the change from one minor number per device to multiple minor numbers per device. As you might expect, `spull_gendisk->part` is used as the source of the partition size.

```
case BLKGETSIZE:
  /* Return the device size, expressed in sectors */
  err = ! access_ok (VERIFY_WRITE, arg, sizeof(long));
  if (err) return -EFAULT;
  size = spull_gendisk.part[MINOR(inode->i_rdev)].nr_sects;
  if (copy_to_user((long *) arg, &size, sizeof (long)))
  return -EFAULT;
  return 0;
```

The other *ioctl* command that is different for partitionable devices is `BLKRRPART`. Rereading the partition table makes sense for partitionable devices and is equivalent to revalidating a disk after a disk change:

```
case BLKRRPART: /* re-read partition table */
  return spull_revalidate(inode->i_rdev);
```

But the major difference between *sbull* and *spull* is in the *request* function. In *spull*, the *request*function needs to use the partition information in order to correctly transfer data for the different minor numbers. Locating the transfer is done by simply adding the starting sector to that provided in the request; the partition size information is then used to be sure the request fits within the partition. Once that is done, the implementation is the same as for *sbull*.

Here are the relevant lines in *spull_request*:

```
ptr = device->data +
      (spull_partitions[minor].start_sect + req->sector)*SPULL_HARDSECT;
size = req->current_nr_sectors*SPULL_HARDSECT;
/*
 * Make sure that the transfer fits within the device.
 */
```

```
if (req->sector + req->current_nr_sectors >
                spull_partitions[minor].nr_sects) {
    static int count = 0;
    if (count++ < 5)
      printk(KERN_WARNING "spull: request past end of partition\n");
    return 0;
}
```

The number of sectors is multiplied by the hardware sector size (which, remember, is hardwired in *spull*) to get the size of the partition in bytes.

Interrupt-Driven Block Drivers

When a driver controls a real hardware device, operation is usually interrupt driven. Using interrupts helps system performance by releasing the processor during I/O operations. In order for interrupt-driven I/O to work, the device being controlled must be able to transfer data asynchronously and to generate interrupts.

When the driver is interrupt driven, the *request*function spawns a data transfer and returns immediately without calling *end_request*. However, the kernel doesn't consider a request fulfilled unless *end_request*(or its component parts) has been called. Therefore, the top-half or the bottom-half interrupt handler calls *end_request* when the device signals that the data transfer is complete.

Neither *sbull* nor *spull* can transfer data without using the system microprocessor; however, *spull* is equipped with the capability of simulating interrupt-driven operation if the user specifies the `irq=1` option at load time. When `irq` is not 0, the driver uses a kernel timer to delay fulfillment of the current request. The length of the delay is the value of `irq`: the greater the value, the longer the delay.

As always, block transfers begin when the kernel calls the driver's *request* function. The *request* function for an interrupt-driven device instructs the hardware to perform the transfer and then returns; it does not wait for the transfer to complete. The *spull request* function performs the usual error checks and then calls *spull_transfer* to transfer the data (this is the task that a driver for real hardware performs asynchronously). It then delays acknowledgment until interrupt time:

```
void spull_irqdriven_request(request_queue_t *q)
{
    Spull_Dev *device;
    int status;
    long flags;

    /* If we are already processing requests, don't do any more now. */
    if (spull_busy)
            return;

    while(1) {
        INIT_REQUEST;  /* returns when queue is empty */

        /* Which "device" are we using? */
        device = spull_locate_device (CURRENT);
        if (device == NULL) {
```

```
            end_request(0);
            continue;
        }
        spin_lock_irqsave(&device->lock, flags);

        /* Perform the transfer and clean up. */
        status = spull_transfer(device, CURRENT);
        spin_unlock_irqrestore(&device->lock, flags);
        /* ... and wait for the timer to expire -- no end_request(1) */
        spull_timer.expires = jiffies + spull_irq;
        add_timer(&spull_timer);
        spull_busy = 1;
        return;
    }
}
```

New requests can accumulate while the device is dealing with the current one. Because reentrant calls are almost guaranteed in this scenario, the *request* function sets a `spull_busy` flag so that only one transfer happens at any given time. Since the entire function runs with the `io_request_lock` held (the kernel, remember, obtains this lock before calling the *request* function), there is no need for particular care in testing and setting the `busy` flag. Otherwise, an `atomic_t` item should have been used instead of an `int` variable in order to avoid race conditions.

The interrupt handler has a couple of tasks to perform. First, of course, it must check the status of the outstanding transfer and clean up the request. Then, if there are further requests to be processed, the interrupt handler is responsible for getting the next one started. To avoid code duplication, the handler usually just calls the *request* function to start the next transfer. Remember that the *request* function expects the caller to hold the `io_request_lock`, so the interrupt handler will have to obtain it. The *end_request* function also requires this lock, of course.

In our sample module, the role of the interrupt handler is performed by the function invoked when the timer expires. That function calls *end_request* and schedules the next data transfer by calling the *request* function. In the interest of code simplicity, the *spull* interrupt handler performs all this work at "interrupt" time; a real driver would almost certainly defer much of this work and run it from a task queue or tasklet.

```
/* this is invoked when the timer expires */
void spull_interrupt(unsigned long unused)
{
    unsigned long flags

    spin_lock_irqsave(&io_request_lock, flags);
    end_request(1);    /* This request is done - we always succeed */

    spull_busy = 0;  /* We have io_request_lock, no request conflict */
    if (! QUEUE_EMPTY) /* more of them? */
        spull_irqdriven_request(NULL);  /* Start the next transfer */
    spin_unlock_irqrestore(&io_request_lock, flags);
}
```

If you try to run the interrupt-driven flavor of the *spull* module, you'll barely notice the added delay.

The device is almost as fast as it was before because the buffer cache avoids most data transfers between memory and the device. If you want to perceive how a slow device behaves, you can specify a bigger value for `irq=` when loading *spull*.

Backward Compatibility

Much has changed with the block device layer, and most of those changes happened between the 2.2 and 2.4 stable releases. Here is a quick summary of what was different before. As always, you can look at the drivers in the sample source, which work on 2.0, 2.2, and 2.4, to see how the portability challenges have been handled.

The `block_device_operations` structure did not exist in Linux 2.2. Instead, block drivers used a `file_operations` structure just like char drivers. The *check_media_change* and *revalidate* methods used to be a part of that structure. The kernel also provided a set of generic functions -- *block_read*, *block_write*, and *block_fsync* -- which most drivers used in their `file_operations` structures. A typical 2.2 or 2.0 `file_operations` initialization looked like this:

```
struct file_operations sbull_bdops = {
    read:       block_read,
    write:      block_write,
    ioctl:      sbull_ioctl,
    open:       sbull_open,
    release:    sbull_release,
    fsync:      block_fsync,
    check_media_change: sbull_check_change,
    revalidate: sbull_revalidate
};
```

Note that block drivers are subject to the same changes in the `file_operations` prototypes between 2.0 and 2.2 as char drivers.

In 2.2 and previous kernels, the *request* function was stored in the `blk_dev` global array. Initialization required a line like

```
blk_dev[major].request_fn = sbull_request;
```

Because this method allows for only one queue per major number, the multiqueue capability of 2.4 kernels is not present in earlier releases. Because there was only one queue, the *request* function did not need the queue as an argument, so it took none. Its prototype was as follows:

```
void (*request) (void);
```

Also, all queues had active heads, so `blk_queue_headactive` did not exist.

There was no *blk_ioctl* function in 2.2 and prior releases. There was, however, a macro called `RO_IOCTLS`, which could be inserted in a `switch` statement to implement `BLKROSET` and `BLKROGET`. *sysdep.h* in the sample source includes an implementation of *blk_ioctl* that uses `RO_IOCTLS` and implements a few other of the standard *ioctl* commands as well:

```
#ifdef RO_IOCTLS
static inline int blk_ioctl(kdev_t dev, unsigned int cmd,
                            unsigned long arg)
{
    int err;

    switch (cmd) {
      case BLKRAGET: /* return the read-ahead value */
        if (!arg)  return -EINVAL;
        err = ! access_ok(VERIFY_WRITE, arg, sizeof(long));
        if (err) return -EFAULT;
        PUT_USER(read_ahead[MAJOR(dev)],(long *) arg);
        return 0;

      case BLKRASET: /* set the read-ahead value */
        if (!capable(CAP_SYS_ADMIN)) return -EACCES;
        if (arg > 0xff) return -EINVAL; /* limit it */
        read_ahead[MAJOR(dev)] = arg;
        return 0;

      case BLKFLSBUF: /* flush */
        if (! capable(CAP_SYS_ADMIN)) return -EACCES; /* only root */
        fsync_dev(dev);
        invalidate_buffers(dev);
        return 0;

        RO_IOCTLS(dev, arg);
    }
    return -ENOTTY;
}
#endif  /* RO_IOCTLS */
```

The `BLKFRAGET`, `BLKFRASET`, `BLKSECTGET`, `BLKSECTSET`, `BLKELVGET`, and `BLKELVSET` commands were added with Linux 2.2, and `BLKPG` was added in 2.4.

Linux 2.0 did not have the `max_readahead` array. The `max_segments` array, instead, existed and was used in Linux 2.0 and 2.2, but device drivers did not normally need to set it.

Finally, *register_disk* did not exist until Linux 2.4. There was, instead, a function called *resetup_one_dev*, which performed a similar function:

```
resetup_one_dev(struct gendisk *gd, int drive);
```

register_disk is emulated in *sysdep.h* with the following code:

```
static inline void register_disk(struct gendisk *gdev, kdev_t dev,
                unsigned minors, struct file_operations *ops, long size)
{
    if (! gdev)
        return;
    resetup_one_dev(gdev, MINOR(dev) >> gdev->minor_shift);
}
```

Linux 2.0 was different, of course, in not supporting any sort of fine-grained SMP. Thus, there was no `io_request_lock` and much less need to worry about concurrent access to the I/O request queue.

One final thing worth keeping in mind: although nobody really knows what will happen in the 2.5 development series, a major block device overhaul is almost certain. Many people are unhappy with the design of this layer, and there is a lot of pressure to redo it.

Quick Reference

The most important functions and macros used in writing block drivers are summarized here. To save space, however, we do not list the fields of `struct request`, `struct buffer_head`, or `struct genhd`, and we omit the predefined *ioctl* commands.

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

> These functions are in charge of device registration in the module's initialization function and device removal in the cleanup function.

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

> The first function initializes a queue and establishes the *request* function; the second is used at cleanup time.

`BLK_DEFAULT_QUEUE(major)`

> This macro returns a default I/O request queue for a given major number.

`struct blk_dev_struct blk_dev[MAX_BLKDEV];`

> This array is used by the kernel to find the proper queue for a given request.

`int read_ahead[];`
`int max_readahead[][];`

> `read_ahead` contains block-level read-ahead values for every major number. A value of 8 is reasonable for devices like hard disks; the value should be greater for slower media. `max_readahead` contains filesystem-level read-ahead values for every major and minor number, and is not usually changed from the system default.

`int max_sectors[][];`

> This array, indexed by both major and minor number, holds the maximum number of sectors that should be merged into a single I/O request.

`int blksize_size[][];`
`int blk_size[][];`
`int hardsect_size[][];`

These two-dimensional arrays are indexed by major and minor number. The driver is responsible for allocating and deallocating the row in the matrix associated with its major number. The arrays represent the size of device blocks in bytes (it usually is 1 KB), the size of each minor device in kilobytes (not blocks), and the size of the hardware sector in bytes.

MAJOR_NR
DEVICE_NAME
DEVICE_NR(kdev_t device)
DEVICE_INTR
#include <linux/blk.h>

These macros must be defined by the driver *before*it includes `<linux/blk.h>`, because they are used within that file. `MAJOR_NR` is the major number for the device, `DEVICE_NAME` is the name of the device to be used in error messages, `DEVICE_NR` returns the minor number of the *physical* device referred to by a device number, and `DEVICE_INTR` is a little-used symbol that points to the device's bottom-half interrupt handler.

spinlock_t io_request_lock;

The spinlock that must be held whenever an I/O request queue is being manipulated.

struct request *CURRENT;

This macro points to the current request when the default queue is being used. The request structure describes a data chunk to be transferred and is used by the driver's *request*function.

INIT_REQUEST;
end_request(int status);

`INIT_REQUEST` checks the next request on the queue and returns if there are no more requests to execute. *end_request* is called at the completion of a block request.

spinlock_t io_request_lock;

The I/O request lock must be held any time that the request queue is being manipulated.

struct request *blkdev_entry_next_request(struct list_head *head);
struct request *blkdev_next_request(struct request *req);
struct request *blkdev_prev_request(struct request *req);
blkdev_dequeue_request(struct request *req);
blkdev_release_request(struct request *req);

Various functions for working with the I/O request queue.

blk_queue_headactive(request_queue_t *queue, int active);

Indicates whether the first request in the queue is being actively processed by the driver or not.

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

Provides a function to handle block I/O requests directly out of the kernel.

end_that_request_first(struct request *req, int status, char *name);
end_that_request_last(struct request *req);

Handle the stages of completing a block I/O request. *end_that_request_last* is only called when all buffers in the request have been processed -- that is, when *end_that_request_first* returns 0.

```
bh->b_end_io(struct buffer_head *bh, int status);
```

Signals the completion of I/O on the given buffer.

```
int blk_ioctl(kdev_t dev, unsigned int cmd, unsigned long arg);
```

A utility function that implements most of the standard block device *ioctl* commands.

```
int check_disk_change(kdev_t dev);
```

This function checks to see if a media change has occurred on the given device, and calls the driver's *revalidate*method if a change is detected.

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

The generic hard disk allows Linux to support partitionable devices easily. The `gendisk` structure describes a generic disk; `gendisk_head` is the beginning of a linked list of structures describing all of the disks on the system.

```
void register_disk(struct gendisk *gd, int drive, unsigned minors, struct
block_device_operations *ops, long size);
```

This function scans the partition table of the disk and rewrites `genhd->part` to reflect the new partitioning.

© 2001, O'Reilly & Associates, Inc.